東京大学大学院
情報理工学系研究科 電子情報学専攻
修士論文

RDMA を用いた Proxy 型ネットワークスタック アーキテクチャの提案と実装

Design and Implementation of the Proxy Architecture to Disaggregate Network Stack using RDMA

坂本 裕紀
Hiroki Sakamoto

指導教員　江崎 浩　教授

2018 年 2 月

概要

インターネット上でやりとりされるトラフィックは全世界的に増大傾向にある．こうしたトラフィックを支えるネットワークは広帯域化しており，特にデータセンターにおいてサーバーに要求されるネットワーク I/O 性能は高速化している．ネットワークが高速化するとサーバーにおいて通信のために消費される CPU リソースは増加する．サーバー上で動作する CPU-intensive なアプリケーションに対してより多くのリソースを割けるようにするため，通信において効率的な CPU リソースの利用が求められる．しかし，従来のオペレーティングシステムのネットワークスタックによる処理はメモリコピーやコンテキストスイッチによるオーバーヘッドが大きく，CPU リソースの効率的な利用という要求に十分に応えられていない．そのため，オーバーヘッドの小さい高速な通信を可能にする技術が必要とされる．

オーバーヘッドの小さい高速な通信を実現する既存の技術として，高速パケット I/O 処理技術や Remote Direct Memory Access (RDMA) 技術がある．これらの技術はカーネルでの処理をバイパスしてユーザー空間からゼロコピーでのデータ転送を行い，高性能で効率的な通信を可能にする．しかし，これらは独自のインターフェースを提供しており，従来の通信インターフェースとの互換性が無く実装コストが高い．また，カーネルをバイパスするため，カーネルのネットワークスタックで提供される機能を利用することができない．

そこで本研究では，通信にかかるサーバーの CPU リソース消費を削減するため，TCP/IP ネットワークスタックの処理をアプリケーションサーバーから外部ノードに分離するアーキテクチャ，RoPT を提案する．サーバーと外部ノード間のデータ転送を RDMA で行い，TCP/IP のネットワークスタック処理とコネクションをノードに Proxy させるシステムを設計・実装した．プログラム実行時に Socket API 関数を差し替えることにより，従来のアプリケーションにおいて特別な変更を加えること無く RoPT を用いた通信を行うことを可能にしている．TCP メッセージの送信性能評価を行い，RoPT を用いた通信では従来の TCP/IP 通信に比べ CPU リソースの消費を抑えた通信が可能であることを示した．

目次

図目次

表目次

第 1 章

序論

1.1 本研究の背景と目的

インターネット上において提供されるサービスは日々多様化し，システムの高機能化が進んでいる．たとえば多くの人やモノがインターネットにつながるようになったことで取り扱われるデータの量は増え，そうしたデータの処理分析を行うようなサーバー上のアプリケーションはより多くの計算リソースを必要としている．同時に，クライアント数の増加や提供されるコンテンツのリッチ化にともない，サーバーで処理すべきネットワークトラフィックが増大している．ネットワークトラフィックの増大にしたがい 10 Gigabit Ethernet (GbE) や 40 GbE，さらには 100 GbE といった高速なネットワークインターフェース (NIC) の導入が進み，サーバーに要求されるネットワーク I/O 性能は高速化している．

要求される高速なネットワーク I/O 性能に対して，従来のオペレーティングシステム (OS) カーネル内のネットワークスタックによる処理はオーバーヘッドが大きいことがよく知られている [10, 24]．このオーバーヘッドを生じる主な要素として，カーネル空間とユーザー空間の間で発生するメモリコピー，システムコールや受信割り込みによって発生するコンテキストスイッチが挙げられる．これらは CPU サイクルを消費し，高速なネットワーク I/O 処理の実現への障壁となっている．サーバー上で動作するアプリケーションが計算リソースを十分に活用できるようにするため，高速でかつより効率的なオーバーヘッドの小さい通信を可能にする技術が必要とされる．

そこで，ソフトウェアでの高速なパケット I/O 処理を可能にするライブラリやフレームワークが開発された [5, 15, 18, 26]．これらの技術はオーバーヘッドの大きいカーネルでの処理をバイパスし，ユーザー空間において高速に直接パケットバッファへアクセスすることを可能にする．これらの技術は高いネットワーク I/O 処理性能を達成できるが，課題もある．これらのフレームワークはそれぞれ独自の Application Programming Interface (API) を持ち実装コストが高い．また，パケットに直接アクセスするためプロトコルスタック自体も実装しなくてはならない．従来のアプリケーションをこうしたフレームワークを用いて再開発する場合，無視できない大きなコストとなる．

カーネルをバイパスしゼロコピーでのデータ転送を実現するもうひとつのアプローチとし

て，Remote Direct Memory Access (RDMA) 技術がある．RDMA は高性能コンピューティングの分野でノード間のインターコネクトにおけるデータ転送に用いられる通信技術である．RDMA ではノード間でメモリからメモリへ直接データ転送を行う [25] ため，データ転送にかかる CPU 負荷の関与が最小限に抑えられ低遅延で高速な通信が可能となる．それまでは RDMA を用いるには InfiniBand の設備が必要であったが，近年 Ethernet 上で RDMA を用いる技術があらわれ [19]，データセンターネットワークでの RDMA の利用が以前より容易となった．この技術を利用して実環境のデータセンターネットワークで RDMA を運用した研究も行われている [17]．しかし RDMA による通信は TCP/IP 通信とは通信アーキテクチャが全く異なるため，双方のノードにおいて RDMA で通信を行うことが保証されていなくてはならず一般的な End-to-End 通信に用いることはできない．

このような背景を踏まえ，本研究では，既存のアプリケーションやプロトコルスタックといった資産を利用できるようにしつつ，通信にかかるサーバーの CPU リソース消費を抑えた高速な通信を実現することを目的とする．本研究では，図 1.1 に示すようにサーバーのネットワークスタックの処理を分離するアーキテクチャによって，通信にかかるサーバーの CPU リソース消費を外部へオフロードする．本研究で提案するアーキテクチャでは，従来はサーバー自身で行う TCP/IP ネットワークスタック処理を外部ノードに担わせる．サーバーと外部ノードとの間のデータ転送を RDMA で行い，そのノードに TCP コネクションを Proxy させる．また，サーバー上ではアプリケーションに対して従来の通信インターフェースと互換性のあるインターフェースを提供する．このような Proxy 型のアーキテクチャによって本研究の目的とする通信を実現する．

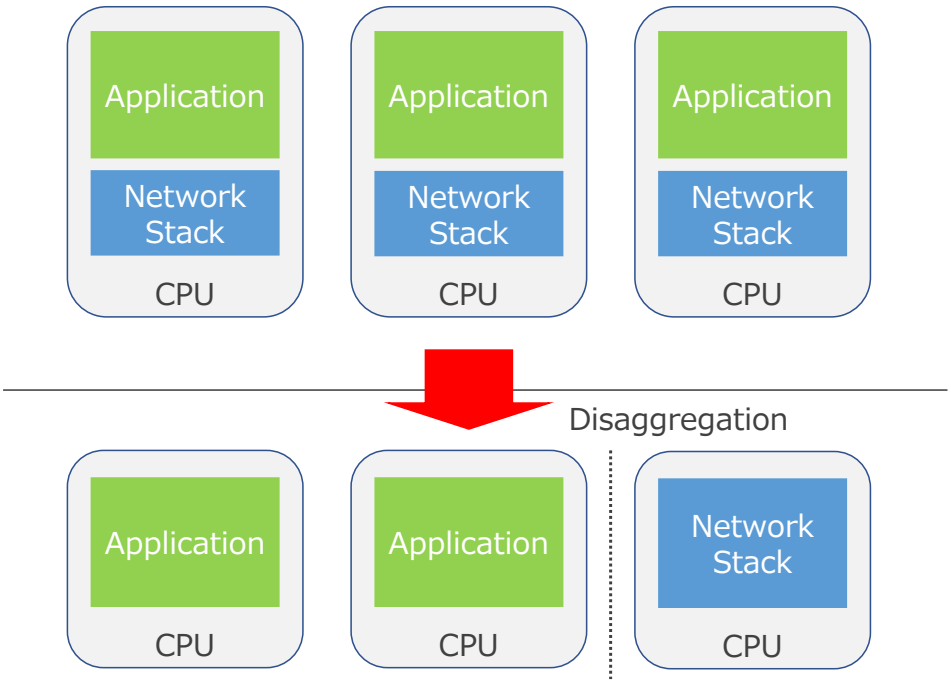

図 1.1. ネットワークスタックの処理を分離するアーキテクチャ

1.2 本論文の構成

本論文の構成は次のとおりである．まず 2 章において，高速化するネットワークの現状と高機能化するアプリケーションによる要求を述べ，それに対して従来のオペレーティングシステムのネットワークスタックはオーバーヘッドが大きいという課題があることについて述べる．次に 3 章で，オーバーヘッドの小さい高速な通信を実現する技術として，ソフトウェアによる高速パケット I/O 処理技術と RDMA 技術について述べる．これらの技術の特性と課題を整理し，本研究において達成されるべき目的を示す．4 章では本研究で提案するアーキテクチャの概要と設計を述べ，5 章でその実装について示す．6 章では実装したアーキテクチャの評価を行い，考察する．最後に 7 章で本論文をまとめ，今後の課題を述べる．

第 2 章

ネットワークの高速化とオペレーティングシステム

本章ではまず，高速化するネットワークの現状について述べる．次に，サーバー上で動作するアプリケーションの高機能化によってリソースがより効率的に利用されることが求められていることを述べる．最後に，従来のオペレーティングシステム (OS) のネットワークスタックによる処理はオーバーヘッドが大きく，高速化とリソースの効率的な利用という要求に応えられていないことを述べる．

2.1 高速化するネットワーク

インターネット上でやりとりされるトラフィックは全世界的に増大傾向にある．2017 年の Cisco の調査分析 [2] によると，2016 年から 2021 年にかけての 5 年間で 約 2.9 倍のトラフィック増大が見込まれている．この予想にはスマートフォンの普及および Machine-to-Machine (M2M) 通信の増加にともなうモバイルトラフィックの増大と，既にトラフィックの 7 割を占める動画トラフィックのさらなる増大が大きく寄与している．スマートフォンの普及および M2M 対応機器の増加によって接続ユーザー数が増加を続けるだけでなく，ユーザー当たりのトラフィック量も同様に増大を続ける．こうしたネットワークトラフィックの増大を支えるのは，それに対応して広帯域化するネットワークである．

ネットワークの広帯域化にともない，データセンター内のネットワークには帯域や速度，遅延といった観点でより高い性能が要求される．近年はサービスを提供するアプリケーションのクラウド環境への移行が進み，インターネット上で提供されるサービスはデータセンターにおいて稼働するものが多い．リソースがデータセンターに集約されることにより，ネットワークトラフィックもデータセンターへ集中する．また，データセンター内のネットワークでやりとりされるパケットは仮想的なオーバーレイのネットワークを構成するために多重にカプセル化されている．そのため転送されるデータの実質的な中身のサイズよりも実際に転送されるパケットサイズの方が大きく，見かけのトラフィックはより増大する．

ネットワークに要求される広帯域性を達成するため，データセンターでは高速な NIC の導入

が進んでいる．データセンター内のサーバーにおいて 10 GbE の NIC は今や広く用いられており，40 GbE や 100 GbE といったさらに高速な NIC の導入も進められている [4,7,13,17]．これによって，以前はネットワークリンクがボトルネックであったようなシステムにおいても状況が変化しつつあり，高速なネットワークのトラフィックを扱うためにサーバーに要求されるネットワーク I/O 性能が高速化している．

2.2 高機能化するアプリケーションによる要求

インターネット上において提供されるサービスは日々多様化しており，それによって新たな機能が次々と実現されている．多くの人やモノがインターネットにつながるようになったことで取り扱うデータの量が増え，それらのデータの分析や検索といった処理がサーバー上で行われている．また，画像認識や音声認識といった機械学習や深層学習を用いた機能もインターネット上で提供されるようになっている [1,3,6]．このようにサーバー上で動作するアプリケーションは高機能化が進んでおり，こうしたアプリケーションはより多くの計算リソースを必要とする．

理想的なのはサーバーにおける CPU リソースをすべてアプリケーションに対して割り当てることが可能なことである．しかし実際には，通信のためにネットワーク処理でも CPU リソースが消費される．ネットワークが高速化するにしたがって処理すべきネットワークトラフィックも増大するため，この処理に消費される計算リソースも増加する．サーバー上で動作する CPU-intensive なアプリケーションに計算リソースを割けるようにするため，通信において効率的な計算リソースの利用が求められる．

2.3 OS のネットワーク処理におけるオーバーヘッド

要求される高速なネットワーク I/O 性能と効率的な計算リソースの利用に対して，従来の OS カーネル内のネットワークスタックによる処理はオーバーヘッドが大きい [10,24]．このオーバーヘッドを生じる主な要素として，カーネル空間とユーザー空間の間で発生するメモリコピー，システムコールや受信割り込みによって発生するコンテキストスイッチが挙げられる．また，プロトコルスタック自体も複雑なものになっている．本節ではこれらのオーバーヘッドについて述べる．

2.3.1 メモリコピー

Linux のメモリ空間においてカーネル空間とユーザー空間は分離されている．カーネルではリソースを管理してアプリケーションに提供できるようハードウェアに近い処理を行う．そのためカーネルには高い実行権限が与えられており，カーネルが使用するメモリ空間はユーザーから直接アクセスできないように保護されなくてはならない．メモリ空間が分離されることでこうしたセキュリティが確保されている．セキュリティが確保される一方で，ユーザー空間に

あるデータをカーネル空間でも共有するためにはメモリコピーが必要となる．

従来のネットワーク処理においては，ユーザー空間で動作するアプリケーションがカーネル内のネットワークスタックを用いてデータの送受信を行う．このとき，プロトコル処理をカーネル内で行うため，データが一度カーネル空間にコピーされる必要がある．これはメモリ上の異なる領域間でのメモリコピーであり，CPU サイクルを消費して行われる．

2.3.2　コンテキストスイッチ

コンテキストスイッチとは，CPU がそれまで行なっていた動作から切り替えて別の動作を始めるときに CPU の状態を切り替える過程のことである．CPU が動作を切り替えるときにはレジスタなどの状態を復元できるように保存したり保存された状態を復元したりする仕組みが必要となる．たとえば，特権命令を実行するためなどの理由から，カーネル空間での動作とユーザー空間での動作とを切り替えるときにコンテキストスイッチが発生する．また，マルチタスクを実行している間のスレッドの切り替わりや外部割り込みによっても発生する．この仕組みがあることによって，複数のタスクがひとつの CPU を共有して用いることが可能となる．

コンテキストスイッチにはオーバーヘッドがともなう．CPU レジスタは状態を保存および復元しなくてはならず，また，OS カーネルによるスケジューラーが実行されなくてはならない．このために CPU サイクルが消費される．さらに，タスクが切り替わることによりアドレス空間が切り替わるため，キャッシュの内容や Translation Look-aside Buffer (TLB) のエントリも切り替わることになる．こうしたコストは無視できない [14].

