Transtracer: 分散システムにおけるTCP/UDP通信の終端点の監視によるプロセス間依存関係の自動追跡

坪内 佑樹[†1,a)]　古川 雅大[†2]　松本 亮介[†1]

概要：Web サービスの利用者による多様な要求に応えるために，Web サービスを構成する分散システムが複雑化している．その結果，システム管理者が分散システム内のプロセス間の依存関係を把握することが難しくなる．そのような状況では，システムを変更するときに，変更の影響範囲を特定できず，想定よりも大きな障害につながることがある．そこで，システム管理者にとって未知のプロセス間の依存関係を自動で追跡することが重要となる．先行手法は，ネットワーク接続を終端するホスト上で Linux のパケットフィルタを利用してトランスポート接続を検知することにより依存関係を発見する．しかし，Linux カーネル内のパケット処理に追加の処理を加えることになるため，アプリケーションの通信に追加の遅延を与えることになる．そこで，本論文では，サーバ用途で広く利用されている Linux を前提に，TCP/UDP 接続の終端点であるネットワークソケットに含まれる接続情報を監視することにより，未知のプロセス間の依存関係を網羅的に追跡可能なアーキテクチャを提案する．このアーキテクチャにより，プロセスが Linux カーネルの TCP/UDP 通信機構を利用する限り，未知のプロセスの依存を見逃さずに追跡できる．また，接続情報の監視処理は，ソケットがすでに保持する接続情報を読み取るだけとなり，アプリケーションの通信処理とは独立するため，アプリケーションの通信遅延に影響を与えない．最後に，先行手法との比較実験を行い，応答遅延オーバーヘッドとリソース負荷を評価した結果，応答遅延オーバーヘッドを 13–20%，リソース負荷を 43.5%低減させていることを確認した．

Transtracer: Automatically Tracing for Processes Dependencies in Distributed Systems by Monitoring Endpoints of TCP/UDP

Yuuki Tsubouchi[†1,a)]　Masahiro Furukawa[†2]　Ryosuke Matsumoto[†1]

Abstract: Distributed systems in Web services are becoming more complex to respond to various demands by the Web services users. As a result, it becomes difficult for system administrators to understand the processes dependencies in a distributed system. In such a situation, when system administrators add changes to parts of the system, the area of influence by the change cannot be identified, which may lead to a more significant outage than expected. Therefore, system administrators need to trace dependencies between unknown processes automatically. The previous method discovers the dependency by detecting the transport connection using the Linux packet filter on the hosts at ends of the network connection. However, since the method adds the additional processing to the packet processing in the Linux kernel, it gives extra latency to the application traffic. In this paper, we propose an architecture that traces the dependency between unknown processes by monitoring network sockets, which are endpoints of TCP/UDP connection. Our proposed architecture enables tracing without missing unknown processes as long as they use the TCP/UDP protocol stack in the Linux kernel. The architecture makes the monitoring processing only reading the connection information from sockets. Since the processing is independent of the application communication, it does not affect the network latency of the application. Finally, as a result of the evaluation of latency overhead and additional resource load compared with the previous method, we confirmed that our architecture reduces the latency overhead by 13–20 % and reduces the resource load by 43.5 %.

1. はじめに

Web サービスが世の中に普及したことにより，利用者からのアクセス数が増大している．Web サービスが人々の生活に欠かせないものとなったために，サービス提供者が 10 年以上の長期間に渡り，サービスを提供し続けてきている．また，Web サービスの利用者による多様な要求に応えるために，単一のサービス提供者が，ソーシャルネットワーク，電子商取引および音声・動画配信など，多様なサービスを提供するようになってきている．さらに，ある Web サービスがもつ一部機能を他の Web サービスから利用する場合，互いにネットワーク接続することにより，機能を共通して利用する事例もある．このような長期間にわたる Web サービスの機能追加，利用者からのアクセス増加および複数のサービスとの接続などの要因により，Web サービスを構成する分散システムが複雑化している．

Web サービスにおける分散システムは，通常，OS 上のプロセス同士がネットワーク通信することにより構成される．Web サービスにおいて分散システムが複雑化しているため，分散システム内のプロセス間のネットワーク依存関係も複雑化し，システム管理者は自身が管理する分散システム内のプロセス間の依存関係を把握することが難しくなる．特定のプロセスに故障や性能劣化などの異常が発生すると，ネットワーク通信先の他のプロセスに影響することがあり，システム管理者がシステムを変更するときに，その変更に問題があった場合に変更箇所に依存する全てのプロセスへ，ネットワークを通じて影響が伝搬する可能性がある．そこで，システム管理者がプロセス間のネットワーク依存関係を知らずに変更すると，システム管理者の予想よりも大きな範囲の障害が発生することがある．したがって，システム管理者はプロセス間の依存関係を把握することにより，変更が影響する範囲を特定する必要がある．そのような調査のための手間を削減するために，システム管理者の手によらず，システム管理者にとって未知のプロセスの依存関係を自動で発見する必要がある．

