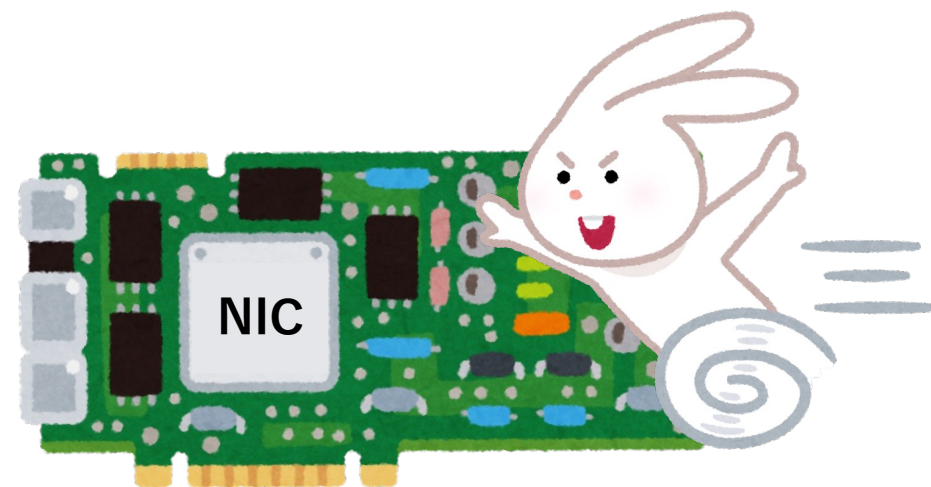


NIC の高速化と システムソフトウェア研究

2010 年くらいからの振り返り

技術研究所 安形



資料

- 発表資料
 - <https://seminar-materials.iijlab.net/iijlab-seminar/iijlab-seminar-20231017.pdf>
 - <https://iijlab-seminars.connpass.com/event/297595/> から辿れます
 - ファイルのサイズが大きいため (16 MB 程度) ダウンロードしていただく場合はご注意ください
- 技術レポート：Internet Infrastructure Review (IIR) Vol. 60
 - システムソフトウェアの通信分野における2010年頃からの研究まとめ
 - 2023年9月26日発行
 - HTML / PDF 版：<https://www.iij.ad.jp/dev/report/iir/060.html>

概要

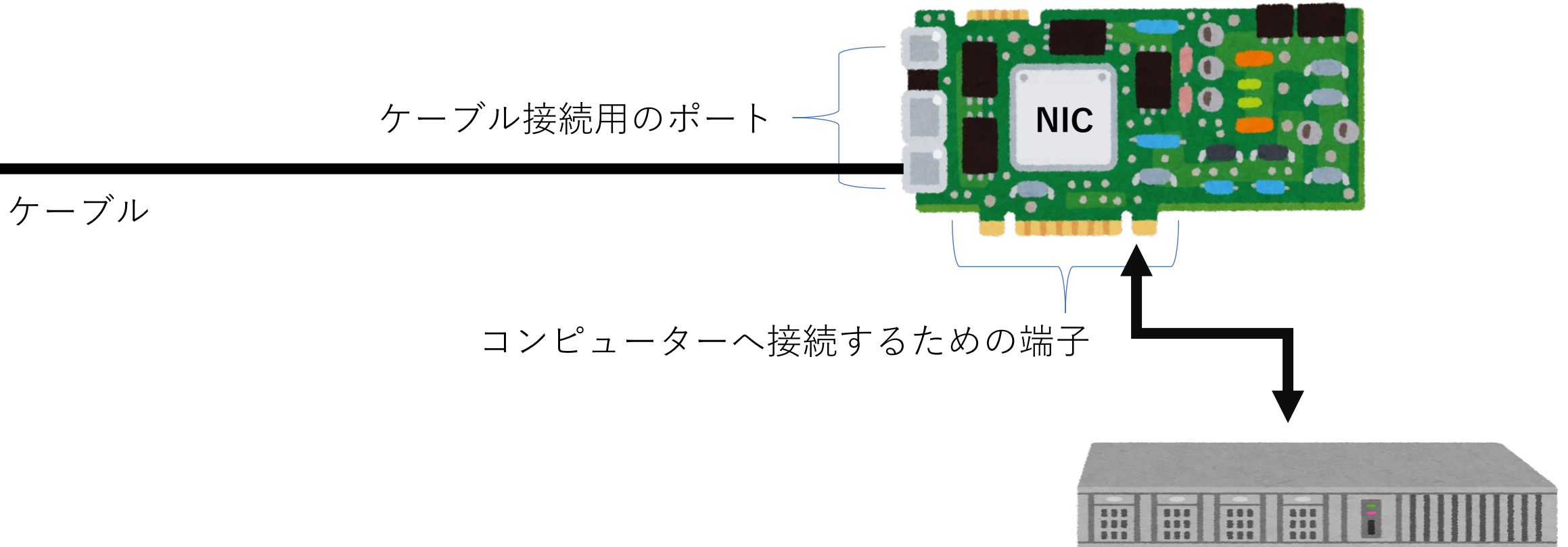
- 2010 年くらいから 10 Gbps を超えるような速度の NIC が比較的安価で入手可能になり、広く利用されるようになった
- 既存のソフトウェアの実装にとって、高速な NIC の性能を十分に引き出すのは難しいという課題が顕著になった
- この課題について、システムソフトウェア分野でのこれまでの取り組みを紹介

背景

高速な NIC と用途

Network Interface Card (NIC)

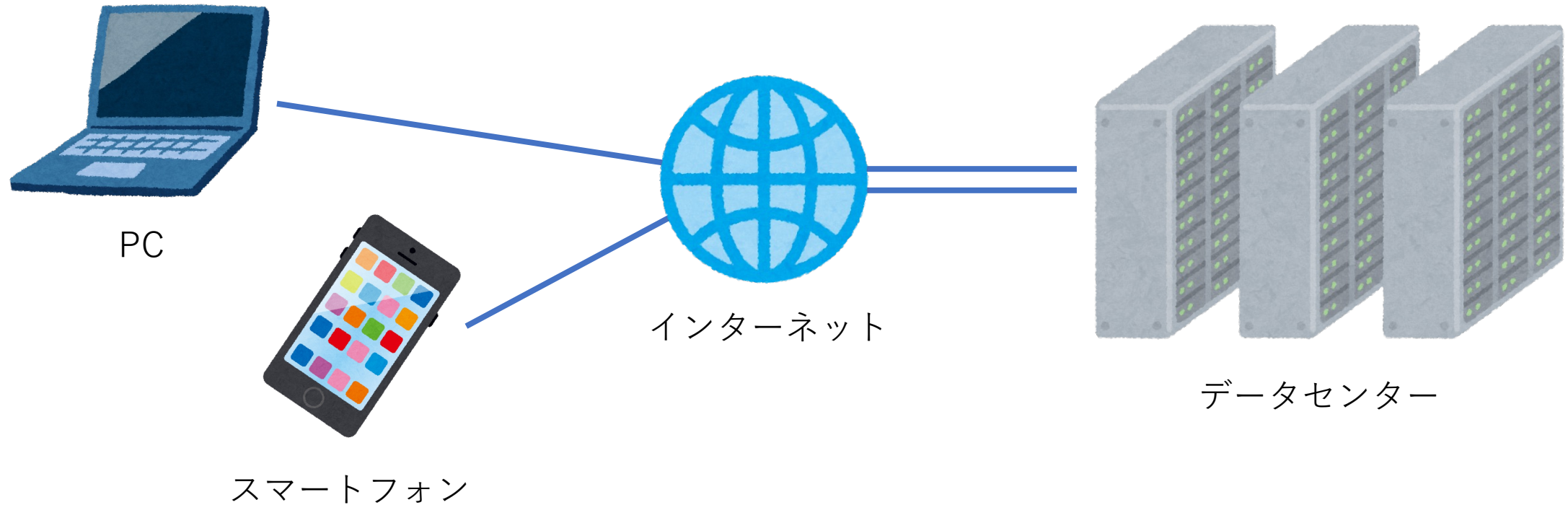
- コンピューターへ搭載可能な通信用ハードウェア



2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側



2010年くらいの利用シナリオ

アプリケーション例

- SNS
- 検索エンジン
- 動画ストリーミング
- クラウドストレージ
- オンラインゲーム
- ビッグデータ分析
- その他たくさんのWeb・スマホ向けサービス

サービス提供側

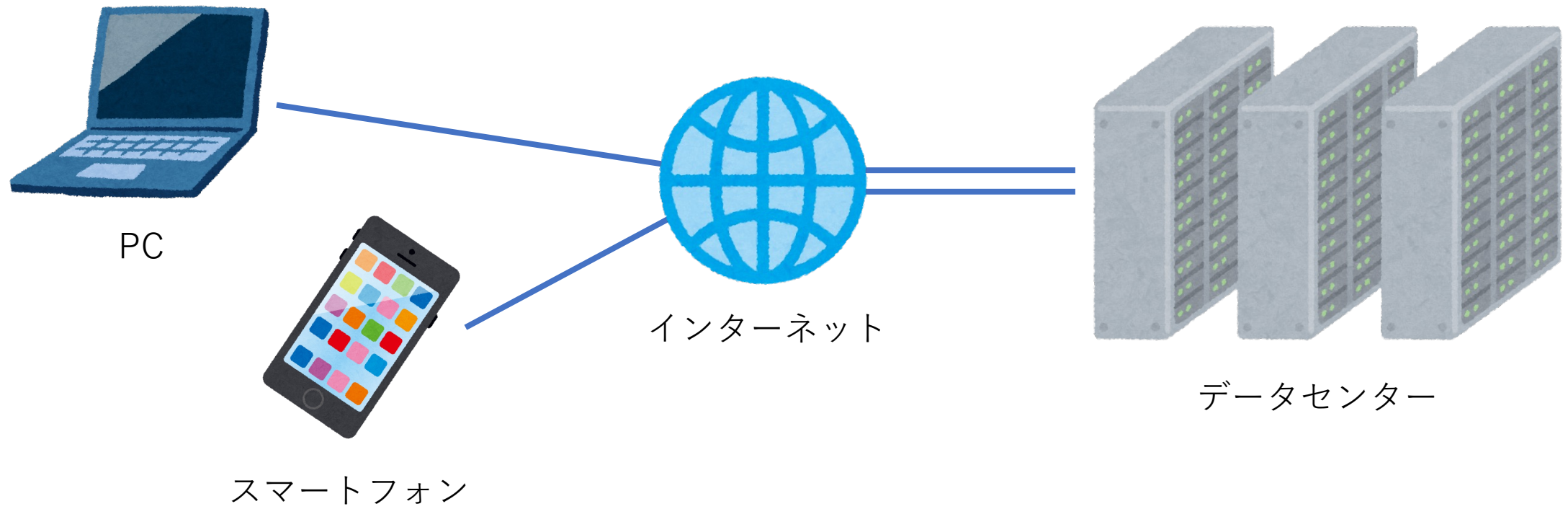


データセンター

2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側



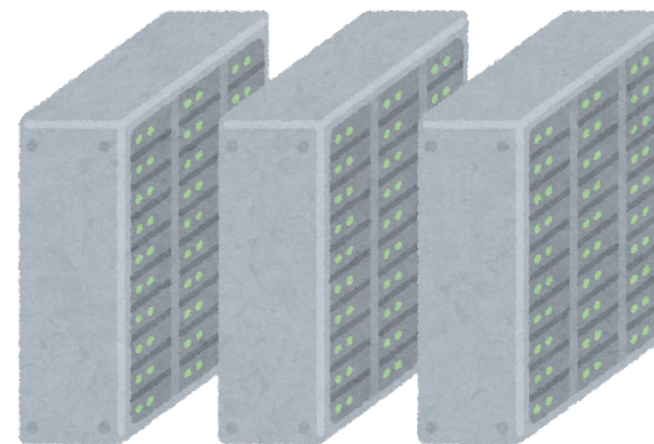
2010年くらいの利用シナリオ

サービス利用者の端末



インターネット

サービス提供側

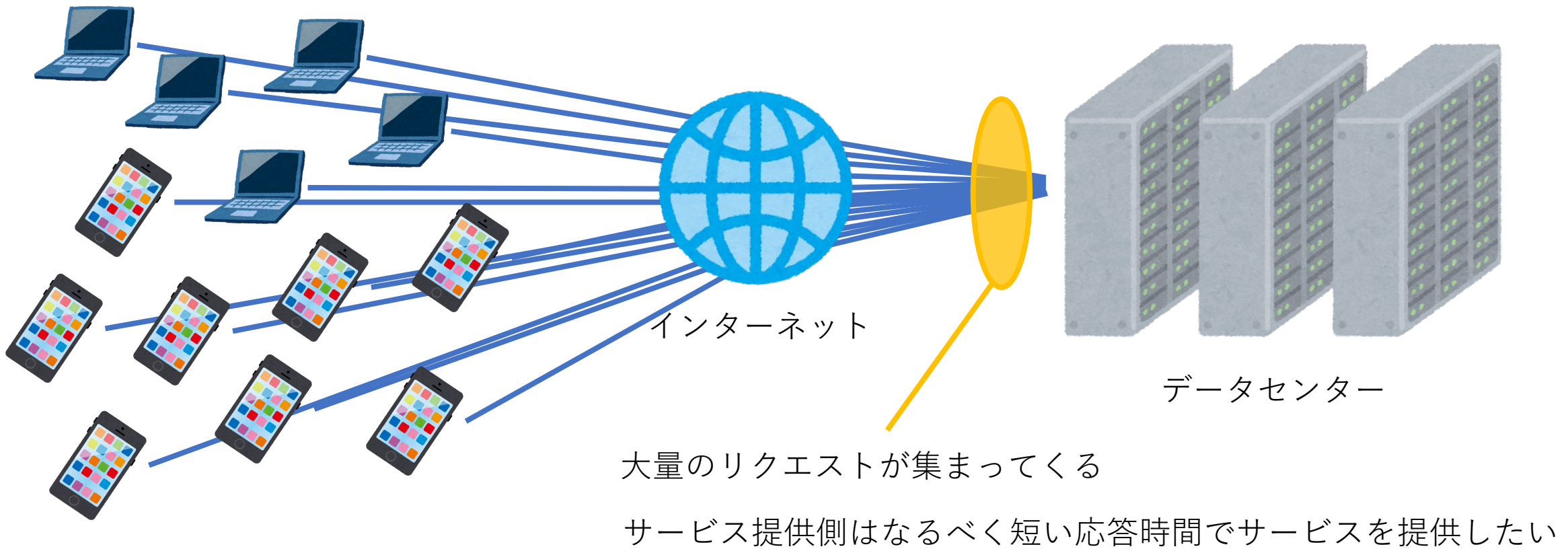


データセンター

2010年くらいの利用シナリオ

サービス利用者の端末

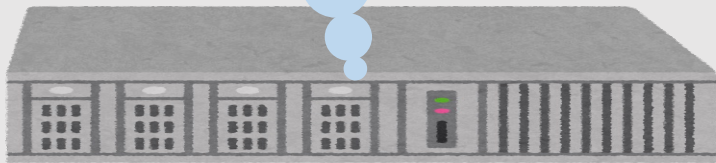
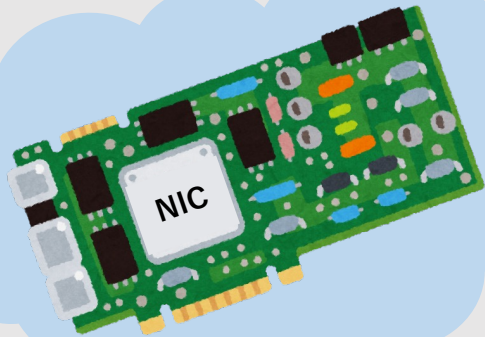
サービス提供側



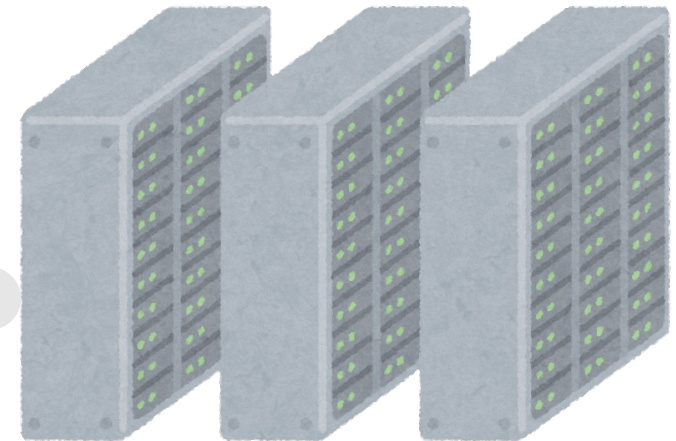
2010年くらいの利用シナリオ

データセンター内のサーバー

10 ~ Gbps



サービス提供側



データセンター

ネット

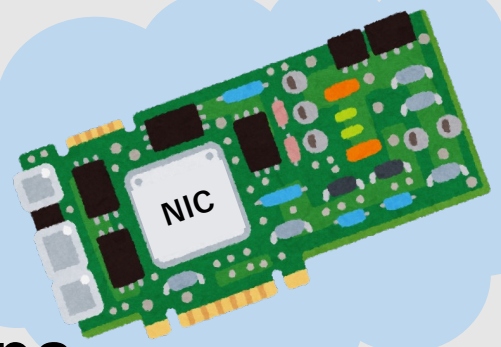
大量のリクエストが集まってくる

サービス提供側はなるべく短い応答時間でサービスを提供したい

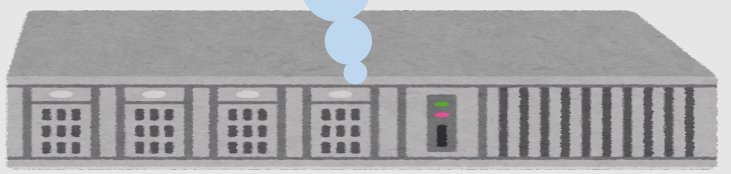
2010年くらいの利用シナリオ

データセンター内のサーバー

サービス提供側



10 ~ Gbps



ネット

要求
大量のクライアントへ
短時間でサービスを提供したい

NICの高速化により
各サーバーが送受信できる
データの量が増加した

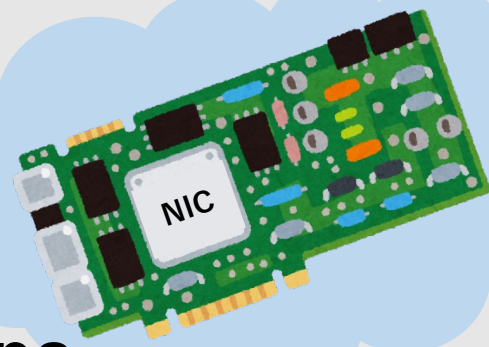
大量のリクエストが集まってくる

サービス提供側はなるべく短い応答時間でサービスを提供したい

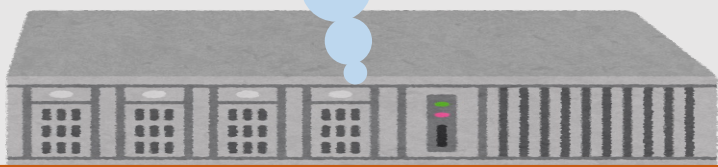


2010年くらいの利用シナリオ

データセンター内のサーバー



10 ~ Gbps



ネット

サービス提供側

要求

大量のクライアントへ
短時間でサービスを提供したい

NICの高速化により

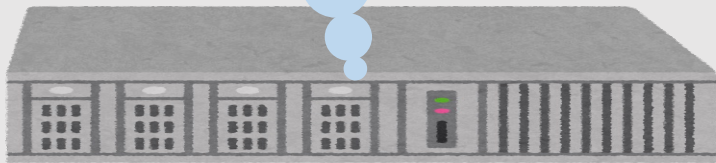
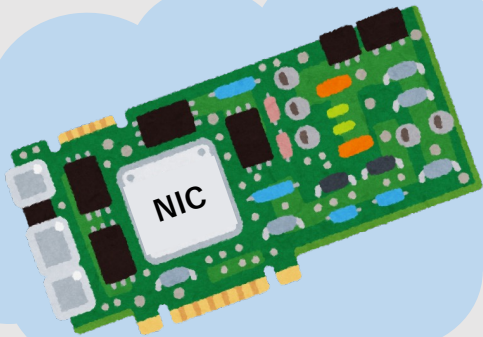
各サーバーが送受信できる
データの量が増加した

しかし、NICが速くなったからといって
大量のクライアントに短時間でサービスを提供できるわけではなかった

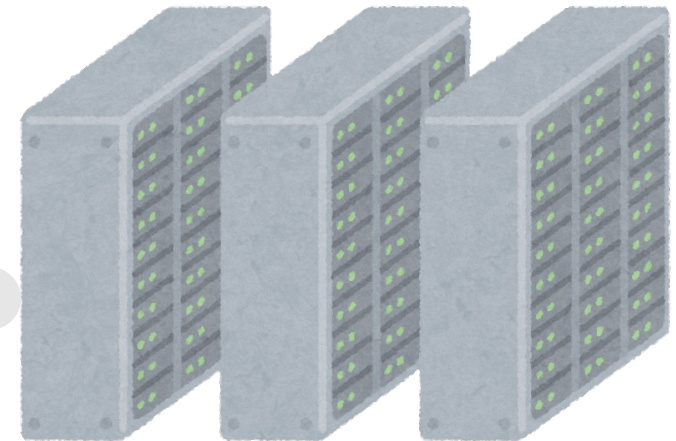
通信関連のシステムソフトウェア

データセンター内のサーバー

10 ~ Gbps



サービス提供側



データセンター

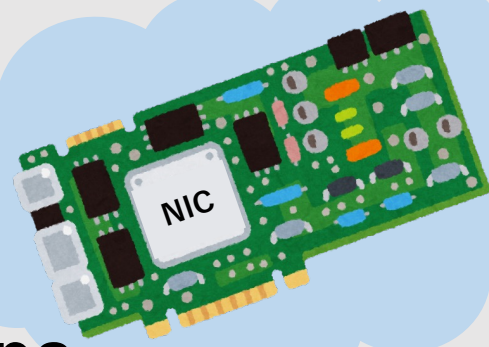
ネット

大量のリクエストが集まってくる

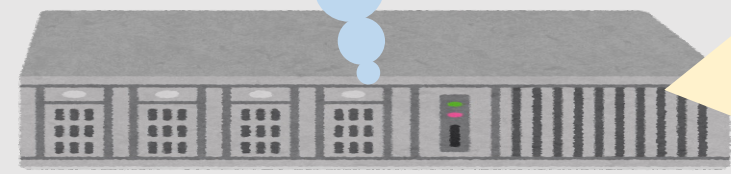
サービス提供側はなるべく短い応答時間でサービスを提供したい

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



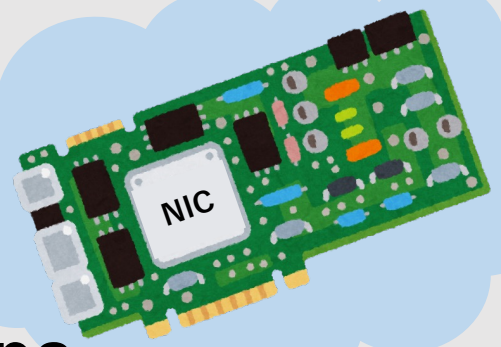
多重のリク
サービス提供

通信関連ソフトウェア

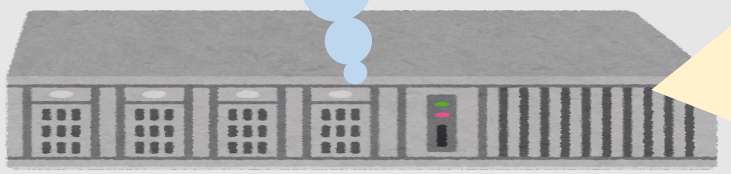


通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



多重のリク
サービス提供

通信関連ソフトウェア

ユーザー空間 **アプリケーション**

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

ホスト

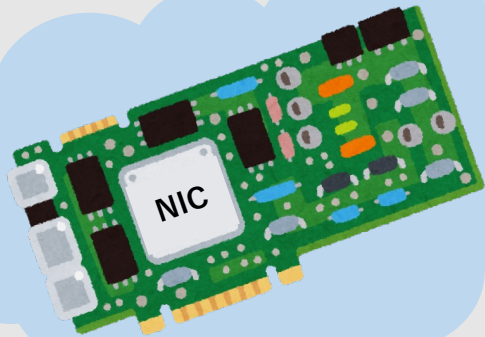
仮想 NIC バックエンド

仮想スイッチ

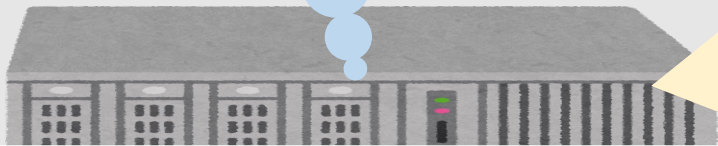
NIC デバイスドライバ

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



ハードウェア (NIC) が速くなった結果
ソフトウェアの効率が更に重要になった

通信関連ソフトウェア

ユーザー空間

アプリケーション

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

ホスト

仮想 NIC バックエンド

仮想スイッチ

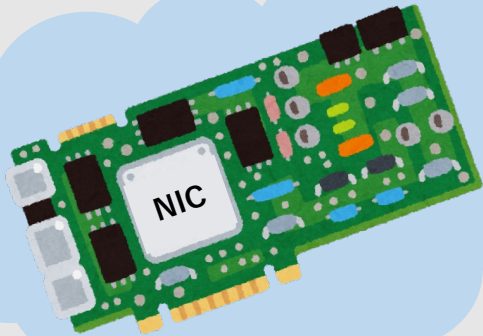
NIC デバイスドライバ

背景

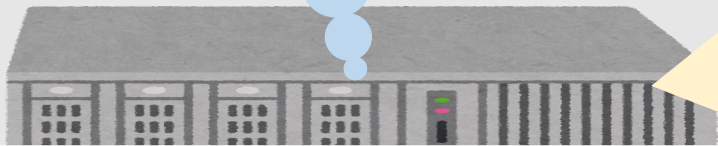
既存のシステムの性能

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



ハードウェア（NIC）が速くなった結果
ソフトウェアの効率が更に重要になった

通信関連ソフトウェア

ユーザー空間

アプリケーション

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

ホスト

仮想 NIC バックエンド

仮想スイッチ

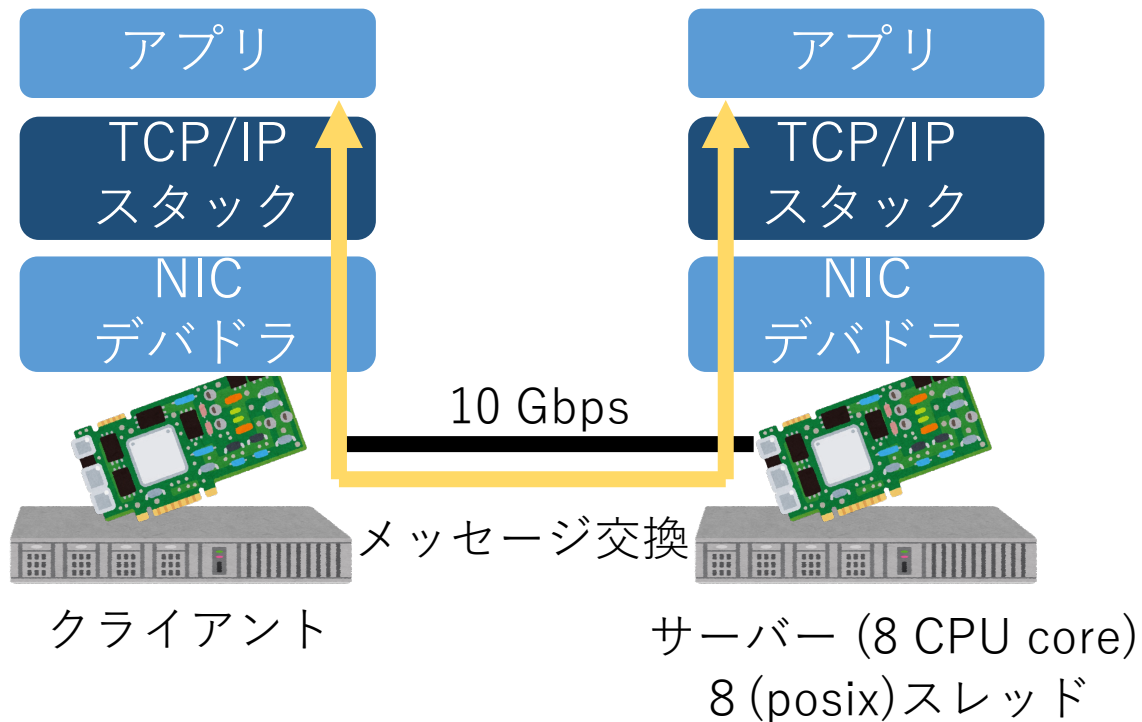
NIC デバイスドライバ

既存の実装の性能

2012 年の論文の発表資料より

Sangjin Han, Scott Marshall, Byung-Gon Chun, Sylvia Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O", OSDI 2012
<https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han>