カーネル内のネットワークスタックによる処理では，システムコールや受信割り込みによってコンテキストスイッチが発生する．通信に用いるソケットを作成しコネクションを確立した後，データを送信するために send や sendto，sendmsg といったシステムコールが呼ばれる．システムコールが呼ばれることによりユーザー空間からカーネル空間へと動作が切り替わる．また，パケットを受信したときには NIC から外部割り込みが入り，割り込みハンドラのスレッドに動作が切り替わる．こうした切り替わりでコンテキストスイッチが発生する．Linux カーネルでは，2.6 以降のバージョンにおいて Linux NAPI [27] という割り込みとポーリングを動的に使い分けて割り込みを減らす仕組みが導入されており，受信割り込みによるコンテキストスイッチのオーバーヘッドを軽減している．

2.3.3　プロトコルスタック

Linux のような汎用 OS はネットワーク処理のために特化して開発されたものではない．カーネルにおいてリソースの管理を行い，OS 上で動くさまざまなアプリケーションに対して抽象化されたインターフェースでネットワーク機能を提供している．アプリケーションによってネットワークに要求される機能はさまざまであるため，カーネル内のネットワークスタックにはそうした要求を満たすための実装も含まれる．これによってプロトコルスタックが複雑化

している．

また，TCP のプロトコル処理自体にも重い処理が含まれる．TCP は信頼性を確保するコネクション指向型のプロトコルであり，プロトコルスタックにはパケットの再送制御やウィンドウ制御，輻輳制御といった処理が実装される．これらは信頼性を保証するために必要な処理であるがコストでもある．さらに，処理はパケットごとに行われる．こうしたプロトコルスタックでの処理においても CPU サイクルが消費される．

以上で述べたように，従来の OS カーネル内のネットワークスタックを用いることによるオーバーヘッドはサーバーの CPU サイクルを消費し，ネットワークの高速化にともない消費する計算リソースが増大する．また，このために高速なネットワーク I/O に対して十分に対応できていない．高速なネットワークに対応し，サーバー上で動作する CPU-intensive なアプリケーションに対してリソースを割けるようにするため，高速でかつより効率的なオーバーヘッドの小さい通信を可能にする技術が必要とされる．

第 3 章

高速な通信を実現する技術

2 章で述べたように，従来のカーネル内のネットワークスタックを用いた通信はオーバーヘッドが大きい．そこで，そうしたオーバーヘッドを回避あるいは軽減することでソフトウェアでの高速なパケット I/O 処理を可能にする技術が開発された．また，高性能コンピューティングの分野でノード間のインターコネクトに用いられている Remote Direct Memory Access (RDMA) 技術をネットワーク I/O として用いる研究も行われている．本章では，オーバーヘッドの小さい高速な通信を実現する既存の技術としてこれらの技術について述べる．そしてこれらの技術における特性と課題を整理し，本研究で提案するアーキテクチャにより達成されるべき要求事項を示す．

3.1 ソフトウェアによる高速パケット I/O 処理技術

汎用のサーバー上においてオーバーヘッドの小さい高速な通信を実現するため，Data Plane Development Kit (DPDK) [5] や netmap [26]，PF_RING [15]，PacketShader [18] I/O engine (PSIO) といった，ソフトウェアによる高速なパケット I/O 処理を可能にするライブラリやフレームワークが開発された．これらの技術はオーバーヘッドの大きいカーネル内のネットワークスタックでの処理をバイパスし，ユーザー空間で動作するアプリケーションから高速に直接パケットバッファに対してアクセスすることを可能にする．カーネルでの処理をバイパスするため，これらの技術を用いる場合プロトコル処理を含め実装することになる．この仕組みを利用して，高速パケット I/O 処理技術を用いて高速な TCP/IP スタックを実現する研究も行われている [11, 21, 30].

高速なパケット I/O 処理技術では従来のカーネルのオーバーヘッドを回避している．このため，通信にかかるサーバーの CPU リソース消費についてカーネルの関与は最小限であると見なせる．サーバーの CPU リソース消費という観点から見ると，理想的にはこの技術はサーバー上においてアプリケーションが動作するほかはネットワークスタック処理のみが動作するというアーキテクチャである．

3.1.1 高速化を実現する仕組み

DPDK や netmap といった高速パケット I/O 処理技術では，2.3 節で述べたような従来のカーネルのオーバーヘッドを回避あるいは軽減することで高速化を実現している．本項では，この高速化を実現する仕組みとして DPDK で用いられるゼロコピー，カーネルバイパス，Poll Mode Driver (PMD)，バッチ処理について述べる．

ゼロコピー

従来のカーネル内のネットワークスタックを利用してパケットを受信する場合，NIC で受信されたパケットはまず NIC のドライバーによってカーネルのネットワークスタックに渡される．そこでプロトコル処理を行い，ユーザー空間で動作するアプリケーションにデータが渡される．この過程で発生するメモリコピーがオーバーヘッドになっていた．

DPDK では独自のドライバーを用いて，NIC で受信されたパケットをあらかじめ確保したメモリプールのパケットバッファに Direct Memory Access (DMA) で書き込む．DMA を用いたコピーでは CPU リソースは消費されない．ユーザー空間で動作する DPDK アプリケーションはこのパケットバッファに対して直接アクセスできるようになっており，CPU リソースを消費するメモリコピーを行うことなくパケットを扱うことが可能である．この仕組みがゼロコピーである．ユーザー空間で動作するアプリケーションにパケットバッファへの直接アクセスを提供するこのような仕組みは，netmap や PF_RING，PacketShader にも同様に存在する．

カーネルバイパス

カーネル内のネットワークスタックでの処理は重く，また，システムコールで発生するコンテキストスイッチにはオーバーヘッドがともなう．高速パケット I/O 処理技術は，カーネルでの処理をバイパスしユーザー空間で直接パケット扱うことによってこれらのオーバーヘッドを回避している．たとえば DPDK アプリケーションは，図 3.1 に示すように完全にユーザー空間のアプリケーションとして動作する．ユーザー空間からパケットバッファへの直接アクセスのために Userspace I/O (UIO) ドライバーを用いている．

このように DPDK や netmap のようなフレームワークはカーネルでの処理をバイパスするため，カーネルによって抽象化されて提供されていた Socket API ではなく，高速にパケットにアクセスするための独自 API を提供している．カーネルにおけるソケットはファイルディスクリプタであり，抽象化されたオーバーヘッドの大きな仕組みである．独自 API はパケットへの高速なアクセスを行うための軽量なもので，こうしたソケットのオーバーヘッドも回避する．しかし，従来のアプリケーションをこの独自 API に合わせて再実装することは無視できない大きなコストである．

また，カーネルでの処理をバイパスするため，アプリケーションにとって必要なプロトコル処理を実装する必要がある．mTCP [21] や IX [11]，StackMap [30] のように，高速パケット

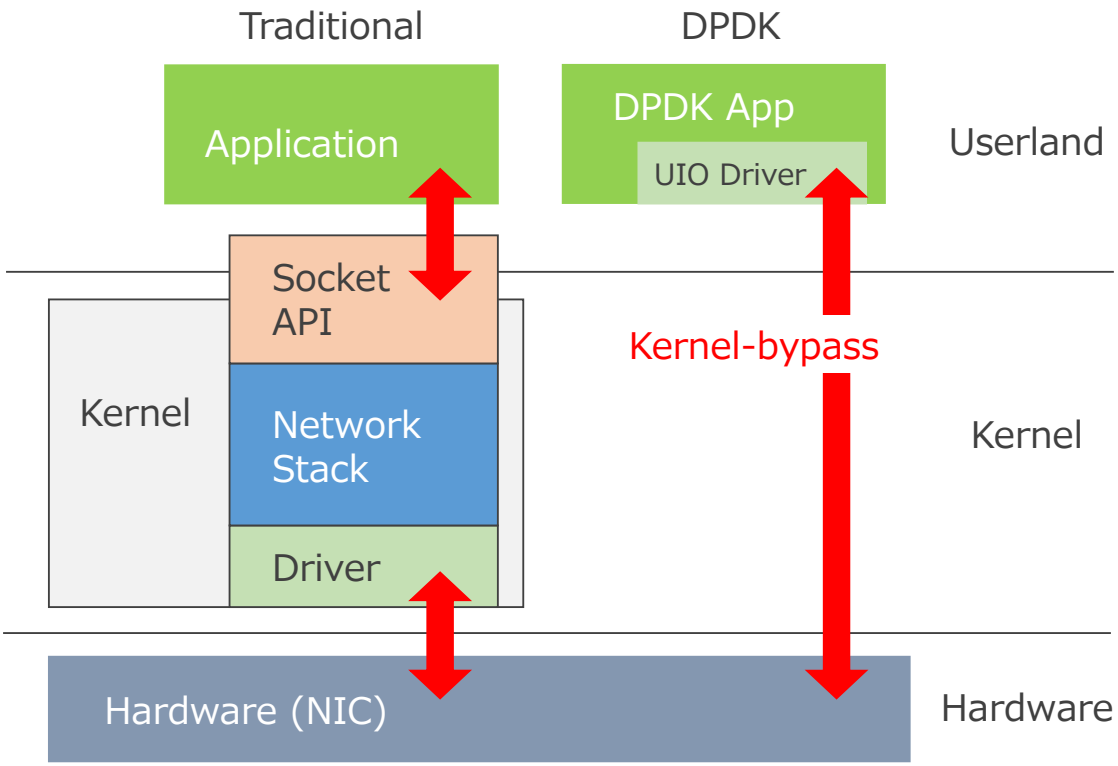

図 3.1. DPDK におけるカーネルバイパス

I/O 処理技術を用いて実装された高速な TCP/IP スタックが存在している．こうした実装によって高速なプロトコル処理を行うことが可能となる．しかしカーネルでの処理をバイパスすることに変わりはなく，iptables のようなカーネルのネットワークスタックに含まれる機能を使うことはできない．

Poll Mode Driver (PMD)

パケット受信時の外部割り込みにより発生するコンテキストスイッチはオーバーヘッドである．DPDK では，受信割り込みを待つのではなくポーリングでパケットを取得しにいくことによってこのオーバーヘッドを無くす．このために Poll Mode Driver (PMD) を用いる．PMD は割り込みを用いずポーリングによって，NIC で受信したパケットを受信用キューに積んでいき，送信用キューに積まれたパケットを NIC から送信する．これによって高速なパケットの送受信を可能にしている．しかしポーリングは CPU サイクルを消費する．高速なネットワークのトラフィックを処理し続けるようなアプリケーションであれば適しているが，パケットが受信されないときにも CPU サイクルを消費してポーリングを行う．DPDK 以外のフレームワークでは，たとえば netmap や PF_RING では PMD を用いていない．

バッチ処理

従来のカーネル内のネットワークスタックにおける処理ではパケットごとに処理を行なっており，これがオーバーヘッドになっている．高速パケット I/O 処理技術では，複数のパケットを一度にまとめて処理するバッチ処理を行うことによってこのオーバーヘッドを軽減している．バッチ処理は，NIC とパケットバッファとの間でパケットを送受信する段階，アプリケーションからパケットバッファにアクセスする段階と，それぞれの段階において適用が可能である．このバッチ処理によって高いスループット性能を達成できるが，まとめて処理するパケット数を増やすことによってレイテンシが大きくなるため，低レイテンシが必要なアプリケーションにおいては注意が必要である．

3.1.2 高速パケット I/O 処理技術における課題

ソフトウェアによる高速パケット I/O 処理を可能にする技術は前項で述べたような仕組みによって高いネットワーク I/O 性能を達成できるが，主にカーネルをバイパスすることによって生じる次のような課題もある．

- 独自 API であるため実装コストが高い
- 従来のカーネル内ネットワークスタックの機能を利用できない

まず，これらのフレームワークはそれぞれ独自の API を持ち実装コストが高いという課題がある．高速なパケットへのアクセスを可能にするため，フレームワークは新たに独自の API を提供している．こうした API は従来の Socket API とは抽象度も内容も異なるため，従来のアプリケーションをこれらのフレームワークを用いて再実装する場合大きな開発コストとなる．

また，カーネルをバイパスしパケットに直接アクセスするため，従来のカーネルのネットワークスタックの機能を利用できないという課題がある．アプリケーションは，カーネル内に実装されているネットワークスタックとは別に必要なプロトコル処理を実装しなくてはならない．つまり高いネットワーク I/O 性能を求めるために，既存の資産でもあるカーネルスタックのコードを使えなくなる．カーネルのネットワークスタックの機能としてたとえば iptables があるが，こうしたカーネルで適用されるフィルタやセキュリティの仕組みも同様に使うことができない．

3.2 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) はネットワークを介してリモートノードのメモリに対して直接アクセスすることを可能にする技術である．この技術は高性能コンピューティングの分野においてノード間のインターコネクトに広く用いられている．RDMA によるデータ転送は低レイテンシで高いスループットを実現し，かつリモートノードのメモリ

へのアクセスにかかる CPU の関与は最小限に抑えられる．RDMA の代表的な実装である InfiniBand [20] は TCP/IP/Ethernet とは異なる通信アーキテクチャであるが，このような通信効率の高さからデータセンター内のノード間通信にも用いられる [12, 16]．それまでは RDMA を用いるには InfiniBand の設備が必要であったが，近年 Ethernet 上で RDMA を用いる技術があらわれ [19]，データセンターネットワークでの RDMA の利用が以前より容易となった．

