Vol.2022-OS-154 No.5
2022/3/14

TCPネットワークコネクションのスケーラブルなライブマイグレーション機構

山田 咲太朗[1,a)]　松原 克弥[1,b)]

概要：クラウド等で稼働するシステムの可用性を向上させる手段として，プロセス・マイグレーション技術が広く用いられている．ネットワーク通信をともなう実行プロセスをマイグレーションする場合，移送すべきプロセスの実行状態として，確立されている TCP ネットワークコネクションが含まれる．Linux の標準プロセスマイグレーション・ツールである CRIU では，マイグレーション対象のプロセスに対する新たなネットワークコネクション要求を遮断したのち，対象プロセスに対応する TCP プロトコルスタックの送受信キューに残るデータを OS カーネルから取り出すことにより，TCP ネットワークコネクションの状態をマイグレーションできる．負荷分散を目的としたプロセスマイグレーションでは，高負荷状態のプロセスがマイグレーション対象になることが想定されるが，Web サーバ等のネットワークシステムにおける高負荷プロセスは，保持している TCP ネットワークコネクションの数も多くなる．その結果，前述のマイグレーション処理にかかるオーバヘッドがネットワークコネクション数に比例して増加するため，マイグレーションにともなうシステム・ダウンタイムに多大な影響を与える．本研究は，TCP ネットワークコネクションに対するマイグレーション処理の最適化手法を検討する．本検討では，CRIU における TCP ネットワークコネクションのマイグレーション処理を分析して，オーバヘッドが大きい処理を特定し，コネクション数に比例してオーバヘッドが増加しない，スケーラブルな処理方式と置き換えることで，マイグレーションにともなうシステム・ダウンタイムを最小限に抑えることを目指す．

キーワード：プロセスマイグレーション，CRIU，パケットフィルタリング，eBPF/XDP

1. はじめに

スマートフォンの普及や IoT の導入拡大にともなって，ネットワークサービスにおける可用性の向上がより重要な課題となっている．サービスの可用性を向上させる手段として，スケールアップやスケールアウトなどの，サービスを提供するシステムが稼働するハードウェア環境を増強する方法が広く知られている．しかし，ハードウェア増強による手段は，機器導入コストやシステム再構築などの運用コストが大きい．また，C10K 問題 [1] のように，ハードウェアだけ増強しても可用性が向上しない場合もあり，ソフトウェアによる解決が求められる．

可用性の向上を実現するソフトウェア技術として，ライブマイグレーションがある．ライブマイグレーションは，負荷分散や高可用性の実現を目的として利用される技術で，プロセスや VM（Virtual Matchine）などを単位として，ある計算機上で実行中のシステムコンポーネントを別の計算機上へ移行して，その実行を継続させる技術である．ネットワークサービスを提供するプロセスが実行されているサーバが高負荷状態になったり障害が発生したりして，サービスの継続が困難な状況においても，そのプロセスを実行状態とともに別のサーバへ移行することで，サービスの完全停止を防ぐことができるようになる．

サービスを提供しているプロセスのライブマイグレーションは，そのプロセスの実行状態を示すデータを抽出して，別の計算機にネットワーク転送し，データから実行状態を復元するまでの処理にかかるオーバヘッドが原因となり，サービスが一時的に停止するダウンタイムが存在する．ネットワークサービスを提供するプロセスの実行状態には，CPU のレジスタやスタック・ヒープなどのメモリに加えて，サービス提供先との通信状態である TCP ネットワークコネクションも含まれる．TCP ネットワークコネクションは，通信先の宛先情報，TCP 通信制御のためのシーケンス番号やウインドウ制御パラメータ，ネットワークスタックの送信キューに残る未送信データや受信キューに残

[1] 公立はこだて未来大学
Future University Hakodate, Hakodate, Hokkaido, Japan
[a)] b1018106@fun.ac.jp
[b)] matsu@fun.ac.jp