これまでに，先行手法として，Linux のパケットフィルタを利用してトランスポート接続の状態を取得することにより依存関係を追跡するコネクションベースアプローチ [1] がある．このアプローチでは，実際に発生した接続を元に依存があると判定するため，偽陽性がなく，Linux カーネルがサポートするトランスポート接続をアプリケーションが利用する限りは依存を検出できる．さらに，偽陽性がないため正確に障害の影響範囲を見積もることができ，Linux の TCP/UDP 接続を利用しているプロセスであればアプリケーションを変更せずに網羅的に追跡可能であるため，未知のプロセスを追跡することに適している．しかし，パケットフィルタを利用すると，Linux カーネル内のパケット処理に介入することになるため，アプリケーション処理に余分な遅延が増える．また，トランスポート接続の開始のみを検出するため，プロセスが一度確立した接続を再利用する場合，接続確立後にはプロセスが再接続するまで当該接続を検出できない．

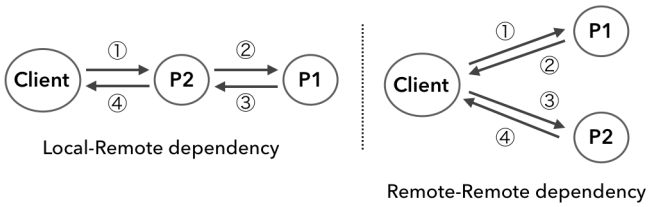


図 1: local-remote 依存と remote-remote 依存

そこで，本論文では，サーバー用途で広く利用されている OS である Linux を対象に，TCP/UDP 接続の終端点であるネットワークソケットに含まれる接続情報を監視することにより，Web サービスの分散システムにおける未知のプロセス間の依存関係を網羅的に追跡可能なアーキテクチャを提案する．このアーキテクチャにより，プロセスが Linux カーネルの TCP/UDP 通信機構を利用している限り，未知のプロセスの依存を見逃さずに追跡できる．先行研究の課題である遅延オーバーヘッドについて，接続情報の監視処理は，ソケットがすでに保持する接続情報を読み取るだけとなり，アプリケーションの通信経路とは独立するため，アプリケーションの通信遅延に影響を与えない．さらに，接続が再利用されていても，接続の終端点であるソケットから接続情報を取得できる．ただし，TCP/UDP 接続情報と OS プロセス情報しか取得しないため，OSI 参照モデルのうちトランスポート層以外のプロトコルがもつ通信単位での依存関係を追跡できないという制約がある．

本論文を次のように構成する．2 章では，分散システムの依存関係の追跡技術について述べる．3 章では，提案する依存関係追跡のためのアーキテクチャについて説明する．4 章では，提案手法の有効性を評価する．5 章では，本論文をまとめ，今後の展望を述べる．

2. 分散システム内の依存関係の追跡技術

Web サービスにおける分散システムが複雑化している状況において，システム管理者がシステム内の依存関係を把握することが難しくなっている．そのような状況では，システムの変更時の影響範囲を特定できず，変更に問題があった場合に，予想を超える規模の障害に発展することがある．そこで，分散システムの依存関係を自動で追跡する手法がこれまでに提案されている．本章では，先行研究のそれぞれの特徴を整理する．

†1 さくらインターネット株式会社 さくらインターネット研究所 SAKURA internet Research Center, SAKURA internet Inc., Ofukatyo, Kitaku, Osaka 530-0011 Japan
†2 株式会社はてな Hatena Co., Ltd.
a) y-tsubouchi@sakura.ad.jp

2.1 プロセス間の依存関係の定義

本論文では，プロセス間の依存関係を，Zand らの研究 [2] と同様に，次のように定義する．プロセス P_1 での遅延，性能低下および故障が，直接または間接的に，プロセス P_2 の遅延，性能低下および故障を引き起こすのであれば，プロセス P_2 はプロセス P_1 に依存する．この定義を元に，Chen らの研究 [3] は，プロセス間の依存関係を local-remote 依存と remote-remote 依存の 2 つに分類している．図 1 にそれぞれの依存を示す．図 1 の番号はリクエストとレスポンスの順番を示しており，プロセスへのリクエストを送る前にネットワーク接続するものとする．まず，P_2 がクライアントからのリクエスト処理の中で P_1 に接続する必要がある場合，P_2 は P_1 に local-remote 依存している．次に，P_2 に接続するために，遠隔にあるクライアントが P_1 に最初に接続する必要があるのであれば，P_2 は P_1 に remote-remote 依存している．

例えば，データベースサーバを P_1，アプリケーションサーバを P_2 とすると，クライアントからリクエストを受けたアプリケーションサーバは，データベースサーバに問い合わせた後に，Web サーバへレスポンスを返却するため，アプリケーションサーバはデータベースサーバに対して local-remote 依存している（図 1 左）．一方で，DNS サーバを P_1，Web サーバを P_2 とすると，クライアントはまず DNS サーバに問い合わせ，IP アドレスを取得したのちに，Web サーバに接続するため，Web サーバは DNS サーバに remote-remote 依存している（図 1 右）．

2.2 依存関係の追跡技術

これまでに分散システムにおける依存関係の追跡手法として，パケットベースアプローチ，コネクションベースアプローチおよびリクエストベースアプローチの 3 種類のアプローチが提案されている．

