Vol.2022-OS-155 No.1
2022/5/26

# 高精度時刻同期機構のカーネル内汎用実装

小堀 和人[1]　李 忠翰[2]　廣津 登志夫[1]

概要：近年普及の進んでいるマイクロサービスアーキテクチャでは，多数のモジュールが目的や状況に応じて柔軟に連携しあうため，構成が複雑になりやすくシステム全体を通した性能分析や負荷状況の把握が困難である．このような分散した複数のモジュールからなるサービスの実行状況の監視や性能の分析のためには分散トレーシングが用いられる．分散トレーシングでは，各マイクロサービス上で収集した統計情報の整合性を確保するため，サーバ間の高精度な時刻同期が重要である．Linux 上の高精度時刻同期プロトコル (Precision Time Protocol: PTP) の実装では，NIC のドライバ上でパケットのタイムスタンプを取得することで時刻同期の精度を向上させせることができるが，これはドライバの実装に依存するためより汎用な仕組みで実現することが望ましい．本研究ではカーネルが提供する機能の一つである eBPF を用い，可搬性の高い高精度時刻同期を実現した．その設計と実装について述べるとともに，PTP マスタとの同期のずれの推移を元に，本提案実装がドライバ依存の実装と同等の性能を持つことを示す．

# A Portable Implementation of High Precision Time Synchronization Mechanism

***Abstract:*** The microservice architecture is widely used in recent years. In this architecture, there are many modules and they collaborate with each other based on their objective and functionality. In order to monitor the microservices, the distributed tracing is generally used. For the precise monitoring results, the high precision time synchronization is important and a timestamping method is normally used on NIC driver. However, the timestamp precision and synchronization depend on NIC and NIC driver implementation. Thus, it is desirable to realize a portable implementation of high precision time synchronization. In this research, we propose a portable precision time synchronization method using extended Berkeley Packet Filter (eBPF). The proposed method filters Precision Time Protocol (PTP) packets in Linux Kernel and provides the time synchronization method using both monotonic and real time timestamps. In our time synchronization experiments with PTP grandmaster, the proposed method has the similar performance in comparison with NIC driver implementation.

## 1. 序論

サービスのスケーラビリティや対障害性，また利用する開発言語の多様性の確保といった観点から，マイクロサービスアーキテクチャによるサービスの構成が広く使われている．マイクロサービスアーキテクチャでは一つのサービスが複数のマイクロサービスの連携で構成されており，個々のマイクロサービスは物理・仮想を問わない任意のサーバ上で実行される．そのため，あるサービスの実行状態の分析をするためには，複数のサーバに渡る多数のマイクロサービスのログを統合することが必要になる．統合したログの整合性を保つためにはサーバ間の高精度な時刻同期が重要である．また，時刻同期の精度が向上することは関連研究 [1] で示されているように分散データベースなどのアプリケーションにおける性能向上にも繋がる．時刻同期はマスタと呼ばれる同期元とスレーブと呼ばれる同期先がパケットを交換し，時刻のずれを計算・補正することでおこなう．時刻のずれを求めるためには最初にマスターススレーブ間の双方向の通信遅延を測定し，その平均から片方向の通信遅延を求める．そして，マスタからスレーブに送られたパケットの送信タイムスタンプから受信タイムスタンプと通信遅延の差をもとめることで，マスタとの時刻のずれを求めることができる．この際，マスタースレーブ間の双方向の通信遅延は等しいと仮定して片方向の通信遅延

1 法政大学大学院 情報科学研究科
Graduate School of Computer and Information Sciences, Hosei University
2 トヨタ自動車株式会社
TOYOTA MOTOR CORPORATION

© 2022 Information Processing Society of Japan

を求めるため，それらの通信遅延の非対称性は時刻同期の精度に影響を与える．

インターネット上での時刻同期において広く使われている Network Time Protocol(NTP) はインターネットを介して通信するため，通信遅延にばらつきが大きく，同期精度は最も良い場合でもミリ秒程度しか実現できない．そこで，より高精度な時刻同期を実現するためには，同期可能なスコープを狭くし，セグメント内のみとした Precision Time Protocol(PTP) が用いられる．PTP において，通信遅延の計算にはパケットが送受信される瞬間のタイムスタンプを用いると規定されている [2] が，サーバのユーザ空間でタイムスタンプを取得する実装では実際にパケットが送受信されるまでに，カーネルや Network Interface Card(NIC) 内のキューによる長さが不定の処理処理が発生し，それが通信遅延の非対称性の原因となる．そのため，Linux では同期精度を向上させるために NIC のドライバやハードウェア機能を利用してタイムスタンプを取得する仕組みを OS が提供する．しかし，NIC のドライバでタイムスタンプを取得するためにはドライバがその機能を用意する必要がある．更に，バージョンアップに対して継続的にメンテナンスをする必要がある．また，ハードウェアでタイムスタンプを取得するためにはその機能を有する NIC を用意する必要がある．そこで特定の NIC に依存しない汎用な仕組みを用いた実装により時刻同期が実現できることが望ましい．