表 1　予備実験の環境

ゲスト（VM）	OS	Ubuntu 20.04.3 LTS
	vCPU	2 core
	vMem	2GB
ホスト PC	OS	Windows 11 Pro
	CPU	Intel Core i7-11800H
	RAM	32GB

図 1　予備実験の構成

る未処理データを移行することによりマイグレーションできる [2]．さらに，これらの情報を保存・復元する間に状態が変化しないようにするために，対象プロセスへ届くパケットの制御が必要となる．しかし，マイグレーション対象となるプロセスがクライアントと通信状態である場合，処理時間は通信中のクライアント数に比例して増加することが分かっている

マイグレーション時間が増加することの問題として，処理時間に応じて対象のシステムにダウンタイムが発生してしまうという問題がある．マイグレーションの処理ではプロセスおよびクライアントとの通信を一時的に停止するため，マイグレーション実行中のシステムはクライアントの処理を捌くことができず，ダウンタイムが発生してしまう．

本研究では，マイグレーションによるオーバーヘッドの削減をすることで，システムの可用性向上へ貢献することを目標とする．本稿では，Linux における標準プロセスマイグレーション・ツールである CRIU を対象として，ネットワークサービスを提供するサーバプロセスをマイグレーションする際のオーバヘッドを分析して，ボトルネックとなる処理を特定する．さらに，TCP ネットワークコネクションを多数保持した高負荷状態のサーバプロセスをマイグレーションした際においても，クライアント数に比例せず，ほぼ一定のオーバーヘッドで処理することが可能なマイグレーション機構の実現手法を検討する．

2.　プロセスマイグレーション処理の分析

Linux におけるプロセスマイグレーションの標準ツールである CRIU における，プロセスマイグレーション処理にかかるオーバヘッドを分析するために予備実験を行った．実験環境を表 1 に示す．本実験環境の構築には，ホスト PC 上に VirtualBox で作成した VM を用いた．Nginx プロセスに対して，Apache Benchmark により負荷をかけた状態で，CRIU を用いて実行中 Nginx プロセスのチェックポイントを保存した後に一旦プロセスを終了させ，その後，

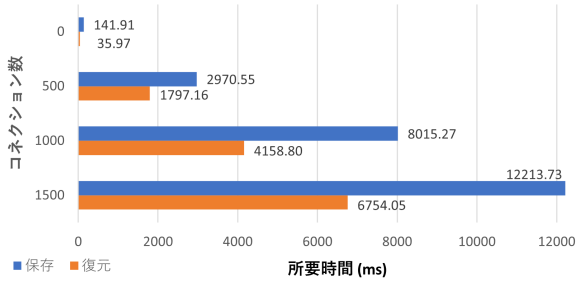

図 2　Nginx のチェックポイントとレストアにかかるオーバヘッド

表 2　CRIU におけるマイグレーション処理の分析

conn.	各プロセスの実行時間 [ms] criu	iptables	sh	other
チェックポイント処理				
0	121.63	13.51	N/A	6.76
500	1,173.07	1,260.40	533.51	N/A
1,000	3,806.45	3,325.54	844.81	N/A
1,500	5,938.31	5,297.09	974.656	N/A
レストア処理				
0	34.02	N/A	N/A	1.95
500	189.60	1,162.94	385.31	59.49
1,000	249.94	2,738.57	1,154.48	15.80
1,500	847.63	4,846.71	1,050.25	N/A

N/A の部分は，全体の 0.01%未満の時間のため測定不可

新たに起動した Nginx プロセスで保存したチェックポイントを用いて実行状態をレストアした（図 1 参照）．その際，パフォーマンスカウンタである perf（Performance analysis tools for Linux）コマンド[*1]を用いて，CRIU の各処理にかかる時間を計測した．Nginx の最大同時接続数設定は，デフォルト値の 1,536 とした．ApacheBench が Nginx へ送信するリクエスト数は，1 回の計測につき 500,000 とし，同時リクエスト数を 0, 500，1,000，1,500 と変化させた．計測は，各設定毎にそれぞれ 10 回ずつ行い，平均を算出した．