- Linux TCP スタックのメッセージ(パケット)サイズごとの性能
 - 論文が提案手法との比較対象としているベースラインの性能

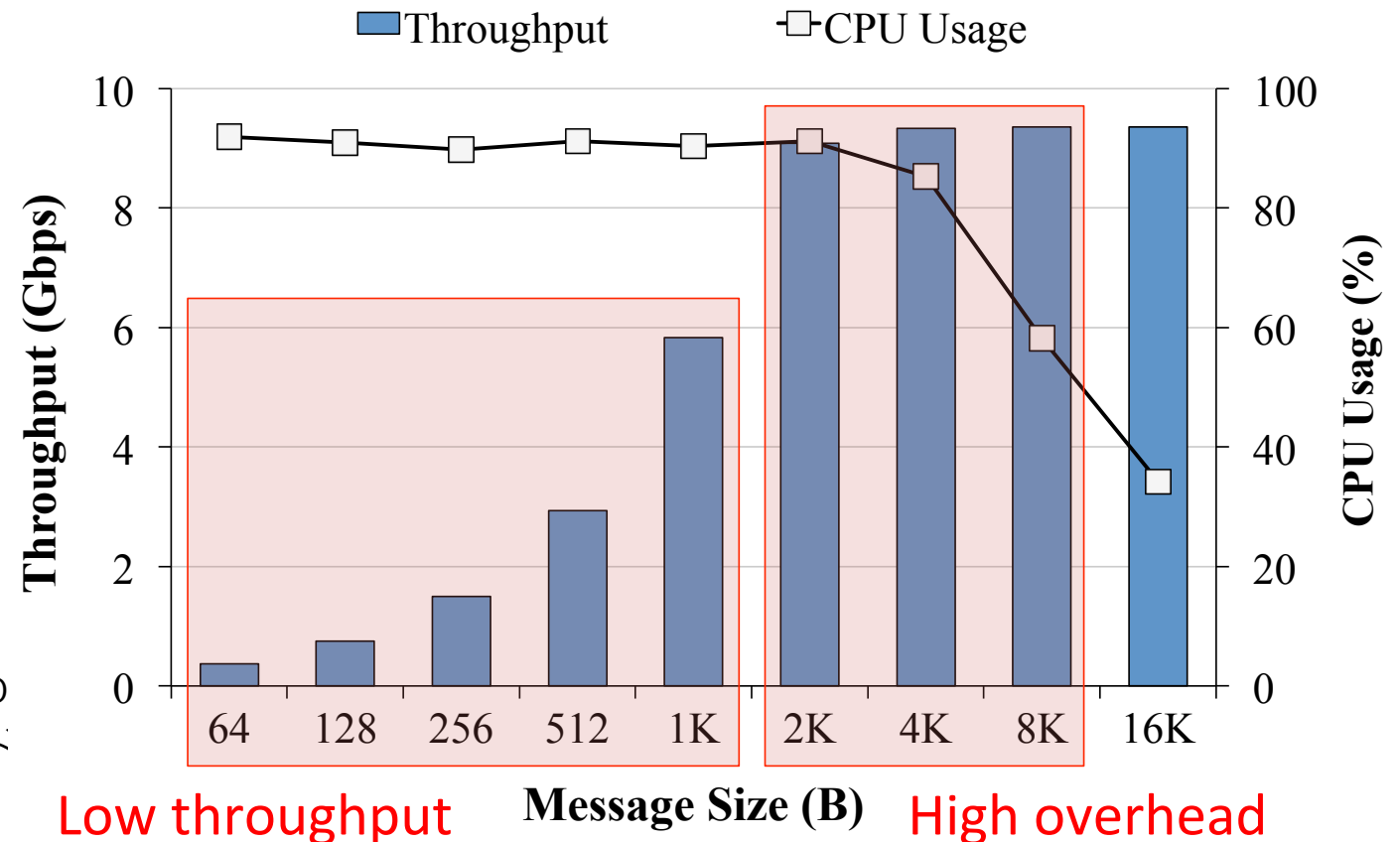
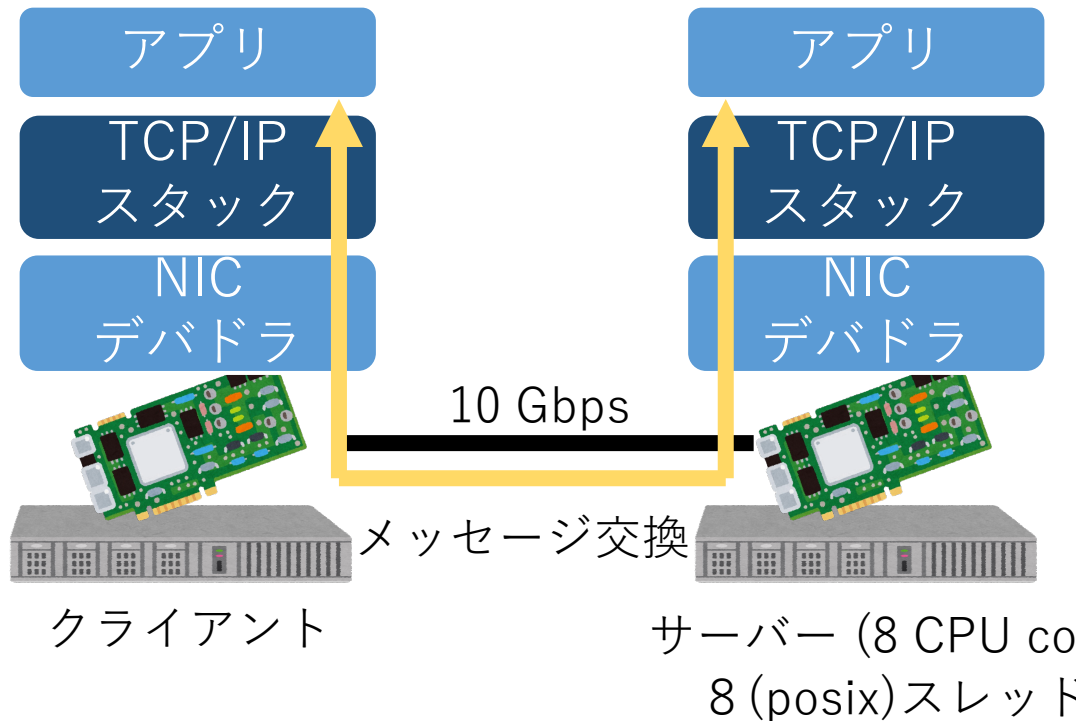


既存の実装の性能

2012 年の論文の発表資料より

Sangjin Han, Scott Marshall, Byung-Gon Chun, Sylvia Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O", OSDI 2012
<https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han>

- Linux TCP スタックのメッセージ(パケット)サイズごとの性能
 - 論文が提案手法との比較対象としているベースラインの性能



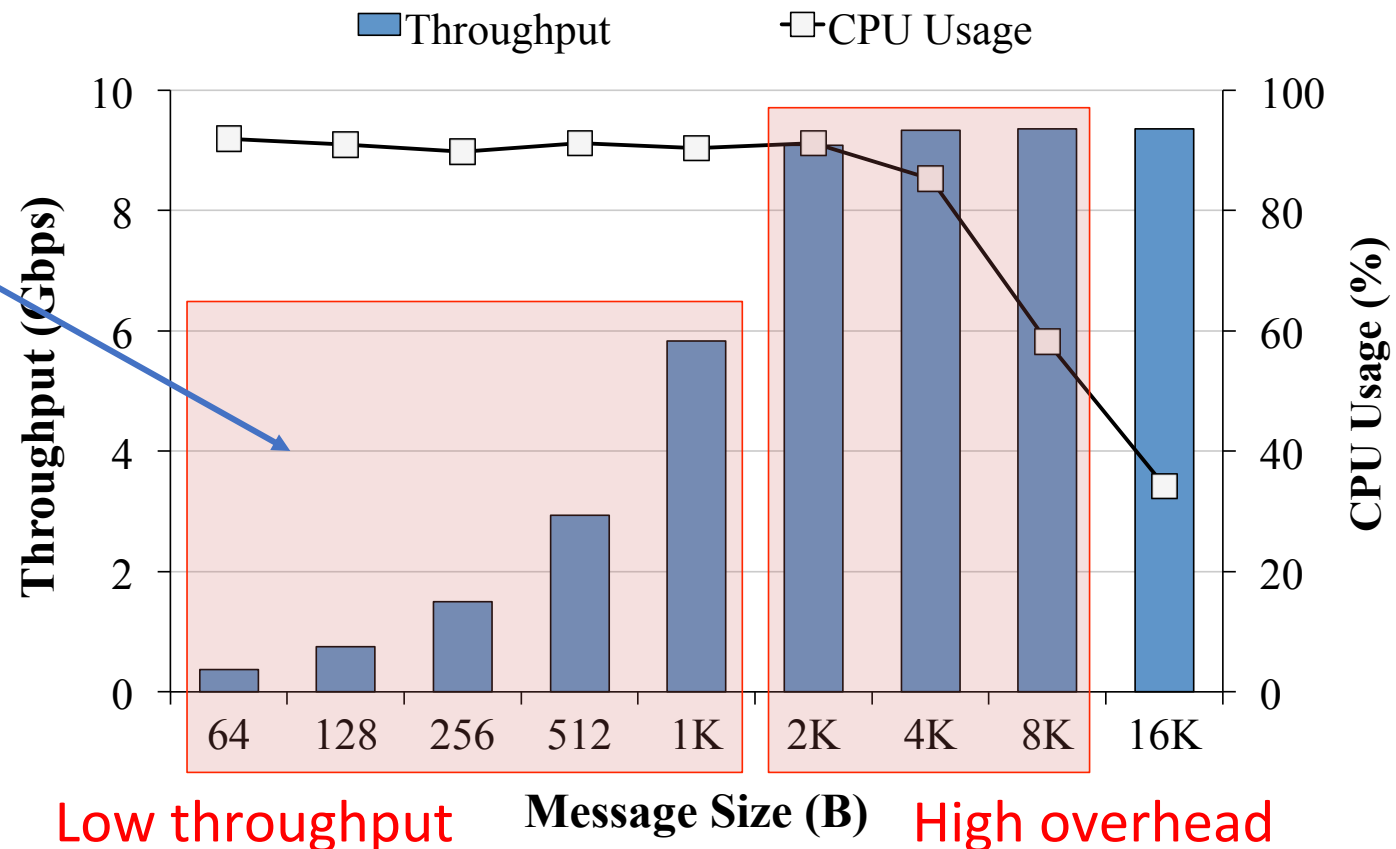
既存の実装の性能

2012 年の論文の発表資料より

Sangjin Han, Scott Marshall, Byung-Gon Chun, Sylvia Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O", OSDI 2012
<https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han>

- Linux TCP スタックのメッセージ(パケット)サイズごとの性能
 - 論文が提案手法との比較対象としているベースラインの性能

- メッセージサイズ $\leq 1K$
 - 10 Gbps を達成できない



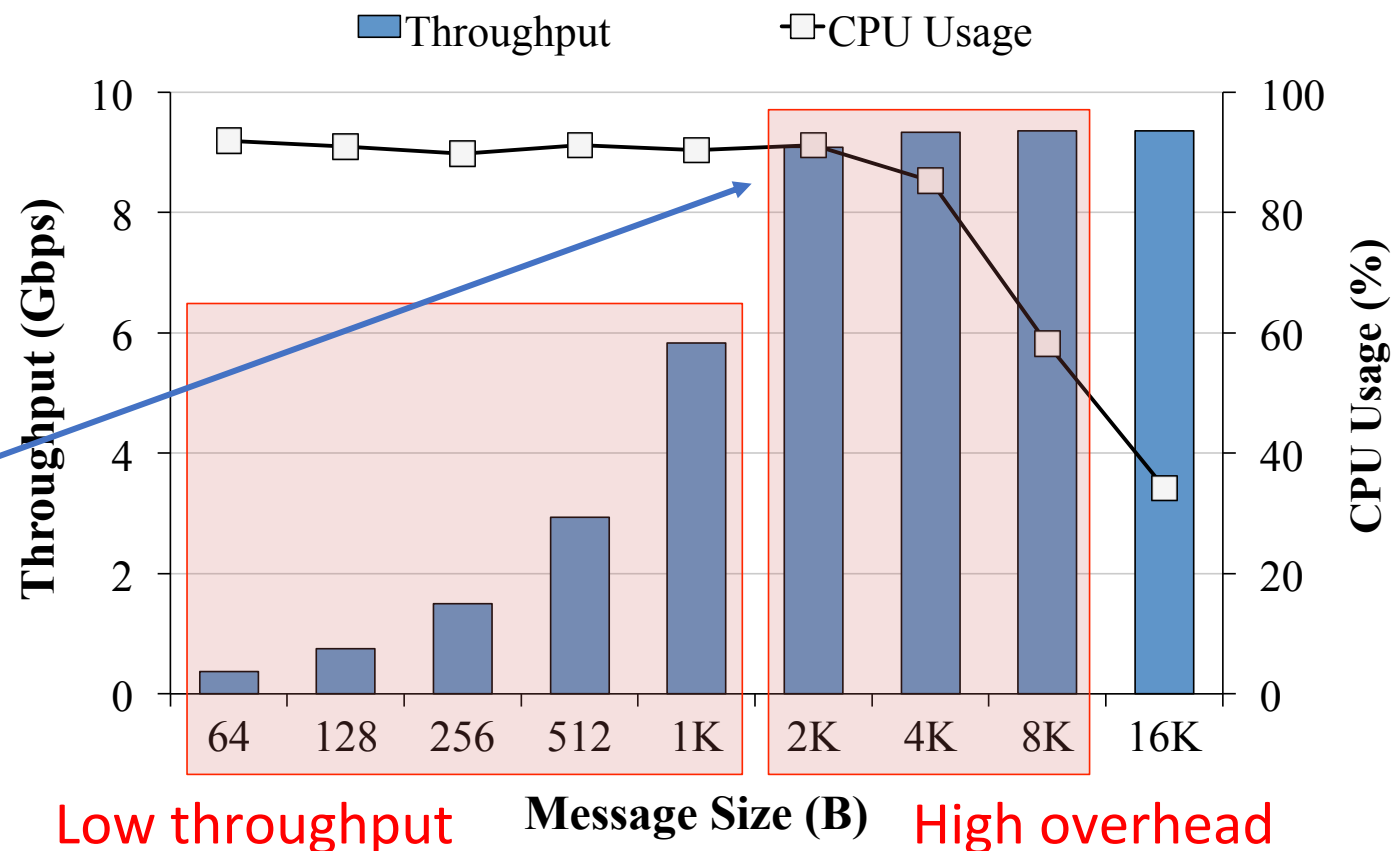
既存の実装の性能

2012 年の論文の発表資料より

Sangjin Han, Scott Marshall, Byung-Gon Chun, Sylvia Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O", OSDI 2012
<https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han>

- Linux TCP スタックのメッセージ(パケット)サイズごとの性能
 - 論文が提案手法との比較対象としているベースラインの性能

- メッセージサイズ $\leq 1K$
 - 10 Gbps を達成できない
- メッセージサイズ $\leq 4K$
 - 依然、高い CPU 使用率



既存の実装の性能

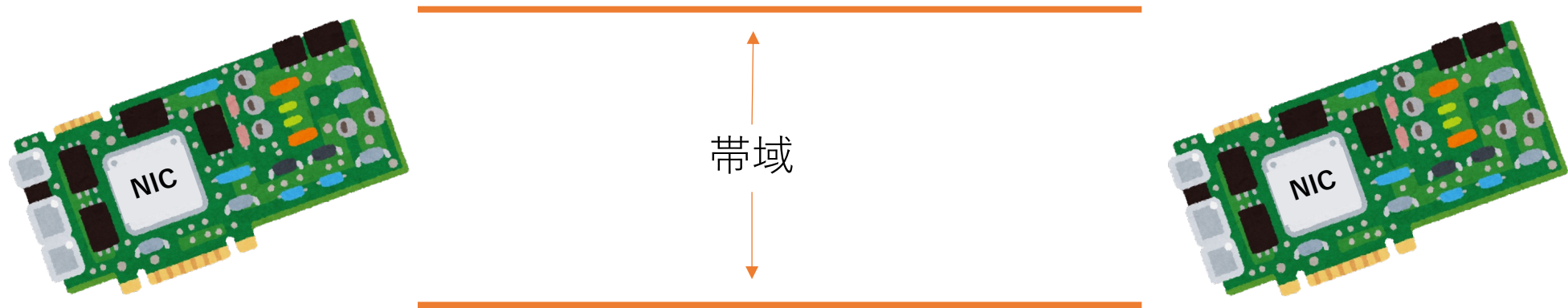
- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する

既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが小さい場合の方が大きい場合より多くのパケットを送れる

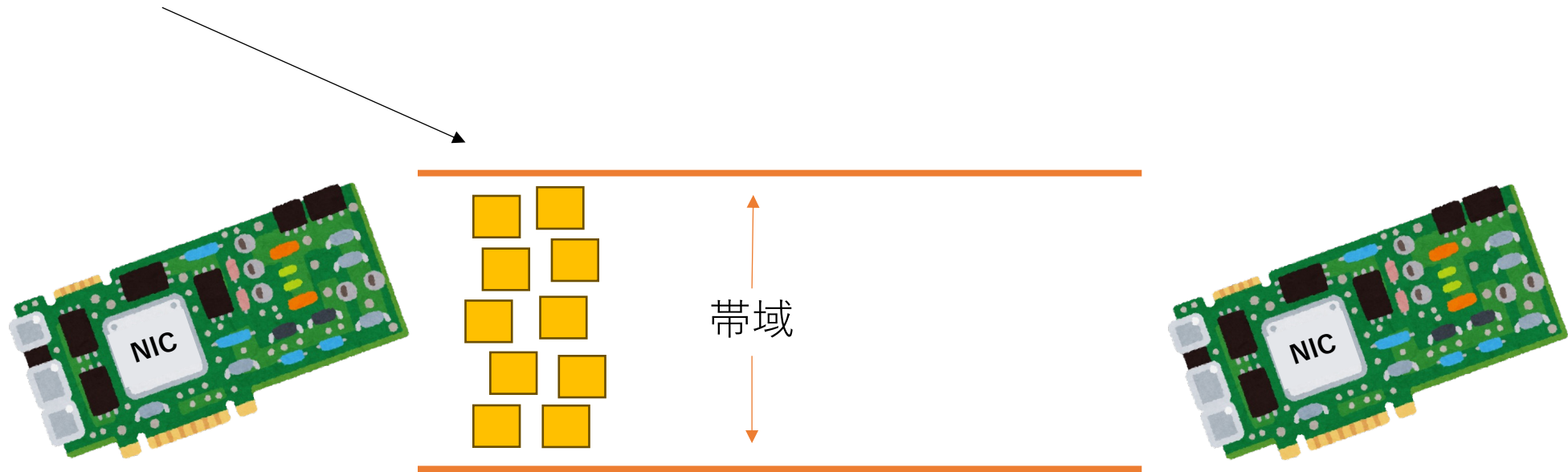
既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが小さい場合の方が大きい場合より多くのパケットを送れる



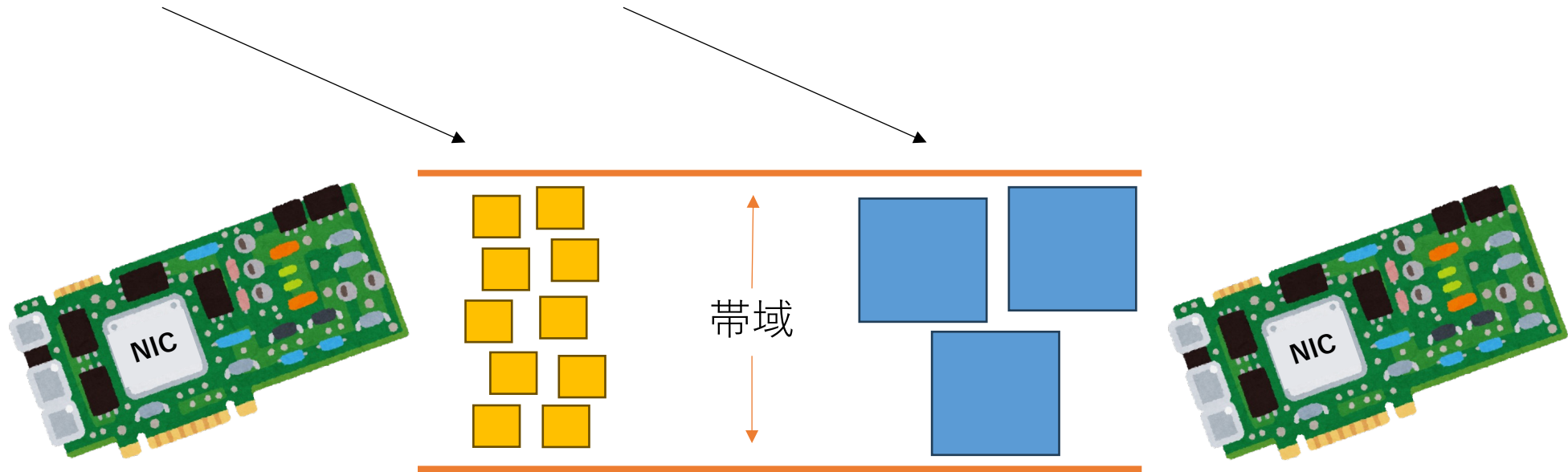
既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が大きい場合より多くのパケットを送れる



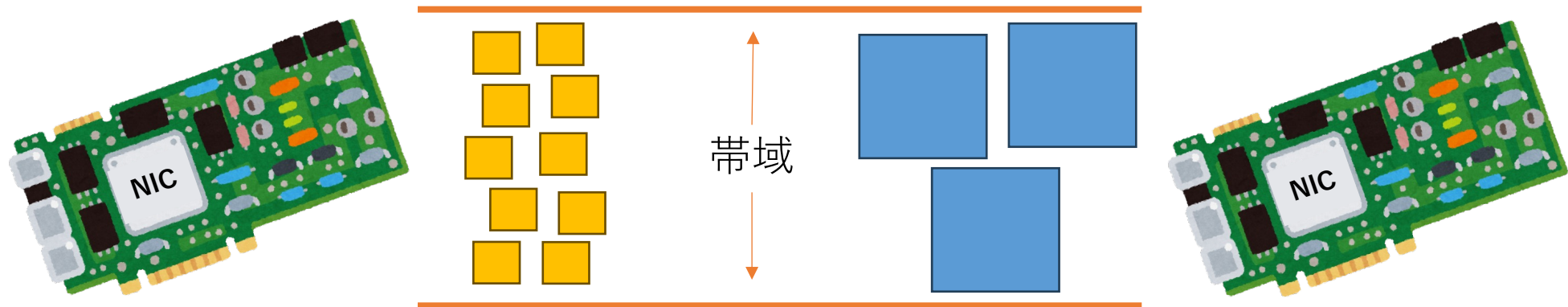
既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



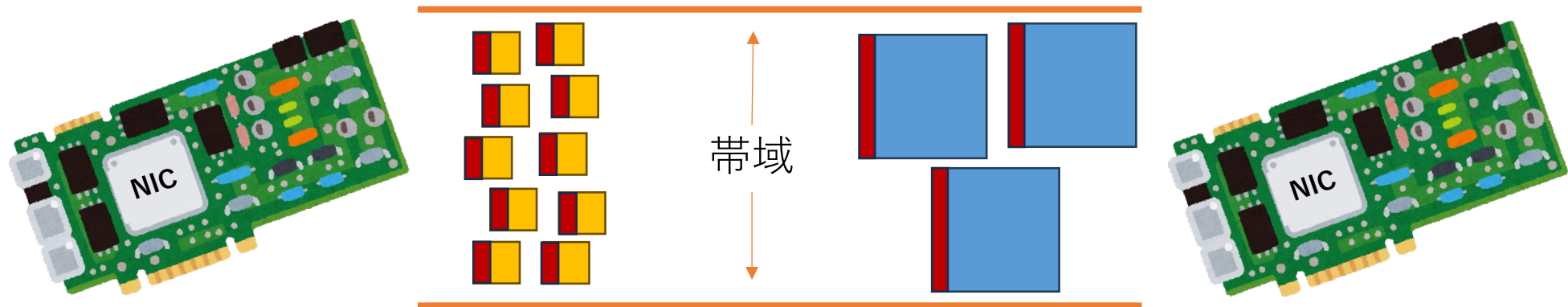
既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



既存の実装の性能

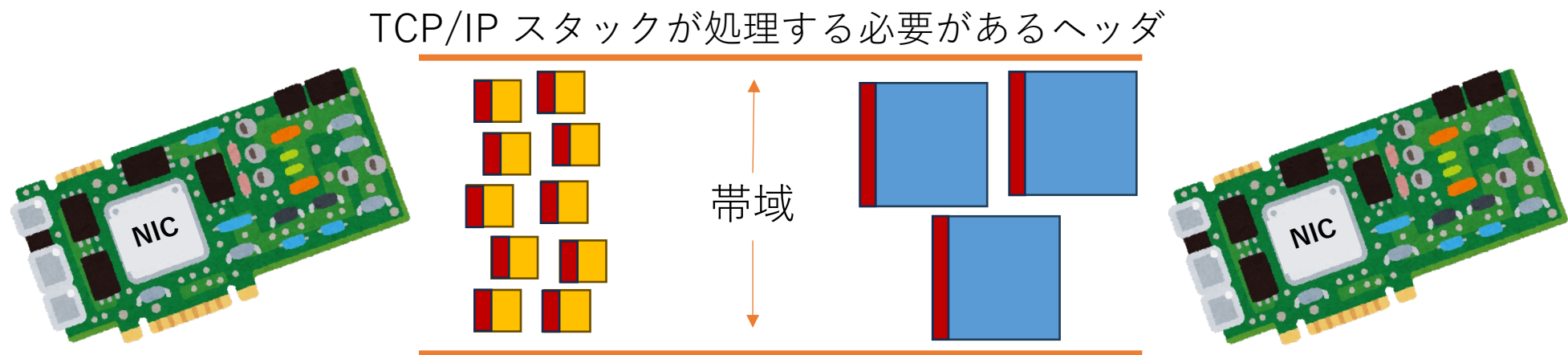
- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の実装の性能

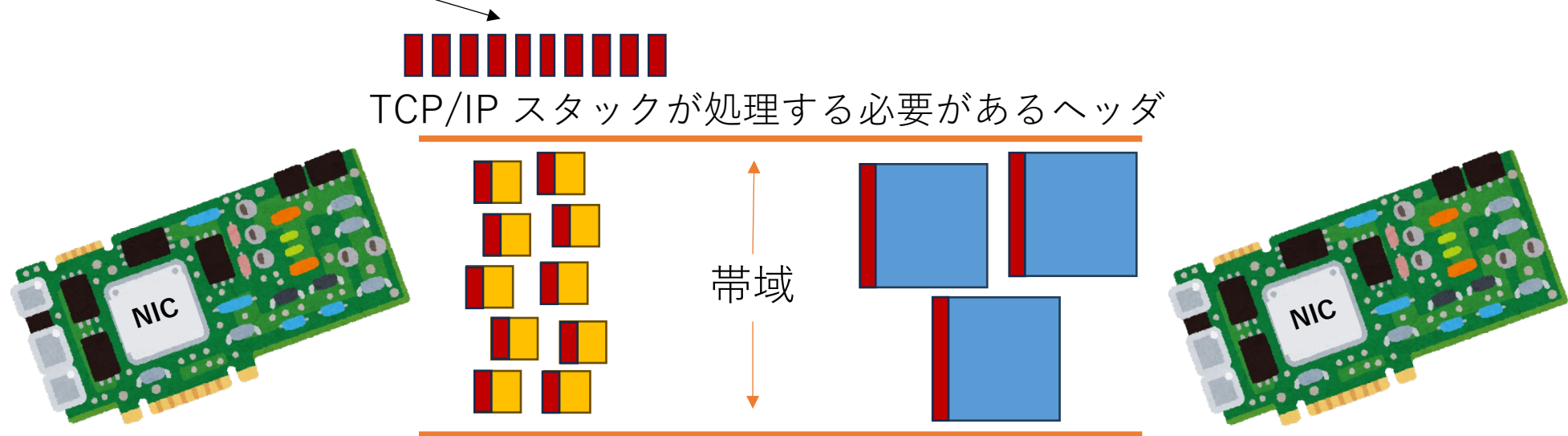
- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の実装の性能