本研究では Linux における PTP 実装である ptp4l の機能拡張をおこない, extended Berkeley Packet Filter(eBPF) によりタイムスタンプを取得することで時刻同期を実現する．eBPF は Linux カーネルが提供する機能で，eBPF を使用することでカーネル内に複数用意されたフックポイントでユーザが作成したプログラムを動的に実行することができる．これにより，NIC のドライバやハードウェア実装に依存しないタイムスタンプの取得が可能となる．評価として既存の実装である， NIC のドライバによるタイムスタンプ取得と精度比較をおこない，提案手法が同等の精度を実現できることを示す．

## 2. PTP

Precision Time Protocol(PTP) はネットワークを介してマスタとスレーブが時刻同期をおこなうためのプロトコルである．PTP は IEEE の標準化 [2] によってパケットのフォーマットや時刻のずれの計算方法などが規定されており，最良の場合にはサブマイクロ秒の同期精度を達成することができる．

時刻同期はマスタとスレーブ間でパケットを交換しておこなわれる．時刻同期におけるパケット交換の概要を図 **1** に示す．まず，Sync メッセージがマスタからスレーブに送信され，マスタは送信時刻を $t_1$ として記録し，スレーブは受信時刻を $t_2$ として記録する．そして，Fllow_Up メッセージにより $t_1$ をマスタからスレーブに伝える．この時点でスレーブは Sync メッセージの送受信時刻である $t_1, t_2$ を保持している．次に，スレーブは Delay_Req メッセージを送信し，その時刻を $t_3$ として記録する．Delay_Req メッセージを受け取ったマスタは受信時刻を $t_4$ として記録し，Delay_Resp メッセージでスレーブに伝える．PTP パケットにはパケットを一意に定めるための sequenceId が記録されており，Sync や Follow_Up などの各メッセージタイプごとに割り当てられる．Follow_Up メッセージに Sync メッセージの sequenceId を格納することで $t_1$ と $t_2$ の対応を取る．同様に Delay_Req メッセージに Delay_Resp メッセージの sequenceId を格納し $t_3$ と $t_4$ を対応付ける．

PTP においてスレーブクロックとマスタクロックの時刻の差を *offsetFromMaster*, 片方向の遅延の平均を *meanPathDelay* と表し，以下の式で計算する．この際，マスタからスレーブ，スレーブからマスタの通信の遅延は等しいと仮定する．

$$offsetFromMaster = t_2 - t_1 - meanPathDelay \quad (1)$$

$$meanPathDelay = \frac{(t_2 - t_1) + (t_4 - t_3)}{2} \quad (2)$$

サーバ上で PTP を使用して時刻同期をする場合には，ソフトウェアまたはハードウェアでタイムスタンプを取得してパケットに送受信時刻を記録する．ソフトウェアで取得した場合，関連研究で示されるようにサーバのスケジューリングが影響して同期の精度が低下する場合がある．これはカーネル内でパケットに時刻を取得する前にオーバヘッドが発生し往復の遅延が非対称になり，求めた meanPathDelay が実際の通信遅延とずれることで，マスタクロックとの時刻の差を正確に推定できなくなるためである．これには，カーネル内で発生する割り込みやコンテキストスイッチなどのオーバヘッドが原因であると考えられる．一方，NIC には自身に搭載された時計を使用して時刻をハードウェア的に取得する実装をもつ場合がある．これによりカーネル内のオーバヘッドによる影響を無視できるため時刻同期の精度を向上させることができる．しかし，仮想マシンで利用される仮想 NIC は物理的に NIC を持つわけではないため，ハードウェア的に取得することはできない．この機能をエミュレートするように仮想 NIC の実装をしたとしても，ソフトウェア処理となるためハードウェア実装と比較して精度は低下すると考えられる．

## 3. 関連研究

Primorac ら [3] はトラフィック生成器の実装方法の違いによる通信遅延測定の精度について示している．ネットワークトラフィックを生成し，サーバやスイッチのパケット処理性能や通信遅延を測定することができるトラフィック生成器は，Field Programmable Gate Array(FPGA) や

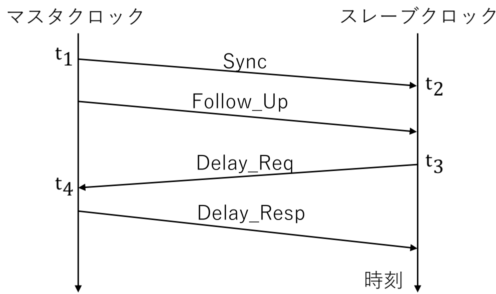


図 **1** PTP におけるパケット送受信の概要
**Fig. 1** Overview of PTP time synchronization mechanism

専用に設計された回路を使用するハードウェア実装と汎用のサーバを使用するソフトウェア実装の二種類がある．通信遅延の測定において，ハードウェア実装では高精度な測定ができるが，利用可能なプロトコルに制限があり，任意のパターンでトラフィックを生成できないなど柔軟性が低く，専用の機器であるため高価である．一方，ソフトウェア実装は任意のプロトコルを利用できるなど柔軟性が高く，一般的なサーバを使用することができるため安価である．しかし，ハードウェア実装と比較して精度が低く，互いにトレードオフの関係にある．