実行中の Nginx プロセスのチェックポイントとレストアにかかるオーバヘッドの計測結果を図 2 に示す．同時コネクション数が 0 の場合，マイグレーション処理にかかる時間は約 170 ミリ秒で，チェックポイント処理に 141.91 ミリ秒，レストア処理に 35.57 ミリ秒かかっている．さらに，同時コネクション数が増えるのと比例して，マイグレーションにかかる時間も増加していた．同時コネクション数が 1,500 の時は，マイグレーション処理に約 18 秒もかかっていることがわかった．同時コネクション数が増加するにしたがってマイグレーション対象プロセスが保持する TCP ネットワークコネクション数が増加するため，それらのチェックポイントとレストア処理にかかる時間がコネクション数に比例して増えているためにマイグレーション処理のオーバヘッドが増大していると考察できる．

[*1] https://perf.wiki.kernel.org/index.php/Main_Page

perf を用いて TCP ネットワークコネクションをもつ Nginx プロセスのチェックポイントとレストアの各処理の解析を行った結果を表 2 に示す．CRIU によるチェックポイント処理とレストア処理は，CRIU のコマンドである criu プロセスと criu プロセスから起動される iptables とシェル（sh）の 3 つのプロセスにより行われている．criu プロセスは，対象プロセスの実行情報に関する情報の収集と実行状態をファイルへ読み書きする処理を担っている．iptables プロセスは，TCP ネットワークコネクションの状態をチェックポイント・レストアする間に状態が変化しないよう，対象プロセスへ届くパケットをブロックするために，パケットフィルタリングおよびネットワークアドレス変換機能である Netfilter に以下の 2 つのルールを追加したり解除したりするために起動されている．

(1) Nginx から ApacheBench クライアントへ送信されるパケットを破棄する
(2) Nginx から ApacheBench クライアントへ送信されるパケットを破棄する

シェルプロセスは，iptables プロセスを起動するために起動されている．実験結果から，iptables とシェルプロセスの実行が，チェックポイント処理全体の実行時間の 52%〜60%を占めており，レストア処理においても全体の 85%〜92%を占める結果となっている．

以上示した予備実験の結果から，TCP ネットワークコネクションを多数持つプロセスを対象とした CRIU のマイグレーション処理におけるボトルネックは，以下の 2 つであることを特定した．第 1 のボトルネックは，シェルと iptables プロセスの起動オーバーヘッドである．ひとつの TCP ネットワークコネクションにつき，シェルを介して iptables を起動していることに加え，それぞれ前述の 2 つのルールを設定するために，その呼び出し回数はコネクション数の 2 倍となる．2 つ目のボトルネックは，iptables による通信制御である．iptables は，Netfilter にルールを追加する際に，一度すべてのルールのデータをユーザランドにコピーしたのちに，すべてを再設定する実装になっている．そのため，設定されているルールが増加するにつれて，新たな設定追加にかかる実行時間が増加してしまう．

以上に特定した 2 つのボトルネックを解消することで，CRIU によるマイグレーション処理のオーバーヘッドを大幅に削減できると考える．

3. スケーラブルな TCP コネクション・マイグレーション機構

本研究では，CRIU によるマイグレーション処理のなかでもオーバーヘッドの大きい処理である，ネットワーク通信制御の設定にかかる時間を削減する．ネットワーク通信制御の設定をプロセスフォークや iptables を使用せずに