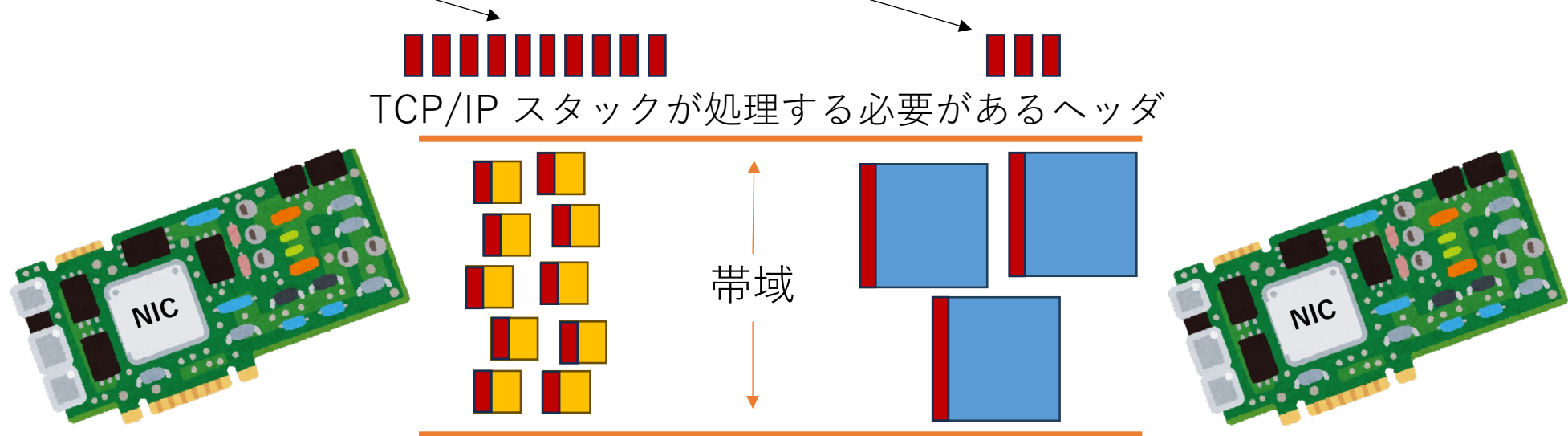
- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の実装の性能

- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる

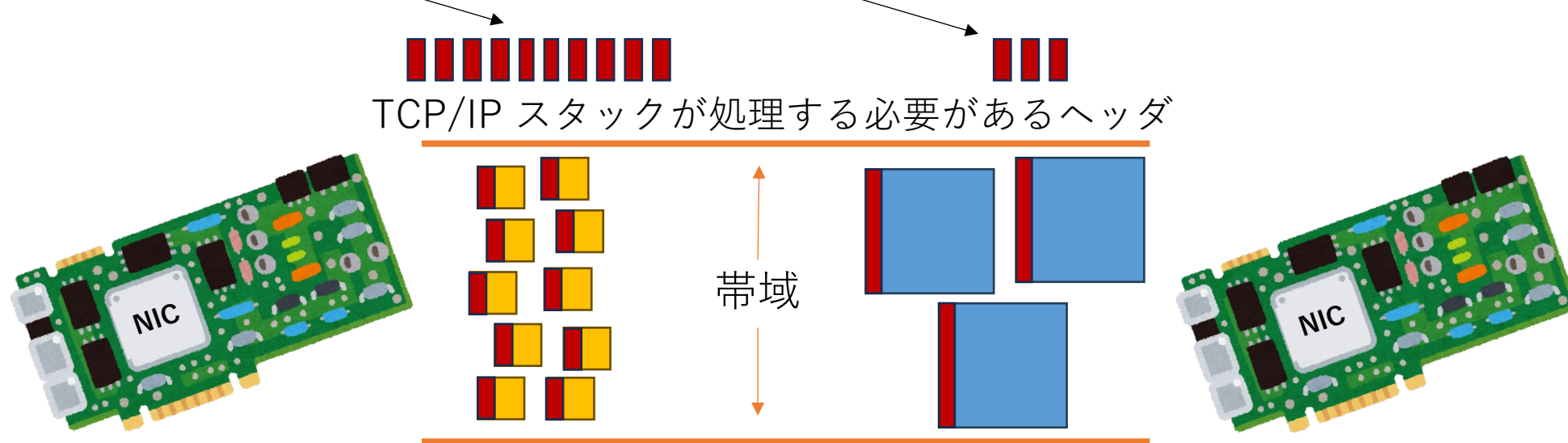


TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の実装の性能

パケットサイズが小さい方が
TCP/IP スタックにとって
最大の仕事の量が多くなる

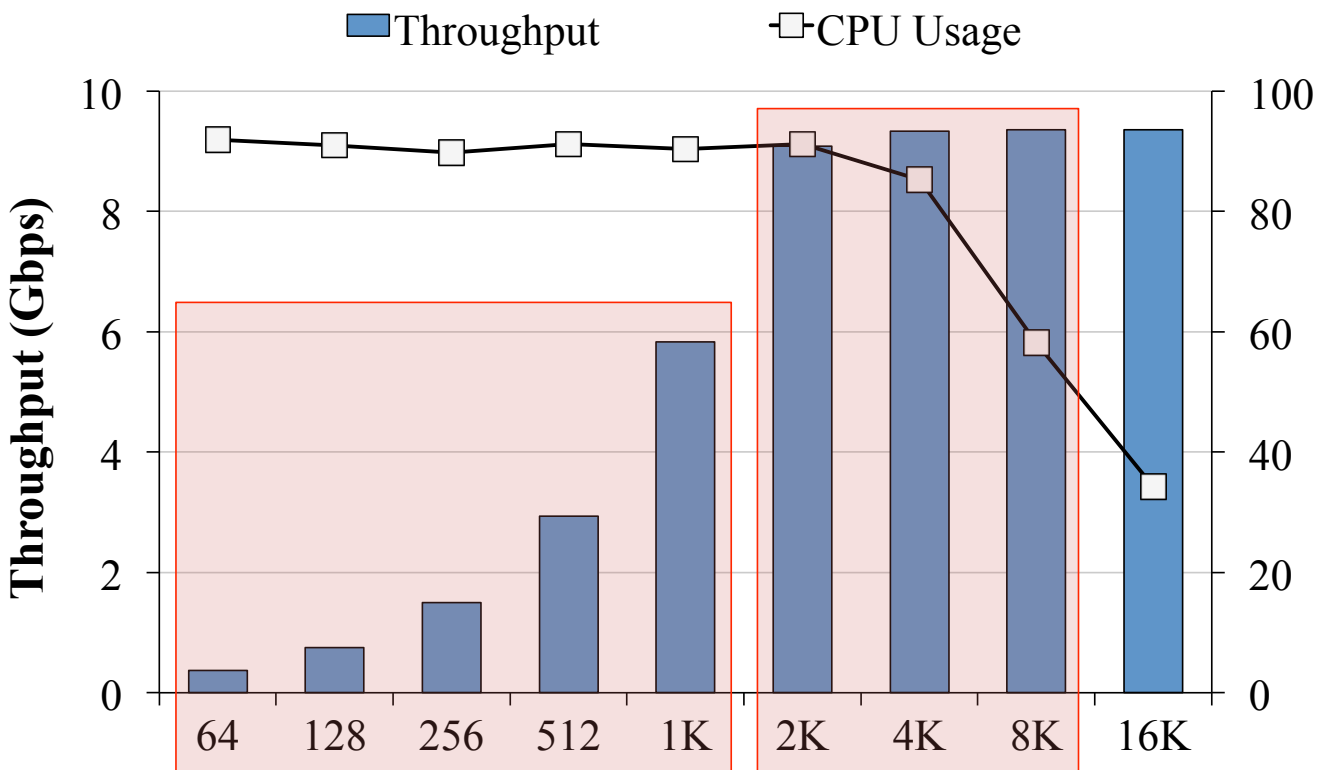
- CPU について考えると、TCP/IP スタックで費やされる時間は（厳密ではないですが概ね）パケット数に依存する
- NIC の特性上、同じ帯域でも、パケットのサイズが **小さい場合**の方が**大きい場合**より多くのパケットを送れる



TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の

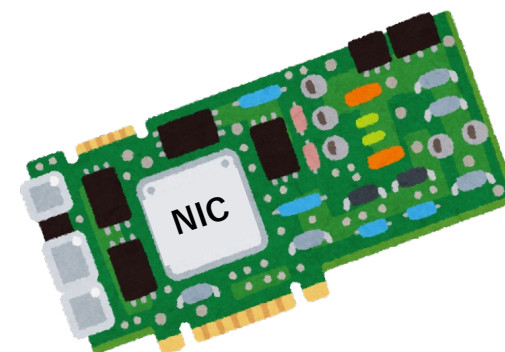
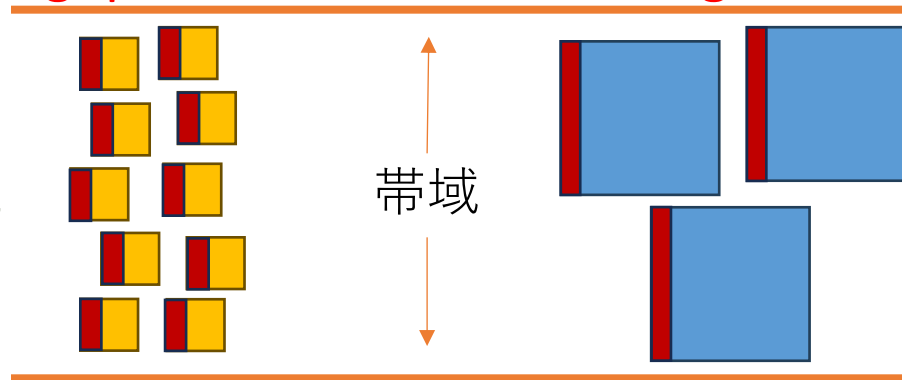
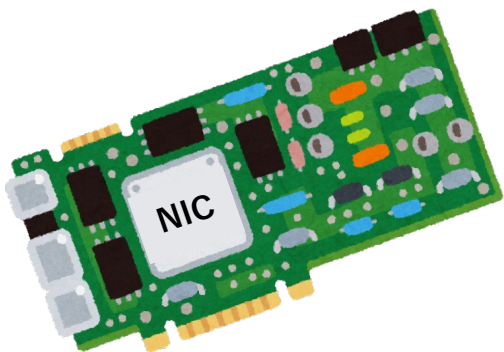
- CPU に
(厳密で
- NIC の特
小さい場



ズが小さい方が
ックにとって
)量が多くなる

られる時間は
)
ミ
:送れる

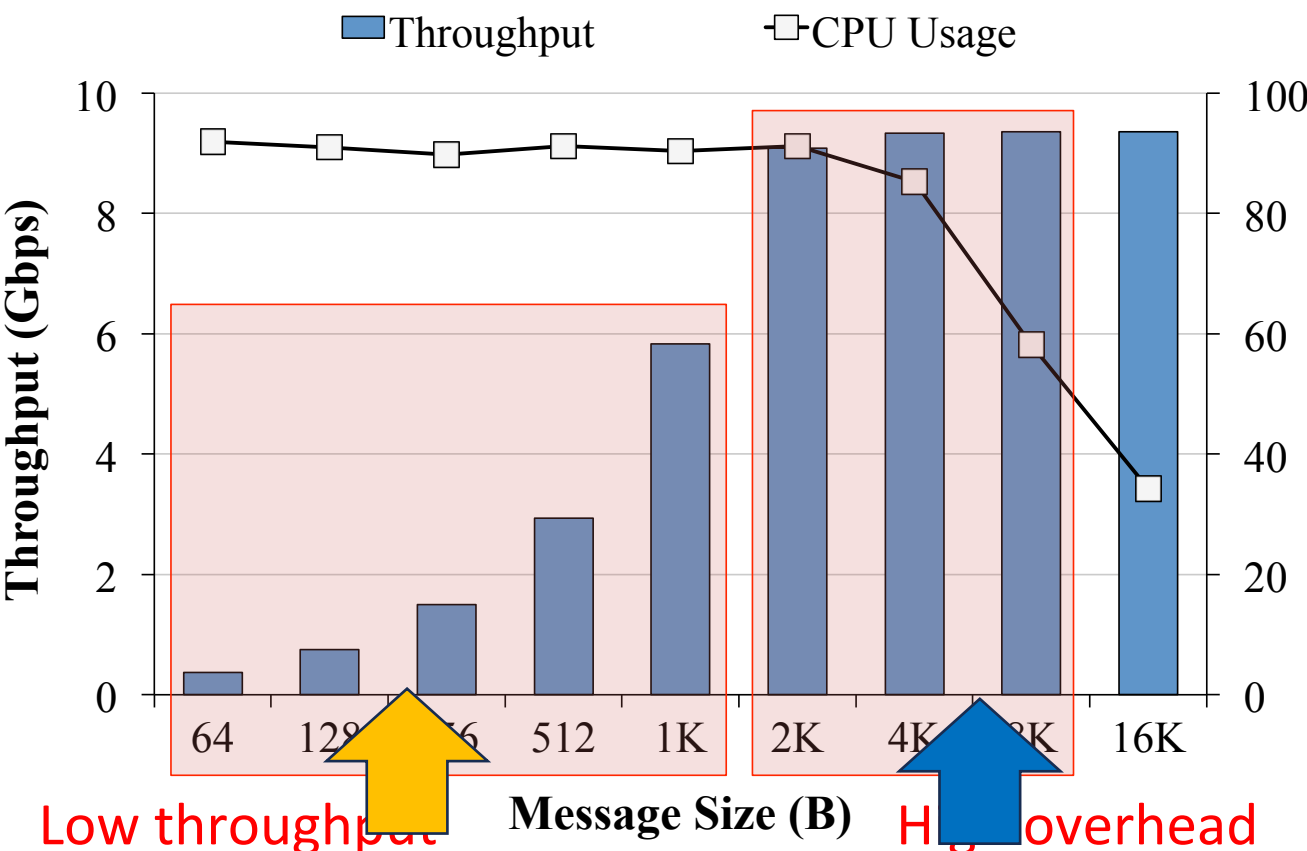
Low throughput Message Size (B) High overhead



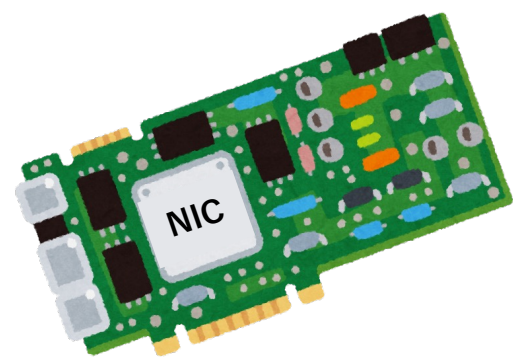
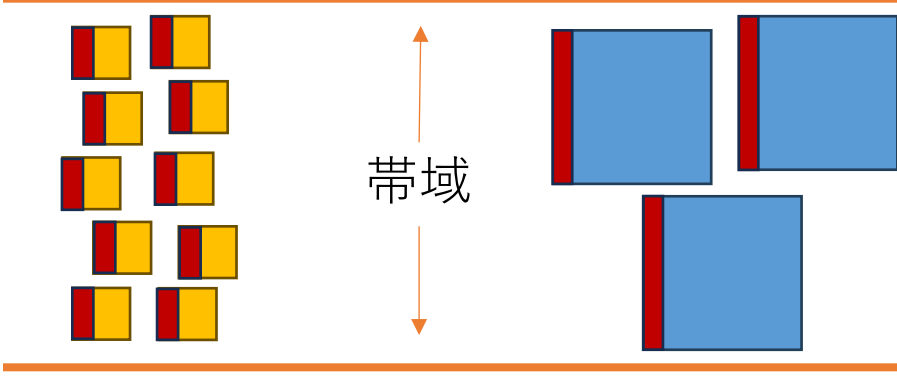
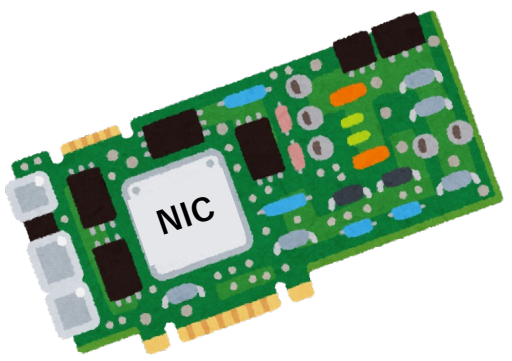
TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の

- CPU による処理 (厳密で)
- NIC の特別な小さい場合



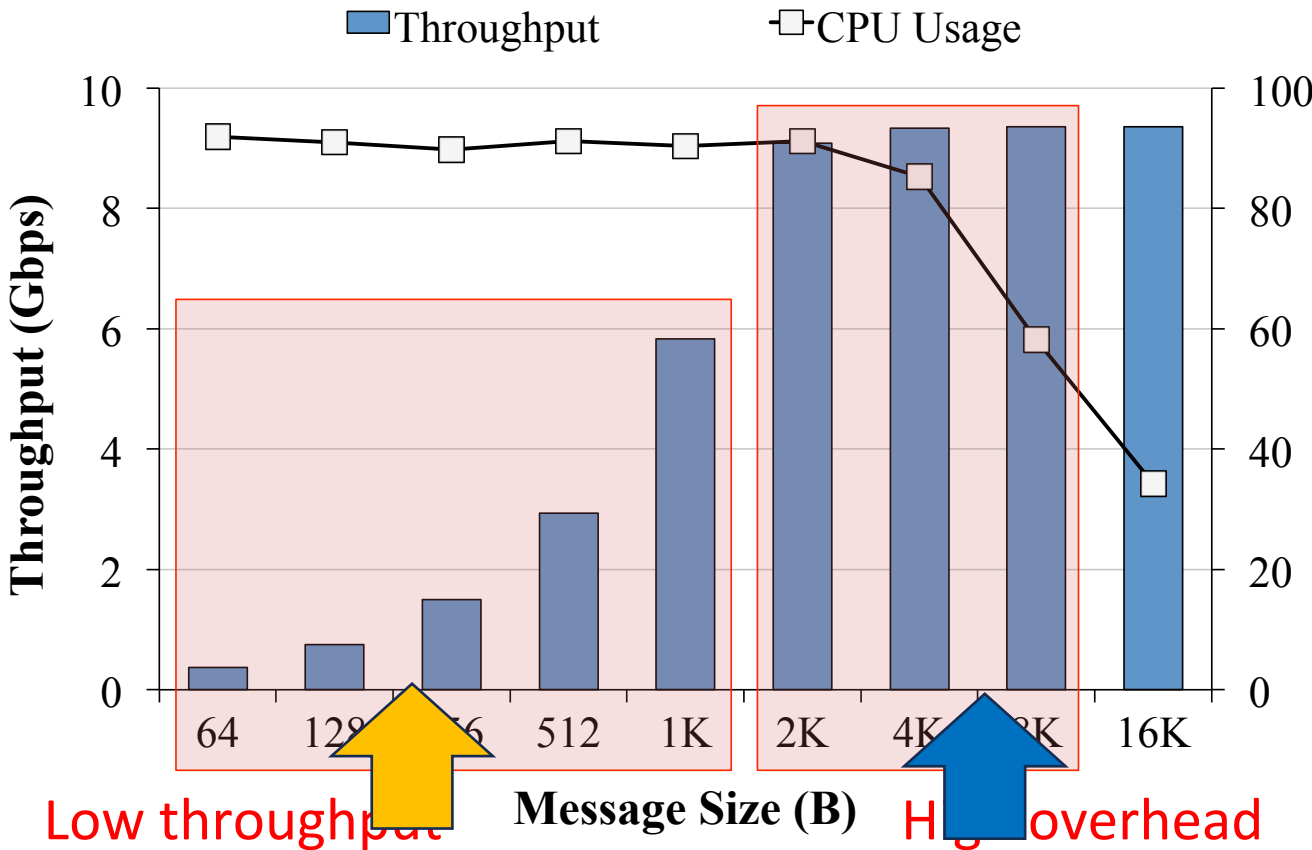
サイズが小さい方が
 パケットにとって
 処理量が多くなる
 送られる時間は
 長い
 送れる



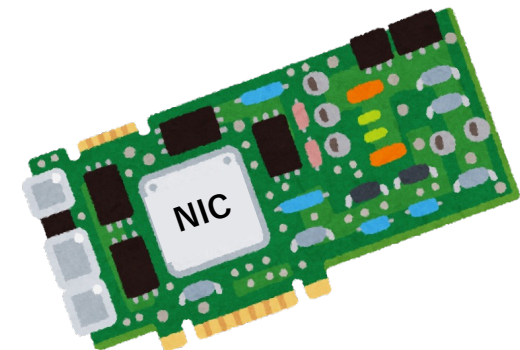
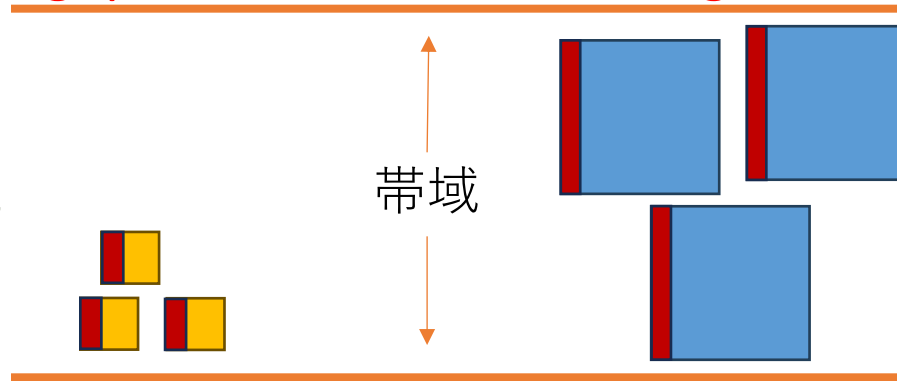
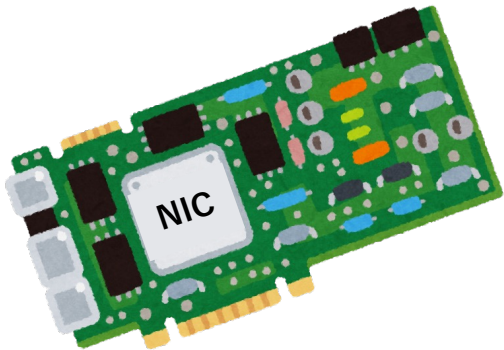
TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の

- CPU による処理 (厳密で)
- NIC の特殊な小さい場合



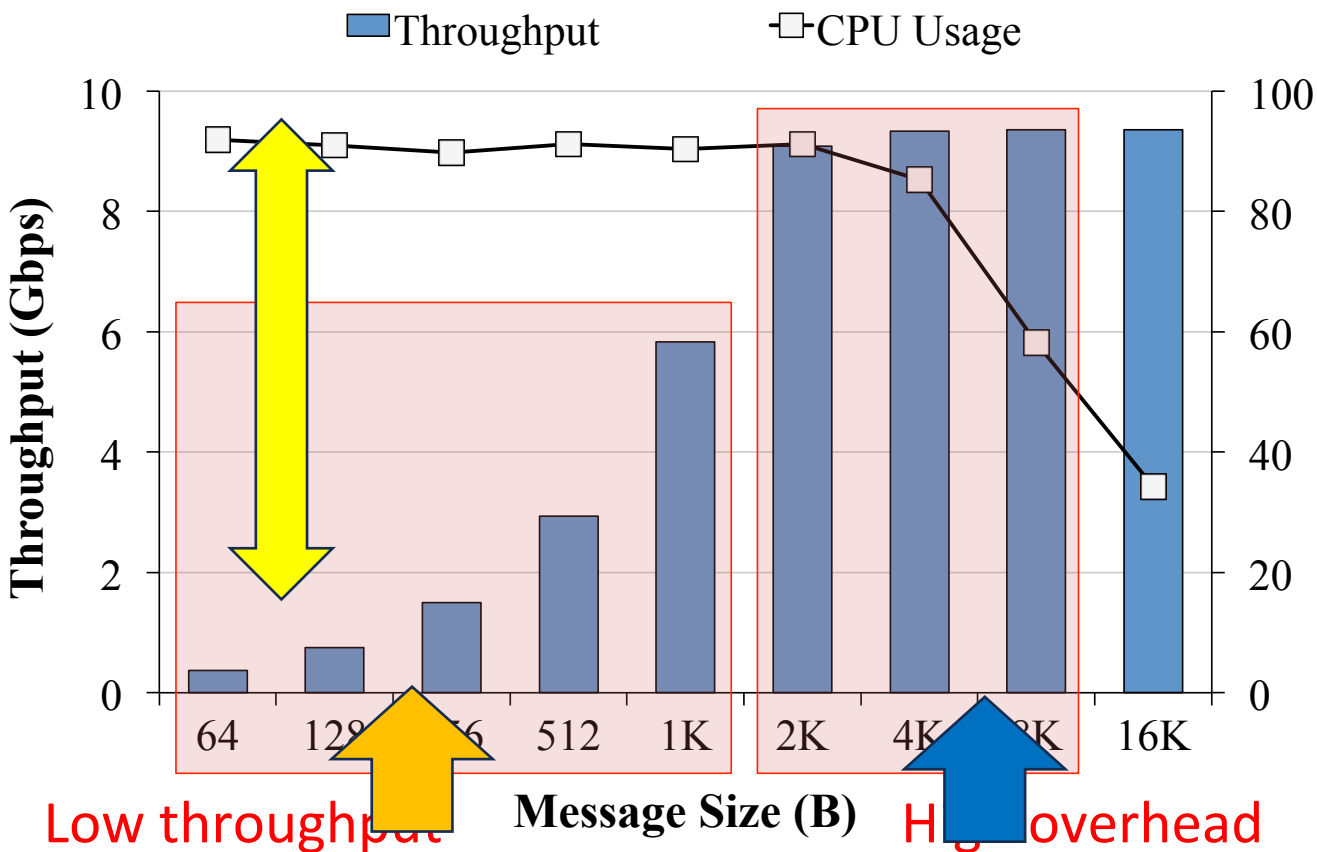
サイズが小さい方が
パケットにとって
オーバーヘッドが多くなる
送られる時間は
長くなる



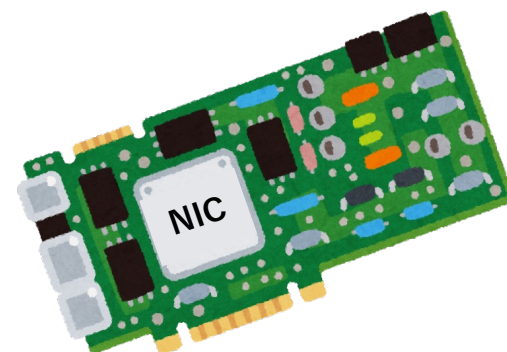
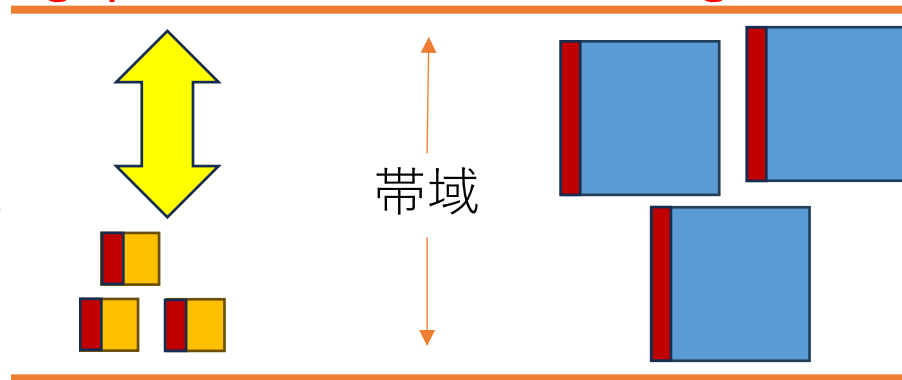
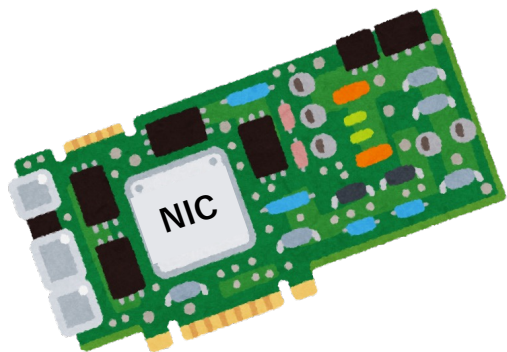
TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

