分散システムにおける故障の自律的な解析

Autonomous Fault Diagnosis in Clustered Distributed Environments

丸山 直也 †
Naoya Maruyama

松岡 聡 †,‡
Satoshi Matsuoka

† 東京工業大学
Tokyo Institute of Technology
naoya.maruyama@is.titech.ac.jp matsu@is.titech.ac.jp

‡ 国立情報学研究所
National Institute of Informatics

我々は実行トレースとそのモデル化に基づく障害解析支援技術を提案する。本手法は、正しく実行された場合の実行時関数呼び出しトレースを取得し、関数の推定出現確率からなる正例実行モデルを構築する。故障発生時には実行トレース中の関数呼び出しを正例実行モデルと比較し、モデルとの乖離を表す異常度を計算する。異常度は高い確率で出現する関数が呼ばれなかった場合、また低い確率で出現する関数が呼ばれた場合に高い値を持つ。計算した異常度の高い順に関数呼び出しを検証することで、より原因箇所の可能性の高い箇所を優先的に検証することができる。提案手法を評価するために、MPI 実装の一つである MPICH に本手法を適用し、InTrigger のような広域環境において観察されている既知のバグを検出可能か評価した。78 ノード、3 拠点からなる広域環境において、正常動作時、故障発生時の関数トレースを取得し、正例モデルの構築、故障トレースの異常度の計算を行った。その結果、 同バグによって通信路が切断された箇所を異常度の高い振る舞いとして特定し、分散システムにおけるバグ検出に有効であることを確認した。

1 はじめに

クラスタやグリッドなどに代表されるように、システムが大規模化、コモディティ化されるにつれて故障の原因特定が問題になってきている。これは大規模化により故障の原因箇所の特定が困難になることが原因としてあげられる。例えば数十ノード規模の小規模なクラスタでは故障箇所は各ノードを管理者が人手での診断も不可能ではないが、数百、数千ノード規模のクラスタでは非現実的である。また、システムを構成するハードウェア、ソフトウェアが多くのコモディティコンポーネントに分割されつつあることも、故障解析が困難になる原因の一つである。これは、構成コンポーネントの増加につれてコンポーネント間の動作検証が困難になるため、またシステム管理者がそれらの多数のコンポーネントからなるシステムを習熟、管理することが困難になるためである。

我々は上述の問題を解決する、クラスタ上の分散ミドルウェア向け故障解析支援技術を提案する。本手法は、システムの実行トレースとそのモデル化に基づく。まず、正しく実行された場合（正例）の実行時関数呼び出しトレースを取得し、コンパクトな正例実行モデルを構築する。故障発生時には実行トレース中の関数呼び出しを正例実行モデルと比較し、モデルとの乖離を表す異常度を計算する。計算した異常度の高い順に関数呼び出しを故障の解析者に提示する。この結果を基に、解析者は故障発生時の関数呼び出しについて異常度順に検証することで、より原因箇所の可能性の高い箇所を優先的に検証することができる。

本モデル化では各プロセスの関数トレースを分割し、分割された単位（実行ユニット）毎にモデルを構築する。分割は以下の仮定に基づく：クラスタ上の分散システムを構成するプロセスは、リモートからのリクエストメッセージを受信し、それに従った処理を行う無限ループから成る。この仮定に基づき、トレースを初期化部、リクエストハンドラ部、終了処理部に分割する。トレースを分割する理由は分散ミドルウェアを構成するプロセスは種々の異なるリクエストを処理するルーチンから構成されるのが一般的であり、関数トレースにはそれらのルーチンが多重化して現れるからである。これらの処理を別個に

モデル化することで、より意味的にまとまりのある単位で学習を行え、モデルの精度向上を期待できる。

実行ユニットからのモデル構築には、各関数の出現確率を用いる。出現確率は、各関数の出現回数をその呼び出し元関数の出現回数で割った値と推定する。この際、トレースよりコールツリーを構築し、各関数についてそのコールパス毎に処理することでコンテキストセンシティブな学習を行い、モデルを精緻化する。また、リクエスト処理実行ユニットについては、そのリクエストが送られてきた接続毎にグルーピングし、同一グループには単一のモデルを構築し、異なる接続同士の実行ユニットは別のモデルを構築する。これは同じ接続においては似た振る舞いが観測されるという予測に基づく。