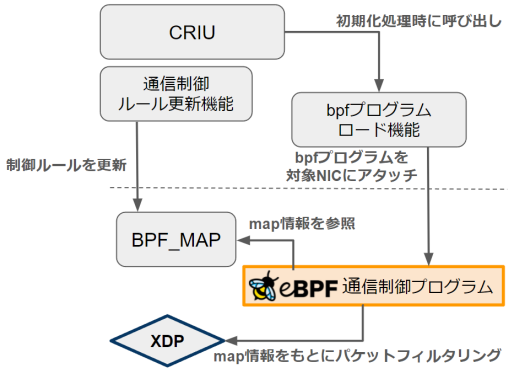

図 3　提案システム概要図

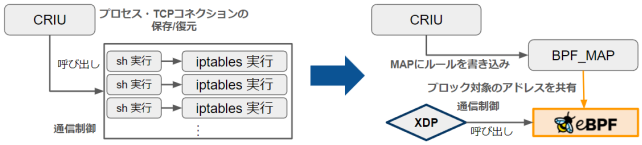

図 4　オリジナルの CRIU との相違点

CRIU プロセス内で処理することで，オーバーヘッドの削減を図る．また，従来の iptables によるネットワーク通信制御の設定やパケット処理アルゴリズムを改善することで，オーバーヘッドの削減が可能だと考える．

図 5 に提案システムの概要図を示す．本システムでは，CRIU のプログラム内に独自のネットワーク通信制御処理を実装するにあたって，eBPF と XDP のを活用することを検討している．

前項で述べたように，ネットワーク通信制御処理を XDP で実装することで，パケットが Linux のプロトコルスタックに到達する前にパケット処理をすることが可能になる．そのため，従来の Netfilter によるパケット処理機構より，高速なパケット処理が可能だと考える．また，ネットワーク通信制御の設定を BPF MAP を使用して Linux カーネル内に共有することで，iptables のように設定をすべてユーザ空間にストアせずとも設定の更新が可能になると考える．

図 4 に従来の CRIU との相違点を示す．iptables を使用したネットワーク通信制御では対象となるプロセスが保持する TCP コネクション 1 つにつき 2 回プロセスフォークとシェルによる iptables の呼び出しがされていたが，eBPF を用いることで，ファイルディスクリプターである BPF MAP に対して 2 回書き込み命令をするだけで同等の処理が可能になる．

3.1 実現手法の検討

前述した提案機構の実現における要である，eBPF/XDP を用いたパケットフィルタリング機構と eBPF ロードプログラムの実装方法について述べる．図 3 は実装するシステムの概要図である．本研究により CRIU に新たに実装する

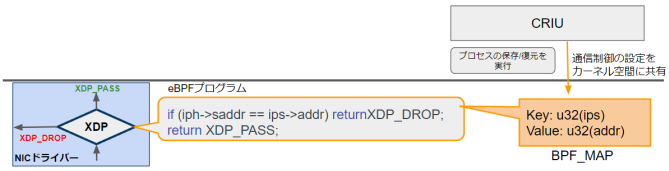

図 5　実装したシステム概要図

機能は，2 つ存在する．一つ目は，bpf プログラムをロードするための機能である．前章で述べた通り，eBPF プログラムをカーネル空間で実行するためには大前提として，eBPF プログラムをカーネル空間にロードする必要がある．そこで，本研究では指定した NIC に対して eBPF プログラムをアタッチとデタッチが可能な関数を実装する．次に，eBPF によるパケット処理機構である．eBPF がアタッチされている NIC に到達したパケットデータを解析し，パケットフィルタリングを行うプログラムを作成する．最後に，BPF Map を介した通信制御の設定を更新する機能を実装する．従来の CRIU では，通信制御の設定を文字列操作処理を用いて iptables を実行するコマンドに書き換える方式であった．しかし，本提案システムでは，eBPF Map を用いるため，文字列操作処理から eBPF Map ファイルディスクリプターに通信制御の設定を書き込む処理に書き換える．以上，3 つの機能を作成して CRIU に導入することで，マイグレーション処理のオーバーヘッドを削減できると考える．

3.1.1　eBPF ロードプログラムの実装

eBPF ロードプログラムを実装するにあたって libbpf ヘルパーライブラリを使用した．eBPF プログラムをロードするには，bpf_prog_load_xattr() 関数を用いる．しかし，bpf_prog_load_xattr() は eBPF プログラムをロードするだけで，NIC にアタッチすることができない．そこで，ロードされたプログラムを NIC にアタッチするために bpf_set_link_xdp_fd() を用いる．bpf_set_link_xdp_fd() 関数は第一引数にデバイスを第二引数に eBPF プログラムのファイルディスクリプターを指定することで，指定したデバイスに eBPF プログラムをアタッチまたは，デタッチが可能になる．また，第 3 引数にはドライバレベルの XDP フックを使用するためのフラグを指定することが可能である．bpf_set_link_xdp_fd() 関数は対象のデバイスにすでに eBPF プログラムがアタッチされている場合，失敗する仕様なため，フラグを使用することでエラーを回避することが可能になる．