そこで，ソフトウェア実装のトラフィック生成器による通信遅延の測定精度を向上させるため，カーネルバイパスネットワーキングや NIC 上でタイムスタンプを取得する機能が利用されている．カーネルバイパスネットワーキングはカーネル内でパケット送受信の処理をおこなわず，NIC とユーザ空間を直接つなげる仕組みである．この時，CPU のコアをそれ専用に割り当て，NIC をポーリングで監視する．これにより割り込みやコンテキストスイッチ，メモリコピーを無くすことができるため処理遅延を削減し，通信遅延の測定精度を向上させることができる．しかし，ユーザ空間のプログラムは NIC を直接制御するためカーネルが提供していた抽象化レイヤを使用できず，自身が NIC の制御やパケット処理を再実装する必要がある．そこで，カーネルバイパスネットワーキングのライブラリとして Data Plane Development Kit(DPDK)[4] が使用される．DPDK を使用すること NIC の制御やパケット処理をライブラリとして共通化できる．NIC 上でタイムスタンプを取得する機能は，NIC がパケット送受信時にハードウェア的にタイムスタンプを取得する仕組みで精度向上につながる．評価ではこれらの仕組みを利用したソフトウェア実装のトラフィック生成器とハードウェア実装において通信遅延を測定し，ソフトウェア実装とハードウェア実装の測定精度の比較をしている．

スイッチによる遅延を測定した評価では，マイクロ秒の精度において NIC でタイムスタンプを取得する機能を利用したソフトウェア実装の生成器がハードウェア実装の生成器と同程度の精度で遅延を測定できることがわかった．これは，NIC 上でタイムスタンプを取得するとカーネル内での処理遅延の影響を受けないためである．次に，ソフトウェアスイッチの設定を変更することによる遅延の変化を測定できるかどうかを評価した．これについても同様にソフトウェア実装の生成器がハードウェア実装の生成器と同程度の精度で遅延を測定できることがわかった．評価より，マイクロ秒の精度においてソフトウェア実装の生成器はハードウェア実装の生成器と同程度の精度で通信遅延を測定できることが示された．

関連研究では NIC によるタイムスタンプ取得機能を使用すると最も高精度に遅延測定ができることが示された．しかし，本研究ではハードウェアの機能に依存しない汎用な手法によりタイムスタンプを取得したいため，これは使用できない．DPDK を始めとしたカーネルバイパスネットワーキングは汎用的にタイムスタンプを取得することができるが，カーネルバイパスネットワーキングでは NIC をポーリングで監視するため，CPU の 1 コアを専有させる必要がある．マイクロサービスにおいてはサービスを提供することが目的であるため，CPU のコアを専有することは望ましくない．そこで本研究では eBPF を用いることでこの問題を解決する．

## 4. ptp4l

ptp4l は The Linux PTP Project による Linux 上の PTP の実装である [5]．時刻同期のためのタイムスタンプ取得をユーザ空間のプロセスで実行するとコンテキストスイッチやキューによる処理遅延の影響で同期精度が低下してしまう．そこで，ptp4l ではタイムスタンプの取得にカーネルが提供する機能を利用している．具体的には NIC のハードウェア実装を利用して取得するハードウェアタイムスタンプと NIC のドライバで取得するソフトウェアタイムスタンプがあり，ptp4l はどちらにも対応している．ハードウェアタイムスタンプはソフトウェアタイムスタンプと比較して高精度な時刻同期をおこなうことができるが，NIC のハードウェアがその機能に対応している必要がある．ハードウェアタイムスタンプとソフトウェアタイムスタンプでどのように同期精度の差が発生するかを明らかにするために，Linux カーネル内におけるパケット送受信時のフローとそれぞれの時刻取得のタイミングについて示す．パケット受信時の処理の概要について図 **2** に示す．パケットを受信する際にはまず，パケットが信号として NIC に届き，受信キューにパケットが保存される．ハードウェアタイムスタンプを使用する場合には，NIC がプリアンブルとパケットの先頭の境界でタイムスタンプをハードウェア的に取得し，パケットに補助情報として書き込む．キューから取り出されたパケットは Direct Memory Access(DMA) でメインメモリに書き込まれ，NIC はハードウェア割り込みを発生させる．割り込みハンドラはソフトウェア割り