2.2.1 パケットベースアプローチ

パケットベースアプローチは，既存のトラヒックからパケットを収集し，パケットヘッダ上の送信元と送信先のそれぞれのホストとポート，パケットの送受信の時刻などの情報を解析することにより，依存を発見する．いずれのパケットベースアプローチにおいても，各プロセス間でトラヒック流量に時間差があることに着目し，各プロセス間のトラヒック性質に相関するパターンを発見することにより，依存を推定する手法がとられている．

パケットベースアプローチの利点は，既存のアプリケーションの変更が不要であることである．さらに，ネットワークスイッチでパケットを観測すれば，スイッチの変更のみで済むため，ホスト上に追加のソフトウェアを配置する必要がない．しかし，全てのパケットを観測すると，パケット流量が大きい場合に，パケットの解析処理のコストが大きくなるため，通常は一部のパケットのみをサンプリングし，パケットの解析コストを低減させる．

しかし，その結果，トラヒック量の小さい依存を見逃す可能性がある．さらに，相関するパターンを発見するときに，相関する度合いを計算することになるため，依存関係の有無を確率で表現することになる．したがって，複数のプロセス間のトラヒックパターンについて，実際に依存はしていないにも関わらず，偶然似ているときに依存があると誤判定するために，偽陽性がある．また，相関する度合いに対して，予め設定した閾値をもとに依存を判定することから，閾値をシステム管理者がチューニングする必要がある．

Sherlock[4]，Orion[3]，NSDMiner[5] は，既存のトラヒックのパケット情報のみを利用する．Macroscope[6] は，ホストまたはネットワークデバイスで収集するネットワークパケットとホスト上のトランスポート接続情報を結合する．Rippler[2] は，ネットワークトラヒックに意図的に遅延を注入することにより，依存するシステムへ遅延が伝搬する特性を利用し，偽陽性を低減させている．

2.2.2 コネクションベースアプローチ

コネクションベースアプローチは，トランスポート接続の情報を接続を終端するホスト上で取得することにより依存関係を追跡する．パケットベースアプローチのように相関するパターンを識別するわけではなく，実際に発生した接続を元に依存があると判定するため，偽陽性がない．一方で，ホスト上にトランスポート接続の情報を取得するための追加のソフトウェアを配置する必要があり，システム管理者の手間が増加する．ただし，実用上，サーバ構成管理ツール [7] を利用して，ソフトウェアの配置を自動化することにより，負担を低減させられる．

コネクションベースアプローチとして Clawson の研究 [1] がある．Clawson の研究では，Linux のパケットフィルタを利用して，ホスト上で終端しているトランスポート接続を監視することにより依存を検出する．Linux がサポートするトランスポートプロトコルをアプリケーションが利用する限りは，依存を検出できる．パケットフィルタを利用すると，Linux カーネル内のパケット処理に介入することになるため，アプリケーション処理に余分な遅延が増えるというデメリットがある．さらに，トランスポート接続の開始イベントのみを検出するため，接続ごとに発生する処理コストを減らすためにプロセスが一度確立した接続を再利用する場合，プロセスが再接続するまで接続を検出できない．

2.2.3 リクエストベースアプローチ

リクエストベースアプローチは，アプリケーション層のリクエストがシステム内のどの経路をたどるかを追跡する．具体的には，各リクエストに識別子を割り振り，通信内容に埋め込んだ上で，後続のプロセスへ伝搬させる．これにより，識別子を頼りに，リクエストがシステム内のど

のプロセスを経由して処理されたかを追跡できる．

リクエストベースアプローチの利点は，コネクションベースアプローチと同様に，実際の接続をもとに依存を判定するため，偽陽性が低いことである．さらに，アプリケーションの処理内容やアプリケーション層のプロトコルに応じた情報を追跡することが可能である．

一方で，リクエストベースアプローチは，リクエストに識別子を埋め込むため，アプリケーションの応答時間に余分な遅延を与える．ただし，識別子の埋め込みをリクエスト単位でサンプリングすることにより，遅延を低減できるが，その一方で，依存の検出について偽陰性が高くなる．さらに，識別子を埋め込むために，アプリケーションを変更する手間，または通信経路上にソフトウェアを配置する手間が発生する．

Pinpoint[8]，Magpie[9]，X-Trace[10]，Dapper[11] は，アプリケーションの処理を変更し，リクエストの識別子を埋め込む処理を加える．サービスメッシュ [12] は，マイクロサービス [13] やプロセスの単位でプロキシを配置し，プロキシを経由してその他のサービスと通信させることにより，アプリケーションを変更することなく，リクエストに識別子を埋め込める．

2.2.4 各アプローチのまとめ

本論文の目的である Web サービスの分散システムにおける未知のプロセスの依存を追跡する観点で，各アプローチの課題を整理する．

まず，網羅的に依存を追跡できることが重要であるため，パケットのサンプリングによりトラヒック量の小さいプロセスとの依存を見逃すこともあるパケットベースアプローチは本研究の目的には適さない．

次に，コネクションベースアプローチであれば，偽陽性がないため，システムに対する変更の影響範囲を実際よりも大きく見積ることがない．しかし，プロセスの接続の再利用により，未知のプロセスの依存を見逃す可能性がある．Web サービスでは，単一のトランスポート接続上で複数の HTTP リクエストを送信する HTTP/2[14] が普及するなどの要因により，プロセスの接続を再利用するケースが増えている．