3.1.2　パケットフィルタリング機構の実装

パケットフィルタリング機構は NIC がパケット取得時に実行されるプログラムのことである．パケット処理の流れを図 6 に示す．パケットフィルタリングをするには取得したパケットからフィルタリングに必要なデータを取得する必要がある．この処理で取得するデータは全部で 4 つ存在する．

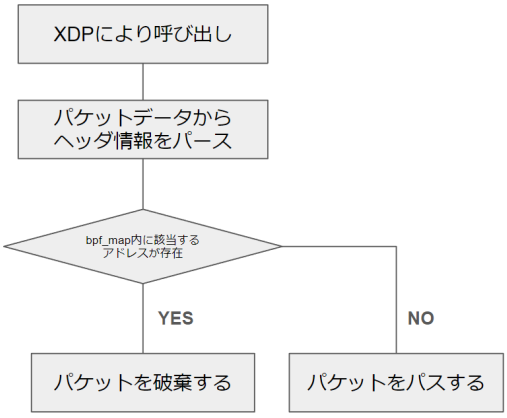

図 6　パケットフィルタリング機構のフロー

(1) 送信元の IP アドレス
(2) 送信先の IP アドレス
(3) 送信元のポート番号
(4) 送信先のポート番号

送信元の IP アドレスと送信先の IP アドレスはそれぞれ iphdr 内の saddr フィールドと daddr フィールドから取得することが可能である．一方，送信元のポート番号と送信先のポート番号は tcphdr 内の source フィールドと dest フィールドから取得することができる．しかし，各ヘッダ内のデータはネットワークバイトオーダー（big endian）形式であるため，パケットフィルタリング処理にデータを使用する際にはホストバイトオーダー形式に変更して扱う．実装する eBPF プログラムではパケットデータをネットワークバイトオーダーからホストバイトオーダーに変換する際には bpf_ntohs() 関数を使用する．パケットフィルタリング処理では，送信元の IP アドレスとポート番号並びに送信先の IP アドレスとポート番号のセットが BPF Map 内に存在するかを検証する．BPF Map 内に該当するデータの組み合わせが存在する場合には，パケットを破棄し，そうでない場合には Linux のプロトコルスタックにパケットデータを転送する．

4.　関連研究

早川ら [3] は，ビデオストリーミングやクラウドバックアップなどのデータ集約型サービスを提供するためのコンポーネントであるオブジェクトストレージシステムの負荷分散を目的とした独自のフレームワークである Prism を考案，実現している．従来のオブジェクトストレージシステムの主要アーキテクチャは Frontend proxy と Content-based routing の 2 種類が存在する．クライアントとオブジェクトデータを保持するバックエンドがフロントエンドマシンを仲介してやり取りをするアーキテクチャである．全てのクライアントからのリクエストを中継しているため，フロ

ントエンドはマシンリソースの大半を利用しているのに対して，バックエンドはマシンリソースの 20%程度しか利用していない．そのため，システム全体のマシンリソースを効率よく利用できるアーキテクチャにする必要がある．Frontend proxy とは異なり，クライアントとバックエンドの中継役を担うフロントエンドの役割をネットワーク Switch が担っている．しかし，Switch では通信プロトコルが TLS による暗号化がされていない UDP ベースのカスタムプロトコルであるため，クライアント - バックエンド間で安全な通信が求められる場面では利用できない．TCP Conection repair を利用して，クライアントの通信先のサーバを動的に切り替えるアーキテクチャを提案．クライアントからのリクエスト処理はフロントエンドが受け取り，バックエンドは直接クライアントにオブジェクトを送信することでマシンリソースを効率的に利用することが可能になるだけでなく，TLS による暗号化された通信が可能になる．TCP コネクションのマイグレーションを利用しすることでシステムの可用性を向上させた研究である．早川らの研究では TCP Connection repair を利用することで，システムの可用性を向上させた．本研究の成果によって TCP コネクションのマイグレーションにおけるオーバーヘッドが削減されることで Prism の可用性とアクセス性能が向上すると考える．