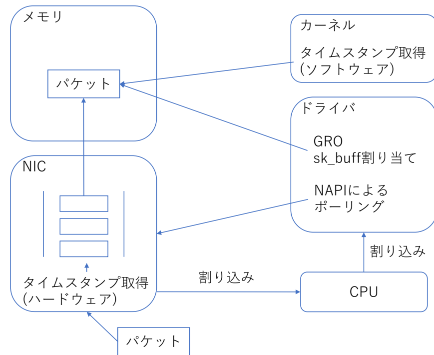


図 2　カーネルにおけるパケット受信処理の概要
**Fig. 2**　Overview of packet receiving in kernel

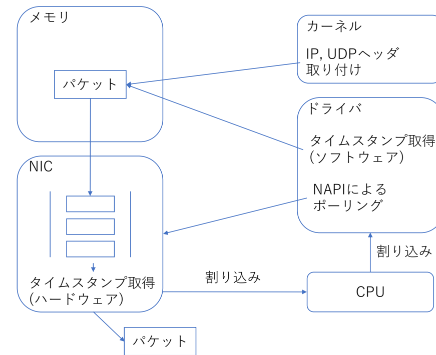


図 3　カーネルにおけるパケット送信処理の概要
**Fig. 3**　Overview of packet transmitting in kernel

込みをスケジュールした後に割り込みをクローズし，ソフトウェア割り込みによって New API(NAPI) が開始される．NAPI はカーネルがポーリングによってパケットを送受信するモードのことで，これにより NIC は一定時間ポーリングで監視され届いたパケットは即座にメモリに転送される．メモリに書き込まれたパケットにはドライバが sk_buff 構造体を割り当て，カーネルのネットワークスタックに渡す．カーネルのネットワークスタックでは有効であれば最初に Generic Receive Offload(GRO) 処理が行われ，キューにキューイングされる．GRO はパケットを集約してから処理する仕組みのことで，Linux ではソフトウェア的にその実装が用意されている．パケットが取りだされると，ソフトウェアタイムスタンプを使用する場合にはタイムスタンプを取得して sk_buff の tstamp フィールドに書き込む．ハードウェアタイムスタンプを使用する場合にはパケットの補助情報を読み取り tstamp フィールドに書き込む．フィールドに書き込まれたタイムスタンプは cmsg 構造体に格納される．パケットは IP や UDP の処理がおこなわれ，アプリケーションに渡される．アプリケーションはソケットオプションを指定し，cmsg 構造体を介してパケットのタイムスタンプを取得する．

パケット送信時の処理の概要について図 **3** に示す．パケットを送信する際に，アプリケーションからカーネルのネットワークスタックに渡されたパケットは UDP や IP ヘッダを付けられた後，ドライバに渡される．そして，ドライバによってメモリから NIC に DMA で書き込まれるが，ソフトウェアタイムスタンプを使用する場合には NIC に書き込まれる直前にタイムスタンプが取得される．取得したタイムスタンプはクローンした sk_buff の tstamp フィールドに書き込まれ，この sk_buff はエラーキューを通じて ptp4l に渡される．これにより送信時のタイムスタンプも ptp4l に伝えることが可能となる．NIC では送信キューにパケットが保存されたあと送信されるが，ハードウェアタイムスタンプを利用する場合には受信時と同様に NIC からパケットが送信される瞬間のプリアンブルとパケットの先頭の境界でタイムスタンプが取得される．

ハードウェアタイムスタンプを使用するとパケットが NIC から送受信される瞬間にタイムスタンプを取得することができる．ソフトウェアタイムスタンプでは，ドライバやカーネル内でタイムスタンプを取得するため NIC やカーネル内の処理途中にキューが存在し，そこでのパケットが留まる時間は不定である．このため，ソフトウェアタイムスタンプではハードウェアタイムスタンプと比較して往復の遅延の大きさが非対称性になり，同期精度が低下する．

## 5. 設計

NIC のドライバやハードウェア実装に依存しない時刻同期のために，汎用的な手法で PTP パケットの送受信時刻を取得する．そのため，時刻の取得に eBPF を用いることで目的のパケットのフィルタリングと時刻取得をおこない，取得した時刻を使用するように ptp4l を拡張し，時刻同期を実現する．

### 5.1 eBPF

BPF はカーネル内でパケットのフィルタリングをおこなうことでユーザ空間に転送するパケットを減らし，処理の効率を高める仕組みのことであった．それをより汎用的な命令セットの使用できるように拡張したものが現在の eBPF である．eBPF は独自の命令セットやデータを保持するためのマップ，カーネル関数を呼び出すためのヘルパ関数などから構成されており，カーネル内のフックポイントでプログラムを実行できるようにする，仮想マシンのような仕組みのことである．eBPF を利用するにはまず，C 言語で eBPF プログラムを作成し，コンパイラにより eBPF バイトコードを生成する．次に，システムコールにより生成したバイトコードをユーザ空間からカーネル内に渡し，検証器によってカーネルを破壊する可能性がないかどうかを検証する．問題がなければ JIT コンパイラによりネイティブのオペコードに変換され，フックポイントでイベントが発生するたびに実行される．

ⓒ 2022 Information Processing Society of Japan

カーネル拡張のための仕組みであるカーネルモジュールではバグや脆弱性を検査する機構を持たないため，安全性や安定性を担保することが難しいほかカーネル関数に依存するため移植性にも問題がある．そこで eBPF ではループの実行回数や境界外へのメモリアクセスが無いかどうかなどについてバイトコードを検証器によって検証することでカーネルを保護する．また，時刻情報の取得などのシステムの機能を用いる必要がある場合には，eBPF プログラムはヘルパ関数を介してカーネル関数を呼び出すことができる．そのため，すべてのカーネル関数を呼び出せるわけではないが，ヘルパ関数はカーネルと同様にメンテナンスされるため eBPF プログラムの移植性は保証される．