本節では，まず RDMA の通信アーキテクチャについて述べ，従来の TCP/IP/Ethernet による通信との違いを整理する．次に，データセンターネットワークにおける RDMA の利用について述べる．最後に，オペレーティングシステムにおけるネットワーク I/O としての側面から見た RDMA についてまとめる．

3.2.1　通信アーキテクチャ

本項では RDMA の代表的な実装である InfiniBand における通信アーキテクチャについて述べる．RDMA では図 3.2 に示すようにノード間でメモリからメモリへ直接データ転送を行う [25]．図中の Host Channel Adapter (HCA) は Ethernet で構成されるネットワークにおける NIC に相当するハードウェアである．図中のように RDMA でリモートのメモリに書き込む場合，まず送信側の HCA はあらかじめ確保されたメモリ領域から Direct Memory Access (DMA) でデータを読み込み受信側の HCA に転送する．受信側の HCA は同様にあらかじめ確保されたメモリ領域に対して受信したデータを DMA で書き込む．このようにしてノード間におけるメモリからメモリへの直接転送が実現される．また，InfiniBand において，送信側ではユーザー空間で動作するプログラムが確保したメモリ領域を送信する領域に指定できる．これは受信側でも同様である．これによって，異なるノードのユーザー空間アプリケーション同士がカーネルバイパスでかつゼロコピーでの通信を行うことが可能となる．

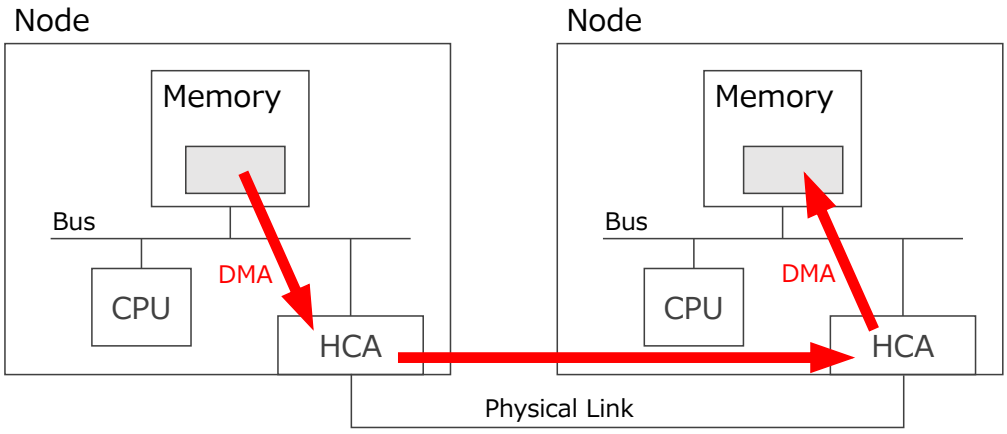

図 3.2. RDMA の概観

InfiniBand では OSI 参照モデルにおける物理層からリンク層，ネットワーク層，トランス

ポート層までを仕様で定めており [20]，従来のネットワークにおける TCP/IP/Ethernet による通信とは異なる通信アーキテクチャで動作する．InfiniBand においてネットワークのプロトコルはハードウェア (HCA) に実装されており，このオフロードによって CPU リソースの消費が抑えられ低レイテンシな通信を実現している．RDMA はリモートノードへのアクセスを行う上でロスレスで低遅延なリンク環境を前提としており，InfiniBand はこれを提供する．また，従来のカーネル内のネットワークスタックを利用した TCP や UDP の通信ではソケットによって通信を行うのに対し，RDMA では Queue Pair (QP) によって通信を行う．以下では，QP による通信と QP による通信でサポートされるオペレーション，トランスポート層のサービスについて述べる．

Queue Pair (QP) による通信

RDMA では Queue Pair (QP) によって通信を行う．QP は HCA によって提供される仮想的なインターフェースで，Send Queue (SQ) と Receive Queue (RQ) のペアからなる．SQ と RQ はそれぞれソケットによる通信の送信バッファと受信バッファに対応するが，SQ や RQ に格納されるのはデータ本体ではなくそれぞれ送信要求と受信要求である．HCA は送信要求の情報に基づき自身のノードのメモリからリモートノードのメモリへデータを転送する．受信要求にはリモートノードから受信したデータを置く場所に関する情報が含まれる．HCA は要求を消費すると完了したことを知らせるエントリを Completion Queue (CQ) に格納していく．プログラムは CQ に積まれたエントリから処理の完了を知ることができる．また，通信先の QP の特定には QP Number (QPN) を用いる．QPN は TCP や UDP におけるポート番号に相当し，ポート番号が 16 ビットであるのに対して QPN は 24 ビットである．ただし QPN は QP 作成時に HCA によって自動的に割り当てられるものであるため，ポート番号とは異なり特定の値を指定することができない．

QP によって RDMA の通信を行うときは，HCA にアクセスさせるメモリ領域をあらかじめ Memory Region として登録しなくてはならない．QP に格納される送信要求や受信要求にはデータを格納するメモリ領域が含まれており，これは仮想メモリアドレスである．あらかじめ Memory Region として登録することによって，HCA が仮想アドレスを物理アドレスに変換して DMA でアクセスすることが可能となる．ローカル側で Memory Region を登録していてもリモート側で登録されていない場合，通信することはできない．

サポートされるオペレーション

QP による通信では Send/Receive，RDMA Write，RDMA Read，Atomic といったオペレーションがサポートされる．これらのオペレーションは QP に格納される要求にしたがって非同期で実行される．このうち RDMA Write と RDMA Read，Atomic が RDMA のオペレーションである．これらのオペレーションの内容について以下に述べる．

- Send/Receive
  送信側のバッファのデータを受信側のバッファに転送する，基本的な送受信モデルのオ

ペレーションである．送信側の Send Buffer オペレーションと受信側の Post Receive Buffer オペレーションとによって通信を行う．送信側の SQ の送信要求には送信されるデータのメモリ領域が含まれ，受信側の RQ の受信要求には受信したデータを書き込むバッファのメモリ領域が含まれる．この通信モデルは，従来のソケット通信で行われる送受信モデルに近い．

- RDMA Write
  ローカルのメモリ領域を読み込み，そのデータをリモートのメモリ領域に書き込むオペレーションである．このオペレーションではリモート側の RQ は要求を消費せず，オペレーションが成功してもリモート側は気付かない．ただし例外として，データの転送とともに 32-bit の即値を送る RDMA Write with Immediate というオペレーションがある．このオペレーションではリモート側の RQ は要求を消費するため，オペレーションの成功をリモート側でも検知できる．
- RDMA Read
  リモートのメモリ領域を読み込み，そのデータをローカルのメモリ領域に書き込むオペレーションである．このオペレーションではリモート側の RQ は要求を消費せず，オペレーションが成功してもリモート側は気付かない．
- Atomic
  リモートの 64-bit のメモリ領域を対象としてアトミック操作を行うオペレーションである．Compare and Swap オペレーションと Fetch and Add オペレーションがサポートされている．このオペレーションではリモート側の RQ は要求を消費せず，オペレーションが成功してもリモート側は気付かない．

トランスポート層のサービス

InfiniBand におけるトランスポート層のサービスには，Reliable/Unreliable，Connection/Datagram の区分によって，Reliable Connection(RC)，Reliable Datagram (RD)，Unreliable Connection (UC)，Unreliable Datagram (UD) といったクラスが存在する．このうち通常は RC と UD が使用される．Reliable なサービスはメッセージの破損検知と確認応答，再送の仕組みによってメッセージの順序性と到達性を保証する．一方，Unreliable なサービスにはそうしたメッセージの順序性や到達性の保証がないが，メッセージの破損検知の仕組みはありメッセージが破損していれば破棄する．また，Connection のサービスは 2 つの QP 間で 1 対 1 での通信を行うもので，Datagram のサービスは多対多での通信が可能である．TCP/IP のトランスポート層における TCP が RC に相当し，UDP が UD に相当する．RC での通信では前述したすべてのオペレーションを使用可能であるが，UD での通信では Send/Receive オペレーションのみを使用できる．

以上で述べた RDMA (InfiniBand) の通信アーキテクチャと従来の TCP/IP/Ethernet による通信のアーキテクチャの比較を改めて表 3.1 にまとめる．このように，RDMA (InfiniBand)

は従来の TCP/IP/Ethernet によるネットワークとは異なる前提に立った通信アーキテクチャである．RDMA は，ロスレスで低遅延であることを前提としたリンク環境でプロトコル処理をハードウェア (HCA) にオフロードし，カーネルバイパスかつゼロコピーなデータ転送を行う．これによって通信にかかる CPU リソースの消費を最小限に抑え，低レイテンシで高いスループットの通信を実現している．

表 3.1. RDMA (InfiniBand) と TCP/IP/Ethernet の通信アーキテクチャの比較

	RDMA (InfiniBand)	TCP/IP/Ethernet
リンク環境	ロスレスで低遅延	比較的遅延が大きい
プロトコル処理	ハードウェア (HCA)	カーネル内スタック
ノード内のデータパス	カーネルバイパス & ゼロコピー	カーネル経由
通信のエンドポイント	Queue Pair (QP)	ソケット
ノード上における エンドポイントの識別	QPN (24-bit)	ポート番号 (16-bit)
送受信データの格納先	登録したメモリ領域	カーネル内バッファ
オペレーション	Send/Receive, RDMA Write, RDMA Read, Atomic	Send/Receive
トランスポート層	RC/RD/UC/UD	TCP/UDP

3.2.2 データセンターネットワークにおける RDMA の利用

高性能コンピューティングの分野においてノード間のインターコネクトに広く用いられる RDMA は，その低レイテンシ性と通信効率の高さからデータセンター内のノード間通信にも用いられる．たとえば，データセンターにおいてはバックエンドで Key-Value 型ストレージや分散型トランザクション処理といったシステムが動作する．これらのシステムにおけるデータ転送部分を RDMA を用いて最適化する研究が行われている [12, 16, 22, 28]．こうした研究において RDMA は特定用途の通信を最適化する目的で用いられる．この点は高性能コンピューティングの分野における RDMA の利用と同様である．

RDMA の利用は InfiniBand に対応したハードウェア (HCA) やスイッチといった設備を導入することで可能であった．RDMA が前提とするロスレスで低遅延なネットワークは InfiniBand によって実現されるもので，これは従来の Ethernet では提供されていなかった．一方で，従来のデータセンターにおけるネットワークは Ethernet の設備が主であり TCP/IP による通信が行われている．RDMA を利用する目的で InfiniBand の設備を新たに導入するのは高いコストであり，このコストが導入障壁のひとつとなっていた．

近年，ネットワークの高速化にともないデータセンターネットワークには高い信頼性と性能を満たすことが求められるようになった．これらの要求を満たすために Data Center

Bridgeing (DCB) のような Ethernet を拡張した規格の標準化が進んでいる [9]．この拡張によって Ethernet においてもロスレスで低遅延なリンク環境を提供することが可能となり，この Ethernet 上で RDMA を用いる RDMA over Converged Ethernet (RoCE) [19] という技術があらわれた．RoCE では InfiniBand の物理層，リンク層の代わりに Ethernet を用いる．さらにネットワーク層に UDP/IP を用いて IP でのルーティングを可能にした RoCE v2 もあらわれた．こうした RoCE の技術を用いることでデータセンターネットワークにおける RDMA の導入および利用は以前より容易となった．

RoCE があらわれたことでデータセンターネットワークにおける RDMA の利用が広まりつつある．実環境のデータセンターネットワークにおいていくつかのサービスを RoCE v2 によって運用した研究がある [17]．この研究では実運用にあたり課題となった点を取り上げており，データセンター内通信の TCP を RDMA で置き換えることの可能性について示唆している．また，ソフトウェアによるサポートもしくは RDMA のソフトウェア実装によって RDMA を柔軟に用いるための研究も行われている [23, 29]．このようにデータセンターネットワークにおいてより広い用途の通信に RDMA を用いる研究が行われている．

3.2.3　OS におけるネットワーク I/O としての RDMA

RDMA での通信は低レイテンシで高いスループットを実現し，データの転送にかかる CPU の関与は最小限に抑えられる．この高いパフォーマンスは，カーネルバイパスでゼロコピーなデータパスであることと，InfiniBand のプロトコル処理がハードウェアにオフロードされていることによって達成される．RDMA はオペレーティングシステムにおいて通信効率面で理想的なネットワーク I/O であると言える．

一方で，RDMA は TCP/IP/Ethernet とは通信アーキテクチャが異なり，それを扱うための API も異なる．ソケットではなく QP による通信であり，あらかじめ Memory Region の登録を行わなくてはならない．そのため，RDMA を用いて通信を行うアプリケーションは，送信側と受信側の双方において RDMA のための API を用いて実装されなければならない．したがって，従来のソケットを用いる一般的な End-to-End 通信にそのまま RDMA を用いることはできない．

3.3　既存技術のまとめ

3.1 節と 3.2 節でそれぞれ述べた高速パケット I/O 処理技術と RDMA について，従来のカーネル内のネットワークスタックを用いた通信とも比較しながら，これら既存の技術の特性と課題を整理する．また，本研究で提案する手法により達成されるべき要求事項を示す．

観点 1 **ネットワークスループット性能**

従来のカーネル内のネットワークスタックを用いた通信はオーバーヘッドが大きく，高速化するネットワークに対して十分なスループット性能を達成できない．それに対し，高速パケット I/O 処理技術や RDMA では十分に高いスループット性能を達成できる．

提案手法においても高いスループット性能を達成できることが求められる．

観点 2 アプリケーション以外での CPU リソース消費

従来のカーネル内のネットワークスタックを用いた通信では，カーネルを経由することによるコンテキストスイッチやメモリコピー，そしてネットワークスタックでの処理自体で CPU リソースを消費してしまう．それに対し，高速パケット I/O 処理技術ではカーネルバイパスによってカーネルのオーバーヘッドが回避される．RDMA では通信にかかる CPU の関与は最小限に抑えられる．提案手法においても CPU リソースの消費を小さく抑えることが求められる．

観点 3 従来の通信インターフェースとの互換性

従来のアプリケーションは Socket API を用いて通信を行う．高速パケット I/O 処理技術では独自の API を提供しているため互換性がなく，再実装する場合には大きな開発コストとなる．RDMA は通信のアーキテクチャ自体が異なるため，通信を行うノードの双方で RDMA を用いるためのプログラムが動作していなければならない．提案手法においては大きな変更を加えること無く従来のアプリケーションを利用できるようにするため，Socket API と互換性のあるインターフェースを提供することが求められる．