Sebastiano ら [4] は，Linux 上で大量のコンテナアプリケーションを運用する場合において，従来よりも効率的なコンテナ内外のネットワーク通信制御を可能にする機構を考案，実現をした．運用するコンテナアプリケーションが増えると同時に，ネットワーク通信制御のルールも増加する．従来までは通信制御のルール更新に iptables が利用されていたが，iptables はルール更新時に全てのルールをユーザランドにコピーするため，ルール数が増加すると同時に処理時間が増加してしまう．そのため，iptables によるネットワーク通信制御はスケーラビリティが低い．iptables によるネットワーク通信制御の代わりを eBPX/XDP を用いて実現している．通信制御における効率的なアルゴリズムである LBVS(Linear BitVector Search) を用いた実装により，通信制御のルール数が，増加した場合においても高いスループットを維持することが可能になった．本研究と Sebastiano らの研究は，アプローチ方法が類似している．しかし，アプローチをする分野で異なっていると考える．Sebastiano らの研究では，コンテナ型アプリケーションのネットワーク通信制御の分野で eBPF/XDP を用いた通信制御が有効な方法であると示した．一方で，本研究では，TCP コネクションのライブマイグレーションに eBPF/XDP を用いた通信制御の有効性を示すものである．

5. おわりに

本研究では TCP コネクションにおけるマイグレーションのオーバーヘッドを削減することでシステムの可用性を向上させることを目的としてる．目的達成のために，TCP コネクションのマイグレーション時に必要なパケット制御を iptables を呼び出すのではなく eBPF/XDP を用いた独自のネットワーク通信処理を CRIU コード内に組み込むことでオーバーヘッドの削減をするシステムの提案をした．また，提案手法が有効であるかを評価するために，実施する実験について検討した．

本稿では，予備実験より CRIU のマイグレーション機構のボトルネックの原因がプロセスフォークと iptabeles を使用したネットワーク通信制御にあると考察し，それらのオーバーヘッドを削減するための機構を検討した．今後は，検討した提案手法を実装したのちに，評価実験を実施することで，提案手法が有効であるかを評価する．また，本研究の提案手法は，CRIU プロセスそのものではなく CRIU によって呼び出されるプロセスのオーバーヘッドを削減するものであった．そのため，CRIU プロセスそのもののオーバーヘッドの削減には至っていないと考える．マイグレーション処理のさらなる最適化として，CRIU プロセスのオーバーヘッドの調査と削減を検討する．さらに，本提案の実現を CRIU や eBPF/XDP などの Linux の機能に依存しない手法に一般化させることで，FreeBSD などの非 Linux 環境との間でも効率的なプロセスマイグレーションが可能な機構を実現したい．

謝辞 本研究は，JSPS 科研費 JP21K11832 の助成を受けたものである．

参考文献

[1] Kegel, D.: The C10K problem, (online), available from ⟨http://www.kegel.com/c10k.html⟩ (accessed 2022-02-21).
[2] 松原克弥，髙川雄平：動的な OS 切替え実現を目指した FreeBSD における Linux プロセスのマイグレーション機構，コンピュータソフトウェア，Vol. 39, No. 1, pp. 18–32 (2022).
[3] Hayakawa, Y., Honda, M., Santry, D. and Eggert, L.: Prism: Proxies without the Pain, *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, pp. 535–549 (2021).
[4] Miano, S., Bertrone, M., Risso, F., Bernal, M. V., Lu, Y. and Pi, J.: Securing Linux with a Faster and Scalable Iptables, *SIGCOMM Comput. Commun. Rev.*, Vol. 49, No. 3, p. 2–17 (2019).