さらに，未知のプロセスの依存を検出するためには，網羅的に計測するために，遅延オーバーヘッドが大きくなることが課題となる．リクエストベースアプローチでは，事前に依存を把握した上で処理や通信の経路上に変更を加えるため，新たに未知のプロセスを発見するには適さない．また，コネクションベースアプローチ同様に，未知のプロセスの依存を検出するためには，計測のための遅延オーバーヘッドが課題となる．

3. 提案する依存関係追跡アーキテクチャ

本論文では，未知のプロセスを含むような複雑化した

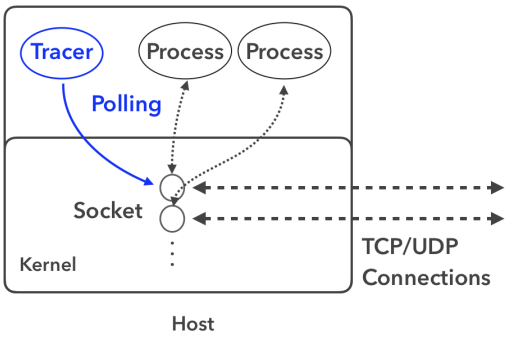


図 2: ソケットの接続情報の取得

Web サービスの分散システムに対して，システム管理者がサーバに対して変更を加えるときに，変更の影響範囲を特定するために依存関係を追跡することを目的とする．サーバで動作するアプリケーションのある特定のリクエスト処理の内容を変更する場合でも，同一プロセス内の処理であればメモリ空間を共有することから，リクエスト処理をするプロセス上の他のリクエスト処理にも変更の影響が伝搬する可能性がある．したがって，変更の影響範囲を特定するためには，ホスト上のプロセス単位での依存関係を取得できれば十分である．

各先行研究の特徴を踏まえると，未知のプロセスの依存を検出するために，偽陰性とアプリケーションに与える遅延オーバーヘッドを低減した上で，システムに対する変更の影響範囲を正確に見積もるために，偽陽性が低いアーキテクチャが必要となる．そこで，本論文では，コネクションベースアプローチを採用することにより偽陽性をなくした上で，Linux カーネル内の各接続の終端点であるソケットに含まれる接続情報を監視することにより，依存関係を追跡するアーキテクチャ Transtracer を提案する．

このアーキテクチャにより，接続情報の監視処理は，ソケットがすでに保持する接続情報を読み取るだけとなり，アプリケーションの通信処理とは独立することになる．したがって，パケットフィルタのようにアプリケーション通信経路中のパケットに対して監視のための処理を追加することがないため，コネクションアプローチの課題である遅延オーバーヘッドを低減できる．また，一度確立した接続を再利用したとしても，接続の終端点であるソケットは存在するため，ソケットから接続情報を取得できる．

3.1 提案するアーキテクチャの詳細

図 2 に Transtracer アーキテクチャにおける TCP/UDP 接続のソケット情報の取得手法を示す．Tracer プロセスをホスト上で動作させておき，Tracer プロセスが Linux カーネルに問い合わせ，各接続に対応するソケットからその瞬間の TCP/UDP の接続状態のスナップショットを取得する．Tracer プロセスは，接続状態と対応するプロセス情報も同時に取得し，接続と接続に対応するプロセスとを紐付ける．新規の接続を検知して取得し続けるために，接続状

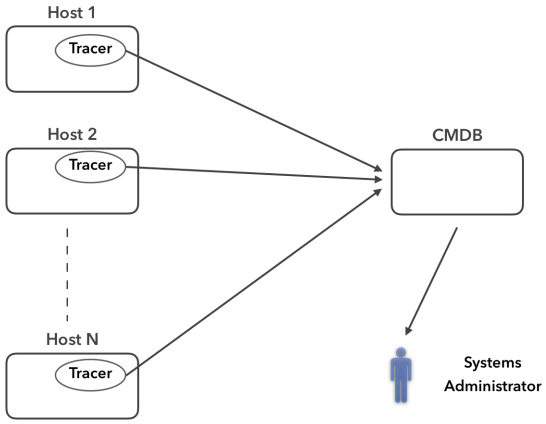


図 3: Transtracer のシステム構成

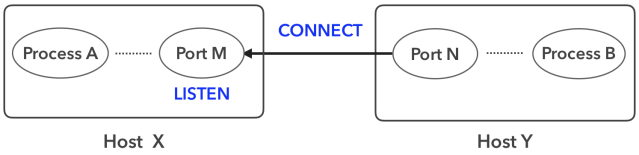


図 4: 依存の方向の識別

態のスナップショットを定期的にポーリング取得する．ただし，ポーリングによる情報取得は，ポーリングの時間間隔分だけ新規の接続検出が遅れるという特徴がある．本論文では，ポーリング間隔を秒単位に設定することを想定しているため，検出が遅れる時間も秒単位となる．システム管理者がシステム変更時に依存関係を確認するという利用ケースであれば，秒単位よりも高速に検出できてもシステム管理者が認識できないため，秒単位で十分である．また，ポーリングによる情報取得はポーリング間隔以内の短命な接続を見逃す可能性がある．ポーリング間隔を小さくすることで，単位時間あたりのハードウェアリソース消費量は増加するが，より短命な接続を検知できる．ポーリング間隔の想定が秒単位であるため，検出可能な接続の生存時間も秒単位を想定している．