構築した実行モデルを用いた故障解析は以下の手順で行う。まず、モデル構築時と同様に対象システムの関数トレースを取得し、トレースを実行ユニットへ分割する。各ユニットについて、モデル中の対応するユニットと比較し、モデルとの乖離を表す異常度を計算する。異常度はモデル中の各関数の推定出現確率を基に、モデル中に高い確率で出現する関数が呼ばれなかった場合、また、低い確率で出現する関数が呼ばれた場合に高い値を持つ。これらの一連の計算はすべてプロセス毎に個別に行い、分散処理可能である。従って、既存研究にあるようにトレースを一箇所に収集する必要がなく、スケーラビリティに優れる。最後に、各ユニットについての異常度を解析者に提示し、解析者による検証を支援する。

既存研究として、我々と同様にデータ解析技術に基づいた故障解析支援技術が提案されている [1,2,5–10]。これらも本提案と同様に正常なシステムの振る舞いと異常な振る舞いを学習することで、故障の検知とその解析を行う。しかしこれらの既存研究では、対象とするシステムが限定された限定されたプログラミングインターフェイスを使用したものでなければならない、解析にシステム全体の情報を一元的に集約する必要があり、スケーラブルでない、などの問題がある。

評価結果提案手法を評価するために、MPI 実装の一つである MPICH [4] に本手法を適用し、既知のバグを検出可能か評価した。斎藤らによって報告されている通り、MPICH のジョブマネージャには広域環境において出現確率が高くなるタイミングバグが存在する [11]。同ジョブマネージャの関数トレースを正例時、故障時共に取得し、正例モデルの構築、故障トレースの異常度の計算を行った。正例時のトレースは単一拠点の 58 ノードを用い、故障発生時のトレースは 3 拠点 78 ノードの資源を用いた。その結果、同バグによって通信路が切断された箇所を異常度の高い振る舞いとして特定し、分散システムにおけるバグ検出に本提案手法が有効であることを確認した。

2 分散システムのモデル化

故障解析に用いるシステムの実行モデルでは、分散システムにおける故障を異常と検知可能であること、その構築が自律的に行われること、広範囲のシステムに適用可能であることを目標とする。我々が提案する故障解析支援では、故障発生時の振る舞いをモデルと比較することで故障原因の解析を支援する。従って人手による解析の手間を最低限にするために、モデル構築は自律的に行われるべきである。また、手法の有用性、適用の手間を削減するために広範囲のシステムに容易に応用可能であるべきである。

本モデル化は、分散システムに対する以下の仮定、観察に基づく。プロセスはメッセージを受信してそれに対応する処理を行う無限ループ（メインループ）で構成される。処理の構成はメッセージのタイプに応じて複数あり、それらが `select` システムコールなどを用いて多重化して実行される。この観察は常に成り立つものではないが、特にクラスタ環境向けの多くのシステムに共通して成立する特徴と期待できる。実際に、4 節で示す評価実験の対象プログラム MPICH ジョブマネージャはこのような構成になっており、我々のモデル化が有効であった。また、クラスタ向けバッチスケジューラのひとつである Torque について予備的にソースコード、関数トレースを用いて解析したところ、同様の振る舞いであることを確認した。同仮定の他のシステムにおける適用可能性の調査は今後の課題である。

以上のシステムに対する観察に基づき、我々は以下の手法によりモデルを構築する。

1. 分散システムの各構成プロセスの関数トレースを取得する。
2. 実行ユニットを似た振る舞いが期待できるグループへ分割する。システム全体を構成するプロセスには役割の異なるものも含まれるのが一般的であるため、プロセス集合を同じ役割を持つもの同士に分割し、それらのプロセスグループ毎にモデルを構築する。 またさらに、トレースを

関連した呼び出しからなる部分トレースに分割し、その部分トレース毎にモデルを構築する。

3. 全プロセスのトレースより関数の出現回数を計算し、各関数の出現確率を推定する。

実行ユニットへのトレースの分割とプロセスグループへのシステムの分割により、似た振る舞いが期待される単位ごとにモデルを構築する。これによりモデルの精度、すなわち関数の出現確率推定の精度を向上させる。以下、各ステップの詳細を述べる。