既存の

- CPU に
(厳密で)
- NIC の特
小さい場



ズが小さい方が
ックにとって
)量が多くなる
れる時間は
送れる



TCP/IP ヘッダはパケットごとについているので、パケットごとに処理が必要

2010年くらいの利用シナリオ

サービス利用者の端末

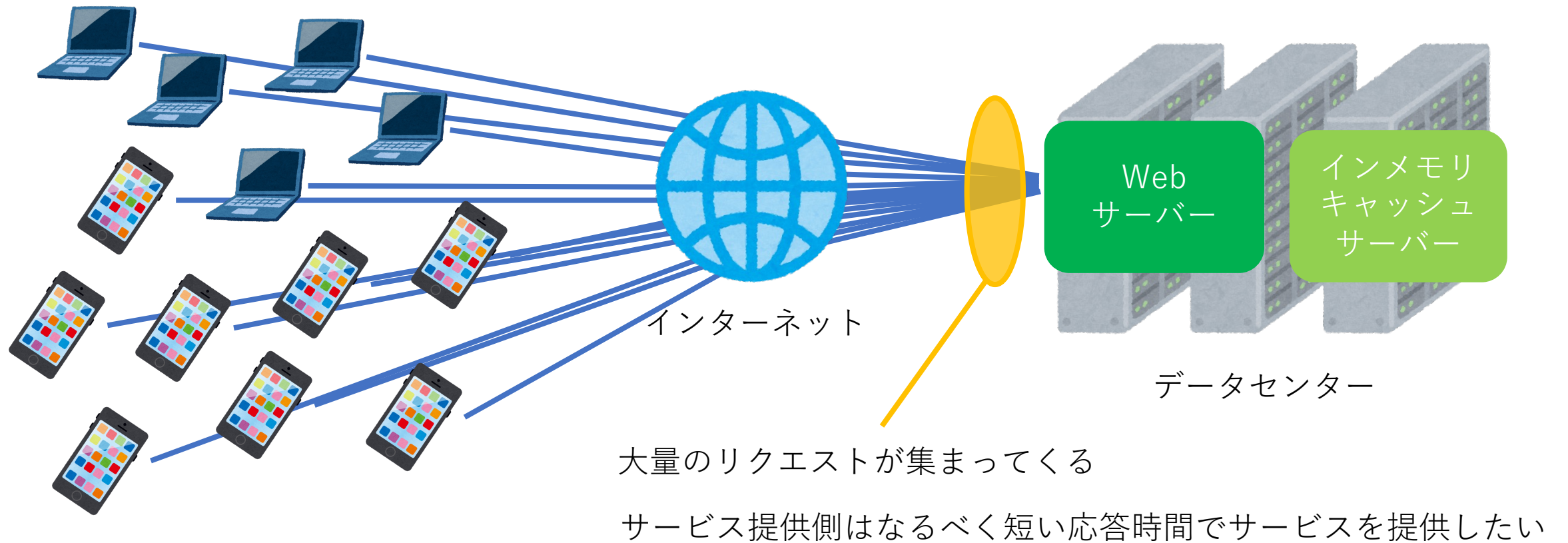
サービス提供側



2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側

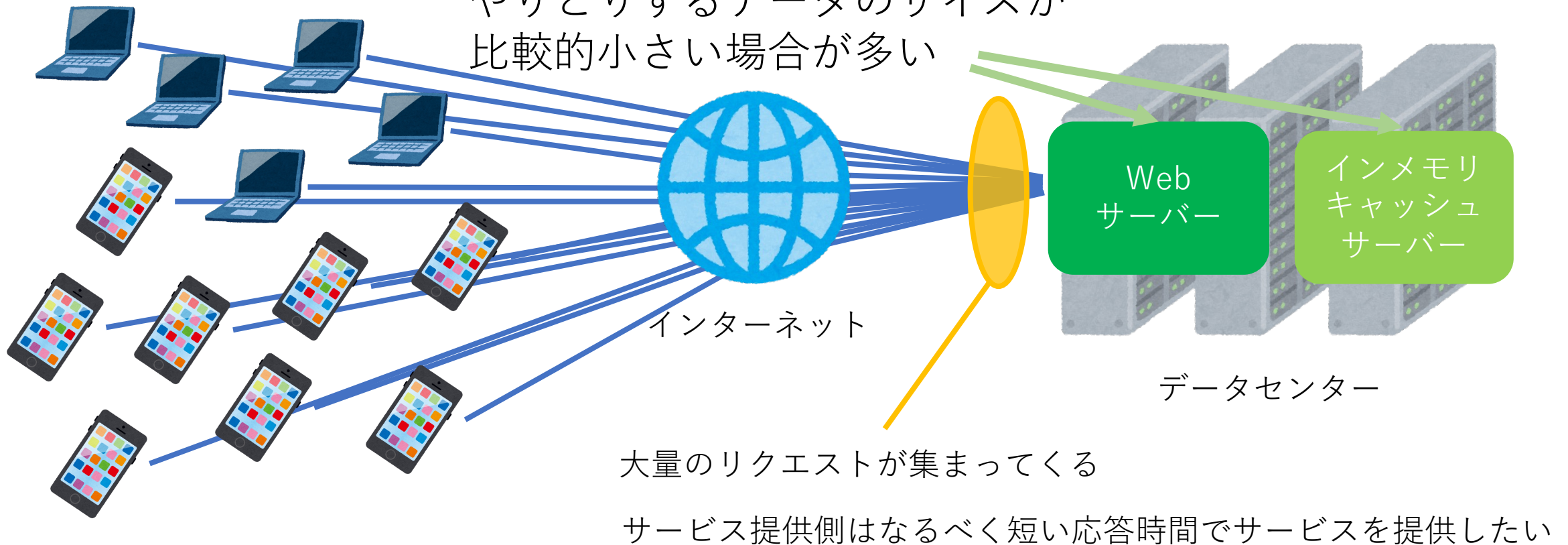


2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側

やりとりするデータのサイズが
比較的小さい場合が多い



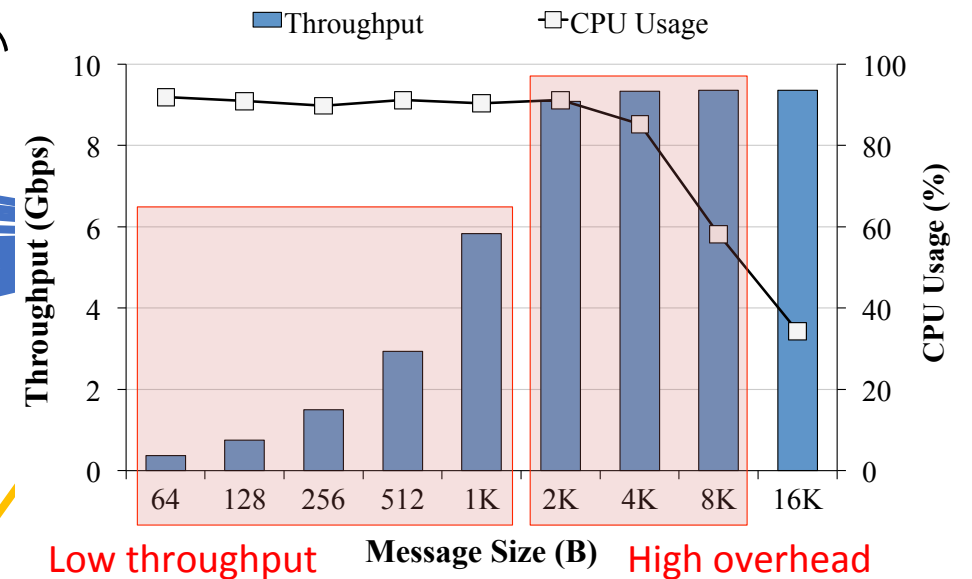
2010年くらいの利用シナリオ

サービス利用者の端末

やりとりするデータのサイズが比較的小さい場合が多い



サービス提供側



大量のリクエストが集まってくる

サービス提供側はなるべく短い応答時間でサービスを提供したい

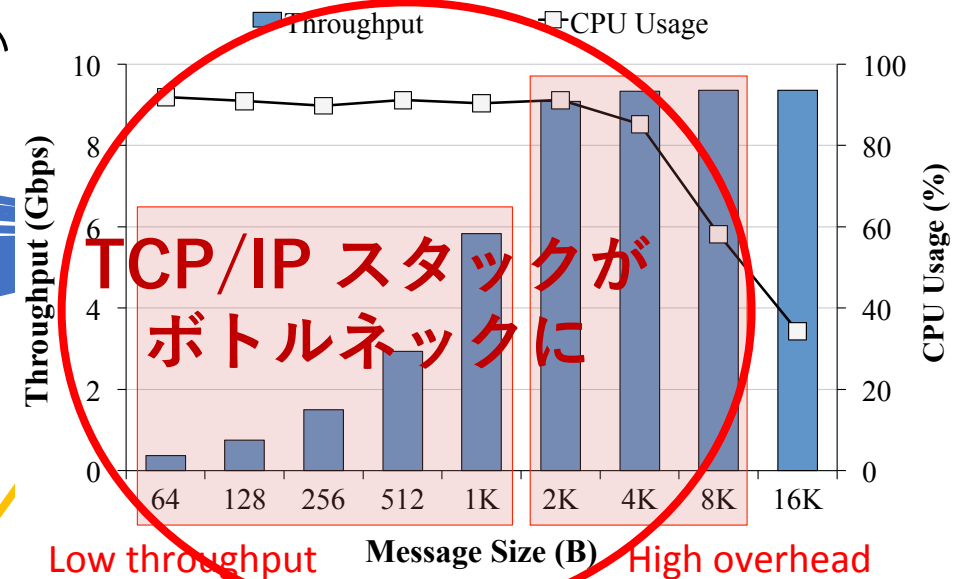
2010年くらいの利用シナリオ

サービス利用者の端末

やりとりするデータのサイズが比較的小さい場合が多い



サービス提供側

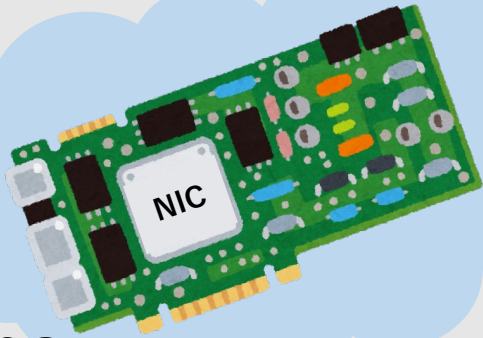


大量のリクエストが集まってくる

サービス提供側はなるべく短い応答時間でサービスを提供したい

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



ハードウェア (NIC) は速くなったので
ソフトウェアの効率が重要になる

通信関連ソフトウェア

ユーザー空間

アプリケーション

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

ホスト

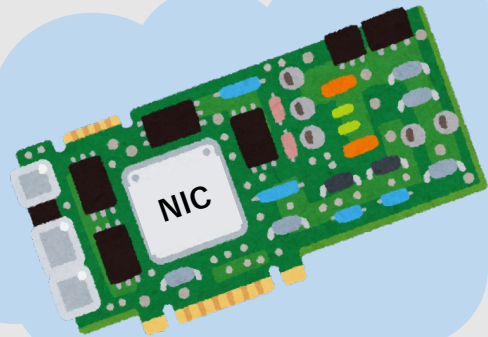
仮想 NIC バックエンド

仮想スイッチ

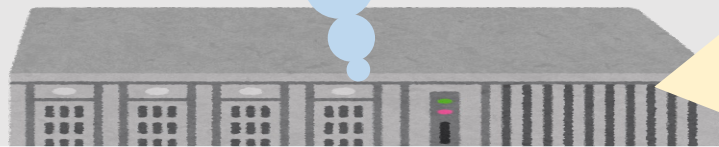
NIC デバイスドライバ

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



ハードウェア (NIC) は速くなったので
ソフトウェアの効率が重要になる

通信関連ソフトウェア

ユーザー空間

アプリケーション

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

ホスト

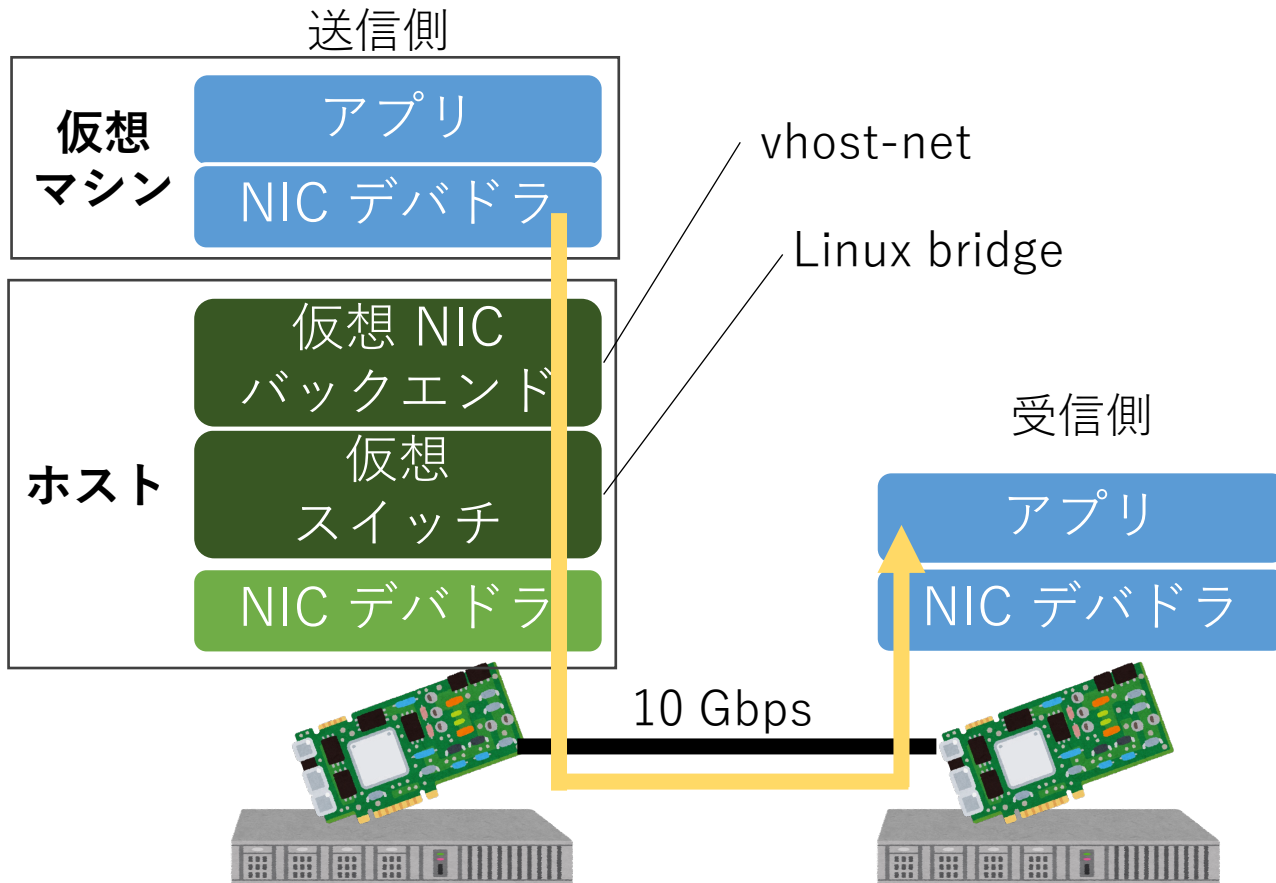
仮想 NIC バックエンド

仮想スイッチ

NIC デバイスドライバ

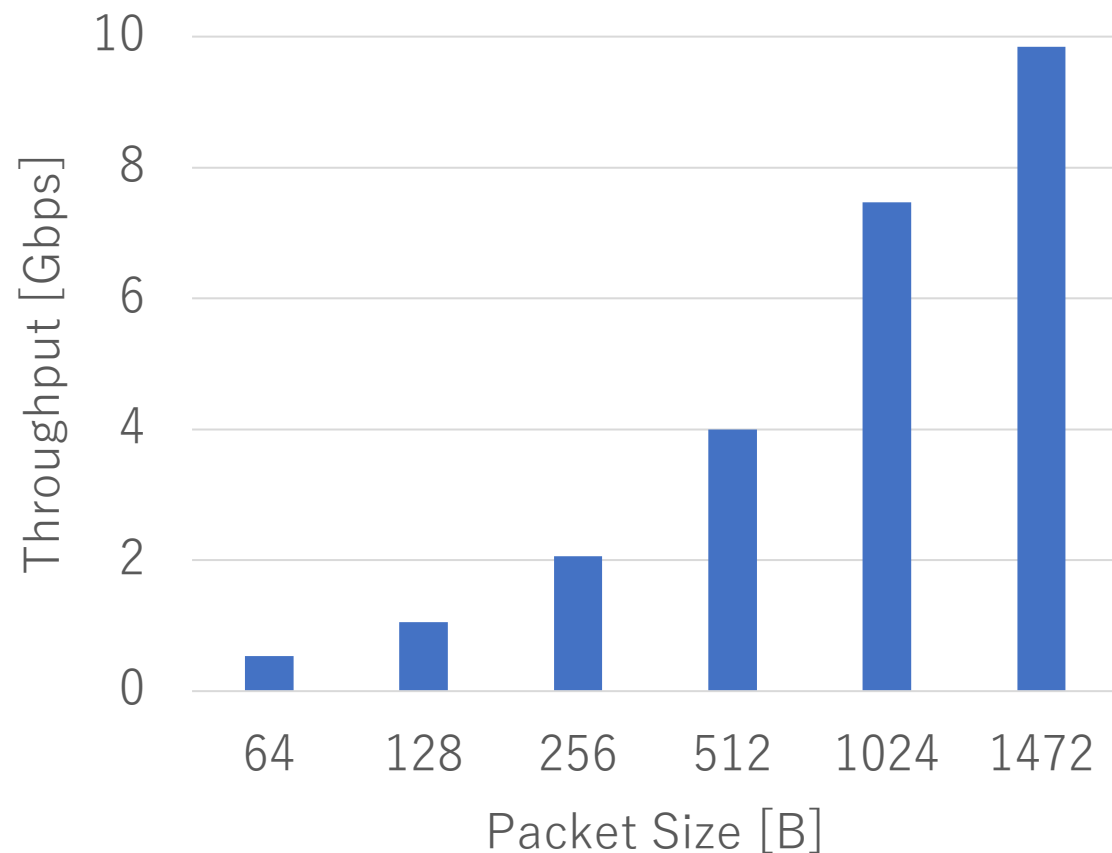
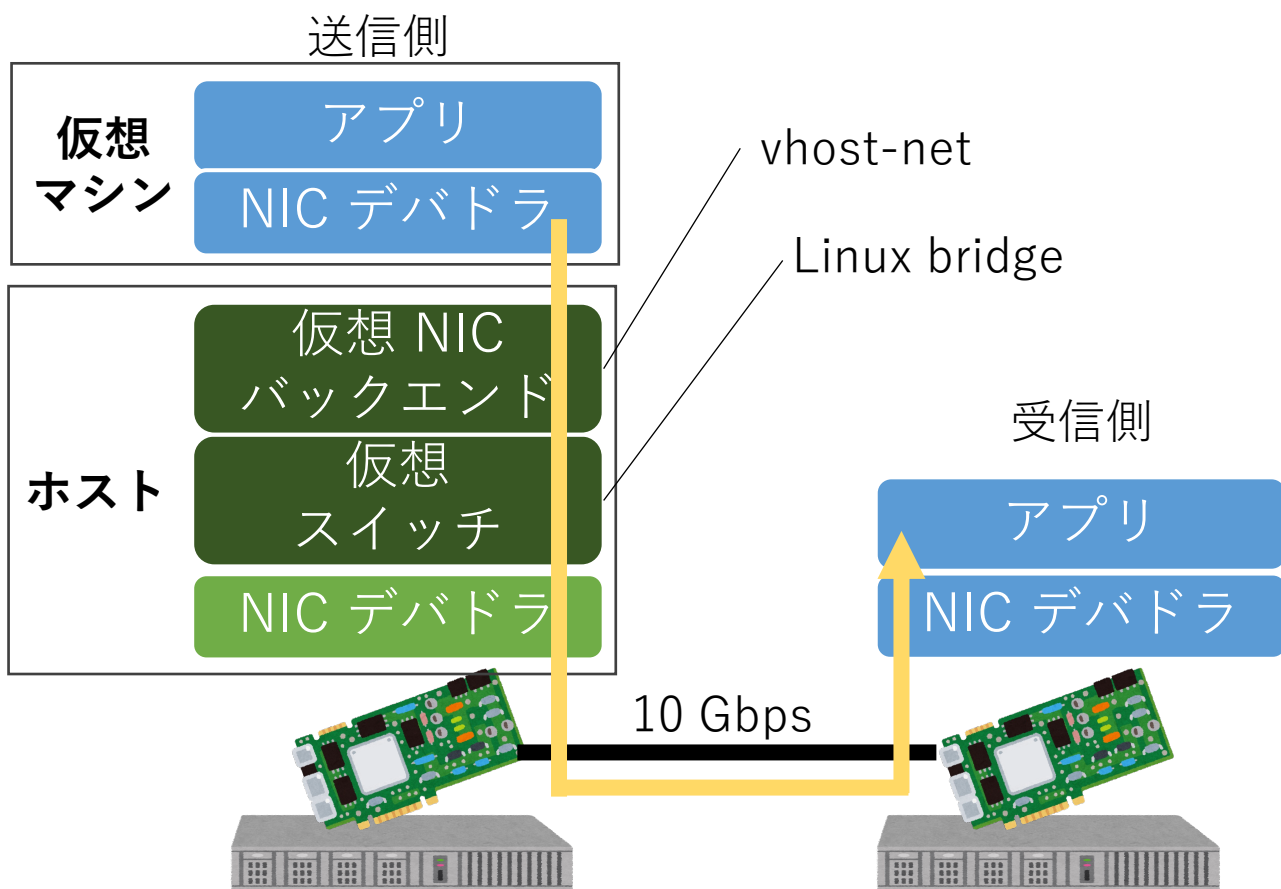
既存の実装の性能