図 3 に，複数のホスト上の接続情報を突き合わせ，依存関係のグラフを作成するためのシステム構成を示す．各ホスト上で動作する Tracer は，取得した接続情報を中央の CMDB（Connection Management DataBase，接続管理データベース）へ送信する．システム管理者または各ホストは，対象のホストやプロセスを一意に識別する情報と時間範囲をパラメータとして，CMDB に問い合わせ，対象に紐づく依存関係を取得する．依存関係は，ネットワーク接続された各プロセスについて，プロセスをノード，ネットワーク接続をエッジとみなしたグラフ構造として表現できる．関係データベースを用いたデータ構造の詳細については，3.3 節にて述べる．

図 4 に，依存の方向を識別する手法を示す．プロセス間の依存の方向を識別するために，プロセス間の TCP/UDP 接続について，次のように，接続を要求する側と接続を待ち受ける側に分けた上で，依存の方向を決定する．ホスト X 上のプロセス A がポート M で待ち受けているとして，ホスト Y 上のプロセス B がホスト X 上のポート M に対して接続するとすると，プロセス A がなければプロセス B は接続処理に失敗するため，プロセス B はプロセス A に依存すると言える．

ホスト上で取得可能な全ての TCP/UDP 接続イベントを収集すると，接続情報の個数が大きくなり，CMDB 上のレコード数が大きくなるという課題がある．そこで，レコード数を削減するために，プロセス間の依存関係を識別する目的に対して冗長となる接続情報を削減する．プロセスを構成する各プロセスやスレッドが OS から割り当てられるランダムな送信元ポートであるエフェメラルポート（Ephemeral Port，短命ポート）を利用して，別のプロセスへ接続することがある．したがって，特定のポートで待ち受けるプロセスに対して，特定のプロセスのクライアントから複数のランダムな送信元ポートを利用して接続されることがあると言える．しかし，依存関係を識別するのみであれば，接続の度に異なるポート番号が割り当てられるエフェメラルポートを一意な接続情報とみなす必要がないため，Tracer プロセスが特定ホスト上の特定ポートに対するエフェメラルポートによる接続情報を重複するものとみなし，それらを集約した上で，CMDB へ送信する．

3.2 提案するアーキテクチャの制約

Transtracer アーキテクチャは，TCP/UDP 接続情報と OS プロセス情報のみ取得するため，OSI 参照モデルのうちトランスポート層の接続以外のプロトコルがもつ通信単位での依存関係を追跡できないという制約がある．具体的には，HTTP などのアプリケーション層におけるリクエスト単位やネットワーク層のパケット単位での依存関係を追跡できない．その結果，アプリケーション層においては，プロセス間の通信の間に，接続を中継するプロセスとしてプロキシサーバを配置するときに，複数の異なる通信経路を中継する場合，各経路の依存関係を個別に認識できなくなるという制約がある．

ただし，プロキシサーバの中継先のプロセスのうちどれか一つでも不調になれば，プロキシサーバがその影響を受け，中継元の全てのクライアントプロセスにも影響が伝搬する可能性がある．したがって，プロキシサーバを配置しても 2.1 節にて定義した依存関係を満たした上で依存を追跡できている．さらに，ネットワーク層においては，プロセス間の通信の間に，NAT(Network Address Translation) 装置を配置すると，片方のホスト上でみえる相手先の IP アドレスやポート番号が実際の接続先の IP アドレスやポート番号ではなくなるため，CMDB 上で接続の両端のプロセスを紐付けられないことがある．

3.3 提案するアーキテクチャの実装

サーバ用途で広く利用されていることから Linux(Linux

```
$ ttctl --dbhost 10.0.0.20 --ipv4 10.0.0.10
10.0.0.10:80 ('nginx', pgid=4656)
└<-- 10.0.0.11:many ('wrk', pgid=5982)
10.0.0.10:80 ('nginx', pgid=4656)
└--> 10.0.0.12:8080 ('python', pgid=6111)
10.0.0.10:many ('fluentd', pgid=2127)
└--> 10.0.0.13:24224 ('fluentd', pgid=2001)
```

図 5: ttctl コマンドの出力

Kernel 4.15) を前提に Tracer プロセスを実装する．また，データベース管理システムとして広く利用される関係データベースシステムのうち，テーブルのデータ型としてネットワークアドレス型をもつために接続情報の IP アドレスを管理しやすい PostgreSQL(バージョン 11.3)[15] を利用し，CMDB を実装する．