2.1 関数トレースによるシステム実行状態の把握

関数呼び出しはシステムの挙動把握とバグ検出に有効であることが報告されている [8]。また関数トレース取得には既存ツールを用いることで対象プログラムの変更を必要とせずに容易に取得可能な場合が多い。例えば、C 言語で記述されたプログラムの場合、gcc コンパイラを用いることでコンパイル時に関数呼び出しに指定した関数をフックとして呼び出すように変更可能である。また Java や Python などのように VM ベースの言語ではランタイムが標準で関数トレースに必要な機能を提供する場合が多い。これらのツールサポートにより本手法を適用する手間が削減される。本手法の関数トレースでは、各関数の呼び出しと終了時に、タイムスタンプ、呼び出し元アドレス、呼び出し先アドレスを記録する。また、Socket API と `fork`, `exec` などの一部の libc 関数についはその引数と返り値も記録する。

2.2 関数トレースの関連した呼び出しへの分割

分散システムの各プロセスについて取得した関数トレースを、それぞれメッセージを受信しそれに応じた処理をする部分を一単位として分割する。また、プロセス初期化部や終了処理部等も別個の一単位とする。同単位を実行ユニットと呼ぶ。関数トレースはプロセス起動時からのすべての関数呼び出しを含むため、意味的に関連の少ない呼び出し同士も同じトレース中に含まれる。実行ユニットに分割することで、関連した関数のみを含んだ部分トレースを構成する。

実行ユニットへの分割は、プログラムの静的解析と関数トレースより自動的に行う。図 1 に例を示す。以下は、対象プログラムが Socket API の `select` を用いて処理が多重化されたループ（メインループ）から構成されている場合のアルゴリズムである。他の API を用いた場合においても同様に適用可能である。

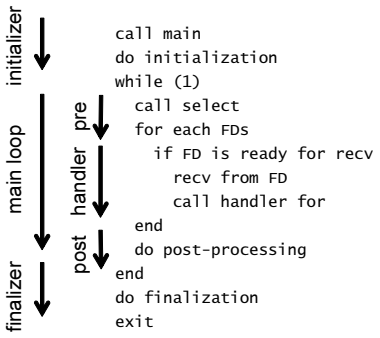


図 1: トレース分割の例

関数トレース中のループの特定 我々は対象プログラムのソースコードやバイナリコードについて静的解析によりループを特定する。ループの特定はバックエッジの検出などによって行う [3]。

メインループの特定 トレース中のループの中から以下の通りにメインループを特定する。メインループのボディは `select` システムコールの呼び出しと、受信可能メッセージを持つ通信路に対する `recv` 処理から構成される。従って、ループ中に `select` と、`recv` の呼び出しがあるループをメインループとして特定する。もしあるひとつの `select` 呼び出しを複数のループが含む場合は、それらの内、インナーループをメインループとして選択する。

ハンドラ実行ユニットの特定 メインループのボディは受信可能な通信に対してメッセージの受信とそれに応じた処理として構成されるのが一般的である。同処理はすべての受信可能通信路について行われるので、ループとして関数トレース中に確認できる。我々はメインループのボディ中に、`recv` システムコールの呼び出しを含むインナーループを特定し、同インナーループのボディをハンドラ実行ユニットとする。

その他の実行ユニットの特定 上記ステップで特定するメインループに到達するまでの部分トレースを初期化実行ユニットとする。また、メインループ以降に呼び出された部分トレースを終了化実行ユニットとする。さらに、メインループのボディにおいてハンドラユニット開始までのトレースを前処理実行

ユニット、ハンドラユニット後の処理を後処理実行ユニットとする。

2.3 実行ユニットのグルーピング

各プロセスにおいて前処理、後処理、ハンドラの各実行ユニッハはメインループの実行回数分存在する。このうち、前処理と後処理は毎回同じ動作をすることが期待できるが、ハンドラは到着するメッセージのタイプとその通信路毎に異なる。従ってすべてのハンドラ実行ユニットにつき単一の実行モデルを構築した場合、意味的に関連の薄いものが含まれ、モデルの精度が落ちる。我々はハンドラ実行ユニットをその通信路毎にグルーピングし、各グループにつき単一のモデルを構築する。各通信路はそれを確立した関数呼び出しパスが等しい場合に同じ通信路とし、接続相手先 IP アドレスやポート等は考慮しない。接続先のアドレスを考慮しない理由は、システムを動作させるノードが異なる場合にもモデルが適用可能とするためである。ポート番号はシステムの設定ファイル等で利用者に応じて変化する場合があるため、これも考慮しない。また、受信したメッセージのペイロード毎にグルーピングすることも考えられるが、そのためにはペイロードのフォーマットやその意味を前もって知る必要がある。我々は対象システムに対する必要予備知識を最低限に抑えるために、メッセージのペイロードのタイプに基づくグルーピングは行わない。