### 5.2 eBPF による同期情報の取得

eBPF プログラムはパケットが送受信される際に 1 パケットにつき 1 回実行される．そこで，全パケットのヘッダを調べ，PTP パケットのタイムスタンプを取得する必要がある Sync メッセージもしくは Delay_resp メッセージをフィルタリングし，monotonic clock を取得して ptp4l に時刻を渡す．eBPF プログラムは検証器の制限により sk_buff の tstamp フィールドにタイムスタンプを書き込むことができない．そのため，従来の実装のようにソケットオプションを指定し，補助情報として cmsg 構造体を介してタイムスタンプを受け渡すことができない．提案手法では tstamp フィールドにタイムスタンプを書き込む代わりに eBPF Map を介してアプリケーションに値を渡す．eBPF Map は eBPF プログラムから読み書きできるマップであり，カーネル内に作成される．カーネル内に存在するため，ユーザ空間のアプリケーションから直接読み書きすることはできないが，pinning と呼ばれる仕組みによってそれが可能になる．これは eBPF Map をファイルとしてアクセスできるようにする仕組みである．ptp4l を拡張して pinning された eBPF Map を読み出すようにする．

### 5.3 real time clock と monotonic clock の変換

eBPF において，時刻を取得するヘルパ関数は 2 種類用されているが，どちらも monotonic clock と呼ばれる種類の時刻を返す関数である．monotonic clock はサーバの起動時からの経過時間を表す時刻のことで，一般的に時間を測るために使用される．そのほかの時刻の種類として real time clock があり，これは現在が何時何分であるかを表す．

eBPF のヘルパ関数には monotonic clock を取得する関数のみが用意されているため，eBPF プログラムは real time clock を取得することができない．通信遅延の計算には往復の通信がどの程度の時間でおこなわれたかを求める必要があり，monotonic clock でタイムスタンプを取得しても計算が可能である．しかし，マスタとの時刻のずれを求める *offsetFromMaster* の計算には real time clock が必要となる．そこで，eBPF によって取得した monotonic clock を real time clock へ変換しなければならない．この変換のために eBPF Map から値を読み出す際に monotonic clock と real time clock を同時に取得し，*offsetFromMaster* の計算に必要な real time clock を推定する．まず，monotonic clock 同士の差を求めることで図 1 における $t_2$ からの経過時間を計算し，そして取得した real time clock から引くことで推定をする．

## 6. 実装

同期情報の取得機能を eBPF で実装し，ptp4l に対して eBPF Map からタイムスタンプを取得し同期に使用する機能を実装した．図 4 は提案手法のタイムスタンプ取得メカニズムである．緑の星で示されるように，カーネル内には eBPF プログラムを実行できるように複数のフックポイントが用意されている．青の四角で示されるようにソフトウェアタイムスタンプはカーネル内のドライバでタイムスタンプを取得し，オレンジの四角で示されるようにハードウェアタイムスタンプは NIC 上でタイムスタンプを取得する．本実装では eBPF を使用してタイムスタンプを取得するが，精度向上のためにもっとも NIC に近いフックポイントである Raw Socket を選択を使用する．アタッチされた eBPF プログラムはパケットを送受信するごとに実行され，PTP パケットかどうかをフィルタリングする．PTP パケットであった場合にはそのメッセージタイプを調べ，Sync と Delay_Req メッセージであった場合にタイムスタンプを取得する．

取得したタイムスタンプは eBPF Map に保存し，ptp4l から読み出して使用する．eBPF Map には Array や Hash といった種類があり，今回は Hash を使用する．キーとして PTP メッセージのタイプと sequenceId をメンバとする構造体を設定し，値としてタイムスタンプを格納する．sequenceId はメッセージタイプごとに一意に割り当てられるため，これのみをキーとするとメッセージタイプを区別することができなくなる．そのため，sequenceID とメッセージタイプを組み合わせることで PTP パケットを一意に定めることができるようにする．eBPF Map は eBPF プログラムをアタッチする際にカーネルによって自動的に作成されるが，pinning はユーザがおこなう必要がある．そのため，アタッチと同時に /sys 以下に pinning をし，ユーザ空間のプログラムからアクセスできるようにする．eBPF Map のサイズは予め決めておく必要があり，動的にサイズの変更をすることはできない．そのため，eBPF Map をすべて使い切ってしまわないように ptp4l が値を読みだしたら削除も同時にするようにする．ptp4l は /sys にマッピングされた eBPF Map を open し，ヘルパ関数を使用して読み書きや削除をすることができる．

ⓒ 2022 Information Processing Society of Japan

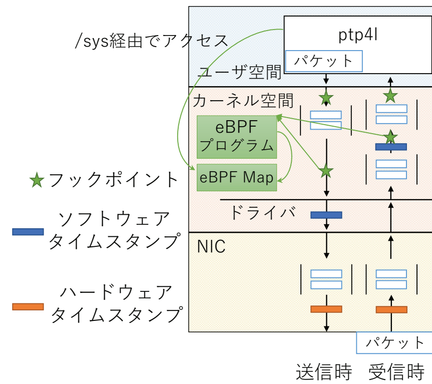


図 4　提案手法のタイムスタンプ取得メカニズム
**Fig. 4** Timestamp achievement mechanism of proposed method