通常，TCP/UDP のソケット情報の取得には，トランスポートプロトコルのソケットを監視する inet_diag モジュールを利用する．inet_diag モジュールにより，ソケットがもつ情報である，宛先 IP アドレス，宛先ポート番号，送信元 IP アドレス，送信元ポート番号，接続の状態，ソケットの inode 番号などを取得可能である．inet_diag モジュールにアクセスするためのインタフェースには，ファイルシステム形式の Process Filesystem(procfs) と，Linux におけるカーネル空間とユーザー空間との間でメッセージ通信するための機構である Netlink[16] がある．本実装では，procfs のようにファイル内容の解析が不要なため，より高速にアクセス可能な Netlink を利用する．

inet_diag モジュールから取得できるソケット情報には，そのソケットを所有しているプロセスの情報は含まれない．そこで，ソケット情報の inode 番号と，プロセスがもつ inode 番号を照合し，一致すれば当該ソケットとプロセスを紐付ける．プロセスがもつ inode 番号は，Process Filesystem(procfs) 上の/proc/[pid]/[fd]/以下のファイルから，取得可能である．

プロセス間の依存関係を PostgreSQL サーバに保存し，CMDB として利用する．プロセス間の依存関係は，プロセスをノードとして，接続を向きをもつエッジとする有向グラフとして表現できるため，ノードを管理するテーブルとエッジを管理するテーブルが必要となる．しかし，接続を集約する都合上，パッシブオープン側のノードはリッスンするポート番号をもつが，アクティブオープン側のノードはエフェメラルポートが集約されており，特定のポート番号をもたない．したがって，アクティブオープン側とパッシブオープン側を同一のスキーマで表現しづらいことから，アクティブオープン側のノードとパッシブオープン側のノードを異なるテーブルとして分離している．

以上のような実装を，筆者らは Transtracer[*1] という名称で開発している．Transtracer では，ホスト上でデーモンとして動作し，ソケットの接続情報を収集しつつ，CMDB に保存する ttracerd と，CMDB に問い合わせて依存関係を取得するための ttctl という 2 つの実行コマンドがある．このように書き込み側と読み取り側のコマンドを分離することにより，利用者が引数オプションを混同せずに指定しやすくできる．利用者は，CMDB となる PostgreSQL サーバをセットアップしたのちに，ttracerd を依存関係を追跡したい全てのホスト上で起動し，CMDB に依存関係を蓄積した上で，ttctl を利用して CMDB に問い合わせ，依存関係を取得する．ttracerd に渡すパラメータに，ポーリング取得のための間隔を秒単位で指定できる．なお，UDP 接続情報の取得と，1 より大きな深さの依存関係グラフの取得は，現時点では未実装である．

[*1] https://github.com/yuuki/transtracer v0.1.0

表 1: 実験環境

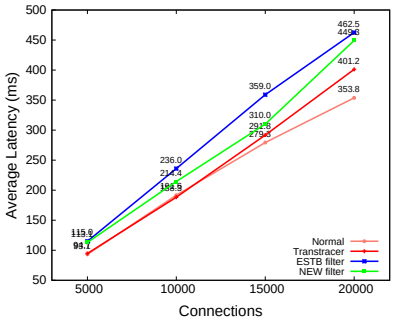
	項目	仕様
Client	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 2core
	Memory	1 GBytes
	Benchmarker	wrk 4.1.0-4
Server	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 4core
	Memory	1 GBytes
	HTTP Server	nginx 1.17.3
CMDB	CPU	Intel Xeon CPU E5-2650 v3 2.30GHz 1core
	Memory	1 GBytes
	Database	PostgreSQL 11.3

利用者が Transtracer をどのように利用するかを示すために，本実装における入出力の例を示す．IP アドレス 10.0.0.10 のホストを起点に深さ 1 の依存関係グラフを取得するには，図 5 のように ttctl の引数に IP アドレスを指定し，CMDB に問い合わせる．このような出力を得た場合，利用者は次のことがわかる．まず，指定した IP アドレス 10.0.0.10 のホスト上では 80 番ポートで待ち受ける nginx[*2] と Fluentd[*3] が動作している．次に，nginx プロセスは，10.0.0.11 のホスト上で動作する HTTP のベンチマークツールである wrk[*4] からの接続を受けつけ，10.0.0.13 のホスト上でポート 8080 番で待ち受ける Python のプロセスに接続している．さらに，10.0.0.10 上の fluentd プロセスは，10.0.0.12 上でポート 24224 番で待ち受ける fluentd プロセスに接続している．また，3.1 節で述べた手法により，アクティブオープン側のプロセスのエフェメラルポートからの接続を同一ポートへの 1 つの接続として集約したものを many と表記している．以上により，Transtracer を利用することにより，対象のホストを変更する前に，当該ホスト上のプロセスを起点とした依存関係を確認できる．

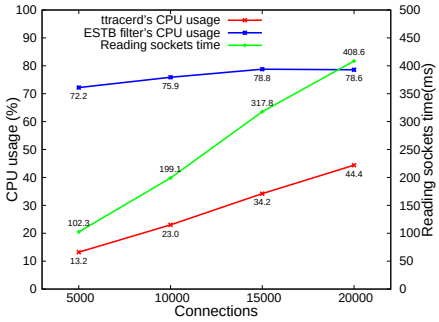
4. 評価

提案する Transtracer アーキテクチャの有効性を確認するために，依存関係の追跡処理がアプリケーションに与える遅延オーバーヘッドとサーバに与えるリソース負荷を 3.3