2.4 プロセスのグルーピング

システム全体のモデルを構築するために、各プロセスのトレースをシステム全体について集計する。その際、プロセスのシステム内の役割に応じてプロセスグループを構成し、グループ毎に単一のモデルを構築する。プロセスグループを構成する理由は、クラスタなどの分散環境上のシステムでは一般的にヘッドノードと呼ばれる代表的な役割をもつノードが存在し、その他のノードはワーカーとして動作する場合が多く、それぞれで期待される振る舞いが異なるためである。また、クラスタの最後尾にあたるノードや、ノード間でツリーネットワークを構成した場合などもそれぞれのノード毎に振る舞いが異なる。

プロセスグループへの分割は、各プロセスの初期化実行ユニットに応じて行う。初期化実行ユニットではメインループのセットアップとして、他のプロセスとの通信路やリスニングポートを確立することが期待できる。我々は各プロセスについて、その初期化実行ユニットにおいてセットアップした通信路を求め、その通信路が等しいプロセス同士を同一のグループに割り当てる。これは、Socket API では `connect`, `bind`, `listen`, `accept` へのコールパスに相当する。

2.5 モデル構築

プロセスグループ毎に実行ユニット中の関数 f について、その呼出し元関数 p が呼ばれた場合に f が呼ばれる条件付き出現確率を計算し、モデルとする。出現確率は、その関数 f の出現回数 c_f と呼び出し元関数 p の出現回数 c_p から、c_s/c_p と推定する。この際コールツリーを構成し、コンテキストセンシティブな解析を行う。すなわち、モデル中の各関数について、その関数へのコールスタックの最下位関数からのすべての呼び出しを含むコールパス毎に頻度を計算する。コンテキストセンシティブにすることで、モデルの精緻化が期待できる。

3 モデルを用いた故障解析

故障の原因解析は以下の手順で行う。

1. 正例実行トレースを取得し正例モデルを 2 節の方式で構築する。
2. 故障発生時のトレースを取得する。正例モデル構築時と故障発生時の使用するノード構成が異なっても良い。
3. 各故障トレースをモデル構築の際と同様に実行ユニットへ分割する（被検証実行ユニットと呼ぶ）。
4. 各被検証実行ユニットに対応する正例モデルを特定する。これは、まず被検証トレースのプロセスグループを特定する。等しいプロセスグループに分類されるモデルを正例モデル中に特定し、その正例実行ユニットが被検証実行ユニットに対応するモデルである。ハンドラ実行ユニットの場合は、2 節の場合と同様に、等しい通信路によるハンドラグループのモデルを特定する。
5. 被検証実行ユニットを、対応する正例モデルと比較し、モデルとの乖離を表す異常度を計算する。
6. 各被検証実行ユニットについてその異常度を高い順に故障解析者へ提示する。

以降、異常度の計算について詳細を述べる。

3.1 異常度の計算

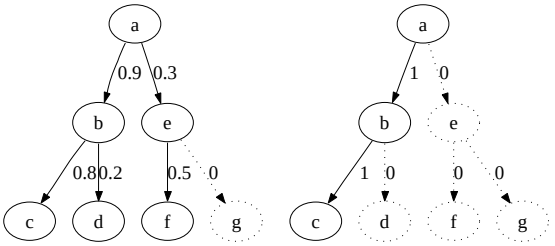
異常度は被検証実行ユニットの正例モデルとの乖離を表す指標である。これは以下の条件に該当する場合に高い異常度を持つように定義する。

- 正例モデル中に対応モデルが存在しない
- 正例モデル中に頻繁に出現かつ被検証実行ユニットに非出現
- 正例モデル中に頻繁には出現せずかつ被検証実行ユニットに出現

我々は上記制約を満たすよう各被検証実行ユニットの異常度を 0 以上 1 以下の範囲で以下の通り計算する。まず対応モデルが存在しない場合は最大値である 1 を割り当てる。存在する場合は以下のステップにより計算する。