観点 4 カーネルのネットワークスタック機能

従来のカーネル内のネットワークスタックを用いた通信では，機能は当然サポートされている．それに対し，高速パケット I/O 処理技術ではカーネルをバイパスするため機能を利用できず必要な処理を実装しなくてはならない．RDMA でも同様である．提案手法においてはカーネルによって提供される機能を既存の資産として利用できるようにすることが求められる．

以上で述べた既存技術の特性の比較および提案手法への要求事項を表 3.2 にまとめる．

表 3.2. 既存技術の比較および提案手法への要求

	観点 1	観点 2	観点 3	観点 4
従来の OS カーネル	遅	カーネルのオーバーヘッドとネットワークスタック処理	✓	✓
高速パケット I/O 処理技術	最適	ネットワークスタック処理		
RDMA	最適	最小限		
提案手法	速	最小限	✓	✓

本節での議論を踏まえ，本研究では，既存のアプリケーションやプロトコルスタックといった資産を利用できるようにしつつ，通信にかかるサーバーの CPU リソース消費を抑えた高速な通信を実現することを目的とする．

第 4 章

RoPT: RDMA for Proxying TCP connections

本章では，既存のアプリケーションやプロトコルスタックといった資産を利用できるようにしつつ，通信にかかるサーバーの CPU リソース消費を抑えた高速な通信を実現するためのアーキテクチャを提案する．まず本研究で想定する環境について述べ，次にアーキテクチャが満たすべき要件をまとめる．そして手法の概要とその設計について述べる．

4.1 想定環境

本研究では，通信にかかるサーバーの CPU リソース消費を抑えた高速な通信を実現することを目的としている．そのため，本研究で提案する手法を適用する環境として，インターネットに接続されたクライアントとの間で TCP/IP 通信を行うデータセンター内のアプリケーションサーバーを想定する．こうしたアプリケーションサーバーにおいては CPU-intensive なアプリケーションが動作し，通信にかかる CPU リソース消費を抑えた場合その効果が大きいと考えられる．また，ネットワークインターフェースの速度としては 10 Gbps あるいはそれ以上の速度を想定する．これは，現在データセンター内のエンドホストにおいて 10 GbE のインターフェースの普及が進んでおり，さらに 25 GbE や 40 GbE といったより高速なインターフェースが導入されていくことが想定されるためである．

4.2 要件

3.3 節において，高速化するネットワークに対して既存技術における特性と課題をまとめた．本節ではその議論を踏まえ，本研究において提案するアーキテクチャが満たすべき要件を以下に整理する．

1. CPU リソースの消費を抑えた通信
   通信にかかるサーバーの CPU リソースの消費を小さく抑えることが必要である．従来の OS カーネル内のネットワークスタックによる処理はオーバーヘッドが大きく，ネッ

トワーク I/O が高速化するほどその処理で多くの CPU リソースを消費してしまう．サーバー上で動作する CPU-intensive なアプリケーションに対して計算リソースをより割けるようにするため，データの送受信にかかる CPU リソースの消費を抑えるような設計を行わなくてはならない．効率的な通信が可能であればそれだけ高速な通信を実現することも可能となる．

2. **既存のアプリケーション資産の利用**
   サーバー上で動作する従来のさまざまなアプリケーションにおいてアーキテクチャが利用可能である必要がある．同様にカーネルによって提供されるネットワークスタック内の機能を利用できるようにする必要がある．サーバー上では，たとえば静的あるいは動的なコンテンツを配信する Web サーバーや Web アプリケーションといった，ネットワークを利用する各種アプリケーションが動作している．高速化するネットワーク I/O を十分に活用できるように高速パケット I/O 処理技術を用いようとする場合，その独自 API に対応しなくてはならない．この場合アプリケーション側に大きな変更が必要となり，開発上の大きなコストとなる．また，これらの技術はカーネルをバイパスするため従来カーネルによって提供される機能を利用することができない．したがって，既存のアプリケーションに大きな変更を加えること無くネットワークの機能を利用できるように設計を行わなくてはならない．

4.3 手法の概要

本研究では，4.2 節で示した要件を満たすアプローチとして，RDMA for Proxying TCP connections (RoPT) を提案する．RoPT は，RDMA を用いて TCP/IP ネットワークスタックの処理を分離する Proxy 型のアーキテクチャである．このアーキテクチャの概観を図 4.1 に示す．

従来用いられるネットワークスタックはカーネル内に実装されており，サーバー（アプリケーションノード）上ではアプリケーションでの処理とネットワークスタックでの処理がそれぞれ行われる．ネットワークの高速化が進み処理すべきトラフィック量が増大すると，ネットワークスタックでの処理により多くの CPU リソースが消費されてしまう．そこで本研究では，従来はアプリケーションノード上で行なっていたネットワークスタック処理を外部のノード（ネットワークスタックノード）に分離する．アプリケーションノードとネットワークスタックノードとの間のデータ転送を RDMA で行うことによって，アプリケーションノードにおけるネットワーク処理にかかる CPU リソース消費を抑える．アプリケーションノードとインターネット上のクライアントとの間の TCP コネクションはネットワークスタックノードによって Proxy される．

また，アプリケーションノードにおいて従来のアプリケーションに大きな変更を加えること無く RoPT を用いた通信を可能にするため，標準の Socket API システムコール関数をプログラム実行時に差し替える．従来のカーネル内ネットワークスタックを用いるプログラムは

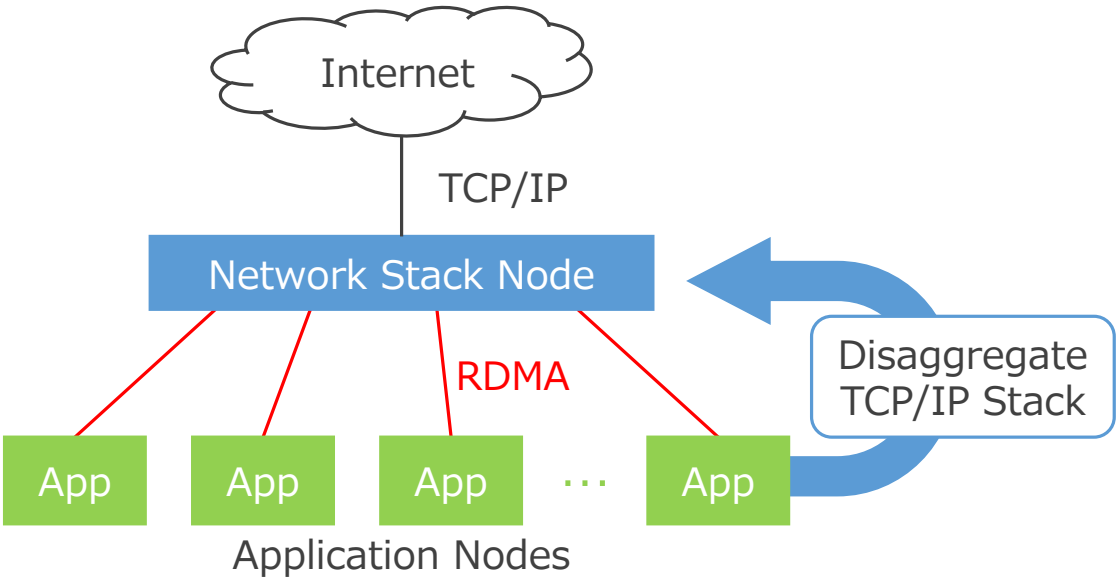

図 4.1. RoPT のアーキテクチャ概観

Socket API を用いてネットワーク機能を利用しており，大きな変更を加えること無く利用できるためには Socket API 互換なインターフェースを提供する必要がある．プログラム実行時に Socket API システムコール関数を差し替えることによってアプリケーション側の変更が不要となる．

RoPT は，アプリケーションノード上で動作する RoPT Client と，ネットワークスタックノード上で動作する RoPT Manager によって構成される．RoPT Client はサーバーアプリケーションの実行時に Socket API をハイジャックし，RoPT Manager との間で RDMA 通信を確立する．RoPT Manager は接続先であるクライアントとの間に TCP ソケット通信を確立し，対応する RDMA 通信と紐付けてデータの転送を行う．以上のようにして RoPT は，サーバーアプリケーションに対してはクライアントとの間で従来のとおり TCP ソケット通信を行なっているかのように見せつつ，実質的にはネットワークスタックノードで Proxy された TCP コネクションを扱わせる．

4.4　RDMA を用いた TCP/IP ネットワークスタックの分離

RoPT では，アプリケーションノードで行なっていた TCP/IP ネットワークスタックの処理をネットワークスタックノードに分離する．従来のカーネル内ネットワークスタックによる処理はオーバーヘッドが大きく CPU リソースを消費してしまうという課題がある一方，そのカーネル内ネットワークスタックにより提供される機能を利用できることが求められる．RoPT では，これを解決するためネットワークスタックの処理と機能を外部のネットワークスタックノードに分離するアーキテクチャとする．この分離によってアプリケーションノードとネットワークスタックノード間でデータ転送が必要となるが，この転送に従来の TCP/IP 通

信の代わりに RDMA によるデータ転送を用いることによってアプリケーションノードにおけるデータの送受信にかかる CPU リソースの消費を抑える．

ネットワークスタックを分離したアーキテクチャである RoPT を用いた通信と従来の TCP/IP 通信のデータパスの比較を図 4.2 に示す．従来の TCP/IP 通信では，サーバーアプリケーションから送信されるデータはアプリケーションノードの TCP/IP スタックで処理され，クライアントへと送信される．一方，RoPT を用いた通信では，アプリケーションから送信されるデータは RoPT Client によって RDMA でネットワークスタックノード上の RoPT Manager に転送される．この部分ではカーネルをバイパスしており，ユーザー空間から直接データ転送を行う．さらに，RoPT Manager に転送されたデータは対応する TCP ソケットに渡され，ネットワークスタックノードの TCP/IP スタックで処理されてクライアントへと送信される．

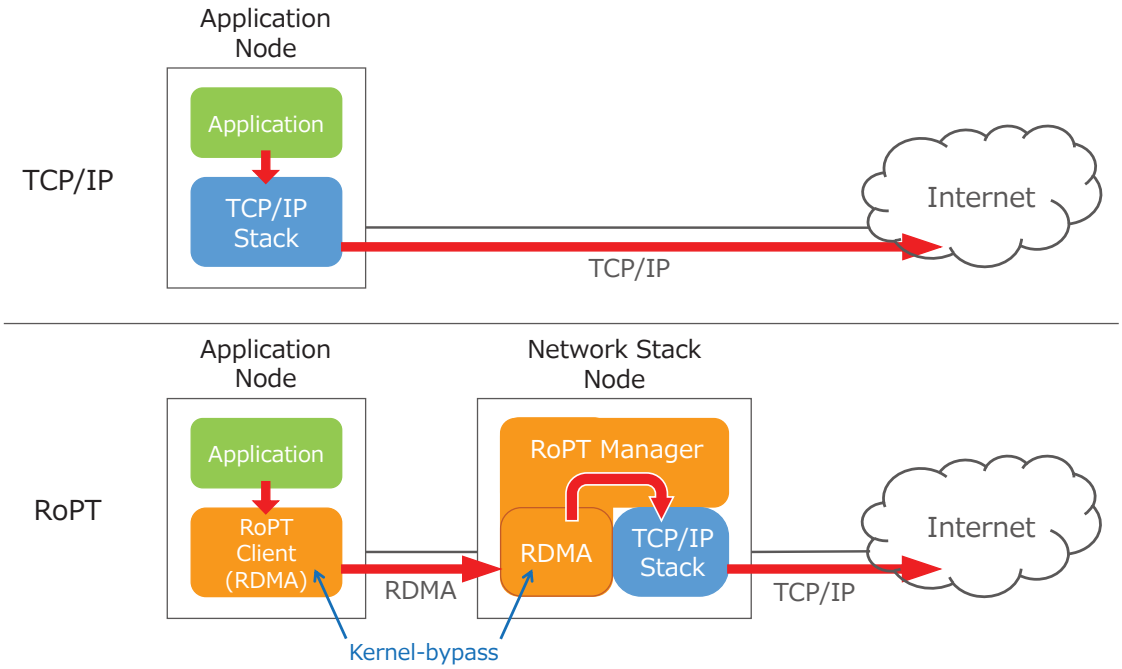

図 4.2. 従来の TCP/IP 通信と RoPT を用いた通信のデータパスの比較

また，RoPT ではネットワークスタックの処理を分離しているため，アプリケーションノードとネットワークスタックノードとの間で接続先すなわちクライアントの IP アドレスやポート番号といったコネクション情報を共有する必要がある．RoPT を用いた通信において，サーバーアプリケーションが確立しようとするクライアントとの TCP コネクションは Proxy されていて，ネットワークスタックノードとクライアントとの間で実際の TCP コネクションが確立される．このコネクションを確立するためにコネクション情報の共有が必要である．

4.5　コネクションの確立と通信フロー

本節では RoPT を用いてアプリケーションノード上で動作するサーバーアプリケーションが外部のクライアントとコネクションを確立し通信を行う手順について述べる．アプリケーションノードで行なっていた TCP/IP スタックの処理をネットワークスタックノードに分離するにあたり，サーバーアプリケーションとクライアントとの間の通信を成立させるためにはコネクション情報と送信されるデータ本体があれば必要十分である．前者のコネクション情報についてはサーバーアプリケーションがコネクションを確立するタイミングで共有し，その後は確立されたコネクションの上で通信を行うこととする．したがって確立されたコネクションの上で通信を行なっている間は後者のデータ本体の転送だけを考えればよい．RDMA でのデータ転送によってデータ本体を送り，コネクション情報は別の経路で受け渡す．RoPT ではこのコネクション情報の受け渡しに TCP/IP を用いる．

本節では，まずアプリケーションノード上のサーバーアプリケーションが RoPT を用いて外部へコネクションを確立する場合の手順について述べる．次に，それとは逆にサーバーアプリケーションが RoPT を用いて外部すなわちクライアントからの接続を待ち受け，コネクションを確立する場合の手順について述べる．最後に，いずれかの手順を経てサーバーアプリケーションが外部とのコネクションを確立した後，RoPT を用いて通信を行うときの通信フローについて述べる．