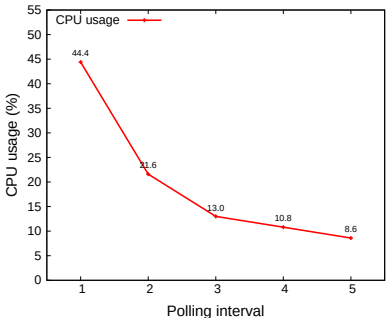
[*2] https://nginx.org/
[*3] https://fluentd.org
[*4] https://github.com/wg/wrk



(a) 応答時間



(b) CPU 利用率と接続情報取得時間



(c) ポーリング間隔に対する CPU 利用率

図 6: 実験結果

章で述べた実装を用いた実験により評価した．表 1 に実験環境を示す．実験環境の各ロールをさくらのクラウド[*5]上の仮想サーバを用いてそれぞれ 1 台ずつ構築した．各ロールの OS は全て Ubuntu 18.04.2 Kernel 4.15.0 であり，各仮想サーバ間の帯域は 1 Gbps である．実験では，Server 上で ttracerd プロセスを動作させ，Client から Server 上の nginx に対して，HTTP のベンチマーカーである wrk によるベンチマークを行った．

コネクションベースアプローチの先行手法である Clawson の研究 [1] との比較のために，Server 上で Linux のパケットフィルタである iptables を利用して検知した TCP/UDP 接続のログを Ubuntu 18.04 のデフォルトで採用されているログ管理プロセスの systemd-journald に流すようにした上で同様のベンチマークを行った．接続検知のためパケットフィルタ方式として，1 つ目に，iptables により Client からの新規に接続されたときのみログを残すようにする NEW フィルタ方式があり，NEW フィルタ方式はログの量が少ない代わりに，再利用された接続をあとから検知できない．2 つ目に，Client からのすでに確立済みの接続を流れるパケット全てのログを残すようにする ESTB フィルタ方式があり，ESTB フィルタ方式は NEW フィルタ方式と比べてログの量が多い代わりに検知漏れがない．Clawson の研究における実験では，ESTB（ESTABLISHED）フィルタ方式を採用しているが，1 分あたり 20 パケットのみのサンプリングとなっており，パケット流量の少ない接続を検知できない可能性があるため，本論文の実験ではサンプリングを無効とするように設定にした．

以降の各実験に共通する設定を整理する．nginx は静的に配置した 612 バイトの HTML ファイルを返却するのみである．nginx のみで CPU コアを使い切らないように，nginx のワーカープロセス数は，Server の 4 コアのうち 2 コアになるように設定した．また，wrk のスレッド数を 1 に設定し，60 秒のベンチマークを行った．wrk は HTTP Keep-Alive により，一旦確立した接続を再利用するようになっているため，nginx 側でも HTTP Keep-Alive を有効に設定し，接続のタイムアウト時間をベンチマーク時間よりも大きな値とすることにより，ベンチマーク中の接続数が一定の値をとるようにした．プロセス単位の CPU 利用率を計測するために，Linux の pidstat コマンドを利用し，pidstat コマンドに指定する計測間隔をポーリング間隔と同一の値に設定した．以降で特に指定がない場合，ttracerd プロセスがソケットから接続情報を取得するときのポーリング間隔は 1 秒とした．なお，以降で計測したリソース消費量と実行時間の値は 1 回のベンチマーク中に連続で 5 回計測したときの平均値である．

まず，Transtracer が追跡対象のサーバ上のアプリケーションに与える遅延オーバーヘッドを評価した．具体的には，wrk の同時接続数パラメータを 5,000 から 20,000 まで増やしたときに，依存関係を追跡しない通常状態，ttracerd プロセスを起動した状態，ESTB フィルタ方式，NEW フィルタ方式のそれぞれの平均応答時間をグラフ化した．この実験の結果を図 6(a) に示す．図 6(a) より，Transtracer は，いずれの接続数においても ESTB フィルタ方式の値の 13–20%小さい値，NEW フィルタ方式の値の 5.8–16.2%小さい値をとった．また，通常状態と比較し，Transtracer の応答時間は 1.7–13.4%大きい値となった．以上により，Transtracer により先行手法と比較して，応答時間の遅延オーバーヘッドを低減できていることがわかる．

次に，Transtracer の導入により，依存関係を追跡可能となる代わりに，追跡対象のサーバにどの程度のリソース負荷のオーバヘッドを与えるかを評価した．具体的には，wrk の同時接続数パラメータを 5,000 から 20,000 まで増やしたときの ttracer プロセスと ESTB 方式の CPU 利用率の変化と接続情報の取得時間の変化をグラフ化した．この実験の結果を図 6(b) に示す．図 6(b) より，ESTB フィルタ方式と比較すると，ESTB フィルタ方式はいずれの接続数においても CPU 利用率が 70%を超えている一方で，ttracer プロセスの CPU 利用率は 20,000 接続時においても，44.4%となる．したがって，ESTB フィルタ方式に対して，Transtracer は依存関係追跡のための CPU 負荷を 43.5%低減している．