1. 被検証実行ユニットについてそのトレースからコールツリーを構築する。ツリーのルートから各ノードへのパスはコールパスに相当する。各エッジに整数値 1 を割り当てる。
2. 正例モデルについても同様にコールツリーを構成する。各エッジには起点ノードが出現した際の宛先ノードの条件付き出現確率を割り当てる。
3. 正例モデルのコールツリーと被検証実行ユニットのコールツリーについて各エッジに割り当てた値の差 Δ を求める。両方にエッジが存在する場合は割り当てた値の差を計算する。一方にのみ存在する場合は存在しないツリーのエッジを 0 として $Delta$ 求める。
4. 全エッジについて計算した $Delta$ の平均を求める。ただし、呼び出し元パスにおいて既に該当するノードがもう一方に存在しない場合は平均の計算に含めない。これは、呼び出し元パスが存在しないため、該当パスが存在しないのは当然だからである。呼び出し元パスによる違いは既に呼び出し元パスでの差分に含まれているため、同パスについてさらに違いを加えるのは意味的に冗長だからである。

これにより、各被検証実行ユニットについて 0 以上 1 以下の異常度を計算する。計算された値は、出現確率との差分を用いることで上記制約を満たす値となる。



(a) 正例モデル (b) 被検証実行ユニット

図 2: 正例モデルと被検証実行ユニットとの比較: 点線のエッジとノードは同ノードが出現しなかったことを表す。

図 2 に例を示す。図 2(a) が正例モデルのコールツリーを表し、図 2(b) が被検証実行ユニットのコールツリーを表す。各ノードのラベルは関数を表し、エッジのラベルは正例モデルの場合は条件付き出現確率、被検証実行ユニットの場合は 1 であればそのパスが出現したことを意味する。図中の点線で囲まれたノードとエッジはそれらが出現しなかったことを意味する。この被検証実行ユニットの異常度は以下の通りに計算される。まず、$a \rightarrow b$ は双方に存在するため $\Delta_{a \rightarrow b} = 0.1$ である。同様に、$\Delta_{b \rightarrow c}$ は 0.2 である。$b \rightarrow d$ は正例モデルには存在するが被検証実行ユニットには存在しない。従って $\Delta_{b \rightarrow d} = 0.2$ である。また、同様に $a \rightarrow e$ は正例モデルにのみ存在し、$Delta_{a \rightarrow e} = 0.3$ である。被検証ユニットでは関数 e が呼ばれなかったため、f, g も呼ばれていない。従ってこれらのノードは差分平均の計算には加えない。結果、同被検証実行ユニットの異常度は 0.2 である。

4 実験

提案手法の故障解析に対する有効性を評価するために MPICH v1.2.5p4 [4] の既知のバグに本手法を適用した。MPICH は MPI の代表的な実装のひとつであり、利用実績、可搬性などの点で優れたライブラリである。しかし、多拠点のクラスタを同時に使用した場合に高い確率でジョブを正常に実行できず、ジョブマネージャがハングアップするバグがあることが報告されている [11]。単一拠点のクラスタのみを利用した場合は発生しない。斉藤らによる人手による解析の結果、同バグの原因は Socket API の `recv` 関数の返り値をチェックせずにメッセージ受信をしたと誤って判断するためであることがわかっている。同バグは既に開発者に報告されバグであると確認され

ている。

InTrigger 上の 3 拠点 78 ノード上で MPICH ジョブマネージャを実行し、故障トレースを取得した。正例トレースは InTrigger 上の単一拠点のクラスタを用いて取得した。取得したトレースより異常度を計算した結果、同バグの発生によって通信路が切断された部分を高い異常度を持つ振る舞いとして特定した。通信路が切断された部分は 78 プロセス中 1 プロセスの実行ユニットにおいて発生した。自動解析の結果を基に人手によって通信路切断の原因を調査した結果、同プロセスと接続された別のプロセスにおいてバグが発生したために切断されたことがわかった。以上の結果より、本提案手法が分散システムにおけるバグ解析に有効であることを確認した。以降、対象システムへの適用手法、結果について詳細を述べる。

4.1 トレース取得

同バグを本手法で解析するために、MPICH のジョブマネージャである MPD に関数トレーサを適用し、正常実行時と故障発生時のトレースを取得する。MPD は Python で記述されているため、我々は Python プログラム用関数トレーサを実装した。同トレーサは以下の 3 つの機能により対象プログラムを変更なしに関数呼び出しをローカルディスクに保存する。