- 仮想マシンからの単純なパケット転送性能：Linux vhost-net



既存の実装の性能

- 仮想マシンからの単純なパケット転送性能：Linux vhost-net

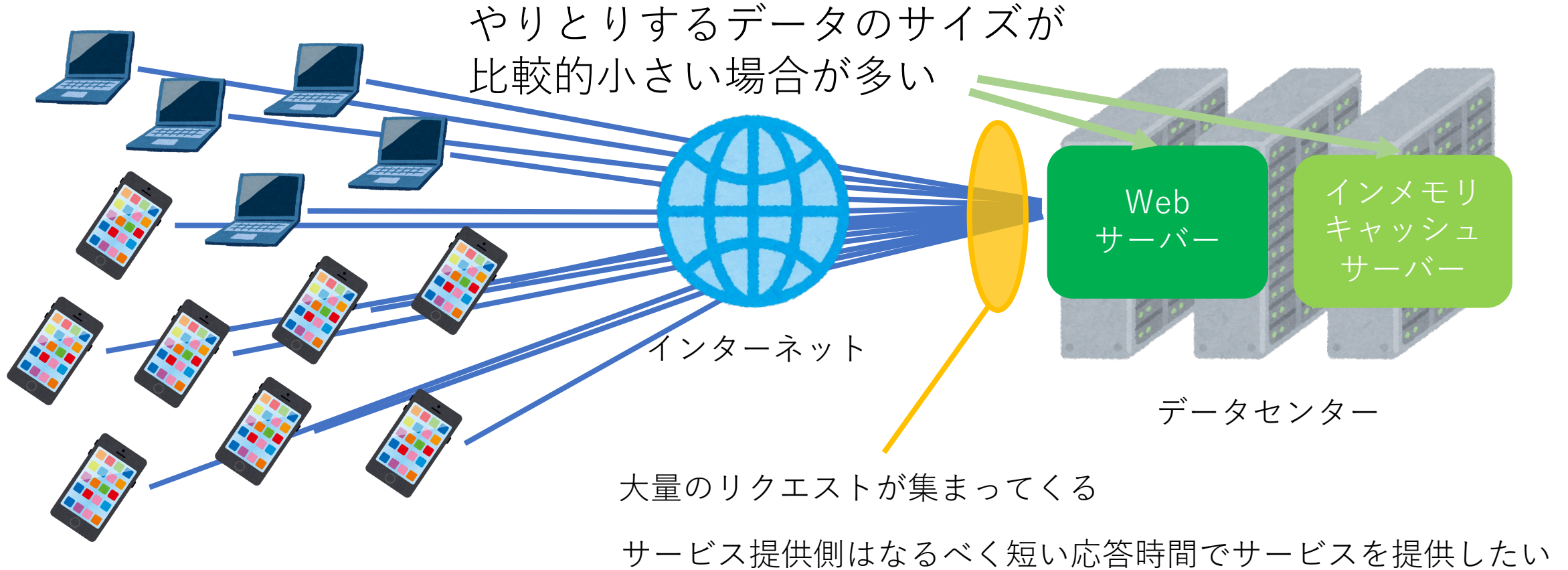


2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側

やりとりするデータのサイズが比較的小さい場合が多い

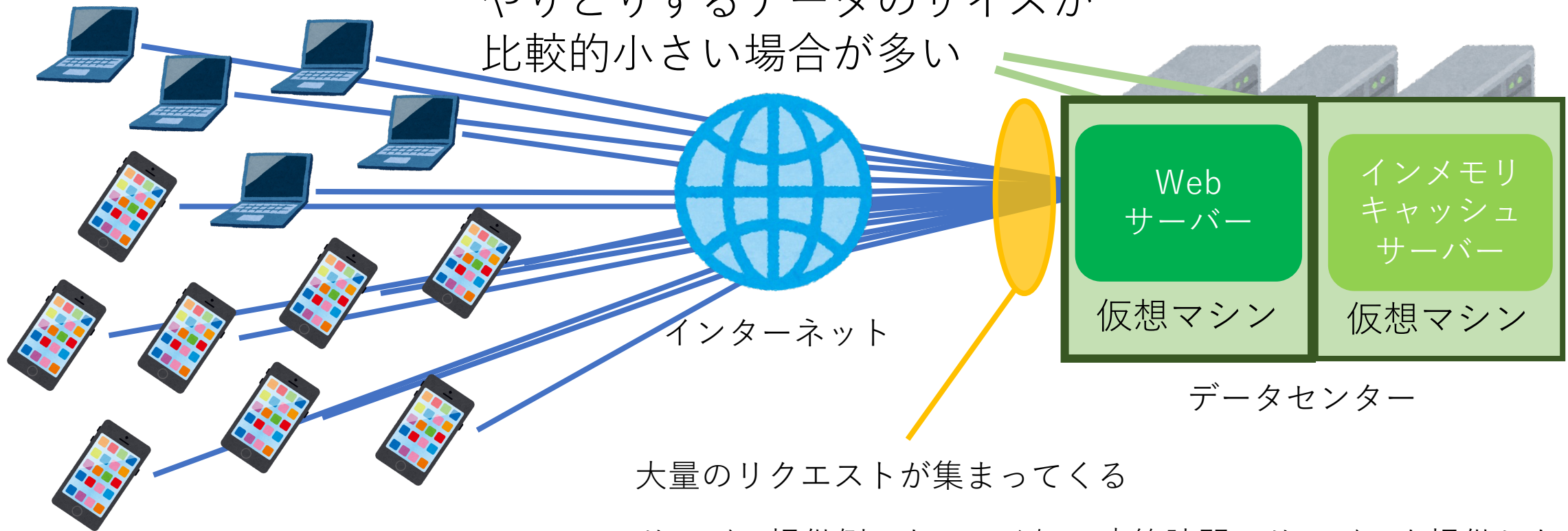


2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側

やりとりするデータのサイズが
比較的小さい場合が多い



インターネット

Web
サーバー

仮想マシン

インメモリ
キャッシュ
サーバー

仮想マシン

データセンター

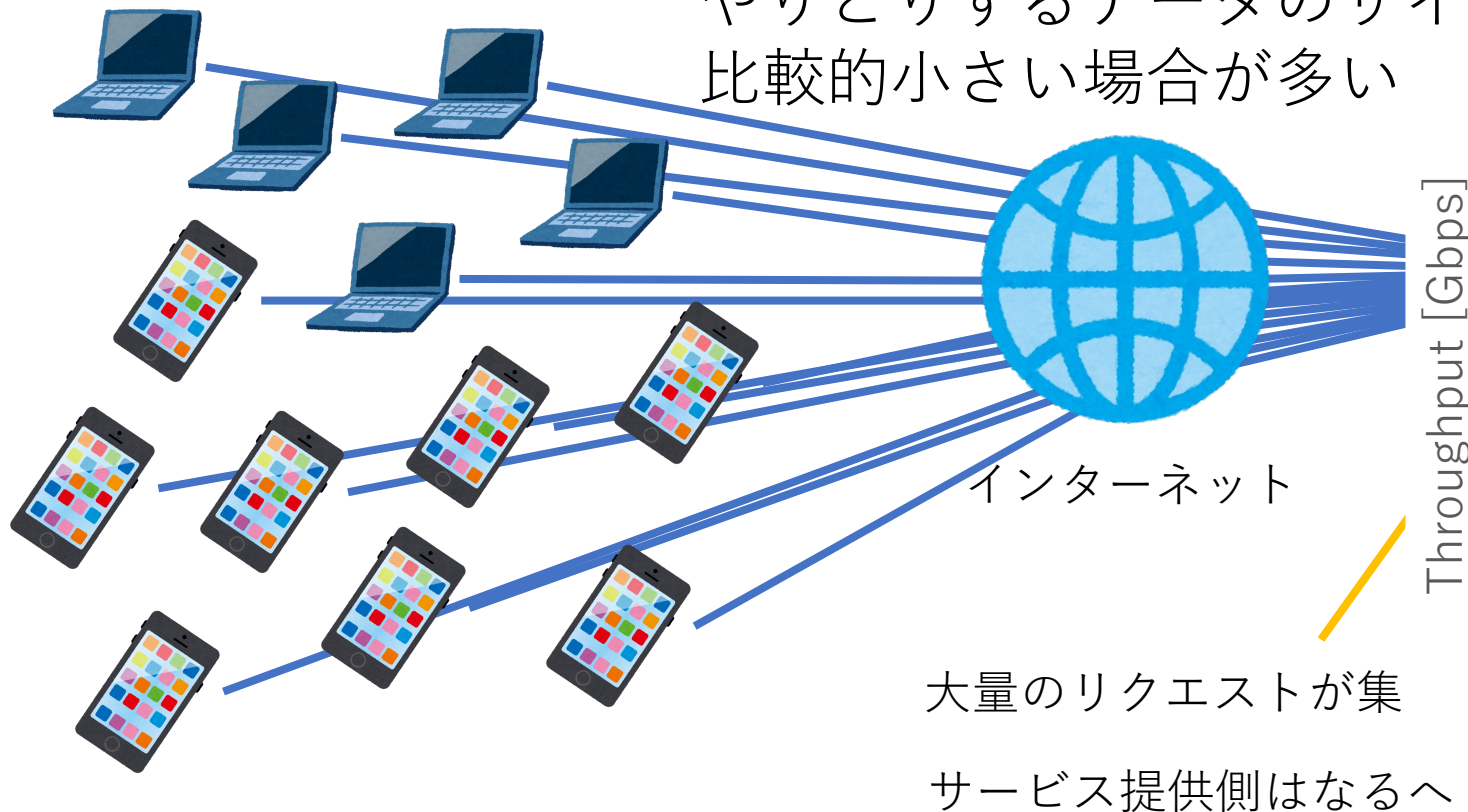
大量のリクエストが集まってくる

サービス提供側はなるべく短い応答時間でサービスを提供したい

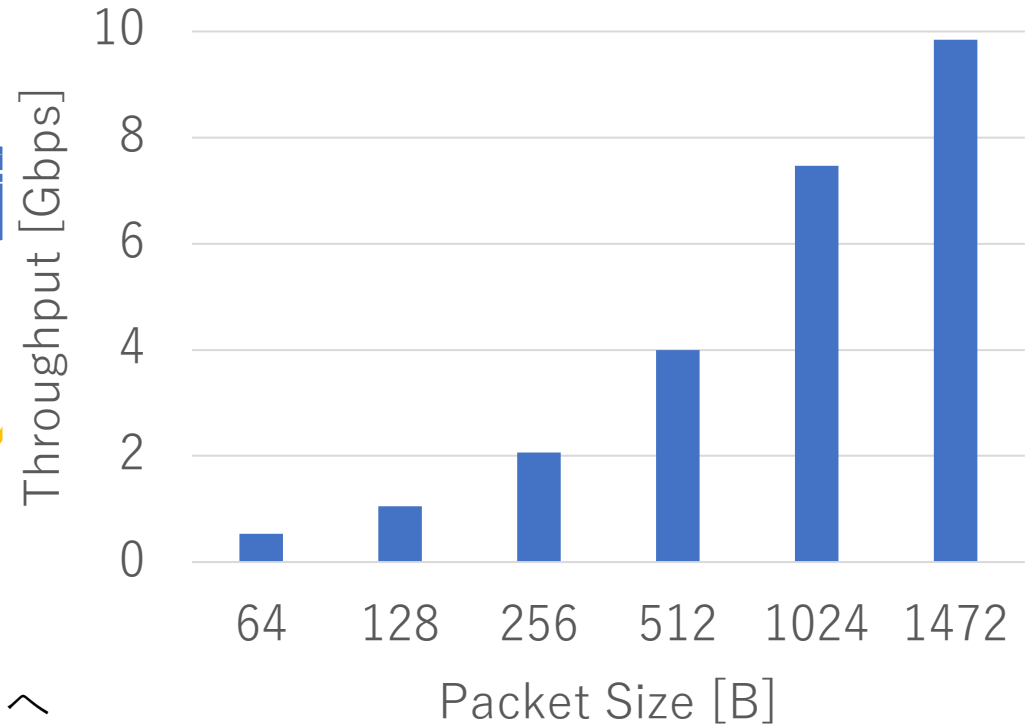
2010年くらいの利用シナリオ

サービス利用者の端末

やりとりするデータのサイズが比較的小さい場合が多い



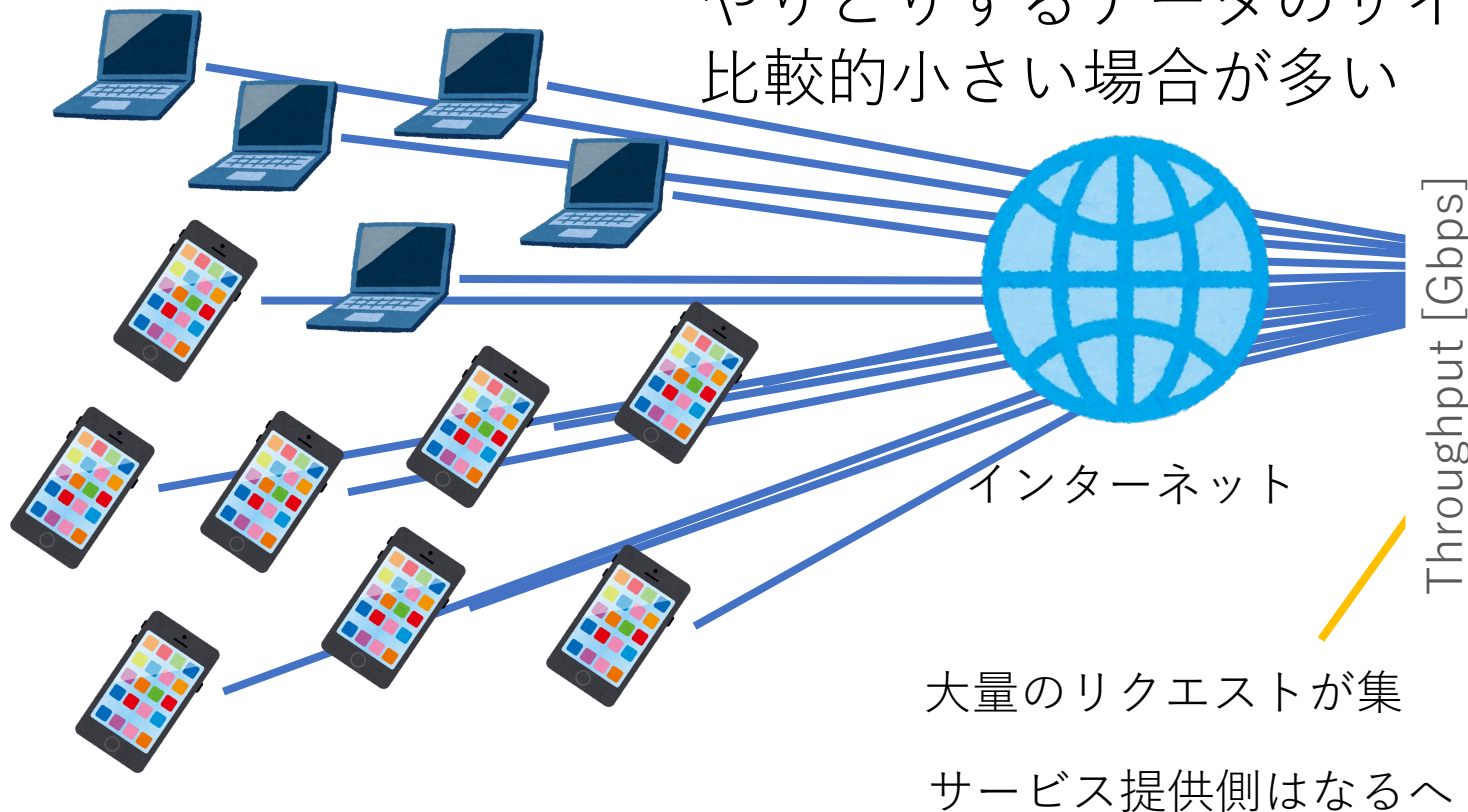
サービス提供側



2010年くらいの利用シナリオ

サービス利用者の端末

やりとりするデータのサイズが
比較的小さい場合が多い



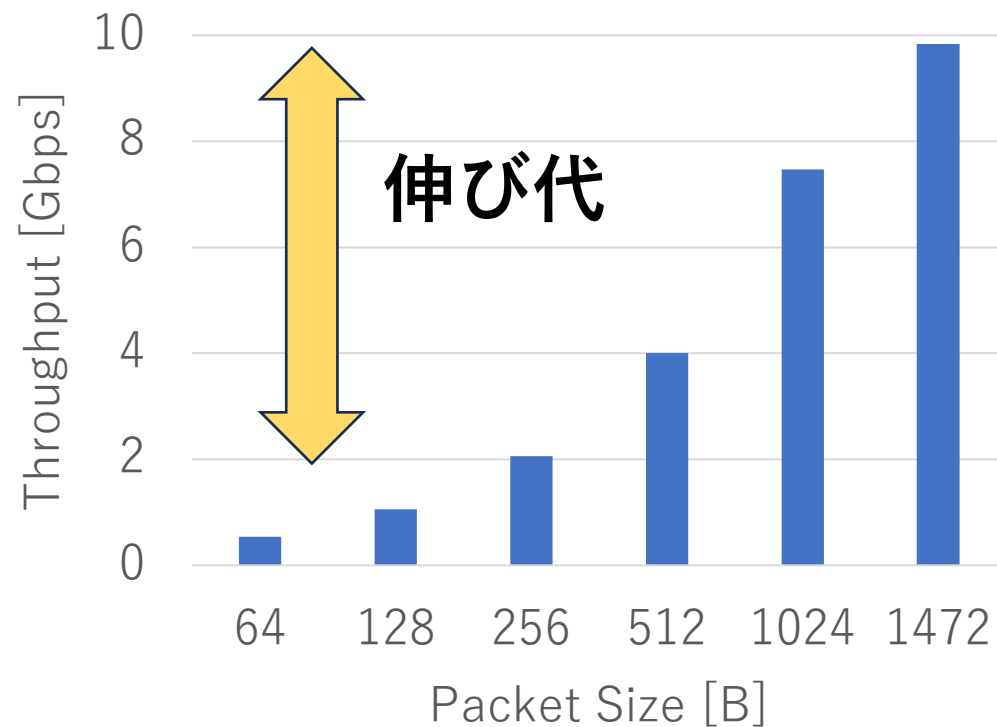
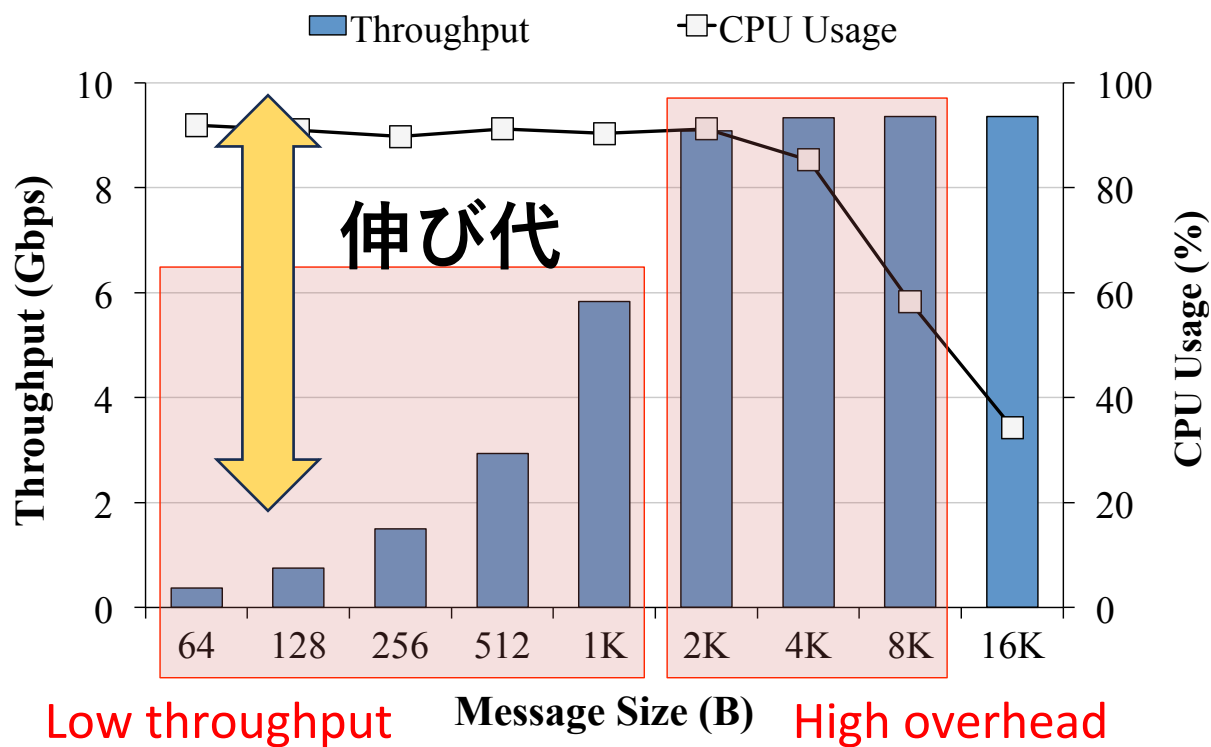
サービス提供側

仮想マシンのI/Oが
ボトルネックに



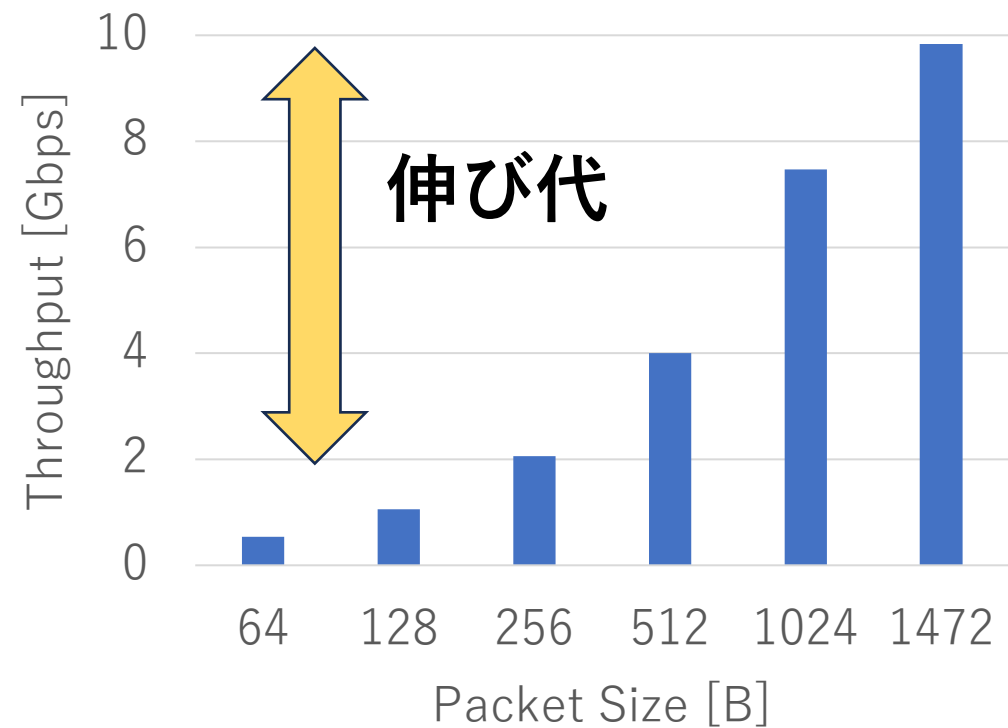
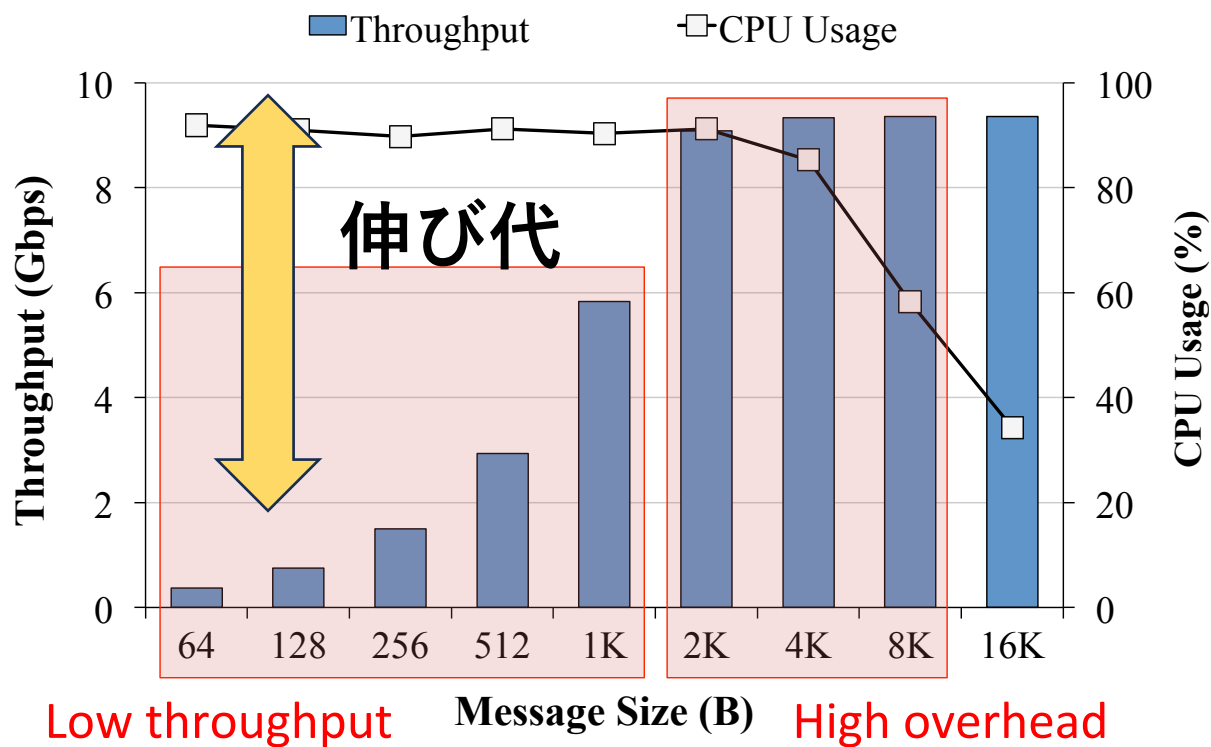
10 Gbps NIC の普及

- ソフトウェアの視点から、性能について大きな伸び代ができた



課題

- 伸び代をどのように引き出して有効活用するか？



研究紹介

システムコール呼び出しコストについて

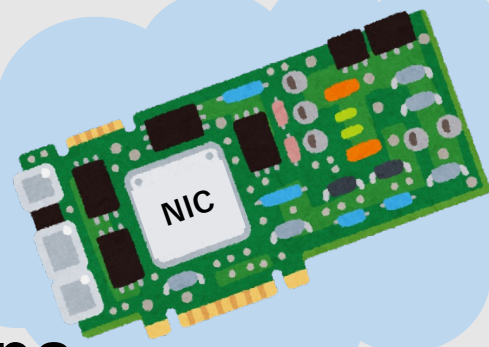
基本的な仕組みの説明

システムコールの呼び出しコスト

- システムコール
 - ユーザー空間プログラムがカーネル空間の機能呼び出すためのインターフェース

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps

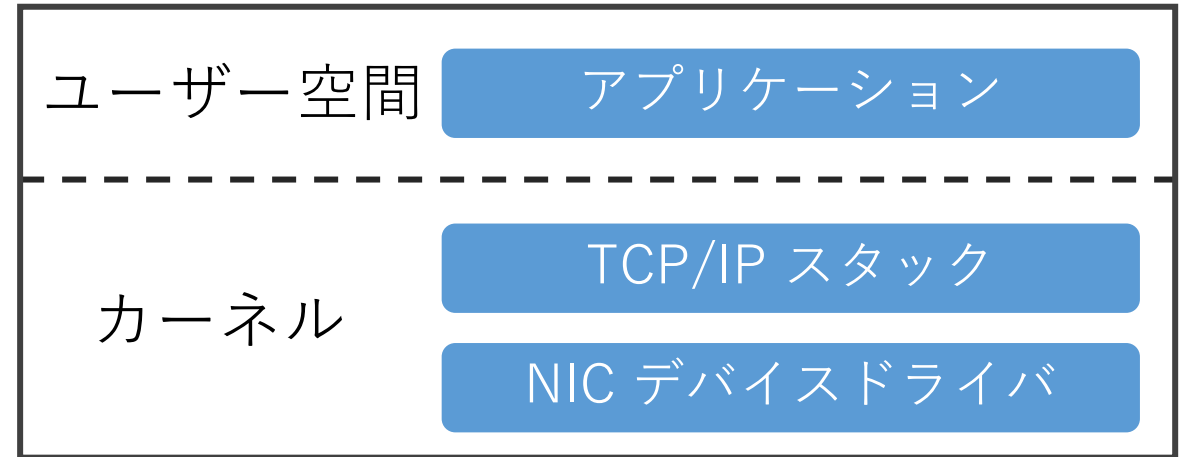


多重のリク
サービス提供保

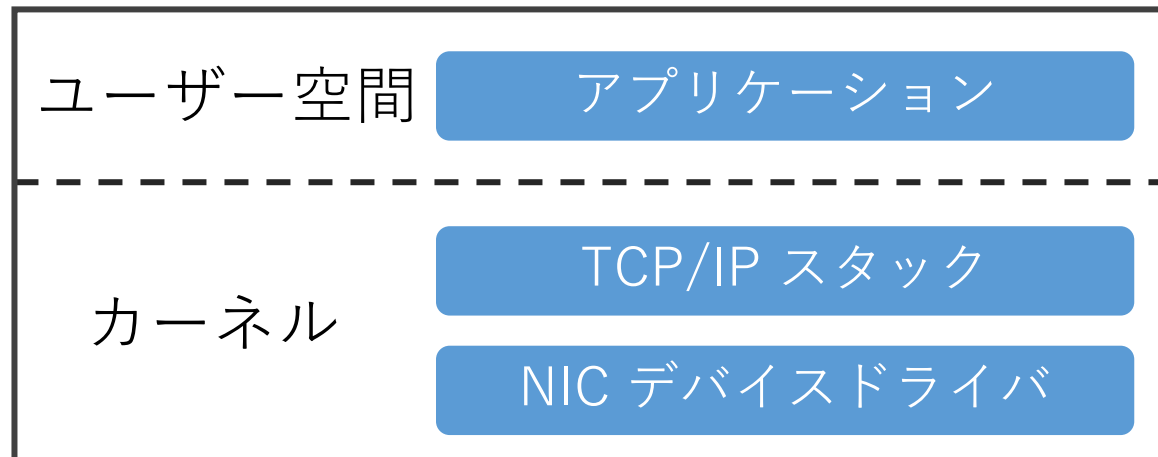
通信関連ソフトウェア



通信関連のシステムソフトウェア

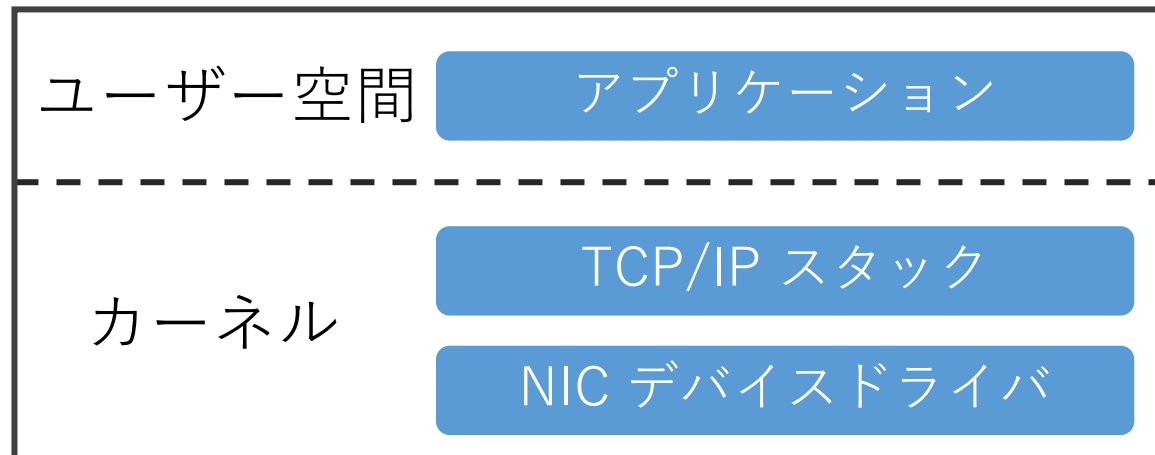


システムコールの呼び出しコスト



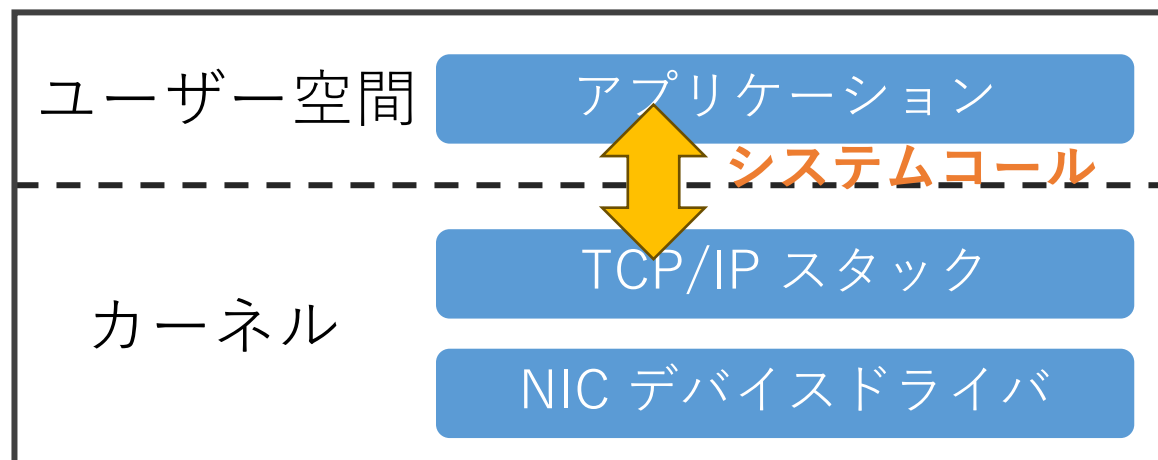
システムコールの呼び出しコスト

- システムコール
 - ユーザー空間プログラムがカーネル空間の機能呼び出すためのインターフェース
- ユーザー空間プログラムはカーネルに実装されている TCP/IP スタックを、システムコールを通して利用する



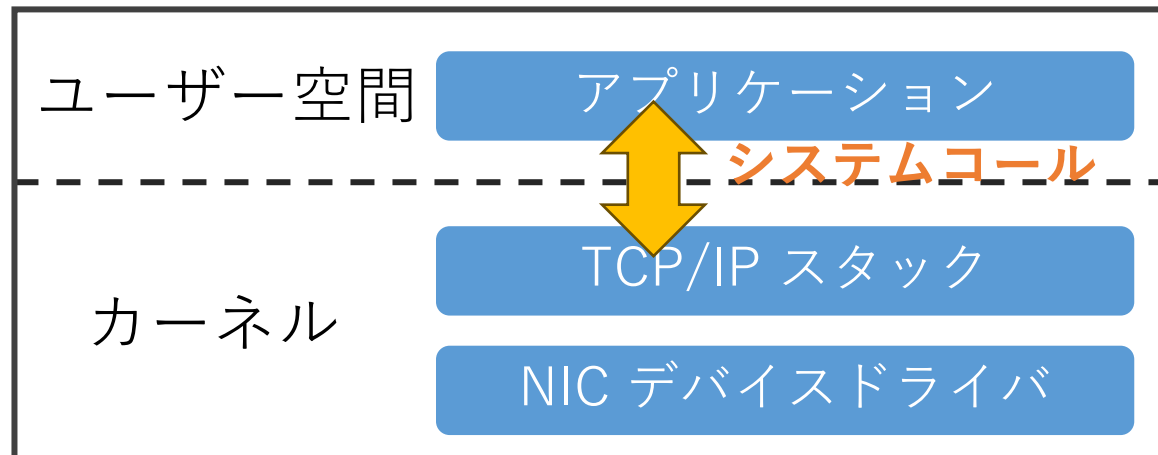
システムコールの呼び出しコスト

- システムコール
 - ユーザー空間プログラムがカーネル空間の機能呼び出すためのインターフェース
- ユーザー空間プログラムはカーネルに実装されている TCP/IP スタックを、システムコールを通して利用する