4.5.1　サーバーアプリケーションからのコネクション確立

まずアプリケーションノード上のサーバーアプリケーションが RoPT を用いて外部へコネクションを確立する場合の手順を述べる．このときのコネクション確立手順を図 4.3 に示す．

1. RoPT Client からの接続を待ち受け
   はじめは RoPT Manager が RoPT Client からの接続を待ち受ける状態にある．ここで待ち受けているのは，コネクション情報の受け取りに用いる TCP/IP ソケット通信と，データの転送に用いる RDMA 通信である．
2. アプリケーションによる connect 関数の呼び出し
   サーバーアプリケーションがクライアントに対して TCP/IP のコネクションを確立しようと connect 関数を呼び出す．RoPT Client はこの関数をハイジャックし，接続先であるクライアントに関するコネクション情報を取得する．
3. コネクション情報の送信
   RoPT Client が RoPT Manager に対して TCP/IP のコネクションを確立し，アプリケーションから取得したコネクション情報を RoPT Manager に送信し共有する．
4. 接続先へのコネクションを確立
   RoPT Manager は RoPT Client から受け取ったコネクション情報を用いて，接続先のクライアントに対して TCP/IP 通信のコネクションを確立する．ここで確立した

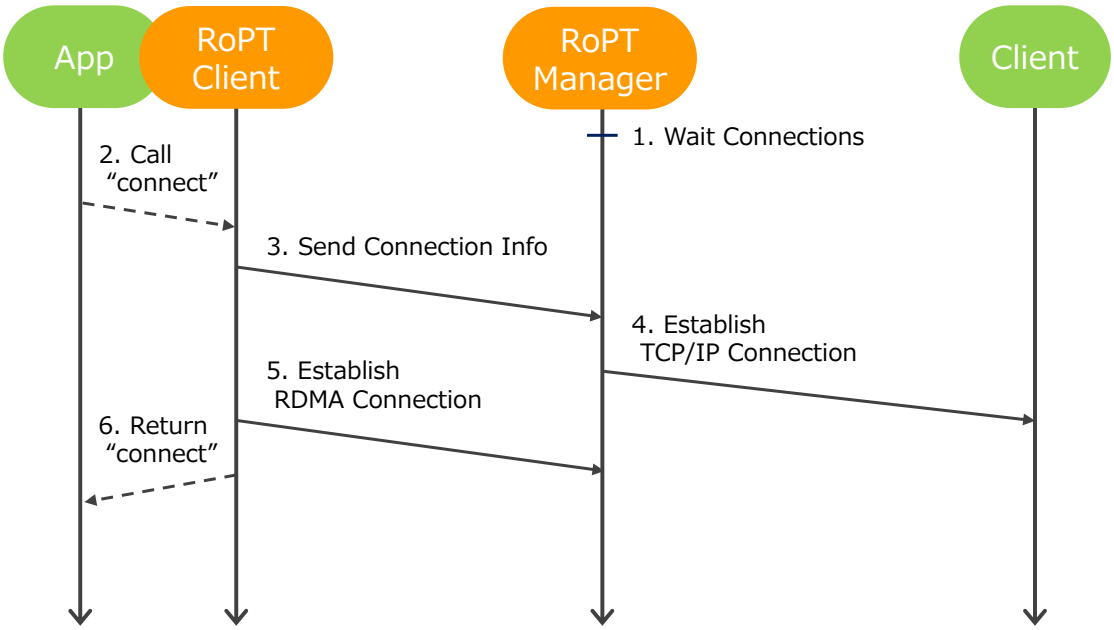

図 4.3. サーバーアプリケーションからのコネクション確立手順

TCP/IP 通信のコネクションは，次で確立する RDMA 通信のコネクションと紐付けられる．

5. RDMA 通信のコネクションを確立
   コネクション情報を送信した RoPT Client は次に RoPT Manager に対して RDMA 通信のコネクションを確立する．通信中のデータ転送はこの RDMA 通信のコネクションを用いて行われる．
6. コネクション確立完了
   以上で接続先のクライアントまでのコネクション確立が完了したとみなし，RoPT Client はサーバーアプリケーションに対して connect 関数を返す．

4.5.2　クライアント側からのコネクション確立

逆に，サーバーアプリケーションが RoPT を用いて外部すなわちクライアントからの接続を待ち受け，コネクションを確立する場合の手順を述べる．このときのコネクション確立手順を図 4.4 に示す．

1. RoPT Client からの接続を待ち受け
   はじめは RoPT Manager が RoPT Client からの接続を待ち受ける状態にある．ここで待ち受けているのは，RoPT Manager がクライアントからの接続を待ち受けるのに必要な情報の受け取りに用いる TCP/IP ソケット通信である．
2. アプリケーションによる bind 関数の呼び出し

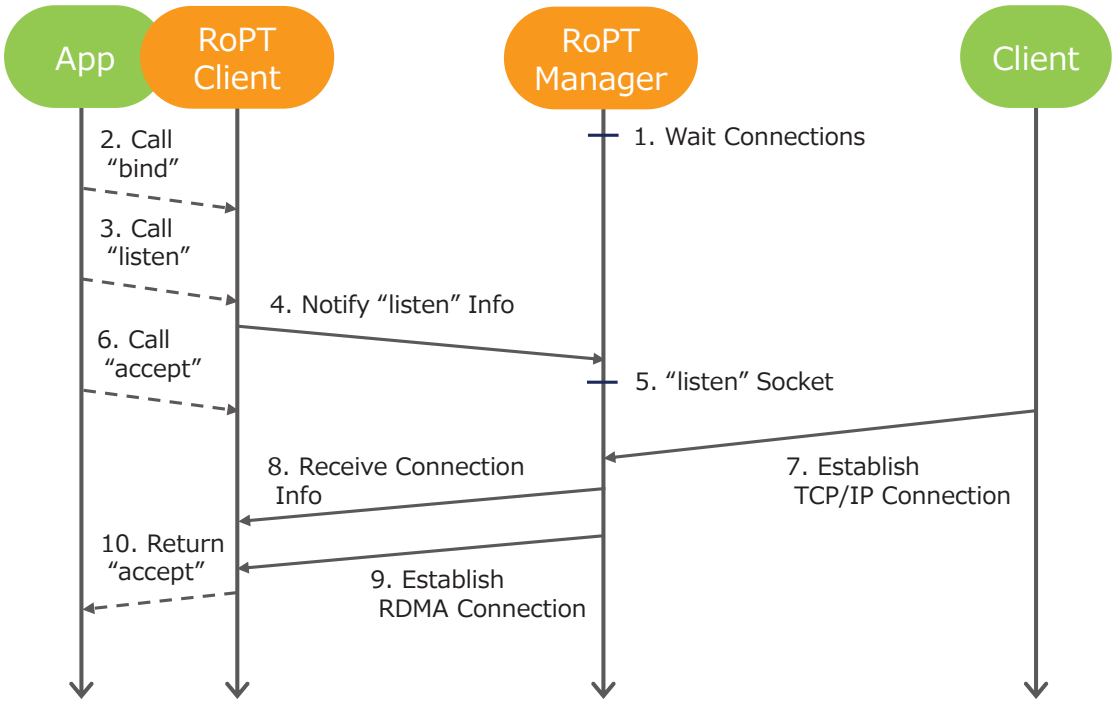

図 4.4. クライアント側からのコネクション確立手順

サーバーアプリケーションが TCP/IP の通信を待ち受けるために bind 関数を呼び出してアドレスやポート番号を指定する．RoPT Client はこの関数をハイジャックし，クライアントからの接続の待ち受けに必要な情報を取得して保管する．

3. **アプリケーションによる listen 関数の呼び出し**
   サーバーアプリケーションが通信の待ち受けを開始するために listen 関数を呼び出す．RoPT Client はこの関数をハイジャックする．
4. **待ち受けに必要な情報の送信**
   RoPT Client は RoPT Manager に対して TCP/IP のコネクションを確立し，bind 関数の処理時に取得していた待ち受けに必要な情報を RoPT Manager に送信する．同時に，RoPT Client は RoPT Manager からの接続の待ち受けを開始する．ここで RoPT Client が待ち受けるのは，コネクション情報の受け取りに用いる TCP/IP ソケット通信と，データの転送に用いる RDMA 通信である．
5. **クライアントからの接続の待ち受け**
   RoPT Manager は RoPT Client から受け取った待ち受けに必要な情報を用いて，クライアントからの接続を待ち受ける．
6. **アプリケーションによる accept 関数の呼び出し**
   サーバーアプリケーションがクライアントからの接続の受け入れを行うために accept 関数を呼び出す．RoPT Client はこの関数をハイジャックし，RoPT Manager からの接続を待つ．

7. クライアントとのコネクション確立
   RoPT Manager がクライアントからの接続を accept して TCP/IP 通信のコネクションを確立する．ここで確立した TCP/IP 通信のコネクションは，後で確立する RDMA 通信のコネクションと紐付けられる．
8. コネクション情報の送信
   RoPT Manager が RoPT Client に対して TCP/IP のコネクションを確立し，接続してきたクライアントの情報をコネクション情報として RoPT Client に送信し共有する．
9. RDMA 通信のコネクションを確立
   コネクション情報を送信した RoPT Manager は次に RoPT Client に対して RDMA 通信のコネクションを確立する．通信中のデータ転送はこの RDMA 通信のコネクションを用いて行われる．
10. コネクション確立完了
    以上でクライアントから RoPT Client までのコネクション確立が完了したとみなし，RoPT Client は受信したコネクション情報を accept 関数の呼び出し時に指定された先に格納し，サーバーアプリケーションに対して accept 関数を返す．

4.5.3　コネクション確立後の通信フロー

最後に，サーバーアプリケーションが RoPT を用いて外部とのコネクションを確立した後の通信フローについて述べる．この通信フローを図 4.5 に示す．

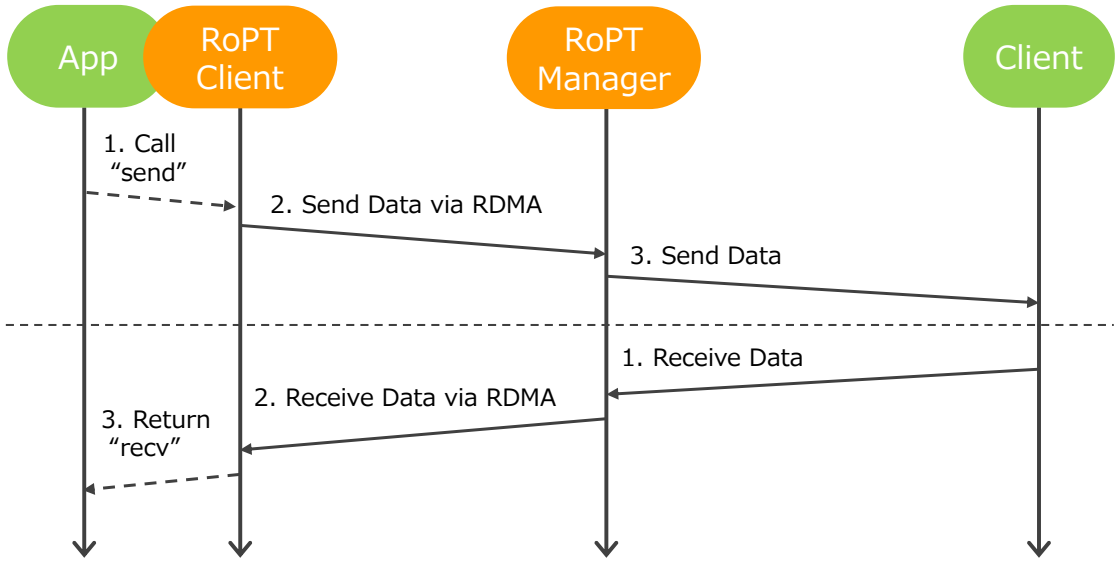

図 4.5. コネクション確立後の通信フロー

サーバーアプリケーションからクライアントへのデータ送信

以下のようにしてサーバーアプリケーションからクライアントへのデータ送信を行う．

1. アプリケーションによる send 関数の呼び出し
   サーバーアプリケーションがクライアントに対してデータ送信を行うためにバッファにデータを格納し send 関数を呼び出す．RoPT Client はこの関数をハイジャックし，送信されるデータの格納先を取得する．
2. RDMA を用いたデータ転送
   RoPT Client は，既に確立された RDMA 通信のコネクションを用いて RoPT Manager へデータを転送する．
3. RoPT Manager によるデータ送信
   RoPT Manager は，RoPT Client から転送されたデータを，その RDMA 通信のコネクションと紐付けられた TCP/IP 通信のコネクションを用いてクライアントへと送信する．

クライアントからサーバーアプリケーションへのデータ送信

以下のようにしてサーバーアプリケーションはクライアントからのデータ受信を行う．

1. クライアントからのデータ受信
   RoPT Manager がクライアントから送信されたデータを受信する．
2. RDMA を用いたデータ転送
   RoPT Manager は，そのクライアントとの間の TCP/IP 通信のコネクションと紐付けられた RDMA 通信のコネクションを用いて RoPT Client へデータを転送する．
3. RoPT Client によるデータ受信
   RoPT Client は，RoPT Manager から転送されたデータを，ハイジャックした recv 関数で指定されていた先のバッファに格納し，recv 関数を返す．

第 5 章

実装

本章では，まず RoPT Client と RoPT Manager に共通する実装環境を述べる．その後，RoPT Client と RoPT Manager の実装詳細についてそれぞれ述べる．最後に，本研究で実装を行なった RoPT について，適用可能なアプリケーションに制限があるため注意点としてまとめる．

5.1 実装環境

RoPT は Linux 環境において動作するライブラリおよびアプリケーションとして，C 言語を用いて実装を行なった．表 5.1 に実装環境を示す．RoPT では，アプリケーションノードとネットワークスタックノードとの間で RDMA 通信を行う必要があるが，この部分の実装には librdmacm ライブラリ，特にその中の rsocket を利用している．rsocket は Socket API と同等のレベルで RDMA の通信を抽象化した RDMA Socket API を提供するライブラリで，データの転送を RDMA Write with Immediate によって行なう．また，本研究では RoPT Manager でコネクションの Proxy を行う TCP/IP 通信について IPv4 のみを実装している．そのためコネクション情報として，Socket API において用いられる sockaddr_in 構造体を使用する．sockaddr_in 構造体には IP アドレスやプロトコル，ポート番号といった接続に必要な情報が含まれる．