ファイルマップによる固定長トレースバッファ 対象プロセスのメモリ空間に固定長トレースバッファを割り当てるライブラリを用意し、動的リンカにより対象プロセスへロードする。本実験では 32MB のバッファを利用した。バッファがフルになった場合とプロセスの実行終了時にローカルディスクへ中身を書き出す。プロセスが KILL シグナルなどで強制終了された場合、同ライブラリに制御を移すことができず、バッファの中身をローカルディスクへ保存できない。本手法では強制終了時のトレースを失わないために、トレースバッファをローカルディスクのファイルをマップすることで割り当て、強制終了された場合は同ファイルからトレースを復帰させる。

標準 API を用いた Python 関数呼び出しのトレース Python にはデバッガ実装用 API として関数呼び出し、例外発生、等のイベント発生時にフックとして指定された関数を呼び出す機能が標準で提供されている (`sys.settrace`)。我々のトレーサは同 API を利用して関数の呼び出しと終了時に呼び出し先関数アドレス、呼び出し元命令アドレス、タイムスタンプをトレースバッファへ記録する。対象プログラムコードを本トレーサから呼び出すことで対象プログラム自体には変更を加えることなくトレーサを適用可能である。

動的リンカを利用した libc 関数呼び出しのトレース Python インタプリタからの socket API などの呼び出しをトレースするために、同 API をラップするライブラリを動的リンカの機能を用いて対象プロセスにロードする。Socket API について呼び出し元、呼び出し先アドレス、タイムスタンプの他に、接続先アドレス、送受信サイズ等の関数ごとの引数、返り値も記録する。

表 1: InTrigger ノードの構成

CPU	Intel Core2Duo 2.33GHz
RAM	4GB
Interconnect	GigE
Kernel	Linux kernel v2.6.18 x86_64
OS	Debian v3.1

本トレーサを適用した MPD を InTrigger 上の各ノードに配備し、MPI プログラムを投入した。InTrigger のハードウェア、ソフトウェア構成は表 1 の通りである。[11] で報告されている場合と同様に InTrigger の本郷、千葉、大久保の各拠点に配備されているクラスタ上の計 78 ノードで実行した場合、MPI プログラムが開始されず各ノード上で MPD プロセスがハングアップした状態になった。MPI 付属にする異常時用 MPD 強制終了コマンドを用いてシステム終了させ、全ノードのトレースを回収した。正常実行時のトレースは千葉のクラスタのみ計 58 ノードを実行させた場合より取得した。トレースのサイズは平均 1.4MB、最大 1.8MB であった。

4.2 トレース解析結果

収集した正例トレースよりグローバルモデルを構築した。構築されたモデルのデータサイズは 170KB 程度であった。同モデルと各故障トレースの実行ユニットとを比較し、異常度を計算した。図 3 に故障時のトレースについて各実行ユニットの異常度を計算した結果を示す。X 軸はジョブ投入時からの経過時間（秒）を表し、Y 軸は各ユニットの異常度を表す。

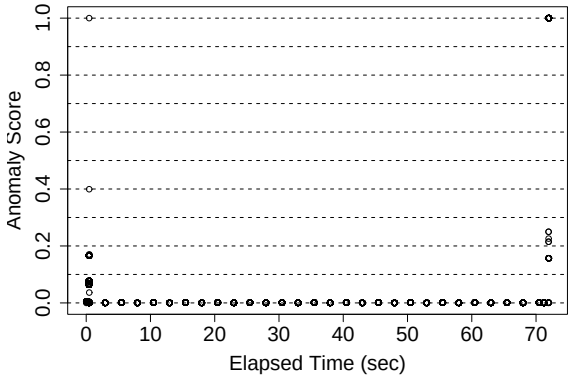


図 3: 故障発生時トレースの異常度: X 軸はジョブ投入時からの経過時間を表し、Y 軸は各実行ユニットの異常度を表す

同グラフより、実行開始直後と終了間際のみ異常度の高い状態が確認でき、かつその間はモデルとの乖離が少ないことがわかる。開始直後の異常状態のうち、もっとも高い異常度 1.0 を持ったユニットが経過時間 0.5 秒の時点に 1 つ存在し、ノード chiba121 のユニットであった。同ユニットではノード hongo102 と接続された通信路について `recv` システムコールが失敗し、通信路を切断した。同通信路についてのメッセージ受信は正例モデル中には発生しなかったため、高い異常度を持つと判断された。終了間際の異常度高い状態は、強制終了によるものであった。正例モデルの構築では MPD を正常に終了させたが、故障発生時は異常終了させたため、終了時のルーチンが異なり、モデルと呼び出される関数が異なったためであった。以上より、chiba121 において hongo102 との接続が切断された部分が原因であると推測できる。