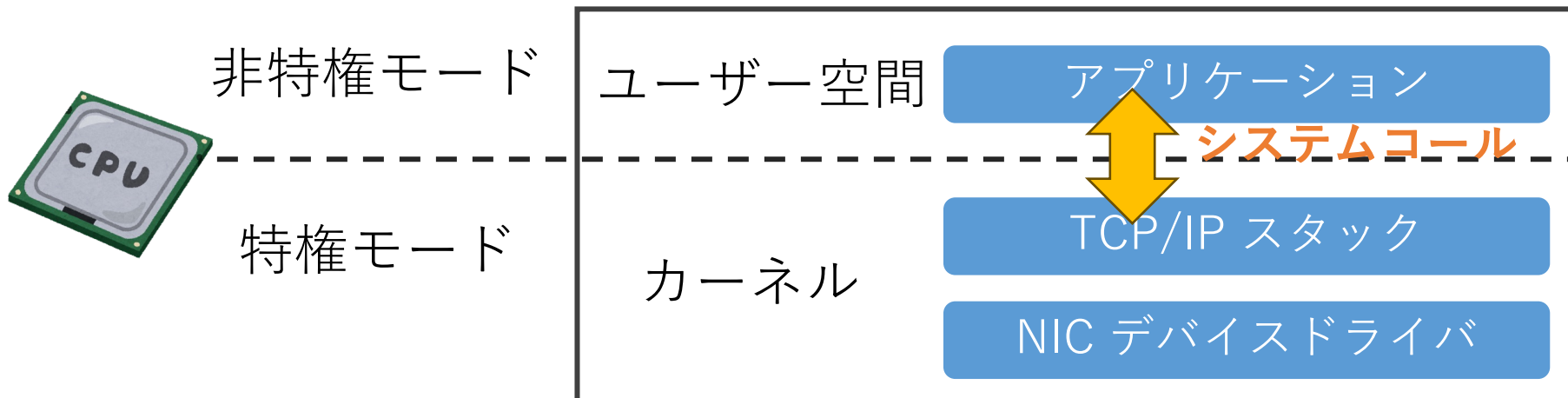
システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い



システムコールの呼び出しコスト

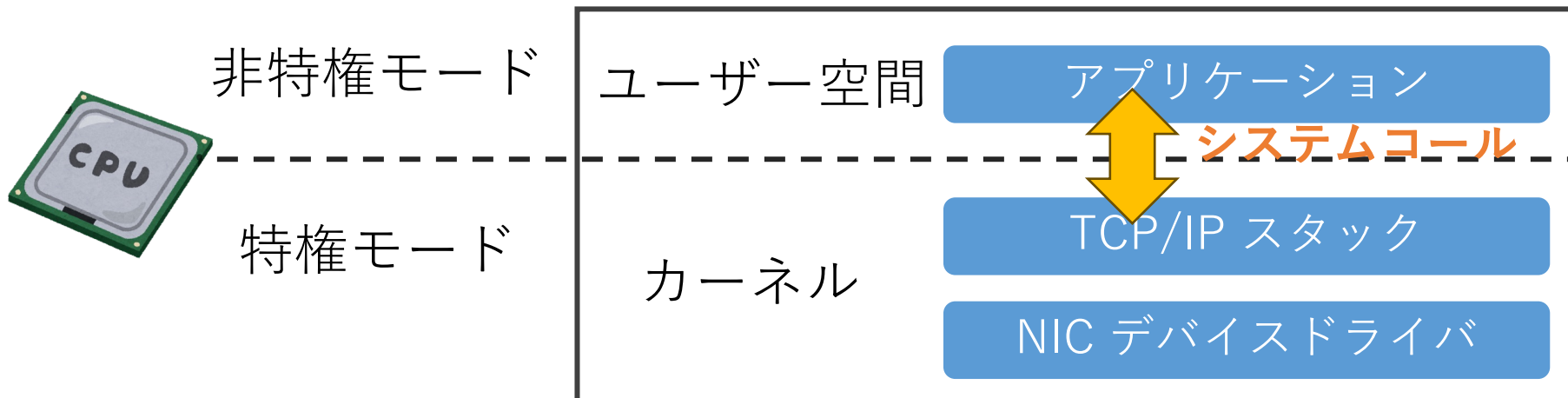
システムコールと通常関数呼び出しの違い



システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

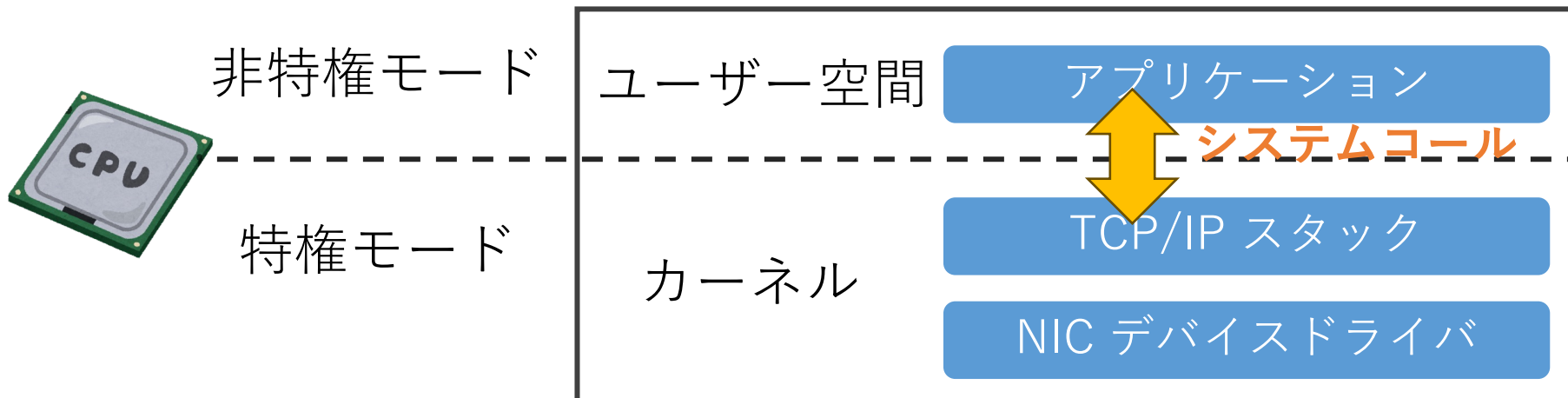


システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えはx86-64であれば **syscall** というCPU命令を利用

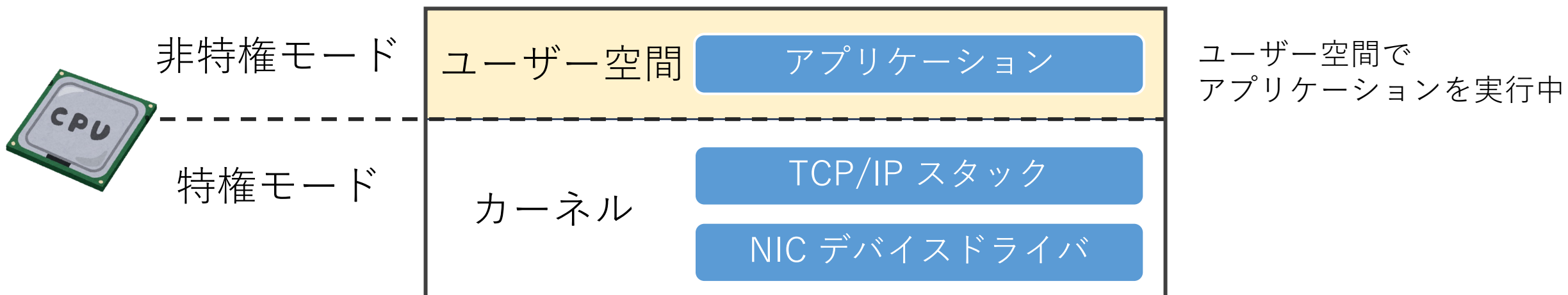


システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えは
x86-64 であれば **syscall** という CPU 命令を利用



システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えはx86-64であれば **syscall** というCPU命令を利用



システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えは
x86-64 であれば **syscall** という CPU 命令を利用

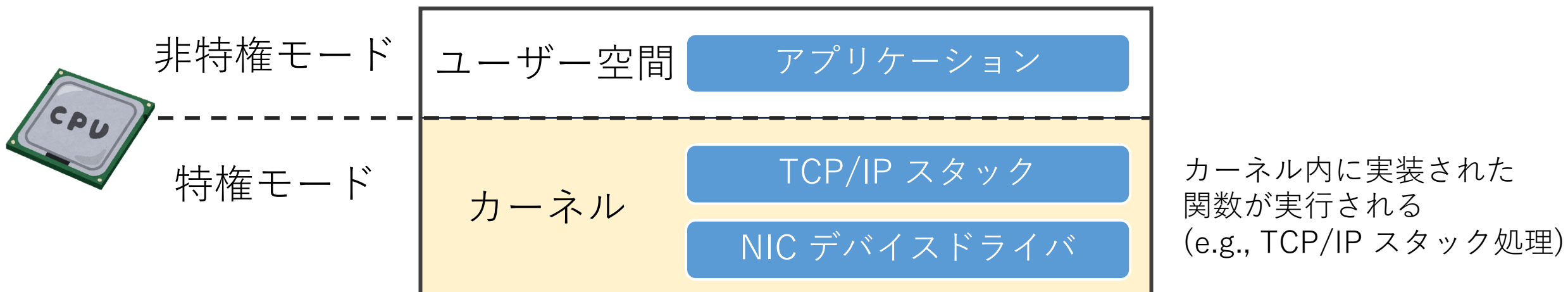


システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えは
x86-64 であれば **syscall** という CPU 命令を利用

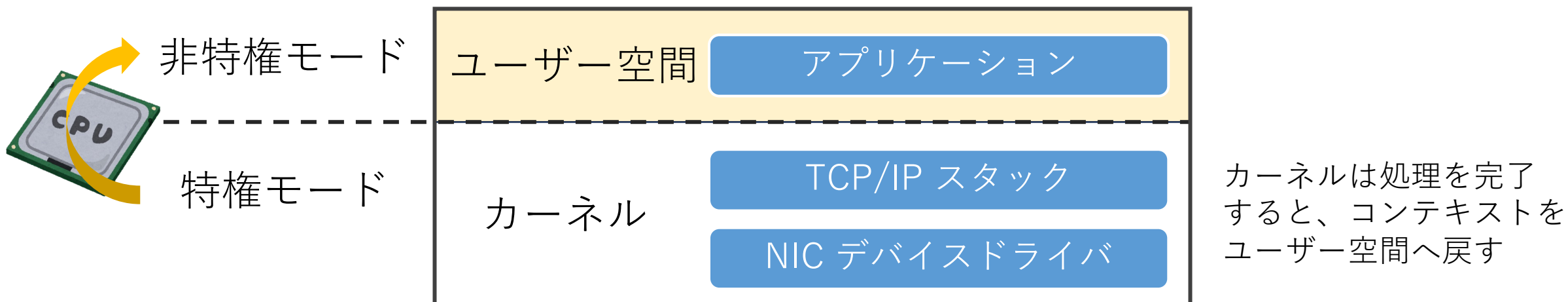


システムコールの呼び出しコスト

システムコールと通常の実数呼び出しの違い

システムコールは、CPU のモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替えた後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えは x86-64 であれば **syscall** という CPU 命令を利用

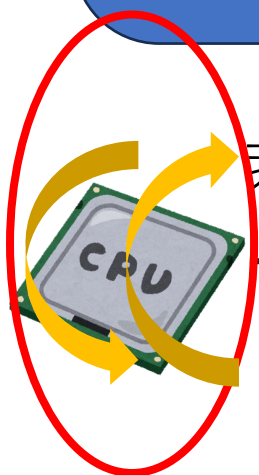


システムコールの呼び出しコスト

システムコールと通常関数呼び出しの違い

システムコールは、**CPUのモードを非特権モード（ユーザー空間）から特権モード（カーネル）への切り替え**した後、カーネルに実装された関数を呼ぶ

システムコールのための特権モード切り替えは
x86-64 であれば **syscall** という CPU 命令を利用



非特権モード

ユーザー空間

アプリケーション

特権モード

カーネル

TCP/IP スタック

NIC デバイスドライバ

ポイント

モード切り替えは
時間がかかる処理



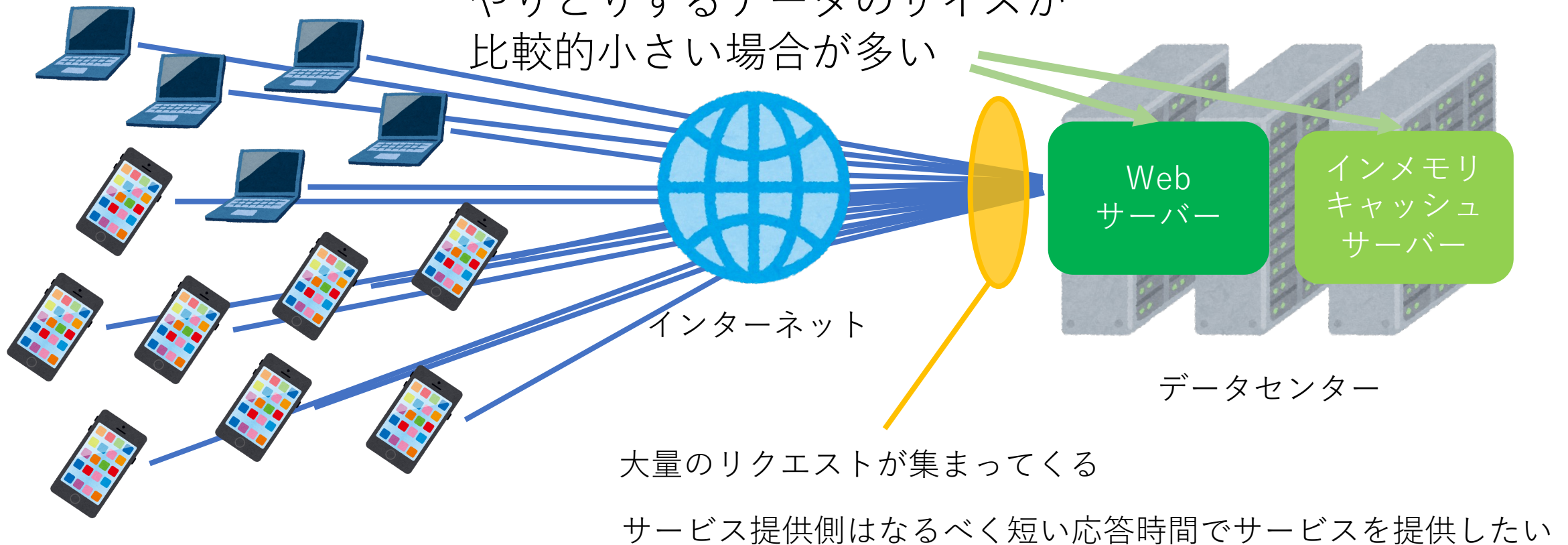
頻繁に呼び出すと
性能劣化に繋がる

2010年くらいの利用シナリオ

サービス利用者の端末

サービス提供側

やりとりするデータのサイズが
比較的小さい場合が多い



典型的なサーバーの実装

{

}

典型的なサーバーの実装

```
{
```

```
/*クライアントからのリクエストデータを読み込み*/
```

```
read(fd, request_buf, sizeof(request_buf));
```

```
}
```

典型的なサーバーの実装

{

```
/*クライアントからのリクエストデータを読み込み*/
```

```
read(fd, request_buf, sizeof(request_buf));
```

```
/*リクエストに応じたレスポンスデータを生成*/
```

```
generate_response(request_buf, response_buf);
```

}

典型的なサーバーの実装

{

```
/*クライアントからのリクエストデータを読み込み*/
```

```
read(fd, request_buf, sizeof(request_buf));
```

```
/*リクエストに応じたレスポンスデータを生成*/
```

```
generate_response(request_buf, response_buf);
```

```
/*レスポンスデータをクライアントへ送信*/
```

```
write(fd, response_buf, response_buf_size);
```

}

典型的なサーバーの実装

```
{  
  /*クライアントからのリクエストデータを読み込み*/  
  read(fd, request_buf, sizeof(request_buf));  
  /*リクエストに応じたレスポンスデータを生成*/  
  generate_response(request_buf, response_buf);  
  /*レスポンスデータをクライアントへ送信*/  
  write(fd, response_buf, response_buf_size);  
}
```

アプリケーション固有の処理 (e.g., HTTP サーバー、キャッシュサーバー)

典型的なサーバーの実装

```
{  
/*クライアントからのリクエストデータを読み込み*/  
read(fd, request_buf, sizeof(request_buf));  
/*リクエストに応じたレスポンスデータを生成*/  
generate_response(request_buf, response_buf);  
/*レスポンスデータをクライアントへ送信*/  
write(fd, response_buf, response_buf_size);  
}
```

OS から提供されるシステムコール

典型的なサーバーの実装

```
{  
    /*クライアントからのリクエストデータを読み込み*/  
    read(fd, request_buf, sizeof(request_buf));  
    /*リクエストに応じたレスポンスデータを生成*/  
    generate_response(request_buf, response_buf);  
    /*レスポンスデータをクライアントへ送信*/  
    write(fd, response_buf, response_buf_size);  
}
```

OS から提供されるシステムコール

典型的なサーバーの実装

```
{  
    /*クライアントからのリクエストデータを読み込み*/  
    read(fd, request_buf, sizeof(request_buf));  
    /*リクエストに応じたレスポンスデータを生成*/  
    generate_response(request_buf, response_buf);  
    /*レスポンスデータをクライアントへ送信*/  
    write(fd, response_buf, response_buf_size);  
}
```

リクエスト読み込みとレスポンス書き出しを頻繁に行うと
システムコールが頻繁に呼び出される

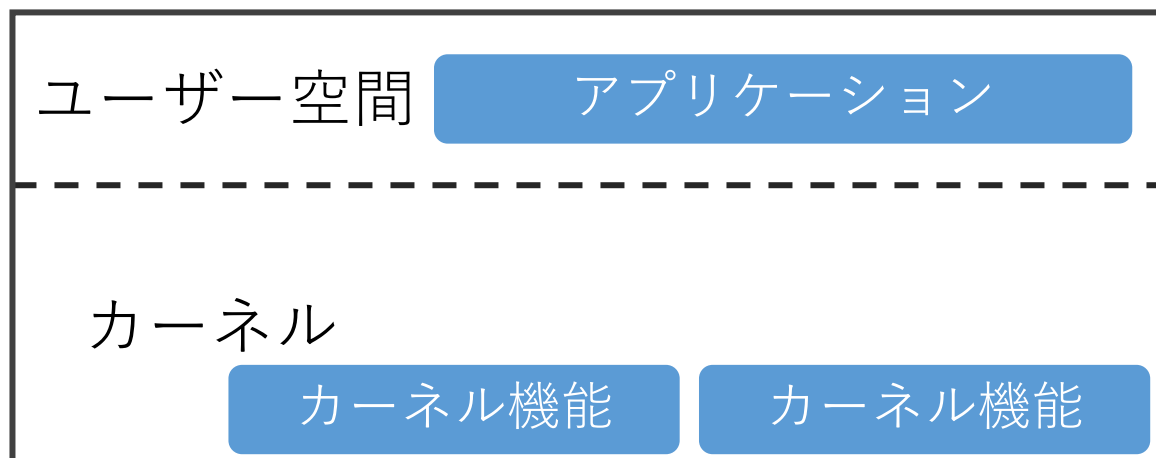
研究紹介

システムコール呼び出しコストについて

システムコールを複数まとめてリクエストできるようにする

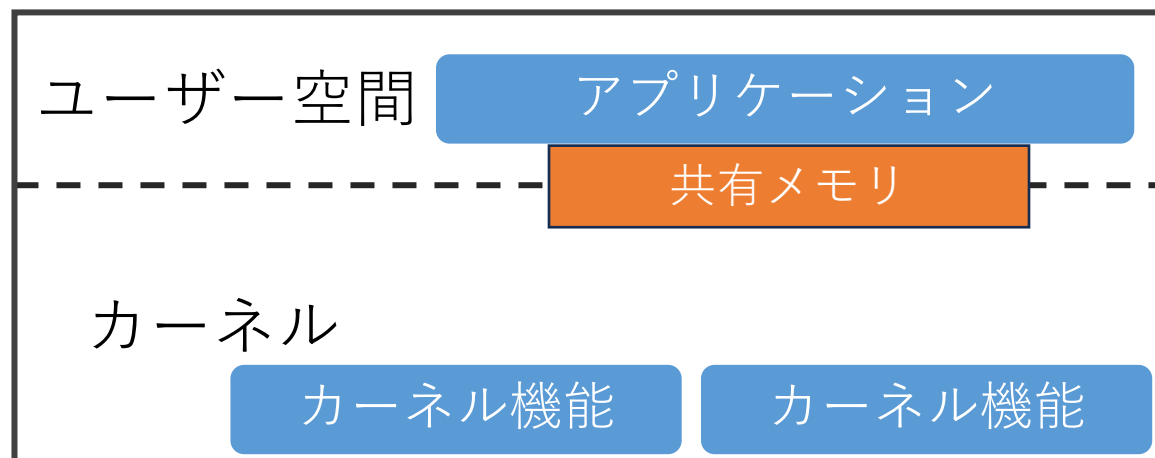
システムコールの頻度を減らす

- FlexSC (OSDI 2010)



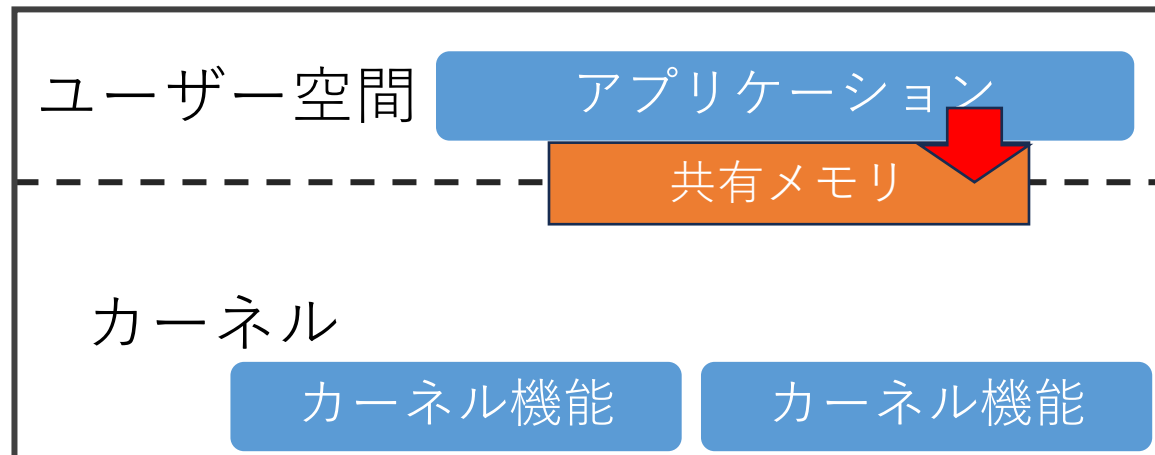
システムコールの頻度を減らす

- FlexSC (OSDI 2010)
 - ユーザー・カーネル空間の間に共有メモリを用意



システムコールの頻度を減らす

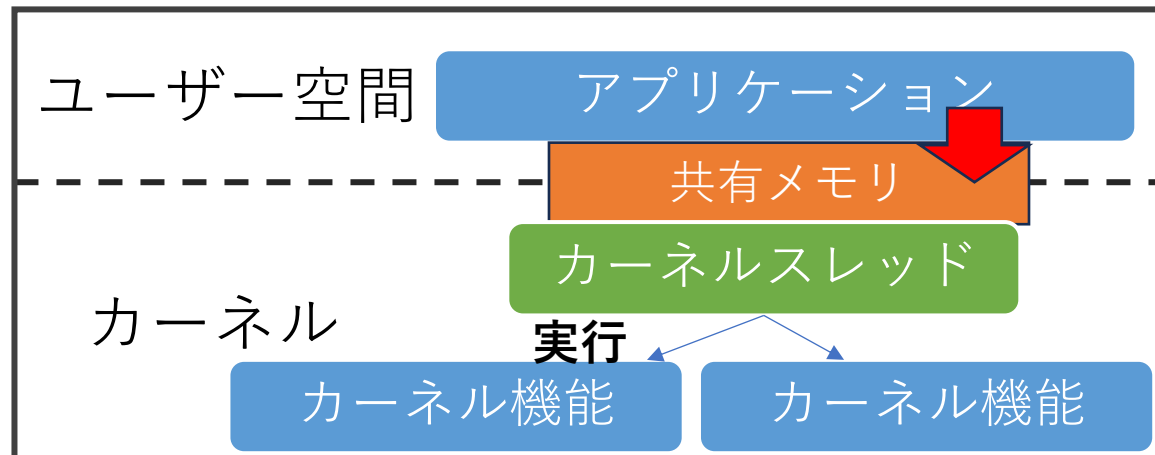
- FlexSC (OSDI 2010)
 - ユーザー・カーネル空間の間に共有メモリを用意
 - ユーザー空間プログラムはリクエスト内容を共有メモリ上に書き込み



システムコールの頻度を減らす

- FlexSC (OSDI 2010)

- ユーザー・カーネル空間の間に共有メモリを用意
- ユーザー空間プログラムはリクエスト内容を共有メモリ上に書き込み
- カーネル内で専用のカーネルスレッドが共有リクエストを読み取り
カーネル機能を実行



Exception-less interface: syscall page

```
write(fd, buf, 4096);  
↓  
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;  
  
while (entry->status != DONE)  
    do_something_else();  
  
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096



Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

通常の write システムコール

```
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096



Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

```
entry = free_syscall_entry(),  
  
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;  
  
while (entry->status != DONE)  
    do_something_else();  
  
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096

FlexSC だとこんなかんじ

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```



```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
	3	fd, buf, 4096	DONE	4096

リクエスト用エントリを取得

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

```
entry = free_syscall_entry();
```

```
/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->sta
```

```
while (ent
do_some
return ent
```

このエントリはユーザー空間と
カーネル空間の共有メモリ上に存在

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096

ユーザー空間

アプリケーション

共有メモリ

カーネルスレッド

カーネル

実行

カーネル機能

カーネル機能

を取得

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```



```
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096



システムコール番号を設定
(write は 1)

Exception-less interface: syscall page

```
write(fd, buf, 4096);  
↓  
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;  
  
while (entry->status != DONE)  
    do_something_else();  
  
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096

引数の数を指定

Exception-less interface: syscall page

```
write(fd, buf, 4096);  
↓  
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096

引数を指定

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```



```
entry = free_syscall_entry();
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```

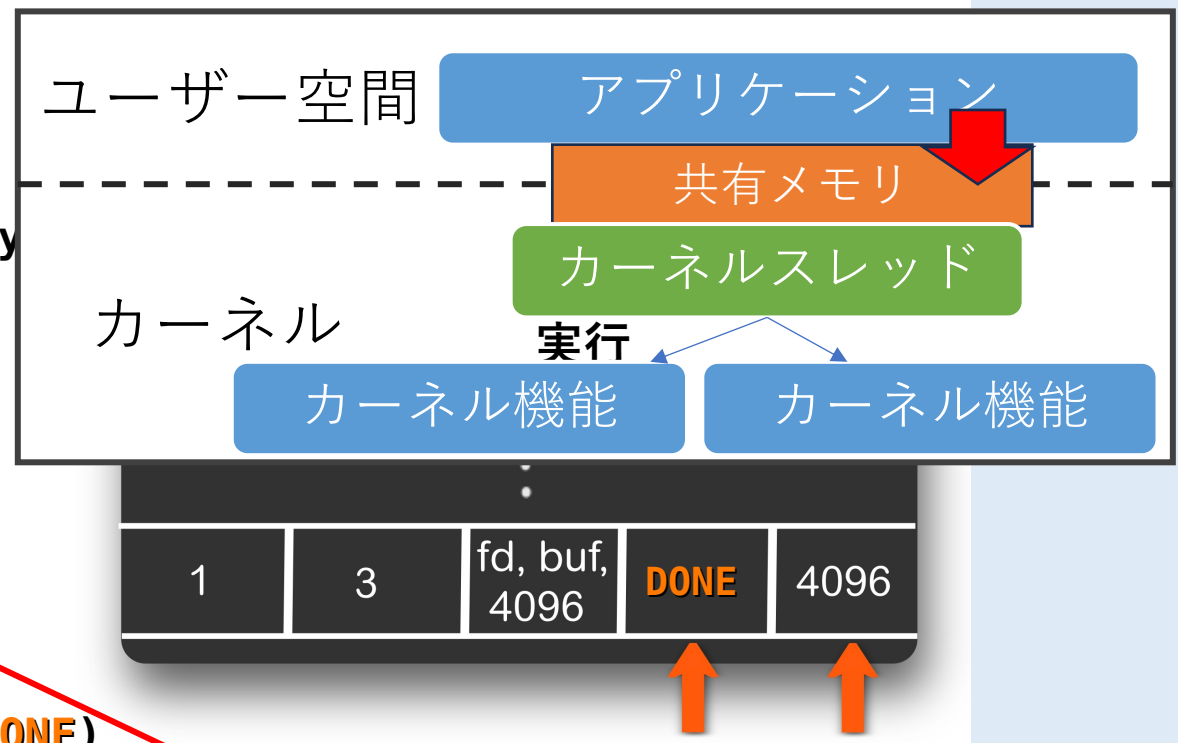
syscall number	number of args	args 0 ... 6	status	return code
		⋮		
1	3	fd, buf, 4096	DONE	4096