## 7. 評価

評価としては eBPF によるハードウェアによらない汎用実装が，従来のドライバレベルの実装であるソフトウェアタイムスタンプと同程度の性能であることを示す．具体的には，高精度の PTP マスタに対してソフトウェアタイムスタンプと eBPF によるタイムスタンプの実装で時刻同期を行い，時刻のずれ（*offsetFromMaster*）の推移を測定する．また，real time clock と monotonic clock の変換における，それぞれの時刻取得にかかるコストを測定し，実装の妥当性を確認する．さらに，C-state や P-state などのプロセス実行の安定性に影響を与えそうな各種設定パラメータを変更した場合の同期精度への影響も示す．

### 7.1 従来実装との同期精度比較

実験ではマスタとスレーブをスイッチを介して接続し，同期をおこなった．マスタには GPS を時刻源とし，高精度な PTP パケットを生成することが可能な Thunderbolt PTP GM200 を使用し，スレーブには Intel Core i7-4790 を搭載した PC に カーネルバージョン 5.13.0-21-generic の Ubuntu 20.04.3 LTS をインストールして使用した．NIC には Intel Ethernet Server Adapter I350-T4V2 を使用した．

*offsetFromMaster* の推移の測定ではまず最初に date コマンドを使用してスレーブの時計を 0.1 秒進め，次にソフトウェアタイムスタンプ及び eBPF 実装の ptp4l を 1800 秒実行し，時刻同期をおこなう．この間， 1 秒ごとに *offsetFromMaster* を記録する．ソフトウェアタイムスタンプを使用する場合も同様にスレーブの時計を進め，時刻同期を実行しながら *offsetFromMaster* を記録する．これをそれぞれ 9 回ずつ繰り返した．

図 **5** はソフトウェアタイムスタンプによる *offsetFromMaster* の推移を $-50000 - 50000ns$ の範囲について示した．図 **6** には提案実装による *offsetFromMaster* の推移を同様に示した．図 **7** はそれぞれの *offsetFromMaster* の相補累積分布関数 (CCDF) であり，縦軸は対数で示す．同期開始直後の時刻のずれは試行によって異なるため，図 7 では 1200 秒経過してからの推移をプロットし，同期が安定してからの推移を比較する．図 5 と図 6 より両実装における特徴的な差は見られない．図 7 より 97% 程度までは同期精度には差がなく，残りの 3% 程度のパケットについては最大で約 $10000ns$ のずれが見られる．

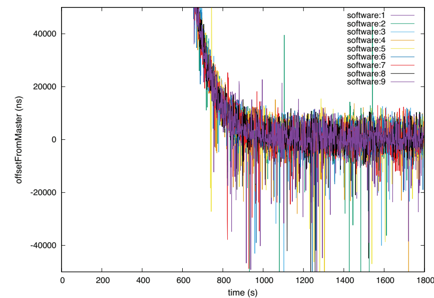


図 **5**　ソフトウェアタイムスタンプによる *offsetFromMaster* の推移
**Fig. 5** Estimation results of *offsetFromMaster* using software timestamp

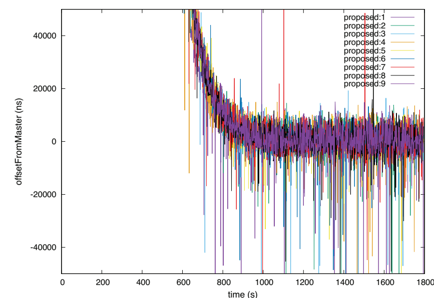


図 **6**　提案実装による *offsetFromMaster* の推移
**Fig. 6** Estimation results of *offsetFromMaster* using proposed method

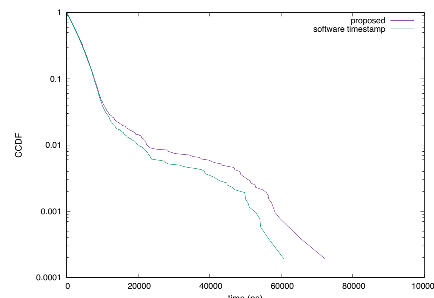


図 **7**　*offsetFromMaster* 推定値の CCDF
**Fig. 7** CCDF of *offsetFromMaster* using both methods

### 7.2 ユーザ空間における時刻取得のコスト

monotonic clock から real time clock に変換するためにはそれらの時刻を同時に取得する必要がある．しかし，そ

ⓒ 2022 Information Processing Society of Japan

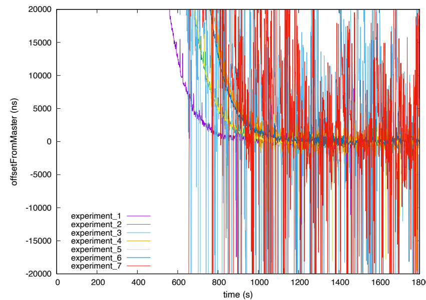

**図 8** スレーブの設定変更による *offsetFromMaster* の推移
**Fig. 8** Estimation results of *offsetFromMaster* with server tuning configurations