*5 https://cloud.sakura.ad.jp

さらに，図 6(b) において，20,000 接続時の 44.4%の CPU 利用率について，CPU コアが少ない環境であれば，nginx プロセスと ttracerd プロセスで CPU リソースを取り合った結果，nginx プロセスの処理性能が低下する可能性がある．そこで，ポーリング間隔を増加させると，その分短命な接続を検出できない可能性が大きくなるかわりに，CPU 利用率をどの程度低減できるかを評価した．具体的には，wrk の接続数パラメータを 20,000 に固定した上で，ttracerd プロセスのポーリング間隔を 1 秒から 5 秒まで増加させたときの ttracerd プロセスの CPU 利用率の変化をグラフ化した．この実験の結果を図 6(c) に示す．図 6(c) より，ポーリング間隔を増加させると，ポーリング間隔に反比例して CPU 利用率が低下する．ポーリング間隔を 5 秒に設定すると，CPU 利用率は 8.6%まで低下する．

以上により，先行手法と比較し，応答遅延オーバーヘッドとリソース負荷オーバーヘッドを低減させていることから，Transtracer アーキテクチャが有効に機能すると考えている．接続数の大きい高負荷環境であっても，低オーバーヘッドで利用可能であることから，高負荷環境のみ依存関係を追跡しないといった実環境の運用上都合による妥協をせずに，網羅的に依存関係を追跡できる．ただし，ポーリングによる取得であるために，秒単位の短命な接続を見逃す可能性がある．Web サービスにおいては，HTTP/2 の利用などにより，接続ごとに実行される処理コストを削減するために，プロセスが接続を再利用することがあることから，実際の依存関係の検出漏れは少ないと考えている．

5. まとめと今後の展望

本論文では，Web サービスの分散システムを対象に，Linux 上で接続の終端点であるソケットに含まれる TCP/UDP 接続情報を監視することにより，システム管理者にとって未知のプロセス間依存関係を追跡可能なアーキテクチャ Transtracer を提案した．Transtracer アーキテクチャにより，サーバ上のリソース消費の負荷を低減しつつ，アプリケーションの通信遅延に影響を与えずに，プロセス間の依存関係を網羅的に検出できる．実験の結果，先行手法と比較し，応答遅延オーバーヘッドを 13–20%，リソース負荷を 43.5%低減させていることを確認し，Transtracer アーキテクチャが有効に機能することを示した．

今後の展望としては，短命な接続であっても確実に依存関係を検出するために，Linux の eBPF(extended Berkley Packet Filter)[17] を利用し，ソケットから接続情報をポーリング取得するだけでなく，ソケットに対する接続イベントを検知をできる手法と組み合わせることを考えている．さらに，コンテナ型仮想化環境においても依存関係を追跡できるようにしていくつもりである．また，CMDB のセットアップのコストを削減するために，各ホスト上の Tracer プロセスがローカルホスト上に接続情報を分散して保存する手法を検討する．最後に，収集した依存関係情報をもって本来存在するはずの依存を発見できない場合に異常とみなすなどの異常検知への応用を進める．

参考文献

[1] J. K. Clawson, Service Dependency Analysis via TCP/UDP Port Tracing, Master's thesis, Brigham Young University-Provo 2015.

[2] A. Zand, et al., Rippler: Delay Injection for Service Dependency Detection, *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 2157–2165 2014.

[3] X. Chen, et al., Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 117–130 2008.

[4] P. Bahl, et al., Towards Highly Reliable Enterprise Network Services via Inference of Multi-Level Dependencies, *ACM SIGCOMM Computer Communication Review*, Vol. 37, No. 4, pp. 13–24 2007.

[5] A. Natarajan, et al., NSDMiner: Automated Discovery of Network Service Dependencies, *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 2507–2515 2012.

[6] P. Lucian, et al., Macroscope: End-Point Approach to Networked Application Dependency Discovery, *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 229–240 2009.

[7] J. O. Benson, et al., Survey of Automated Software Deployment for Computational and Engineering Research, *Annual IEEE Systems Conference (SysCon)*, pp. 1–6 2016.

[8] M. Y. Chen, et al., Pinpoint: Problem Determination in Large, Dynamic Internet Services, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 595–604 2002.

[9] P. Barham, et al., Magpie: Online Modelling and Performance-aware Systems, *17th Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 85–90 2003.

[10] R. Fonseca, et al., X-Trace: A Pervasive Network Tracing Framework, *USENIX Conference on Networked Systems Design & Implementation (NSDI)*, pp. 20–20 2007.

[11] B. H. Sigelman, et al., Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Technical report, Google 2010.

[12] W. Li, et al., Service Mesh: Challenges, State of the Art, and Future Research Opportunities, *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225 2019.

[13] M. Fowler and J. Lewis, Microservices, http://martinfowler.com/articles/microservices.html.

[14] M. Belshe, et al., Hypertext TransferProtocol Version 2 (HTTP/2), Technical report, RFC 7540 2015.

[15] The PostgreSQL Global Development Group, PostgreSQL, https://www.postgresql.org.

[16] P. Neira-Ayuso, et al., Communicating between the kernel and user-space in Linux using Netlink sockets, *Software: Practice and Experience*, Vol. 40, No. 9, pp. 797–810 2010.

[17] D. Calavera, L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*, O'Reilly Media 2019.