ステータスをSUBMITへ変更

Exception-less interface: syscall page

```
write(fd, buf, 4096);  
↓  
entry = free_syscall_entry  
  
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;  
  
while (entry->status != DONE)  
    do_something_else();  
  
return entry->return_code;
```



entry->status = SUBMIT;

ステータスをSUBMITへ変更

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

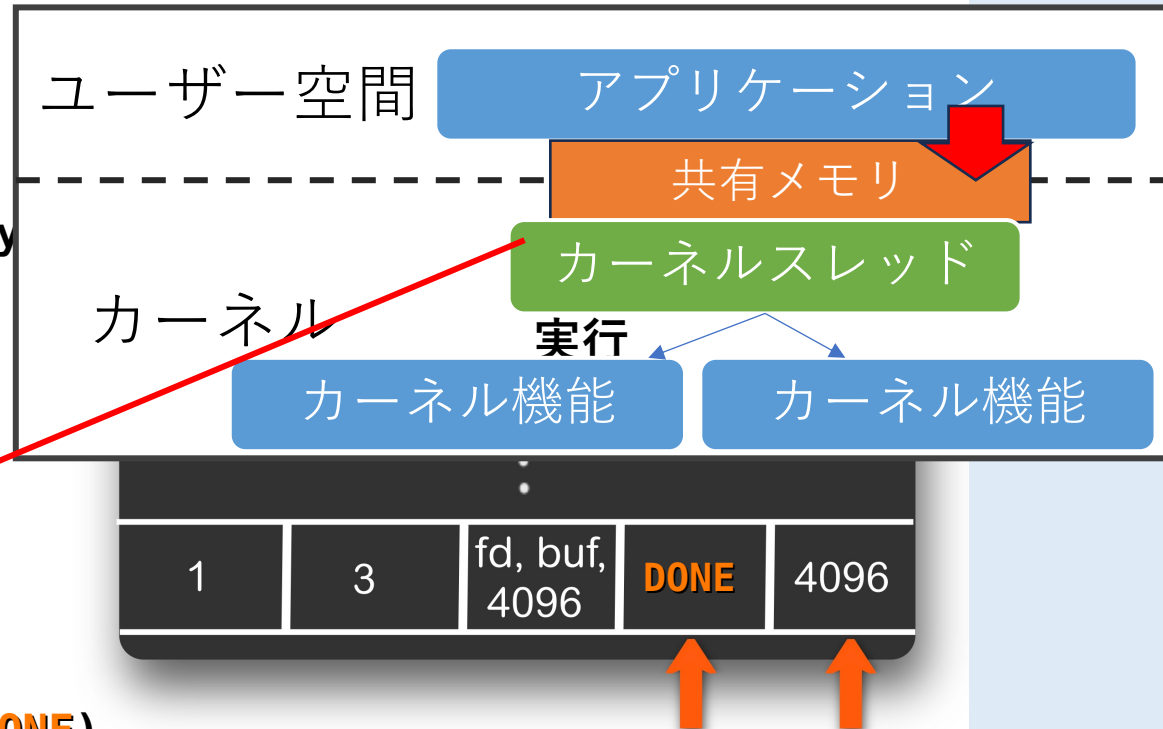


```
entry = free_syscall_entry
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```



専用カーネルスレッドは **SUBMIT** 状態を
読み取り、リクエストの処理を開始する

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

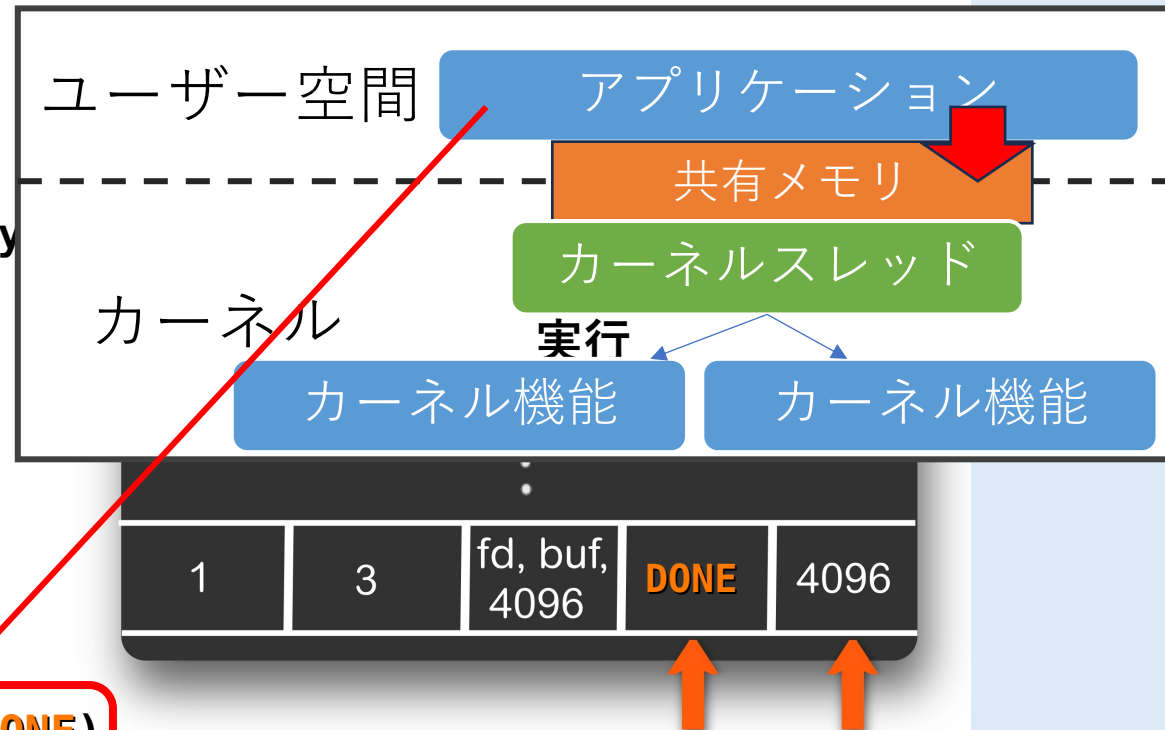


```
entry = free_syscall_entry
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```



アプリケーションはステータスが
DONE になるまで待機

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

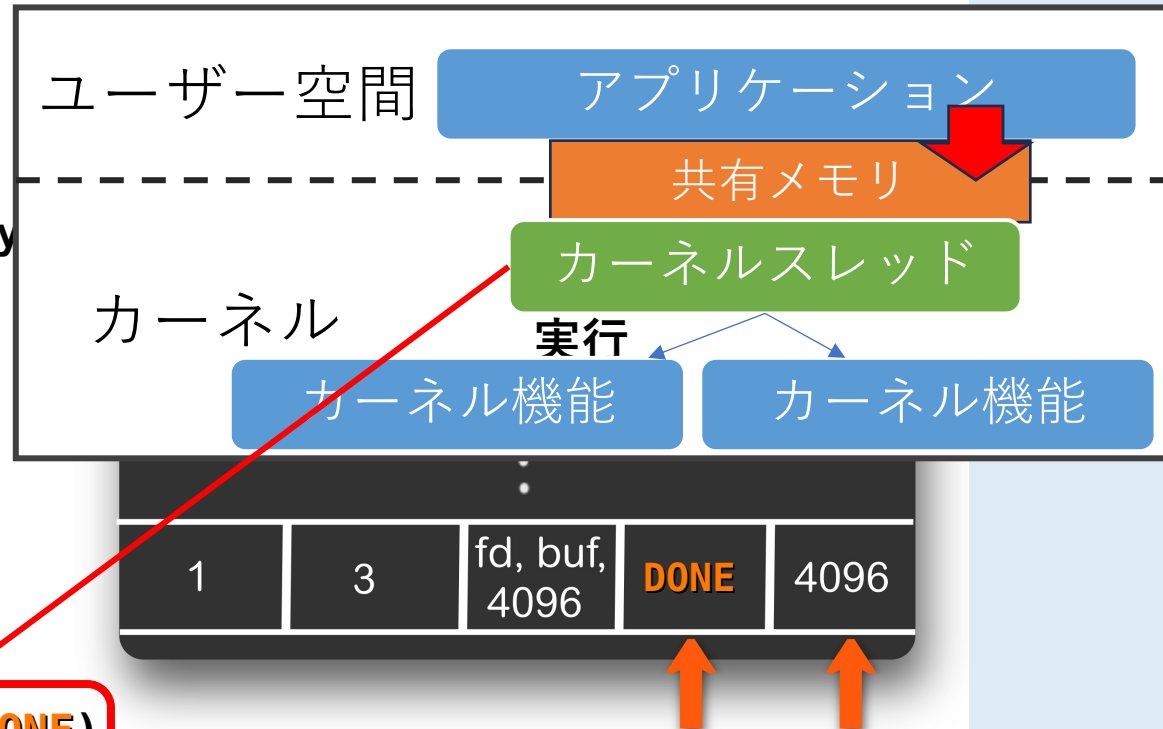


```
entry = free_syscall_entry
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```



カーネルスレッドは処理が完了し次第
結果をreturn_codeに設定した後
ステータスをDONEに変更

Exception-less interface: syscall page

```
write(fd, buf, 4096);
```

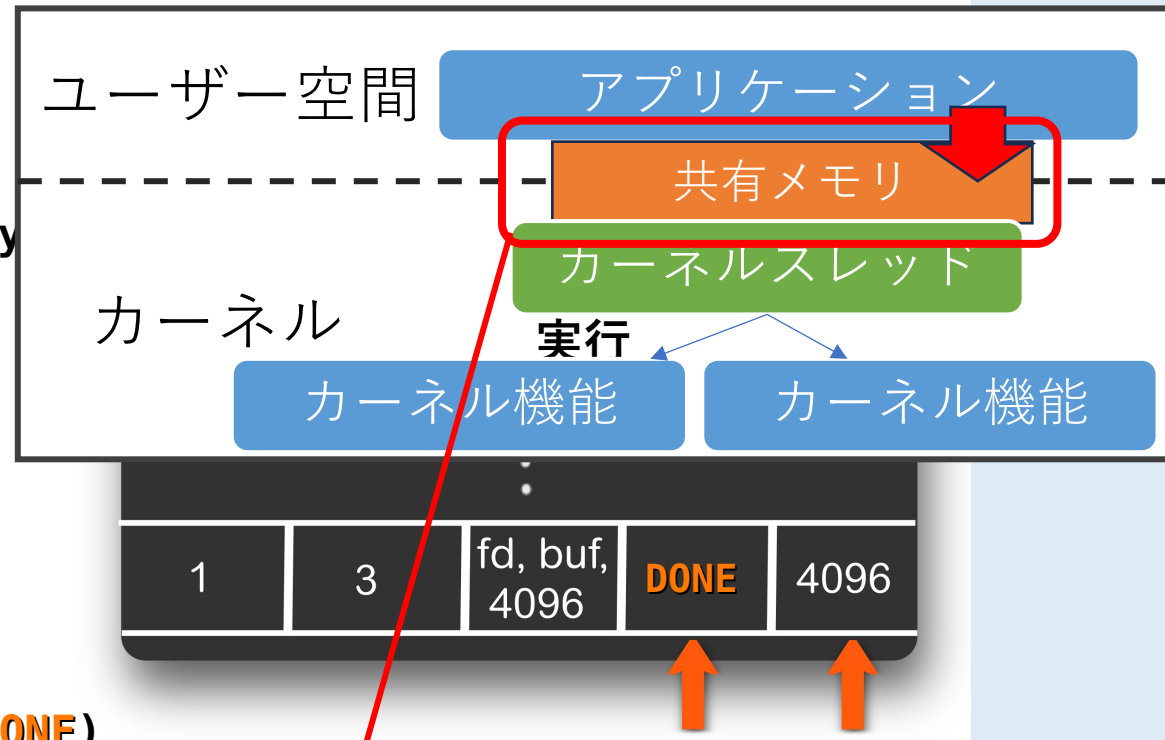


```
entry = free_syscall_entry
```

```
/* write syscall */  
entry->syscall = 1;  
entry->num_args = 3;  
entry->args[0] = fd;  
entry->args[1] = buf;  
entry->args[2] = 4096;  
entry->status = SUBMIT;
```

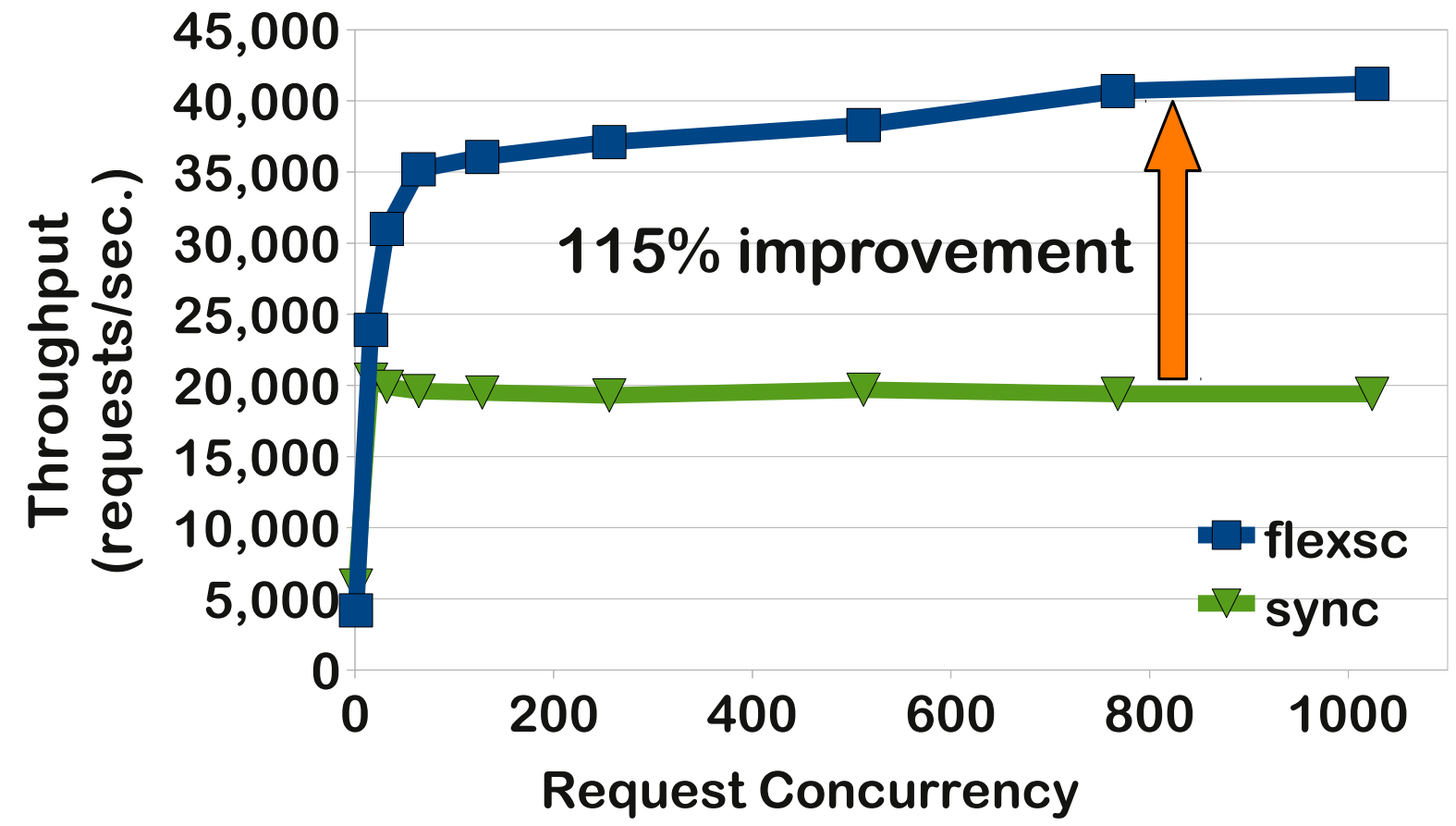
```
while (entry->status != DONE)  
    do_something_else();
```