表 5.1. 実装環境

OS	GNU/Linux Ubuntu 16.04.3 LTS 64-bit
Kernel Version	4.11.12-041112-generic
Library for RDMA	librdmacm 41mlnx1-OFED.4.1.0.1.0.41102

5.2 RoPT Client の実装

RoPT Client は，アプリケーションノード上で RoPT を利用するアプリケーションに読み込まれる共有ライブラリとして実装を行う．プログラム実行時に Socket API システムコール

関数を RoPT Client ライブラリ内の関数に差し替えることによって動作する．標準の Socket API 関数を RDMA Socket API 関数に差し替える実装については，librdmacm ライブラリの中に共有ライブラリ librspreload.so として既に存在する．RoPT Client の実装はこの librspreload.so の実装をベースとし，RoPT Manager との間でコネクション情報を共有して RDMA 通信のコネクションおよびクライアントとの TCP コネクション確立を可能にするための実装を行う．

本節では，まず Socket API システムコール関数をプログラム実行時に差し替える方法について述べる．次に，RoPT Manager とコネクション情報を共有しコネクションを確立するための実装について述べる．

5.2.1　Socket API ハイジャック

既存のアプリケーションに変更を加えること無く RoPT を用いた通信を利用できるようにするため，RoPT Client は標準の Socket API システムコール関数をプログラム実行時に RoPT Client ライブラリ内の関数に差し替えるという形で動作する．Linux 環境では LD_PRELOAD という環境変数に共有ライブラリを指定すると，プログラム実行時に指定されたライブラリがその他のライブラリに先立って読み込まれる．これは動的リンカー/ローダーの仕組みとして提供されている．また，この動的リンカー/ローダーはプログラム実行時，あるシンボル名に対して最初に見つかったものを利用する．そのため，LD_PRELOAD を設定することによって関数を選択的に差し替えることが可能となる．RoPT Client はこの仕組みを利用して標準の Socket API システムコール関数をプログラム実行時に差し替える．したがって，アプリケーションノード上で RoPT による通信を利用する場合には，LD_PRELOAD 環境変数に RoPT Client ライブラリを指定してアプリケーションを実行すればよい．

また，RoPT はコネクション情報の共有のために通常の TCP/IP 通信を用いる設計となっている．それに加え，RoPT が Proxy の対象とする TCP/IP とは異なる通信を意図したソケットやファイルディスクリプタの操作を行う場合，本来の Socket API システムコール関数での処理が必要となる．したがって，差し替える前の本来の関数をプログラム内で呼び出せるようにする必要がある．そのため，ライブラリのプリロード時に dlsym 関数で RTLD_NEXT を指定し本来の関数へのポインタを取得した．

5.2.2　コネクション情報の共有

RoPT Client では，RoPT Manager とコネクション情報を共有しコネクションを確立するための実装を行う．4.5 節で述べたコネクションの確立手順にしたがい，差し替える Socket API の関数のうち bind，listen，accept，connect についてこのための処理を加える．

bind

bind 関数では，サーバーアプリケーションから指定された通信の待ち受けに必要な情報を取得して保管する．bind 関数には引数として sockaddr 構造体へのポインタが渡されており，その中身を通信の待ち受けに必要な情報として取得する．取得した情報を呼び出したソケットと対応付けて保管する．

listen

listen 関数では，bind 関数で取得した情報を RoPT Manager に対して送信し，接続の待ち受けを開始することを通知する．listen 関数が呼ばれると，まず待ち受けに必要な情報を RoPT Manager に渡すための TCP/IP ソケットを開き，このソケットでコネクションを確立する．bind 関数で取得して保管していた情報をそのまま send で RoPT Manager へ送信する．このようにして待ち受けに必要な情報を共有する．その後，rlisten により RoPT Manager からの RDMA 通信の待ち受けを開始する．

accept

accept 関数では，RoPT Manager が接続を受け付けたクライアントについてのコネクション情報を RoPT Manager から受信する．accept 関数が呼ばれると，まずコネクション情報を受け取るための RoPT Manager からの TCP/IP 接続を待ち受け，accept した後受信する．ここで受信されるのは RoPT Manager に対して接続してきたクライアントの情報であり，sockaddr_in 構造体の形式で渡される．その後，RDMA 通信のためのコネクションを raccept により受け入れ，コネクションを確立する．RoPT Manager から受信したコネクション情報を accept 関数の引数として渡されていた sockaddr 構造体へのポインタの先に格納して関数をサーバーアプリケーションに返す．

connect

connect 関数ではサーバーアプリケーションから渡されたコネクション情報を RoPT Manager に対して送信し，共有する．connect 関数が呼ばれると，まずコネクション情報を RoPT Manager に渡すための TCP/IP ソケットを開き，このソケットでコネクションを確立する．connect 関数には引数として sockaddr 構造体へのポインタが渡されており，その中身をコネクション情報としてそのまま send で RoPT Manager へ送信する．このようにしてコネクション情報を共有する．その後，RDMA 通信のためのコネクションを RDMA Socket API の rconnect により確立する．

5.3 RoPT Manager の実装

RoPT Manager は，ネットワークスタックノード上でユーザー空間において動作するアプリケーションとして実装を行う．RoPT Manager は，RoPT Client との間の RDMA 通信

のコネクションとクライアントとの間の TCP/IP 通信のコネクションとを紐付ける機能を持つ．この紐付けられたコネクションを確立するための実装については，4.5 節で述べたコネクションの確立手順にしたがう．本節では，まず RoPT Manager においてコネクションの管理および多重化を行う実装について述べる．次に，紐付けられたコネクション間でデータ転送を行うための実装について述べる．

5.3.1　コネクションの管理および多重化

RoPT Manager は RDMA 通信のコネクションとそれに対応する TCP/IP ソケット通信のコネクションを紐付ける役割を持つ．同時に，紐付けられたコネクションの組をクライアントとサーバーアプリケーションとの間の 1 つのコネクションと見なすとき，そのコネクションは多重化される必要がある．

RoPT Manager の実装ではこのコネクションの管理および多重化を実現するためにスレッドを用いる．1 つのスレッドの中で 1 つの RDMA 通信のエンドポイント，本実装では rsocket，とそれに対応するコネクションの TCP/IP ソケットを扱うことによって，コネクションの紐付けを実現する．また，マルチスレッドによりコネクションの多重化を実現する．このような実装とすることによって，ネットワークスタックノードにおいてマルチコアの環境を生かすことが可能となる．

一般にネットワークプログラムにおいてコネクションの多重化を実現するには，マルチスレッド以外にも I/O の多重化や fork によるマルチプロセスといった方法が存在する．しかし，select や poll といったシステムコールによる I/O の多重化を用いる方法ではマルチコアの環境を生かすことができない．また，5.4 節で後述するように，プロセス空間を複製する fork システムコールは librdmacm ライブラリによって RDMA を扱うプログラムでは用いることができないという問題がある．以上の理由から，RoPT Manager におけるコネクションの多重化の実装にはスレッドを用いている．

5.3.2　データ転送

サーバーアプリケーションとクライアントとの間でコネクションが確立された後，RoPT Manager では新規にスレッドを開始する．RoPT Manager はこのスレッド内でデータ転送を行う．スレッドには紐付けられた 2 つのコネクションのそれぞれのエンドポイント，すなわち rsocket と TCP/IP 通信のソケットが渡される．RoPT Manager で行う必要があるのは，一方のソケットから受信したデータをもう一方のソケットから送り出す単純なデータ転送である．高速なデータ転送を可能にするため，送受信ともにノンブロッキングに設定している．また，双方向通信を可能にするため，スレッド内で自身と同等のスレッドを開始し，rsocket から受信 (rrecv) して TCP ソケットへの転送 (send) を行うスレッドと TCP ソケットから受信 (recv) して rsocket への転送 (rsend) を行うスレッドを並列に動作させた．さらに，このデータ転送で用いられるバッファのサイズを 65536 bytes と大きく確保した．これによって

TCP Segmentation Offload (TSO) や Large Receive Offload (LRO) といった TCP 通信側でのハードウェアオフロード機能を十分に利用できるようにした．

5.4 適用可能なアプリケーション

RoPT では RDMA 通信部分の実装に librdmacm ライブラリを用いており，このライブラリではプロセス空間を複製する fork システムコールが十分にサポートされていない．これは RDMA 通信を行う仕組みから生じる問題である．RDMA で通信を行う場合にはあらかじめ Memory Region を登録しなくてはならないが，ここで登録されるのは親プロセスのプロセス空間のメモリ領域であり，fork された子プロセスには継承されない．つまり，親プロセスで確立した RDMA 通信のコネクションを子プロセスに継承させることができない．そのため，fork を使用するアプリケーションについて RoPT Client ライブラリを読み込んで実行した場合，RDMA の通信が失敗してしまう．このように RoPT には適用可能なアプリケーションに制限がある．

第 6 章

評価

本章では，提案手法である RoPT の評価を行う．

6.1 評価方針

本研究で設計および実装を行なった RoPT について，4.2 節で示した要件が達成されたかどうかの評価を行う．既存のアプリケーション資産の利用という要件については，RoPT を適用可能なアプリケーションに制限があるものの，TCP/IP スタックの処理と機能をネットワークスタックノードに分離し，アプリケーションノードでは Socket API システムコールを差し替えるという設計によって実現されている．そのため本論文では，CPU リソースの消費を抑えた通信というもうひとつの要件について評価を行う．

本論文では RoPT の通信パフォーマンスについて評価を行う．RoPT では CPU リソースの消費を抑えた通信を実現するため，TCP/IP スタックの処理をネットワークスタックノードに分離しアプリケーションノードとネットワークスタックノード間のデータ転送に RDMA を用いるという設計および実装を行なっている．これによってアプリケーションノードにおいてネットワーク処理にかかる CPU 使用率を低減できているかどうか評価する必要がある．また，RoPT ではネットワークスタックノード上の RoPT Manager によって TCP コネクションが Proxy されているが，このときのスループット性能は十分であるかどうかについても評価する必要がある．本論文では，こうした通信パフォーマンス評価について従来の TCP/IP 通信と比較し，考察する．

6.2 TCP メッセージの送信性能評価

RoPT は TCP/IP の機能をネットワークスタックノードに分離するアーキテクチャである．そのため，本節では，RoPT の通信パフォーマンスとして TCP メッセージの送信性能の評価を行う．RoPT においては CPU リソースの消費を抑えた通信が行われることが要件であり，また，十分なスループットを達成できているかについても評価する必要がある．したがって，パフォーマンス評価のメトリックとしてスループットと CPU 使用率の 2 つを用い

る．これらのメトリックについて，従来の TCP/IP 通信と RoPT を用いた通信のそれぞれについて計測を行なった．

6.2.1 評価方法

実験環境

この評価実験を行う機材の環境を表 6.1 に示す．また，計測対象とする通信のモデルを図 6.1 に示す．ノード間の接続は，それぞれの 40 GbE NIC のポートをアクティブ光ケーブル (AOC) によってスイッチを挟まずに直接接続している．クライアントと接続している Ethernet リンクの MTU は 1500 bytes，アプリケーションノードとネットワークスタックノード間の RoCE (RDMA) リンクの MTU は 4096 bytes に設定する．この値は，4.1 節で述べた想定環境にしたがい，インターネット上のクライアントと接続していることを想定しており，また，アプリケーションノードとネットワークスタックノード間はデータセンター内のリンクであることを想定していることから妥当なものである．この評価実験では，従来の TCP/IP 通信と提案手法である RoPT を用いた通信のそれぞれについて計測を行う．計測に用いた図中の Sender と Receiver は，TCP/IP によるソケット通信を行うことを意図した，ユーザー空間で動作するそれぞれすべて同一のプログラムである．

表 6.1. 実験機材環境

	アプリケーションノード ネットワークスタックノード	クライアント
CPU	Intel Core i3-4160 3.60 GHz (2 cores, Hyper-Threading: off)	Intel Core i7-3770K 3.50 GHz (4 cores, Hyper-Threading: off)
Memory	32 GB (8 GB x4) DDR3 1333 MHz	
OS	GNU/Linux Ubuntu 16.04.3 LTS 64-bit	
Kernel Version	4.11.12-041112-generic	
NIC	Dell / Mellanox CX324A M9NW6 (Dual 40 GbE QSFP)	
Ethernet Controller	Mellanox Technologies MT27500 Family [ConnextX-3]	
NIC Driver	mlx4_en 4.1-1.0.2 (MLNX_OFED)	
HW Offload	TSO: on, GSO: on, LRO: off (fixed), GRO: on	
Library for RDMA	librdmacm 41mlnx1-OFED.4.1.0.1.0.41102	-

評価メトリックと計測方法

この評価実験における評価メトリックはスループットと CPU 使用率の 2 つである．各通信モデルにおける Sender から Receiver までのスループット性能と，スループット性能評価を走らせている間のアプリケーションノードにおける CPU 使用率を計測する．アプリケーションノード側を Sender としているのは，一般にサーバー側からクライアントに対して送信