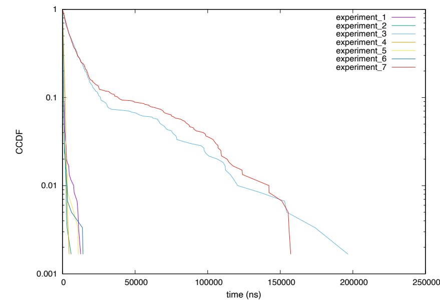

**図 9** スレーブの設定変更による *offsetFromMaster* 推定値の CCDF
**Fig. 9** CCDF of *offsetFromMaster* with server tuning configurations

れぞれの時刻の取得を連続に実行したとしても，個々の処理に時間がかかる場合にはそのずれが時刻同期精度低下の原因になる．そこで，それぞれの時刻取得にかかる時間を測定することで実装の妥当性を確認する．実験ではまず，monotonic clock を取得する．そして測定したい時刻取得関数を実行し，再び monotonic clock を取得する．取得した 2 つの monotonic clock の差から時刻取得関数の実行にかかる時間を求めることができる．これを monotonic clock と real time clock でそれぞれ 100000 回ずつ実行した．結果を表 **2** に示す．実験結果を見ると，取得コストの最小と最大に大きな差があるが，大きなコストを発生させる回数は少ないため平均は小さくなっていると考えられる．

### 7.3 プロセス実行時の安定性による同期精度への影響

スレーブにおけるプロセス実行の安定性によっては時刻同期の精度に影響を及ぼす可能性がある．そこで，Hyper-Threading，C-state，P-state，Turbo Boost，実行コアの制御の設定を表 **1** のように無効化して *offsetFromMaster* の推移を測定した．ここで，実験 1 が最も不安定な要素が少なく，実験 7 が最も不安定な設定になっている．また，実験 2-6 は個々の要素の影響を調べるものである．この実験では実行コアの制御以外の設定をハードウェア的におこなうために Intel Core i5-10500 を搭載したスレーブを使用した．インストールしたソフトウェアや構成は 7.1 節と同じものを使用した．Hyper-Threading と Turbo Boost，P-state は CPU の処理能力を向上させるほか，消費電力を削減するために CPU の動作周波数を動的にさせる．そのため，パケット処理にかかるの時間が変動し，時刻同期の精度に影響を及ぼす可能性がある．C-state は CPU の一部の動作を停止することで消費電力を削減する仕組みで，これが有効になっていると電力削減時にパケットが到着した場合に，電力回復による遅延の影響が出る可能性がある．また，ptp4l はユーザ空間で動作するため OS のスケジューリングによって複数の CPU のコア上で動作する可能性がある．CPU のコアはそれぞれが独立の monotonic clock を保持しているため，参照する CPU コアによって

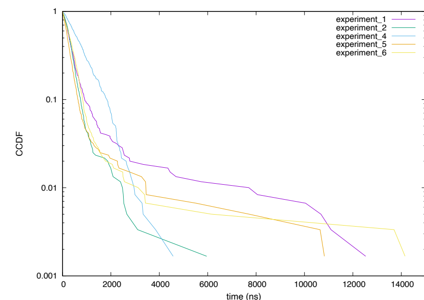

**図 10** 実験 3 と実験 7 を除いた CCDF
**Fig. 10** CCDF without Experiment 3 and Experiment 7

時刻が変動する可能性がある．そこで，実行する CPU のコアを固定することでどのような影響があるかを調べる．この実験ではでは既存の実装であるソフトウェアタイムスタンプを使用した ptp4l を用いる．結果は図 **8** に示す．*offsetFromMaster* が安定した 1200 秒経過後の CCDF を図 **9** にプロットし，特に変動の大きい実験 3 と実験 7 を除いた CCDF を図 **10** にプロットする．実験の結果は，C-state の制御は時刻同期の安定性に大きな影響があることを示唆しており，次に P-state の影響がある．一方，ほかの項目は大きな影響を与えていないことがわかる．

## 8. 考察

ソフトウェアタイムスタンプと提案手法のどちらの手法について図 5 と図 6 より *offsetFromMaster* は時刻同期開始後 1200 秒ほど経過すると約 ±10000 の範囲で安定的に推移している．どちらの手法もパケットを送受信してからタイムスタンプを取得するまでに処理遅延が発生するためこの程度の同期精度となると考えられる．どちらの手法でも *offsetFromMaster* が 1 秒前と比較して大きく変化し，すぐに収束するような推移が見られる．この原因は 7.1 節の実験結果より C-state を有効化していたためと考えられる．両手法を CCDF でプロットしたものが図 7 であり，97 パーセンタイルまでは同等の同期精度を実現している．その後，10000$ns$ の差が発生するがこれは *offsetFromMaster*