同接続が切断された原因を調べるために、hongo102 において chiba121 との接続を切断した実行ユニットについて人手で以下の検証を行った。その結果、[11] によって報告されているバグが原因であることがわかった。同接続を閉じた関数は `MPDMan` クラスの `handle_lhs_input` であり、`MPDSock` クラスの `recv_dict_msg` 関数により chiba121 よりメッセージ受信を試みたが失敗したため、接続を閉じた。メッセージ受信に失敗した理由は [11] で報告されているように 8 バイト受信する必要があるところで、3 バイトのみ受信したためであった。

同メッセージ受信に該当する `recv` システムコール呼び出しのトレースについてその返り値を調べたところ、3 バイトのメッセージを受信したことがわかった。[11] で報告されているように、`recv_dict_msg` では 8 バイトのメッセージを受信し、同メッセージを後続のメッセージのサイズを表す整数と解釈する。しかし、故障発生時には先頭 3 バイトのみの受信であったため、無効なメッセージサイズとなり、後続メッセージの受信に失敗した。

5 関連研究

Kiciman らによる PinPoint では、コンポーネントフレームワークで構成されたアプリケーションサーバにおいて、故障発生原因であるコンポーネントを自動的に特定する [6]。これは、アプリケーションサーバへのユーザリクエストについて、リクエスト処理の成功と失敗についてサーバ側のコンポーネント間の処理の流れを学習することで、失敗への寄与率が最も高いコンポーネントをリクエスト処理失敗の原因として検出する。例としては、ある e コマースサイトでユーザが商品を購入する場合、ユーザから HTTP でサイトのフロントエンドウェブサーバへ購入リクエストが届く。フロントエンドは典型的にはロジックを担当するバックエンドへ処理を移譲し、バックエンドが購入リクエストを処理する。この処理には、顧客の情報を持ったデータベースへのアクセスなども含まれる。PinPoint では、このような一つのユーザリクエストを処理する一連のコンポーネント間の制御パスを解析の単位とする。あるリクエストが成功した場合、そのパスを正例とし、失敗した場合負例として学習する。学習には確立文脈自由文法や、決定木を用い、リクエスト処理失敗に最も寄与するコンポーネントを特定する。

PinPoint では、対象システムが特定のアプリケーションフレームワークに従い、コンポーネント指向で構成されているものに制約される。しかし、我々の手法では、繰り返し処理を自動的に検知することで、そのような仮定を置く必要がなくより一般性がある。また、PinPoint ではコンポーネント間の相互作用を解析するだけであり、コンポーネント内部が故障時にどのように動作していたのかは知り得ない。我々は関数トレースを記録することで故障発生時の細粒度な制御フローを得ることができ、より有用性が高い。

Bodik らは、ウェブサイト内の各ページ（HTML、JSP、CGI、など）へのアクセスログを実行時に解析

し、サイト内の異常ページを検知する [2]。各ページについてその単位時間あたりのアクセス数を動的に学習し、その変化を基に異常度を計算する。アクセス頻度が高かったページが突然低くなるなど、突発的な変化が発生した場合、そのページに異常が起きたと判定する。例えば、ある JSP ページをバグを含んだ新しいバージョンに変更した場合、処理部のバグのためユーザからのアクセスがタイムアウトを起こし、アクセス頻度が下がることが予測できる。実行時の学習には χ^2 検定とナイーブベイズ分類器を用い、我々と同様に、過去に同じ振る舞い（アクセス頻度）が観測された場合正常と見なし、そうでない場合に異常と見なす。しかし、彼らの手法ではあるウェブページに異常が起きたことは判定できても、その原因まではわからない。また、アクセス頻度は外的要因（販促キャンペーンなど）にも影響されるため、頻度が突発的に変わったとしても異常とは限らない。