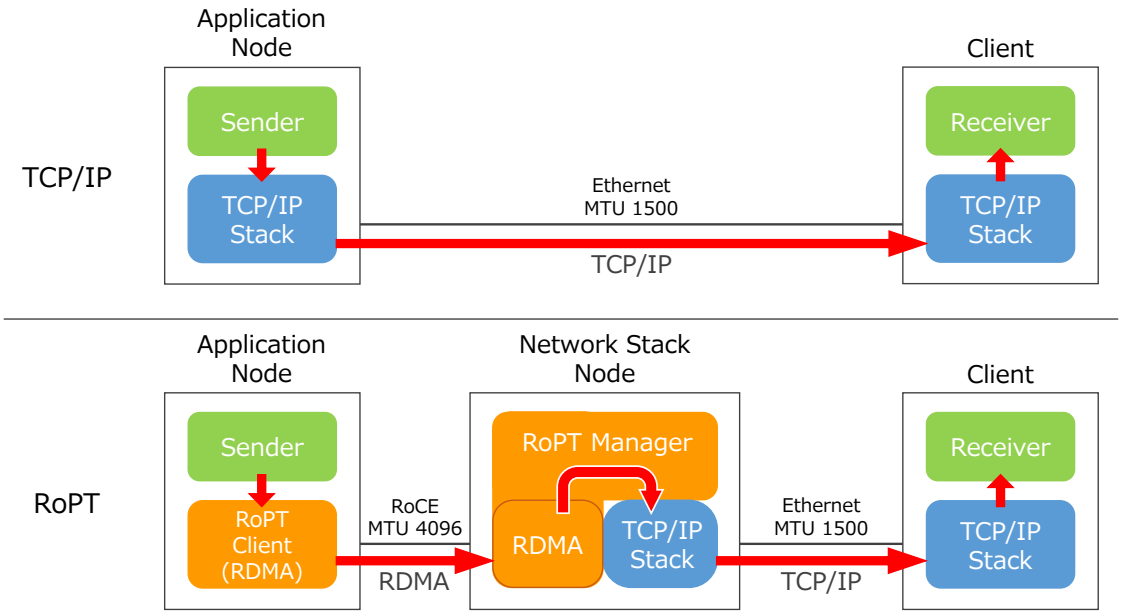

図 6.1. 計測対象とする通信のモデル

されるトラフィック量の方がその逆よりも大きいためである．

スループット性能評価については，Sender から Receiver に対して TCP/IP のテストトラフィックを送信し，単位時間あたりに転送できた TCP メッセージのビット数 (bps: bit per second) を計測した．計測のパラメータとして TCP メッセージサイズを 64，128，256，512，1024，2048，4096，8192，16384 bytes と変化させ，それぞれテストトラフィックを 100 GB 送信し終えるまで流し続けるという試行を各 N (= 10) 回繰り返して行なった．CPU 使用率の評価については，テストトラフィックを送信している間のアプリケーションノードにおける CPU 使用率，特に Sender プログラムが動作している CPU コアの使用率を mpstat によって毎秒取得した．

また，計測のための Sender と Receiver のプログラムについて，パフォーマンスチューニングとして taskset コマンドを用いてそれぞれが動作するノードの CPU コア 0 に固定した．1 ストリームでの計測のため，RoPT Manager についても同様に CPU コアを固定した．これによって，プロセスが他の CPU コアに移動するコンテキスト移動とそれにともなうオーバーヘッドやパフォーマンスのゆらぎを無くし，計測結果のばらつきを小さくする．また，ソケットオプション SO_SNDBUF と SO_RCVBUF の値を 4M (= 4194304) に設定し，十分なソケットバッファサイズを確保した．

6.2.2　スループット性能の評価結果と考察

スループット性能について計測した結果を図 6.2 に示す．

従来の TCP/IP 通信と RoPT を用いた通信の双方で TCP メッセージサイズが 64 bytes

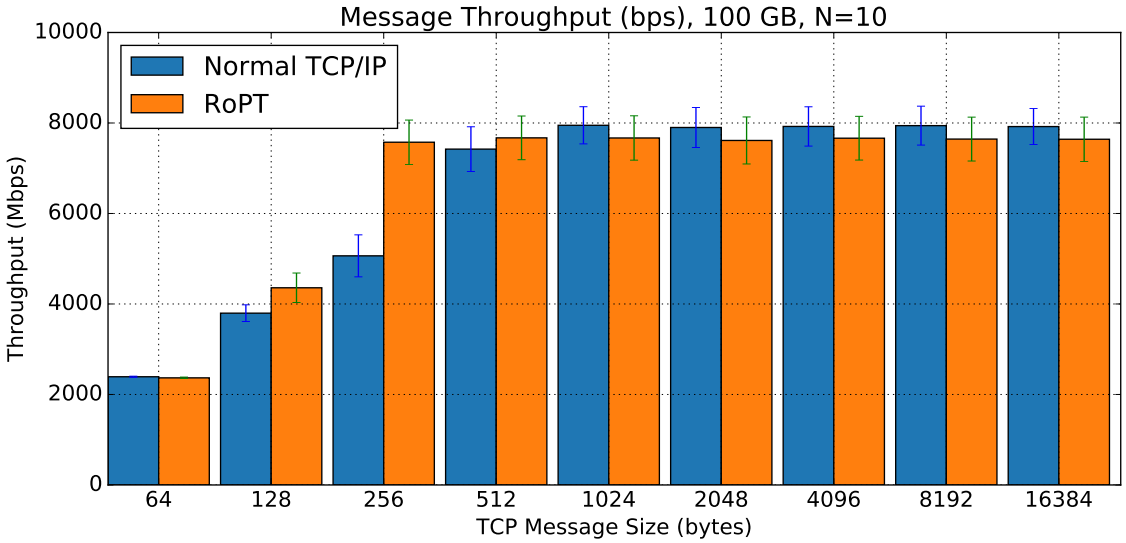

図 6.2. スループット性能の比較

から大きくなるほど次第にスループットも大きくなっているが，従来の TCP/IP 通信ではメッセージサイズが 1024 bytes 以上のとき約 7.9 Gbps でほぼ一定となり，RoPT を用いた通信では 256 bytes 以上のとき約 7.6 Gbps でほぼ一定となっている．RoPT を用いた通信のモデルを考えたとき，スループット性能に対するボトルネックとなるのはネットワークスタックノード上の RoPT Manager によるデータ転送である．RoPT Manager はユーザー空間のアプリケーションとして動作しており，従来の TCP/IP 通信のモデルにおける Sender との差分は同一コア上で RDMA でのデータ受信処理を行なっていることである．この処理がオーバーヘッドとして約 0.3 Gbps のスループット上限の差にあらわれていると考えられる．このように RDMA でのデータ受信処理を同時に行いながらも約 0.3 Gbps の差に収まっており，RDMA での通信はたしかにオーバーヘッドが小さいことが分かる．

また，TCP メッセージサイズが 128，256，512 bytes のときは RoPT を用いた通信の方が従来の TCP/IP 通信のスループットを上回っている．特に 256 bytes のときのスループットは従来の TCP/IP 通信が 5.06 Gbps であるのに対し RoPT を用いた通信では 7.57 Gbps と大きく上回っている．このようなスループットの差があらわれた要因として，RoPT Manager でのデータ転送におけるバッファの効果が考えられる．実装に用いた librdmacm の rsocket による RDMA での受信 (rrecv) では rsocket 内部でバッファリングしているため，アプリケーションノードから受信されたメッセージはある程度まとまって RoPT Manager に渡される．rsocket 内部でバッファリングするため，NIC からはゼロコピーで RoPT Manager に渡されない構造になっている．しかし，このバッファによりまとまって渡されたメッセージを大きな TCP メッセージとしてクライアントに対して送信できる．そのため結果として全体で見たスループットが向上したと考えられる．

6.2.3　CPU 使用率の評価結果と考察

スループット性能評価を走らせている間のアプリケーションノードにおける CPU 使用率を計測した結果について，従来の TCP/IP 通信時の結果を図 6.3 に，RoPT を用いた通信時の結果を図 6.4 に示す．

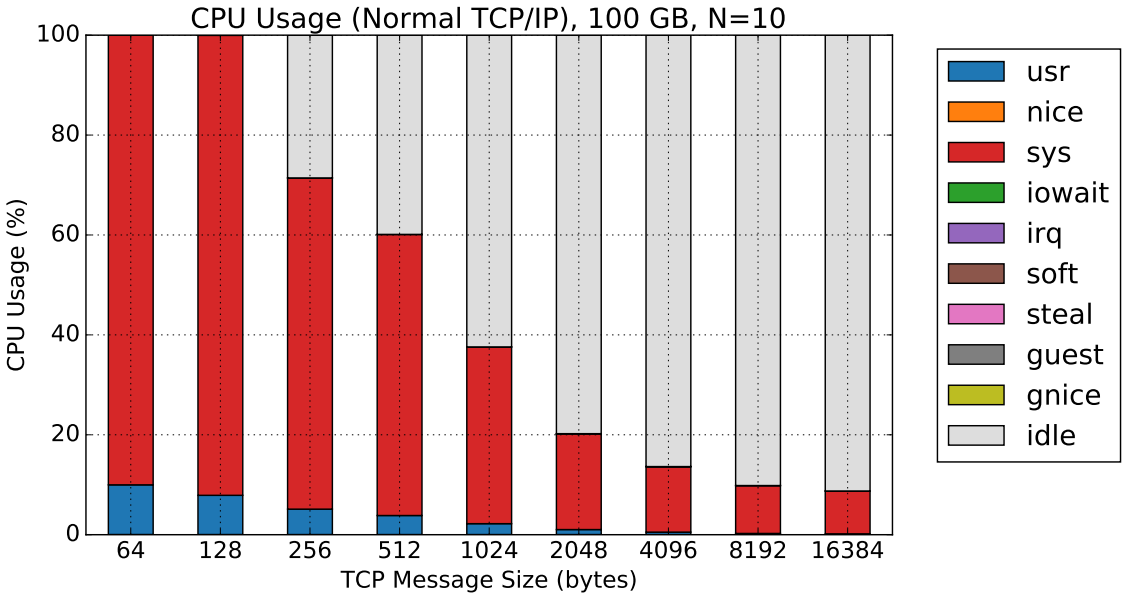

図 6.3. アプリケーションノードにおける CPU 使用率 (TCP/IP 通信時)

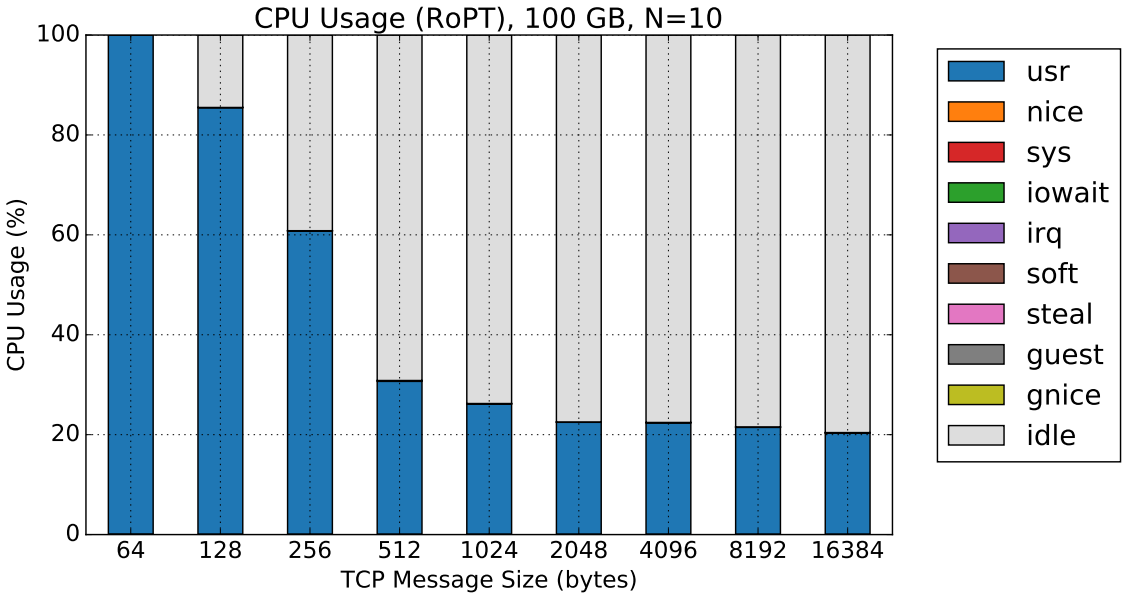

図 6.4. アプリケーションノードにおける CPU 使用率 (RoPT 使用時)

従来の TCP/IP 通信と RoPT を用いた通信のいずれも，TCP メッセージサイズが大きくなるほど CPU 使用率が小さくなっている．これは，TCP メッセージサイズが大きくなるほど単位時間あたりに送り出されるパケット数が減少するためである．アプリケーションノードにおいて，従来の TCP/IP 通信ではパケットごとに send システムコールが呼ばれ，システムコールのオーバーヘッドとカーネルでのネットワークスタック処理が CPU サイクルを消費する．そのため，処理すべきパケット数が減少するにともない CPU サイクルの消費も抑えられる．RoPT を用いた通信では send 関数が呼ばれるたびに rsend 関数を呼び出して RDMA によるデータ転送を行う．こちらはカーネルでの処理によるオーバーヘッドは存在しないが，同様にパケットごとに転送のための処理が行われるため，処理すべきパケット数の減少にともない CPU サイクルの消費が抑えられるのは同じである．

2 つの結果のグラフを比較して最も大きく異なるのは CPU 使用率の内訳である．従来の TCP/IP 通信では sys，すなわちカーネルでの CPU 使用率が idle 以外の高い割合を占めているのに対し，RoPT を用いた通信ではカーネルでの CPU 使用がほぼ見られない．代わりに usr，すなわちユーザーランドでの CPU 使用で占められている．この違いは，TCP/IP ネットワークスタックの処理がカーネル内で行われるのに対して，RoPT の RDMA 通信でのデータ転送はユーザー空間からカーネルをバイパスして行われるというアーキテクチャの違いによってもたらされる．設計における RDMA によるユーザー空間からのデータ転送がたしかに実現されていることを CPU 使用率の評価結果からこうして確認することができた．

結果の具体的な値について見ると，TCP メッセージサイズが 64 bytes のときは従来の TCP/IP 通信と RoPT を用いた通信の双方で 100% の CPU 使用率となった．128 bytes から 1024 bytes までの間では RoPT の方が CPU 使用率が低くなっており，特に 512 bytes のときは従来の TCP/IP 通信での CPU 使用率が 60% あるのに対して RoPT を用いた通信では 30% に抑えられている．しかし，2048 bytes 以上になると逆転し，TCP/IP 通信の方が CPU 使用率が低くなっている．このように，送信する TCP メッセージサイズの大きさに対し，従来の TCP/IP 通信と RoPT を用いた通信の CPU 使用率の変化の傾向は異なるということが分かった．

たとえば，TCP メッセージサイズが 2048 bytes 以上のとき，RoPT を用いた通信での CPU 使用率は 20% 程度でほぼ一定となり小さくなっていない．これは，MTU を超える大きさのメッセージサイズを一度に送信することができず rsend 関数の中で繰り返し送信処理が行われるためだと考えられる．これに対し従来の TCP/IP 通信では，NIC の TCP Segmentation Offload (TSO) 機能が有効になっていればこの機能によってハードウェアで分割送信処理が行われる．TCP メッセージサイズが 2048 bytes 以上のときにおいても TCP メッセージサイズが大きくなるほど CPU 使用率が小さくなる傾向が変わっていないのはこのためだと考えられる．