表 1　サーバ tuning 設定
**Table 1**　A list of server tuning configurations

| | 実験 1 | 実験 2 | 実験 3 | 実験 4 | 実験 5 | 実験 6 | 実験 7 |
|---|---|---|---|---|---|---|---|
| Hyper-Threading の無効化 | yes | no | yes | yes | yes | yes | no |
| C-state の無効化 | yes | yes | no | yes | yes | yes | no |
| P-state の無効化 | yes | yes | yes | no | yes | yes | no |
| Turbo Boost の無効化 | yes | yes | yes | yes | no | yes | no |
| 特定の CPU コアで実行 | yes | yes | yes | yes | yes | no | no |

表 2　時刻取得関数の実行コスト
**Table 2**　Execution cost of the timestamp functions

| | real time clock($ns$) | monotonic clock($ns$) |
|---|---|---|
| 最小 | 25 | 25 |
| 最大 | 20690 | 17840 |
| 平均 | 40 | 27 |
| 標準偏差 | 89 | 81 |

が大きく変化し，すぐに収束するような変動が影響していると考えられる．この結果から，提案手法はソフトウェアタイムスタンプと同等の精度を維持しながらより汎用的に時刻同期を実現できていると考えられる．

時刻取得関数の実行について表 2 より real time clock と monotonic clock のどちらも数十 $ns$ の実行コストが必要であることがわかった．どちらの実行にも最長の時間で数十 $\mu s$ かかる．実験では実行コストを求めるために時刻取得関数の呼び出しの前後に時刻を取得しており，その間に OS のスケジューリングや割り込みが発生し，余分な時間をコストとして数えていると考えられる．そのため，最大に近いコストは実際のコストを反映していないと考えられる．時刻取得関数の実行コストの平均はナノ秒単位であり，事前の予想よりも小さい結果となった．これは vDSO(virtual dynamic shared object) という仕組みを利用しているためである．vDSO はシステムコールをカーネル側で共有オブジェクトとして作成しておき，ユーザ空間から呼び出せるようにする仕組みである．これによりコンテキストスイッチをおこなわずに処理ができるため実行コストの削減につながったと考えられる．平均コストで比較すると real time clock のほうが 10$ns$ ほどコストが大きい結果となった．この差は monotonic clock から real time clock へ時刻を変換するためにユーザ空間で時刻取得関数を呼び出す提案手法に影響を与える可能性があるが，PTP による時刻同期の精度は $\mu s$ 程度であり，実際には影響しないと考えられる．

図 8 を見ると，実験 3 と実験 7 がほかよりも大きく変動する結果となり，これらの共通点は省電力設定である C-state を有効化しているということである．図 9 で明らかなように，C-state を有効にすると 90 パーセンタイルでは 25000$ns$ 以上 *offsetFromMaster* が大きく変動していることがわかる．C-state を有効にすると電力消費を抑えるために CPU のコアが動作を停止するようになる．そのため，PTP パケットを送受信した際に CPU コアを停止状態から動作状態に遷移させる処理が発生する場合があり，この時間が同期精度に影響を与えていると考えられる．実験 3 と実験 7 を除外してプロットした図 10 より，97 パーセンタイルまでは 実験 4 がほかよりも精度が低下している．実験 4 は P-state を有効化しており動作周波数を低下させることで処理に時間がかかり，パケット送受信からタイムスタンプを取得するまでに遅延が発生していると考えられる．ただし，P-state の影響は C-state よりも 2 桁ほど小さい．図 10 より，Hyper-Threading，Turbo Boost，実行コアの制御は同期精度に影響を与えないと考えられる．実験 3 と実験 7 を除く実験において 97 パーセンタイルまでは同期精度が良いが，一時的に精度が低下する場合がある．ソフトウェアタイムスタンプを使用する従来の手法と提案手法はどちらもカーネル内でタイムスタンプを取得するため，設定を変更した場合において，実装の違いによる影響は無いと考えられる．

## 9. 結論

本研究ではカーネルの機能拡張のための汎用的な仕組みである eBPF を使用し時刻同期のためのタイムスタンプ取得をおこなうように拡張することで，NIC のドライバやハードウェア実装に依らない汎用的な時刻同期手法を提案した．また，ソフトウェアタイムスタンプを使用する既存の実装と精度の比較をおこない，同程度の同期精度を実現できることを示した．

### 参考文献

[1] Li, Y., Kumar, G., Hariharan, H., Wassel, H., Hochschild, P., Platt, D., Sabato, S., Yu, M., Dukkipati, N., Chandra, P. and Vahdat, A.: Sundial: Fault-tolerant Clock Synchronization for Datacenters, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 1171–1186 (2020).

[2] IEEE Standard: Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, Technical report (2020).

[3] Primorac, M., Bugnion, E. and Argyraki, K.: How to Measure the Killer Microsecond, *Proceedings of the Workshop on Kernel-Bypass Networks*, KBNets '17, New York, NY, USA, Association for Computing Machinery, p. 37–42 (online), available from

⟨https://doi.org/10.1145/3098583.3098590⟩ (2017).

[4] DPDK Project: DPDK, DPDK Project (online), available from ⟨https://www.dpdk.org/⟩ (accessed 2022-05-06).

[5] The Linux PTP Project: The Linux PTP Project, The Linux PTP Project (online), available from ⟨http://linuxptp.sourceforge.net/⟩ (accessed 2022-05-06).

ⓒ 2022 Information Processing Society of Japan