Mirgorodskiy らは、クラスタ上のミドルウェアについて関数トレースから故障発生時の解析を行う手法を提案した [8]。我々と同様に、実行中のすべての関数呼び出しを記録し、関数トレースについて異常検知技術を応用したデータ解析を行い、故障の原因を自動的に関数単位で特定する。我々とは主に以下の点で異なる。解析の単位が分散システムを構成する全プロセスであり、プロセス毎の振る舞いの違いを元に故障発生原因を解析する。ここで、分散システム中の構成プロセスはすべて同じ振る舞いを持つものと仮定される。各プロセスについて、固定長リングバッファに保存されたトレース間の非類似度を元に異常度を計算する。しかし、異なるプロセスの固定長バッファ間の違いが有意であるとは限らない。例えば、ある二つのプロセスが、処理部 h_1, h_2, h_3 と順に実行していたとする。しかし、実行タイミングのずれのため、故障発生時のトレースバッファには、一方では h_1 と h_2 のみ、もう一方では h_2 と h_3 のみが残されていたとする。この場合両プロセスとも同一の 3 つ処理部を実行中であり、非類似度は低いと判定されるべきだが、同一部分は h_2 部分のみであり、非類似度が比較的高く判定されてしまう。我々は、解析の単位をメッセージ処理部という細粒度にすることでこの問題を回避する。

6 おわりに

分散環境において故障解析を支援する手法を提案した。同手法は、正しく実行された場合の対象システムの関数呼び出しトレースを取得し、分散システム全体を表す正例実行モデルを構築する。故障発生時にはその関数トレースを正例実行モデルと比較し、異常度を計算する。これにより故障解析者はより故障原因の可能性が高い箇所を優先的に検証できるようになる。提案手法を InTrigger 上で発生した MPICH のバグ解析に適用した結果、同バグによって引き起こされた通信路切断異常を特定でき、バグ解析に有効であることを確認した。

今後の方向性としては、より多くの分散システムへの適用実験があげられる。本論分では MPICH の 1 つのバグについてのみの実験であったが、実際の InTrigger のような分散環境では様々な分散システムが用いられ、必ずしも安定して動作するわけではない。我々はそれらに対しても本手法を適用し故障の診断に有効かどうか検証する予定である。また、本提案手法では関数トレースを用いるが、トレースによる時間的、またデータサイズ的オーバーヘッドが問題になる可能性がある。特に我々が対象とするシステムは主にデーモンプロセスとして常時稼動させるタイプであるため、トレースデータサイズの爆発的な増加が問題になりうる。これに対する解決策としては、実行時に常にトレースを処理し、異常度の高い実行ユニット部のみのトレースを残す方法が考えられる。我々は実行時にトレースを処理し、故障の検知から解析までを統一的に実現することを考えている。

謝辞

故障の再現に協力して頂いた斎藤秀雄氏に感謝する。また実験プラットフォームとして用いた InTrigger 環境を構成、管理されている東京大学近山・田浦研究室に感謝する。本研究の一部は科学研究費補助金特定領域研究 (18049028) による。

参考文献

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89, New York, NY, USA, 2003. ACM Press.

[2] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization. In *2nd IEEE International Conference on Autonomic Computing (ICAC 2005)*, June 2005.

[3] Eli D. Collins and Barton P. Miller. A loop-aware search strategy for automated performance analysis. In *High Performance Computing and Communications (HPCC-05)*, Sorrento, Italy, September 2005.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message pass ing interface standard. *Parallel Computing*, 22(6):789–828, Sep 1996.

[5] Guofei Jiang, Haifeng Chen, C. Ungureanu, and K. Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Second International Conference on Autonomic Computing (ICAC 2005)*, pages 111–122, June 2005.

[6] E. Kiciman and A. Fox. Detecting application-level failures in component-based Internet services. *Neural Networks, IEEE Transactions on*, 16:1027–1041, Sept 2005.

[7] Raissa Medeiros, Walfredo Cirne, Francisco Brasileiro, and Jacques Sauve. Faults in Grids: Why are they so bad and What can be done about it? In *Fourth International Workshop on Grid Computing*, page 18, 2003.

[8] Alexandar V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, Florida, November 2006.

[9] Manos Renieris and Steven P. Reiss. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 30–39, Oct 2003.

[10] M. Steinder and A.S. Sethi. A Survey of Fault Localization Techniques in Computer Networks. *Science of Computer Programming*, 53(2):165–194, November 2004.

[11] 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中健, 高橋慧, 関谷岳史, 頓楠, 柴田剛志, 横山大作, and 田浦健次朗. Intrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境. In **情報処理学会研究報告** *HPC-111 (SWoPP 2007)*, pages 237–242, 旭川, 8 月 2007.