6.3　分析と考察

従来の TCP/IP 通信と RoPT を用いた通信のそれぞれについて TCP メッセージの送信性能の計測と比較を行い，TCP メッセージサイズが 128 bytes から 1024 bytes までの間において，RoPT を用いた通信は従来に比べ CPU リソースの消費を抑えた通信が可能であることを示した．また，RoPT を用いた通信では，従来の通信で CPU を消費していたカーネルでの処理をほぼ無くした通信を実現した．これはカーネルをバイパスしてユーザー空間から RDMA でのデータ転送を行うという設計のとおりの結果が得られたと言える．

しかし RoPT を用いた通信においては，カーネルでの処理が無くなった代わりにユーザーランドでの CPU 使用が増加している．これはユーザー空間でデータ転送を行うためであると考えられるが，高効率な RDMA での通信としては CPU リソースの消費が大きいように見える．本節ではこのユーザーランドにおける CPU 使用について分析と考察を行う．

CPU サイクルがどこで消費されているのかを調査するため，RoPT を用いた通信でスループット性能評価を行なっているときのアプリケーションノードにおいて “perf top” コマンドを実行して確認した．このときのスクリーンショットを図 6.5 に示す．この図は TCP メッセージサイズを 4096 bytes に指定して性能評価のためのベンチマークを走らせたときのものである．perf [8] はシステムのプロファイルを取得するために用いられるツールで，この図ではより上位にあるシンボルの関数が，サンプリングされた CPU サイクルにおいてよく使われていることを示している．

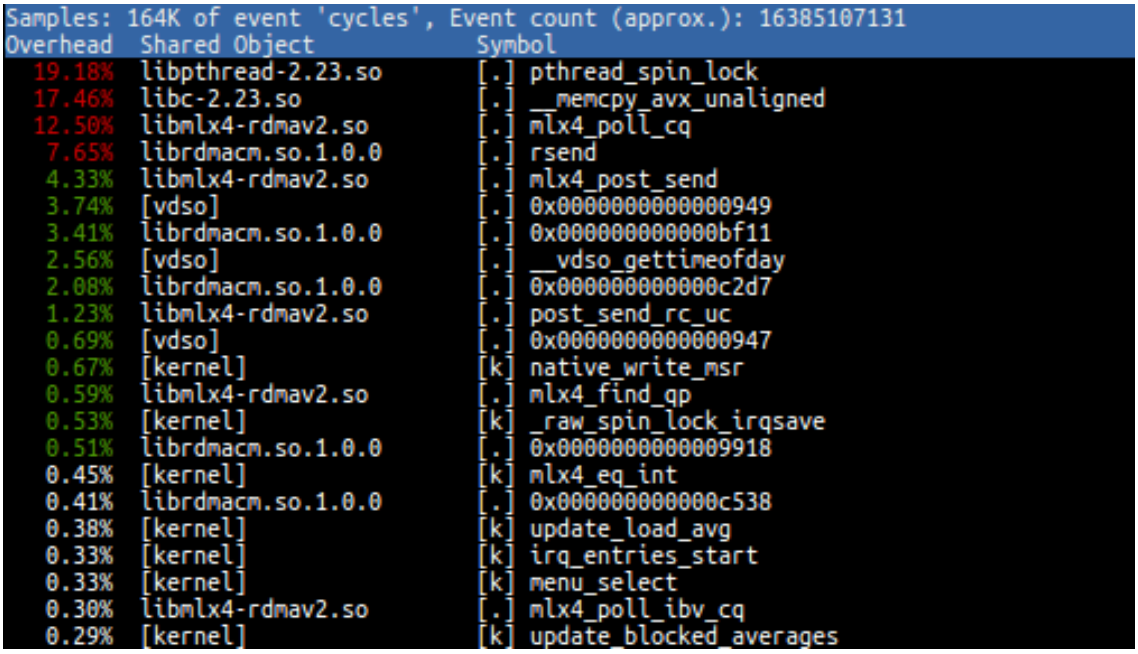

図 6.5. スループット性能評価実行時のアプリケーションノードにおける “perf top” のスクリーンショット (RoPT 使用時)

図によると，スピンロックとメモリコピー，Completion Queue (CQ) のポーリングに CPU サイクルが多く消費されているように見える．これらの処理について考察する．

まず，スピンロックは rsocket においてスレッドセーフを実現するために使用されている．スピンロックはたとえば mlx4_post_send 関数で使用されている．この関数は Send Queue (SQ) に対して送信要求を投げる関数である．librdmacm ライブラリは Queue Pair (QP) のようなリソースを共有して使用するにあたりスレッドセーフな設計が行われており，このためにスピンロックが使用される．しかし，ここで走らせたベンチマークはアプリケーションノードにおいてシングルプロセス・シングルスレッドで実行しており，スピンロックによって CPU サイクルが多く消費されたとは考えにくい．perf によるプロファイルのサンプリングで，頻繁に呼ばれるスピンロックのイベントが多く抽出されたのではないかと考えられる．

次に，メモリコピーは rsocket 内部の送信バッファへのコピーで発生している．rsocket において一定の大きさ以上のデータを rsend で送信する場合，一度 rsocket 内部の送信バッファにメモリコピーされ，そこから RDMA でデータが転送される．RDMA 通信を行うには Memory Region の登録が必要である，RDMA での通信を意図していない従来のアプリケーションでは Memory Region の登録を行なっているはずがない．そのため，ゼロコピーでの転送ではなくなるものの，あらかじめ登録されたメモリ領域として rsocket 内部にバッファを確保しているのは正しいと言える．また，メモリコピーが行われるのは従来の TCP/IP 通信でも同様であり，ここでは差が出ないと考えられる．

最後に，CQ のポーリングは QP に対して投げた要求が処理されたかを確認するために行われる．rsocket では高いスループットと低レイテンシを達成するため，CQ の確認にポーリングを用いている．しかしポーリングでは CPU サイクルを消費するため，これが結果としてユーザーランドにおける CPU 使用を高めていると考えられる．CQ を確認する方法にはポーリングのほかにイベントを通知してもらう方法がある．スループットとレイテンシの性能はポーリングと比較するとやや劣ることにはなるが，CPU リソースの消費を抑えることが可能となる．ポーリングではなくイベント通知によって CQ を確認する方法を導入することにより，CPU リソースの消費をより削減する余地があると考えられる．

第 7 章

結論

本章では, 本論文のまとめと今後の課題を示す.

7.1 まとめ

本研究では，通信にかかるサーバーの CPU リソース消費を削減するため，TCP/IP ネットワークスタックの処理と機能を外部ノードに分離するアーキテクチャ，RoPT を提案した．アプリケーションサーバーと外部ノード間のデータ転送を RDMA で行い，TCP/IP のネットワークスタック処理とコネクションを外部ノードに Proxy させるシステムの設計と実装を行なった．アプリケーション実行時に Socket API 関数を差し替えることにより，従来のアプリケーションにおいて特別な変更を加えること無く RoPT を用いた通信を行うことを可能とした．そして，TCP メッセージの送信性能評価を行い，RoPT を用いた通信では従来の TCP/IP 通信に比べ CPU リソースの消費を抑えた通信が可能であることを示した．

7.2 今後の課題

本研究では，設計および実装を行なった RoPT について適用可能なアプリケーションに制限があった．プロセス空間を複製する fork システムコールを用いるアプリケーションでは RDMA の通信を利用できないという制限である．RoPT を用いた通信をより多くのアプリケーションでも利用できるようにするため，RDMA 通信のコネクションを子プロセスに継承できないことを前提とした仕組みを考える必要がある．たとえば fork によるプロセス空間の複製後でコネクションを確立し直すことが可能であれば，それによって fork を用いるアプリケーションへの適用も可能になると考えられる．

また，本論文の実装では，スループット性能について RoPT Manager でのデータ転送がボトルネックとなっていた．高速化するネットワークに対応するためには RDMA 通信部分の性能を生かしてさらに高速な通信を実現する必要があり，このボトルネックの解消が課題となる．本論文では RoPT Manager についてユーザー空間で動作するアプリケーションとして実装を行なっているが，カーネル内のネットワークスタックを利用することを考えると，

ユーザー空間を一度介するオーバーヘッドの削減のため RoPT Manager をカーネルモジュールとして動作させるといった方法が考えられる．また，カーネル内ネットワークスタックにこだわらなければ，高速パケット I/O 処理技術を用いて実装された TCP スタックを RoPT Manager に導入するという方法も考えられる．

最後に，6.3 節で述べたように，RoPT Client の実装については CPU リソースの消費をより削減するための余地がある．本論文の実装では，RDMA 通信において Completion Queue (CQ) を確認するためにポーリングが用いられておりこれが CPU サイクルを消費していた．ポーリングを用いる代わりにイベント通知による方式を導入することによって，どの程度の CPU リソース消費の削減が可能となるか検討し評価することを今後の課題とする．

参考文献

[1] Amazon Rekognition – AWS. https://aws.amazon.com/jp/rekognition/, (Last Accessed 2018-01-15).

[2] Cisco Visual Networking Index: Forecast and Methodology, 2016–2021 (June, 2017). https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html, (Last Accessed 2018-01-15).

[3] Cognitive Services – Microsoft Azure. https://azure.microsoft.com/ja-jp/services/cognitive-services/, (Last Accessed 2018-01-15).

[4] Dell Networking Z9100-ON Switch Spec Sheet. http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/dell-networking-z9100-spec-sheet.pdf, (Last Accessed 2018-01-15).

[5] DPDK – Data Plane Development Kit. https://dpdk.org/, (Last Accessed 2018-01-15).

[6] Google Cloud Machine Learning at Scale – Google Cloud Platform. https://cloud.google.com/products/machine-learning/, (Last Accessed 2018-01-15).

[7] Mellanox ConnectX-4 EN Adapter Card Single/Dual-Port 100 Gigabit Ethernet Adapter. http://www.mellanox.com/page/products_dyn?product_family=204&mtag=connectx_4_en_card, (Last Accessed 2018-01-15).

[8] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, (Last Accessed 2018-01-15).

[9] IEEE 802.1 Data Center Bridging Task Group. http://www.ieee802.org/1/pages/dcbridges.html, (Last Accessed 2018-01-22).

[10] Pavan Balaji. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *In RAIT workshop '04*. Citeseer, 2004.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pp. 49–65. USENIX Association, 2014.

[12] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.*, Vol. 9, No. 7, pp. 528–539, 2016.

[13] Gautam Chanda. The Market Need for 40 Gigabit Ethernet. *White Paper, Cisco*, 2012.

[14] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context Switch Overheads for Linux on ARM Platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07. ACM, 2007.

[15] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of SANE*, pp. 85–93, 2004.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pp. 401–414. USENIX Association, 2014.

[17] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pp. 202–215. ACM, 2016.

[18] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. *ACM SIGCOMM Computer Communication Review*, Vol. 41, No. 4, pp. 195–206, 2011.

[19] InfiniBand. Architecture Specification Release 1.2. 1 Annex A16: RoCE. *InfiniBand Trade Association*, 2010.

[20] InfiniBand. Architecture Specification Volume 1 Release 1.3. *InfiniBand Trade Association*, 2015.

[21] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pp. 489–502. USENIX Association, 2014.

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pp. 295–306. ACM, 2014.

[23] Mao Miao, Fengyuan Ren, Xiaohui Luo, Jing Xie, Qingkai Meng, and Wenxue Cheng. SoftRDMA: Rekindling High Performance Software RDMA over Commodity Ethernet. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet '17, pp. 43–49. ACM, 2017.

[24] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating*

Systems Design and Implementation, OSDI '14, pp. 1–16. USENIX Association, 2014.

[25] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, 2007.

[26] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, pp. 101–112. USENIX Association, 2012.

[27] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ALS '01. USENIX Association, 2001.

[28] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas, Kornilios Kourtis, and Thomas R. Gross. FlashNet: Flash/Network Stack Co-design. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pp. 15:1–15:14. ACM, 2017.

[29] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pp. 306–324. ACM, 2017.

[30] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC '16, pp. 43–56. USENIX Association, 2016.

謝辞

研究室に配属されて以来，学士と修士の両課程での研究活動にあたり，多くの方にご指導とご協力をいただきました．ここに心より感謝を申し上げます．

指導教員である東京大学情報理工学系研究科教授 江崎浩博士には，幅広い知識と深い経験から的確なご指導をいただきました．自分には無い視点からのコメントで問題についてより深く考えるための機会を与えてくださりました．ここに深く感謝いたします．また，研究の内容や進め方について多くの議論や助言をいただきました情報理工学系研究科准教授 落合秀也博士，生産技術研究所電子計算機室助教山本成一博士，情報理工学系研究科特任助教 塚田学博士に感謝いたします．

研究のアイデアや方針，論文の執筆など，研究活動全般にわたり数多くの助言やご指導をいただきました東京大学情報基盤センターネットワーク研究部門助教 中村遼博士に深く感謝いたします．本研究を進めるにあたり機材を提供していただき，何度も相談に乗ってくださりました．また，研究の方向性について相談に乗ってくださった株式会社 Preferred Networks 浅井大史博士，株式会社 IIJ イノベーションインスティテュート 宇夫陽次朗博士に感謝いたします．両氏にはネットワークやシステムを対象とした研究や評価の方法について議論を通じて多くのことを学ばせていただきました．

研究について助言をいただいたりシステムの触り方を教えてくださった研究室 OB の池上洋行博士，研究室生活においていろいろと相談に乗ってくださった小林諭氏に感謝いたします．同期として共に教え合い励まし合ってきた菰原裕氏，岸本丈氏，北畠知行氏，薗部啓氏に感謝いたします．研究室において研究と運用に関して多くのことを学ばせていただいた先輩の皆様に感謝いたします．また，研究室生活を共に過ごした後輩の皆様に感謝いたします．諸事務を通じて研究室生活をご支援いただいた江崎研究室秘書の高橋富美秘書，岩井愛映子秘書に感謝いたします．

また，これまでの学生生活を支えてくださった家族や友人に深く感謝いたします．

最後に，研究生活や学生生活においてさまざまなご指導とご支援をくださったすべての皆様に感謝いたします．

2018 年 2 月 1 日

坂本 裕紀