```
return entry->return_code;
```



ポイント：ユーザー空間とカーネルの間のやりとりを共有メモリを通じて行うことで syscall 命令に伴うコンテキストの切り替えをなくせる

ApacheBench throughput (4 cores)



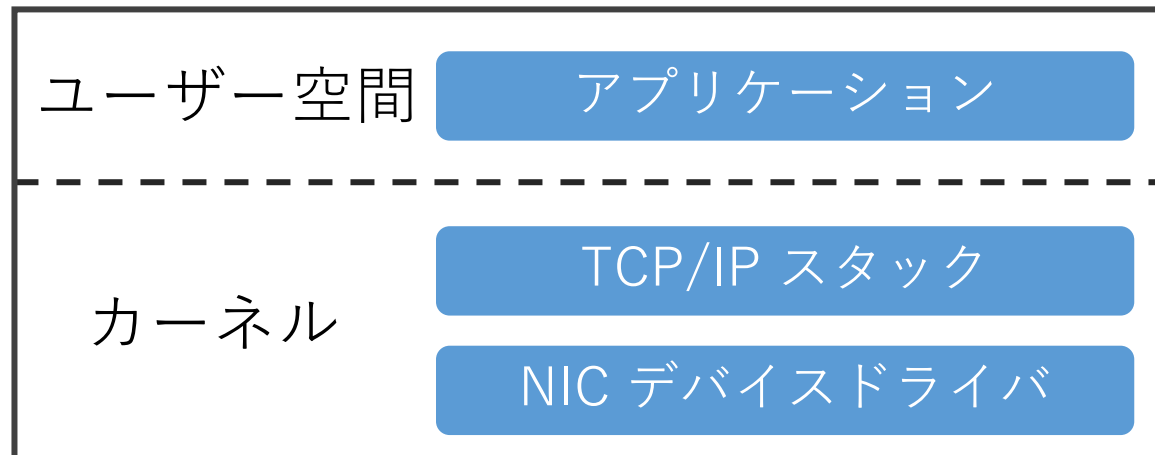
研究紹介

システムコール呼び出しコストについて

ユーザー空間とカーネルの境界をなくす

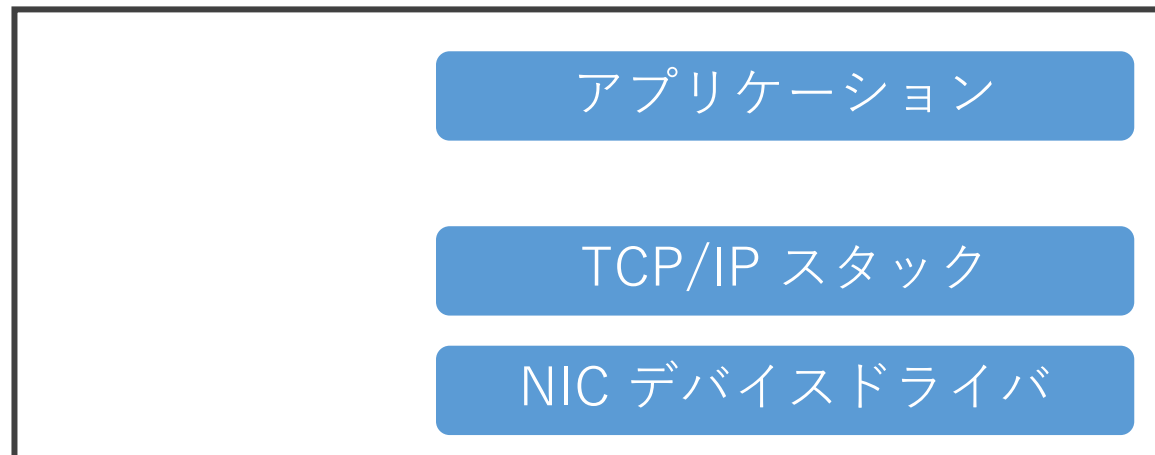
システムコールの頻度を減らす

- アプリとカーネルの境界をなくす



システムコールの頻度を減らす

- アプリとカーネルの境界をなくす



システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

具体的には

一般的にカーネルに実装されている機能をユーザー空間へ移す



システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

もしくは、アプリケーションをカーネル空間で動かす



システムコールの頻度を減らす

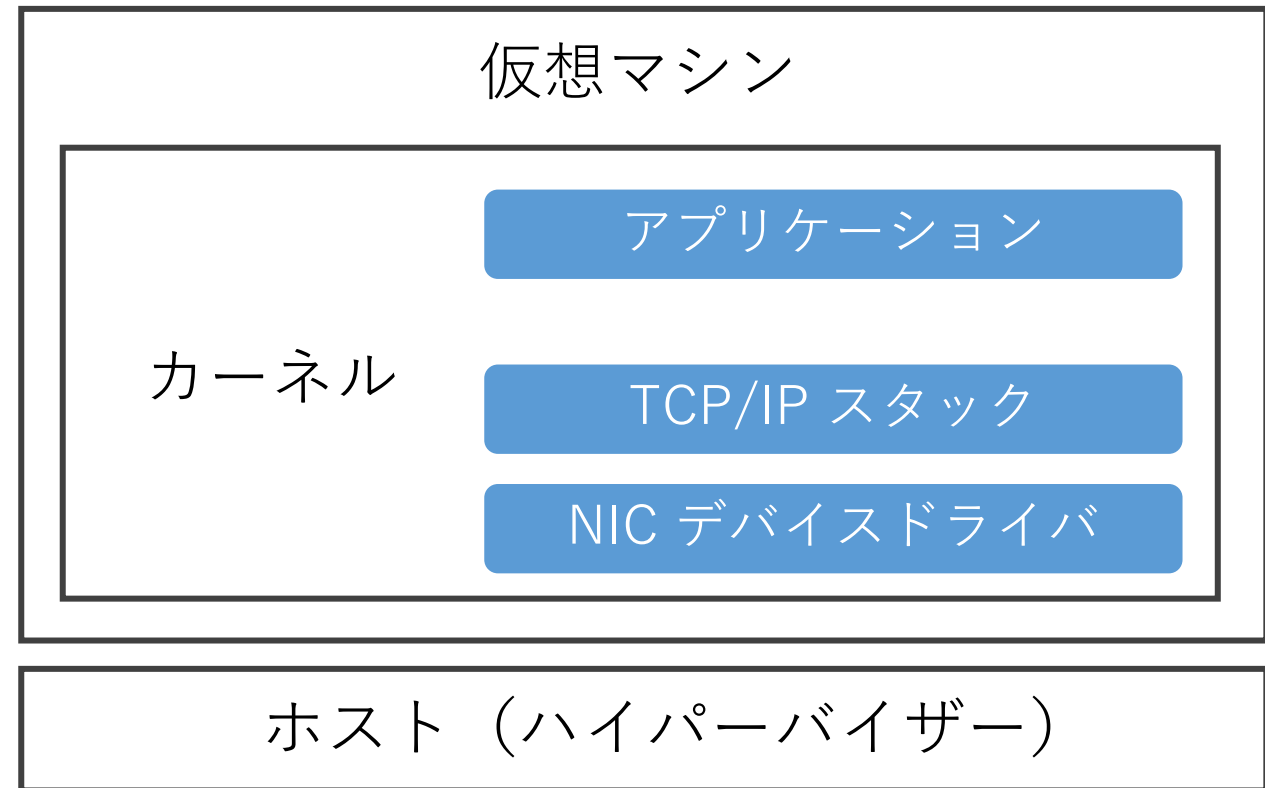
- アプリとカーネルの境界をなくす

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- **Unikernels**

- Mirage (ASPLOS 2013)
- OSv (USENIX ATC 2014)
- Lupin Linux (EuroSys 2020)
- Unikraft (EuroSys 2021)
- Unikernel Linux (EuroSys 2023)



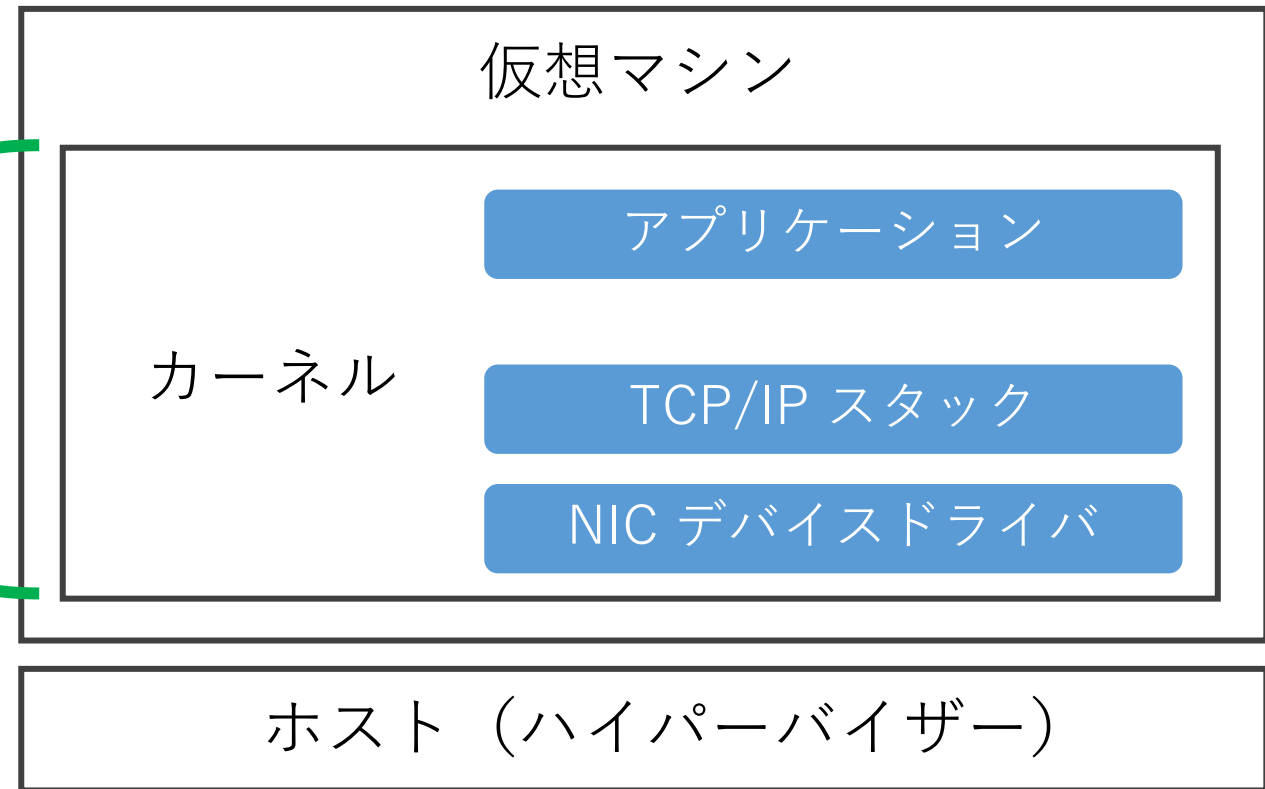
アプリケーションはカーネル空間で動く

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- **Unikernels**

- Mirage (ASPLoS 2013)
- OSv (USENIX ATC 2014)
- Lupin Linux (EuroSys 2020)
- Unikraft (EuroSys 2021)
- Unikernel Linux (EuroSys 2023)



アプリケーションはカーネル空間で動く

システムコールの頻度を減らす

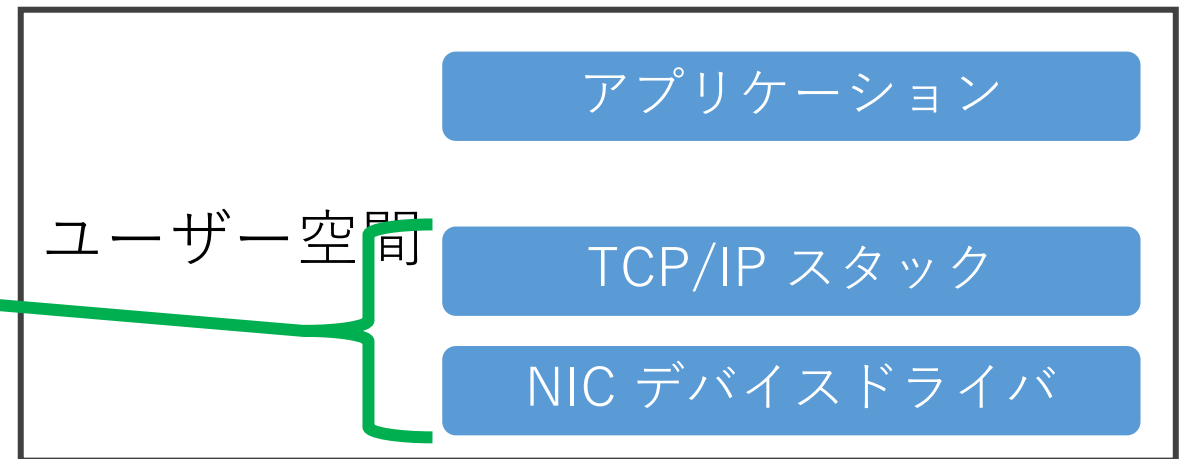
- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - **ライブラリ OS**
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)



OS 機能がユーザー空間で動く

システムコールの頻度を減らす

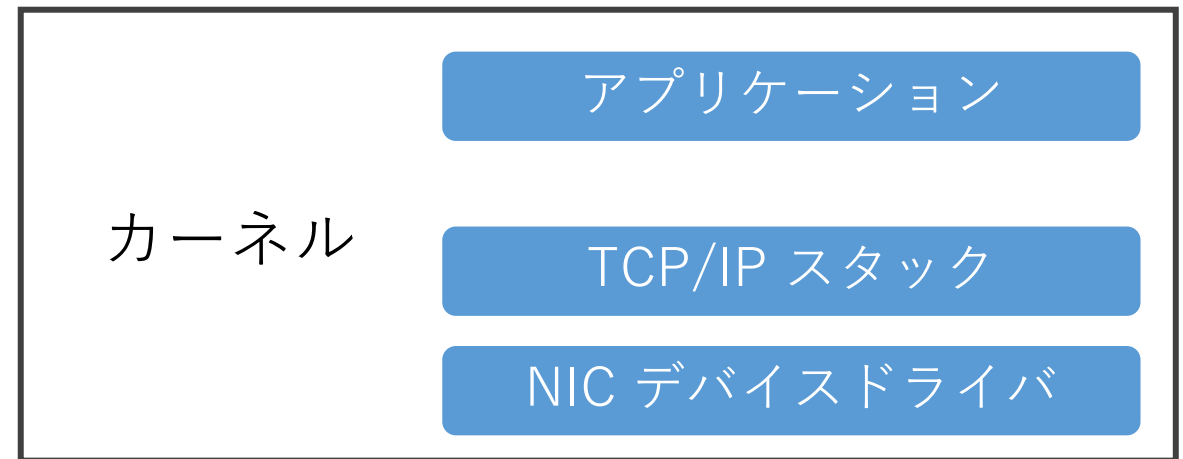
- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - **ライブラリ OS**
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)



OS 機能がユーザー空間で動く

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - ライブラリ OS
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)
 - アプリのコードを検証後カーネルで実行
 - Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)



アプリケーションはカーネル空間で動く

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- Unikernels

- Mirage (ASPLOS 2013)
- OSv (USENIX ATC 2014)
- **Lupin Linux (EuroSys 2020)**
- Unikraft (EuroSys 2021)
- Unikernel Linux (EuroSys 2023)

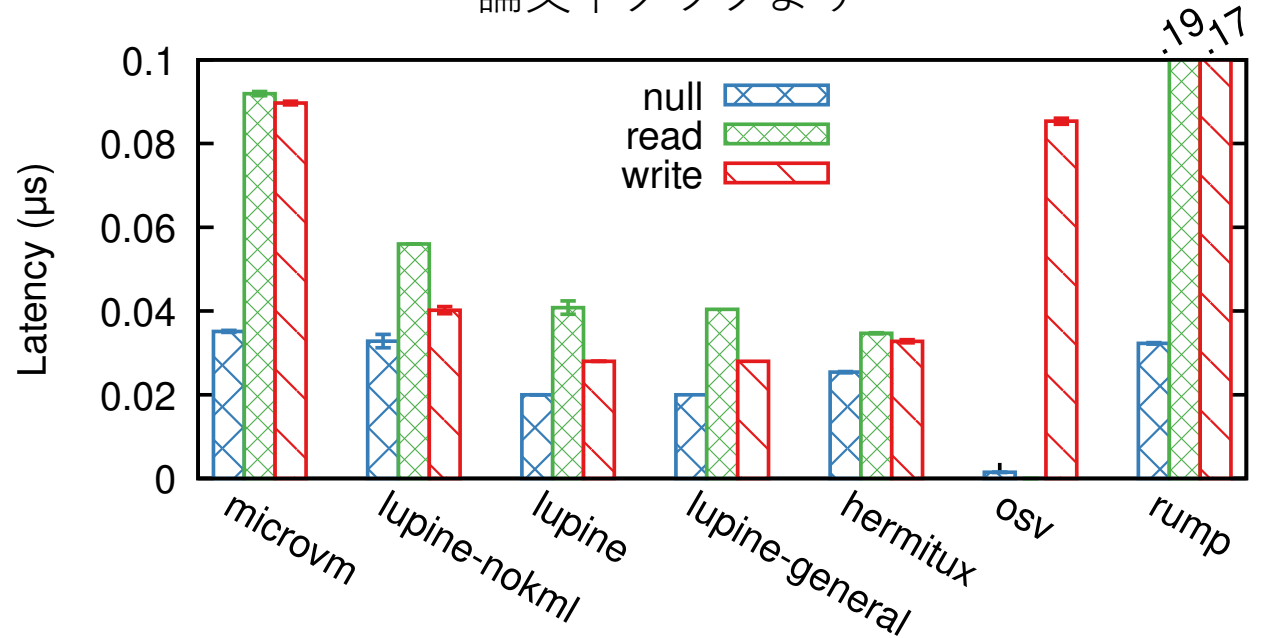
- ライブラリ OS

- VirtuOS (SOSP 2013)
- EbbRT (OSDI 2016)
- Demikernel (SOSP 2021)

- アプリのコードを検証後カーネルで実行

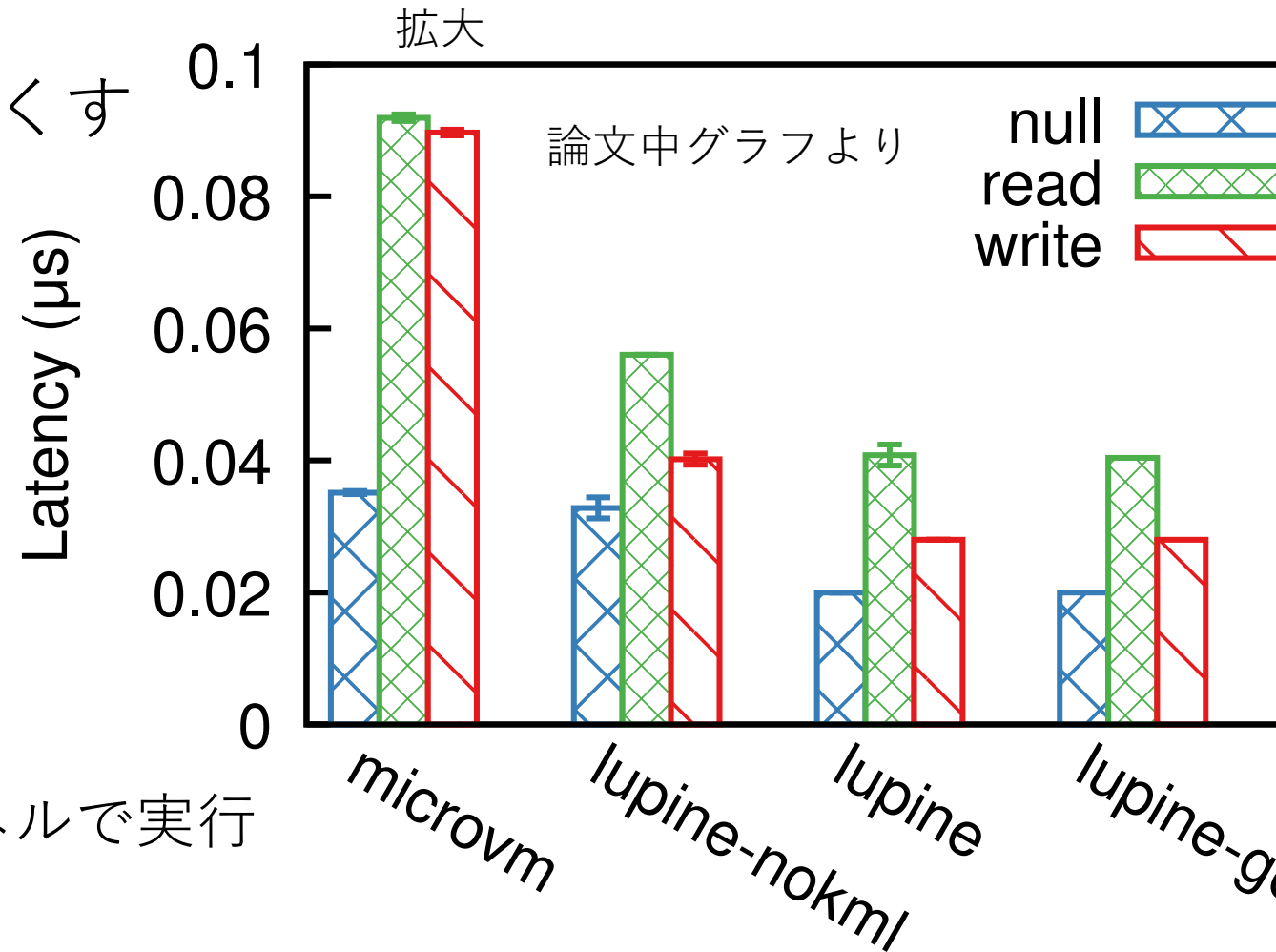
- Privbox (USENIX ATC 2022)
- Userspace bypass (OSDI 2023)

論文中グラフより



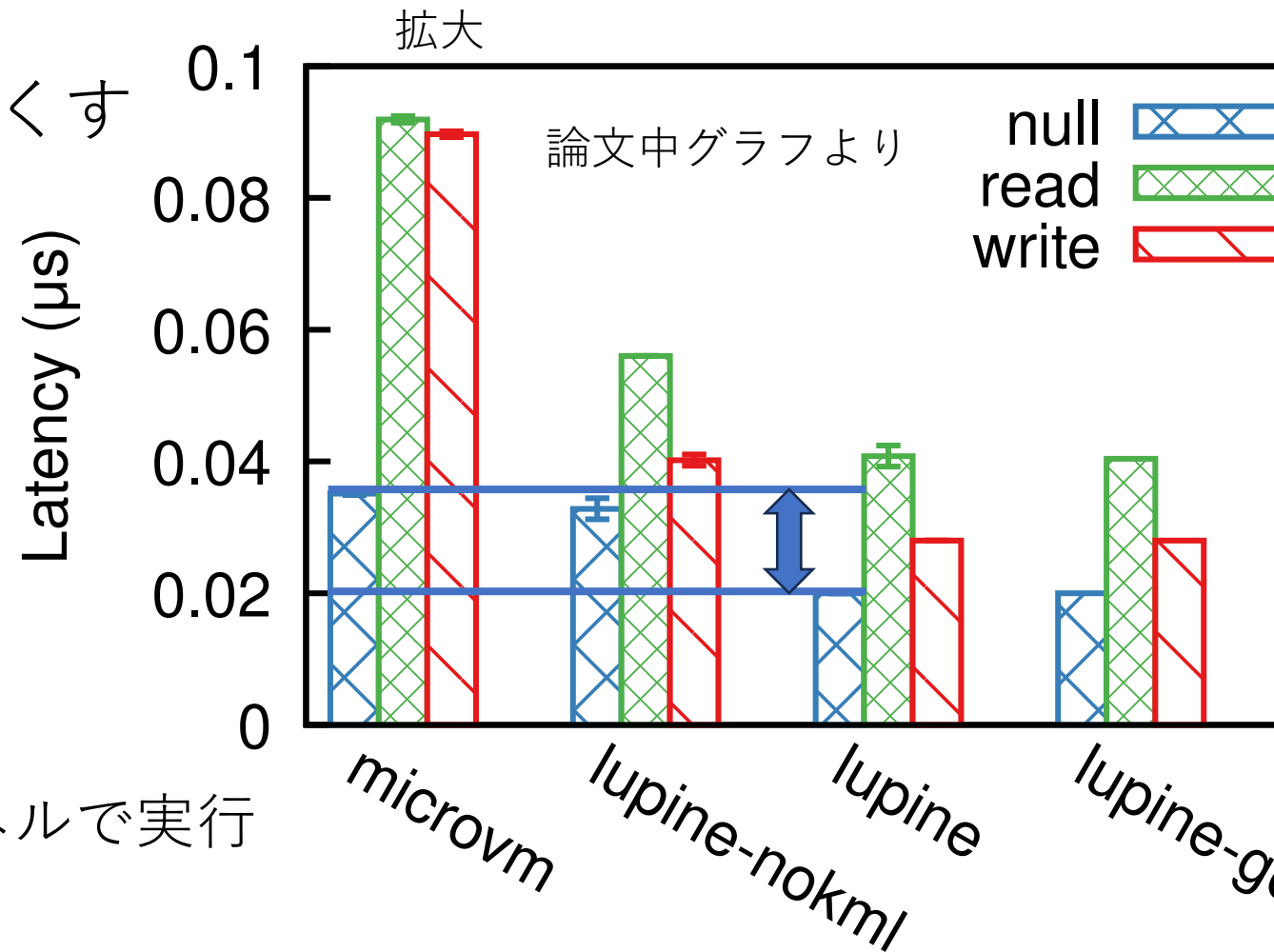
システムコールの頻度を減らす

- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - **Lupin Linux (EuroSys 2020)**
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - ライブラリ OS
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)
 - アプリのコードを検証後カーネルで実行
 - Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)



システムコールの頻度を減らす

- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - **Lupin Linux (EuroSys 2020)**
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - ライブラリ OS
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)
 - アプリのコードを検証後カーネルで実行
 - Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)



システムコールの頻度を減らす

- アプリとカーネルの境界をなくす
 - Unikernels
 - Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)
 - ライブラリ OS
 - VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)
 - アプリのコードを検証後カーネルで実行
 - Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- Unikernels

- Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)

既存の Linux 実装を使いたい

- ライブラリ OS

- VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)

- アプリのコードを検証後カーネルで実行

- Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- Unikernels

- Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)

既存の Linux 実装を使いたい

開発を簡単にしたい

- ライブラリ OS

- VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)

開発を簡単にしたい

- アプリのコードを検証後カーネルで実行

- Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- Unikernels

- Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)

既存の Linux 実装を使いたい

開発を簡単にしたい

- ライブラリ OS

- VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)

開発を簡単にしたい

I/O デバイスの差異を吸収したい

- アプリのコードを検証後カーネルで実行

- Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)

システムコールの頻度を減らす

- アプリとカーネルの境界をなくす

- Unikernels

- Mirage (ASPLOS 2013)
 - OSv (USENIX ATC 2014)
 - Lupin Linux (EuroSys 2020)
 - Unikraft (EuroSys 2021)
 - Unikernel Linux (EuroSys 2023)

既存の Linux 実装を使いたい

開発を簡単にしたい

- ライブラリ OS

- VirtuOS (SOSP 2013)
 - EbbRT (OSDI 2016)
 - Demikernel (SOSP 2021)

開発を簡単にしたい

I/O デバイスの差異を吸収したい

- アプリのコードを検証後カーネルで実行

- Privbox (USENIX ATC 2022)
 - Userspace bypass (OSDI 2023)

既存のカーネル機能が使える

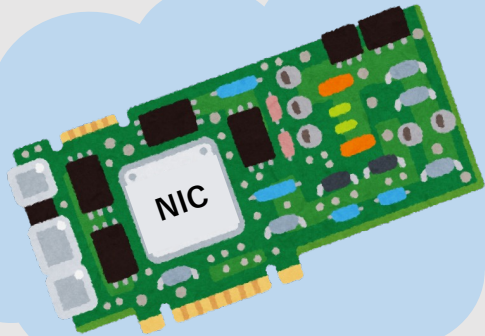
研究紹介

パケット I/O 性能について

基本的な仕組みの説明

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



多重のリク
サービス提供

通信関連ソフトウェア

ユーザー空間

アプリケーション

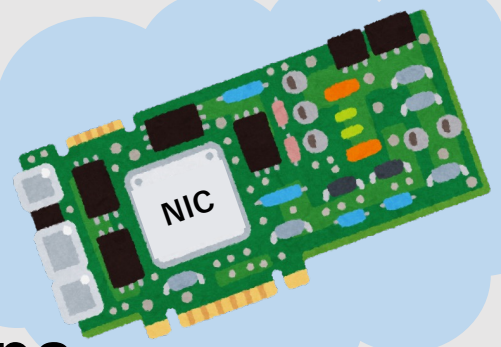
カーネル

TCP/IP スタック

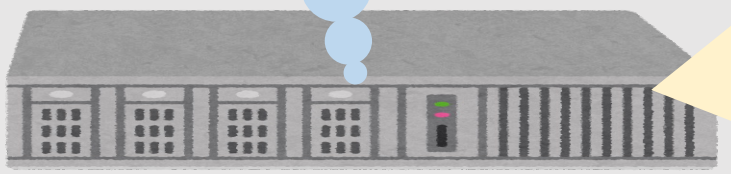
NIC デバイスドライバ

通信関連のシステムソフトウェア

データセンター内のサーバー

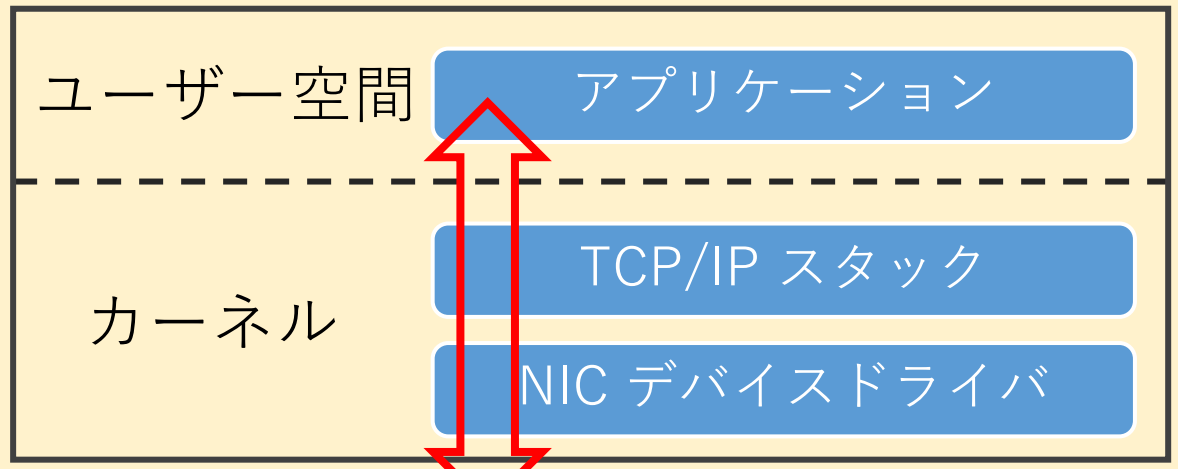


10 ~ Gbps



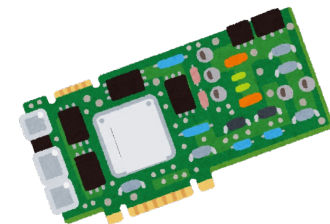
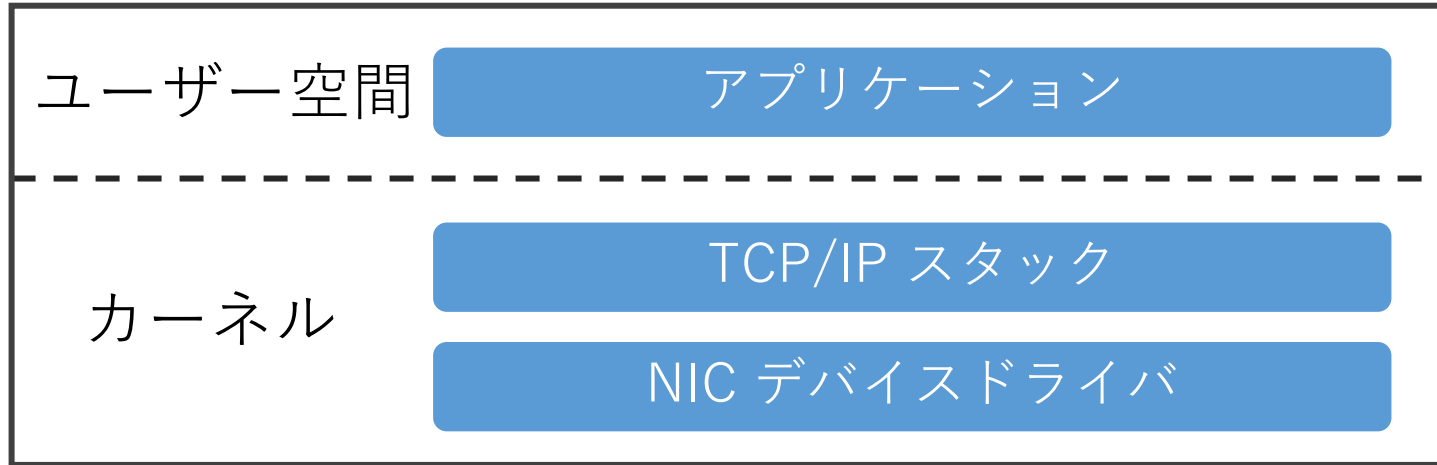
多重のリク
サービス提供

通信関連ソフトウェア

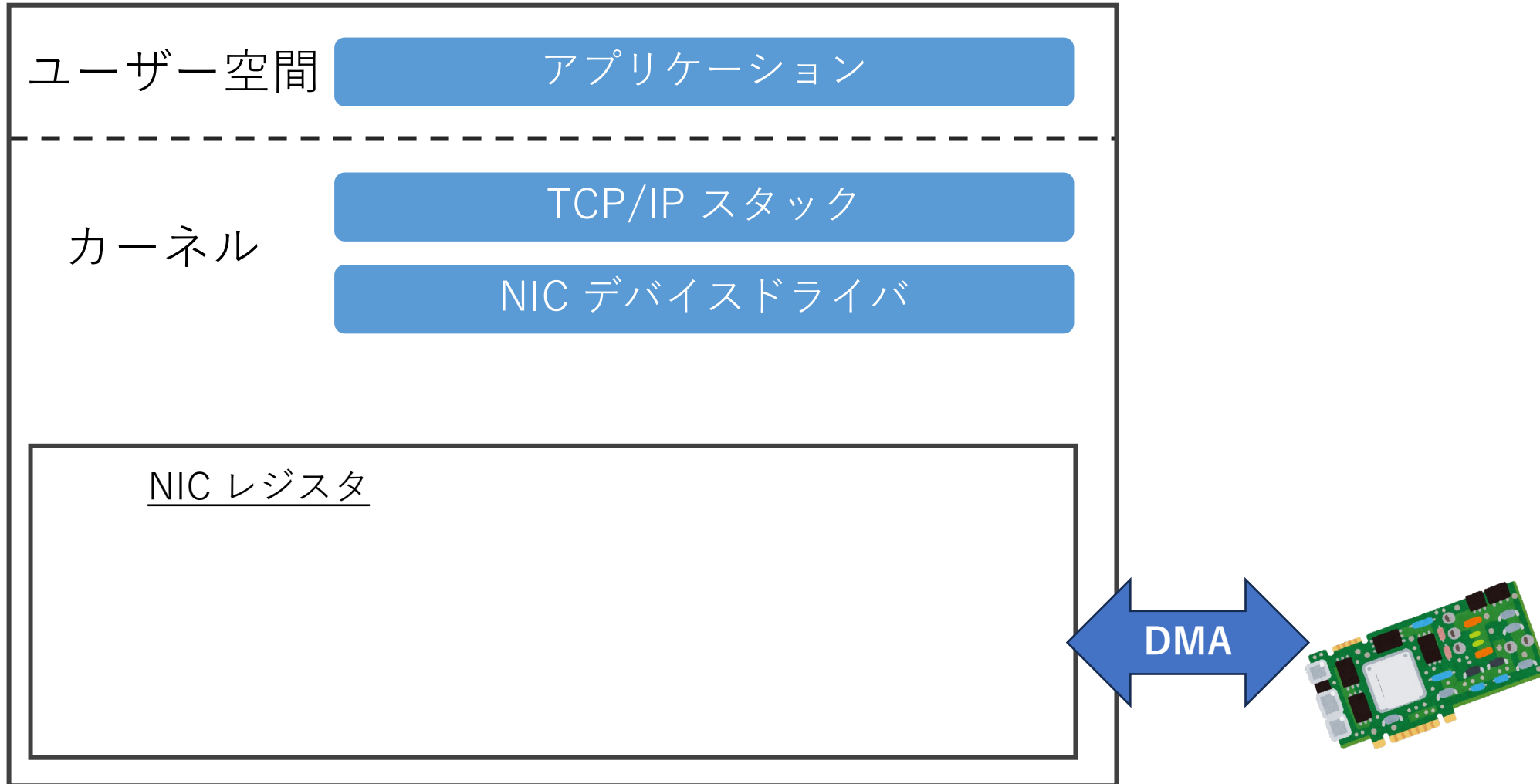


既存の仕組みだと比較的処理が軽い UDP でも
小さいパケットを高速にやりとりできなかった

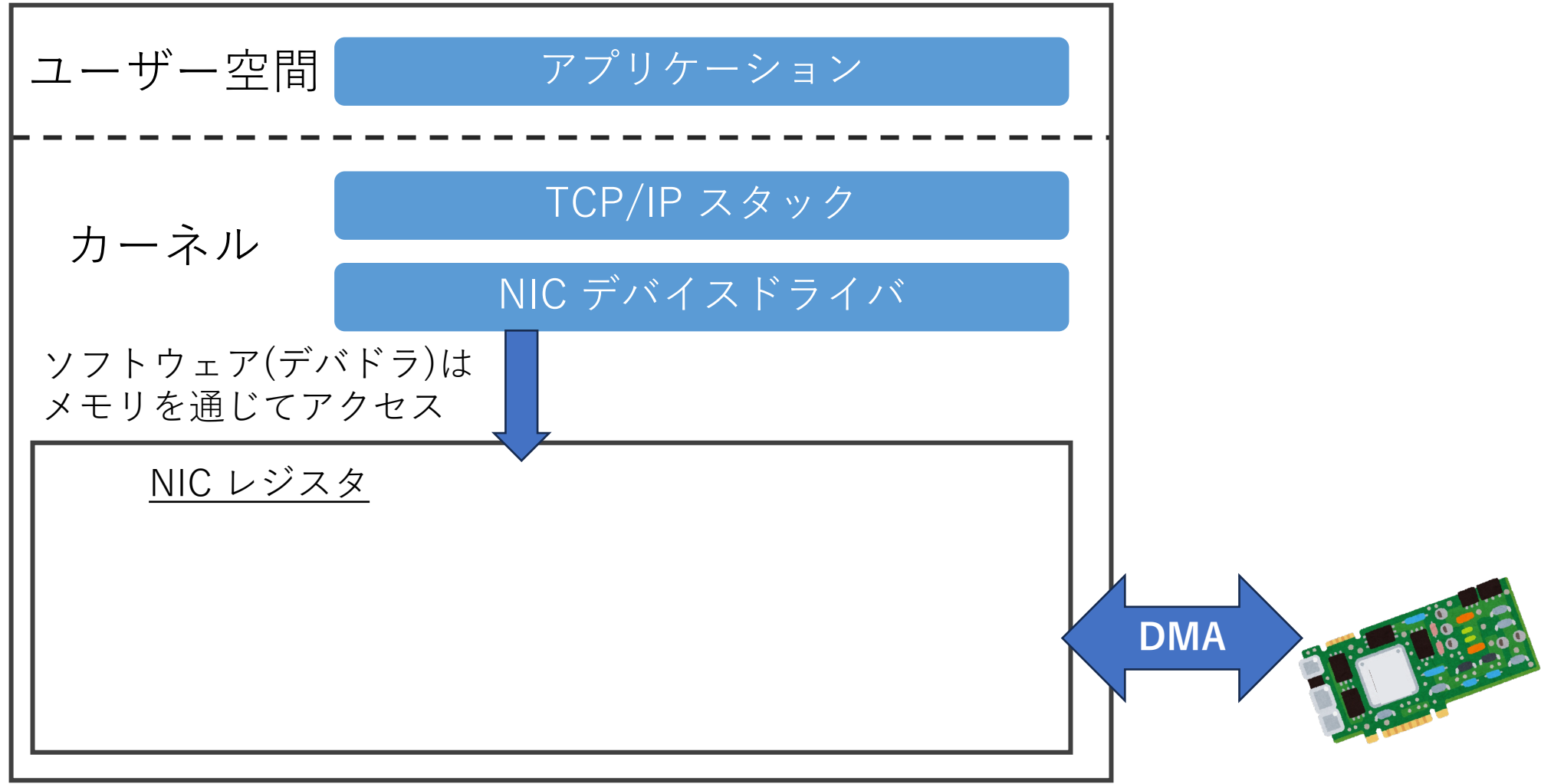
NIC と通信関連プログラムの構成



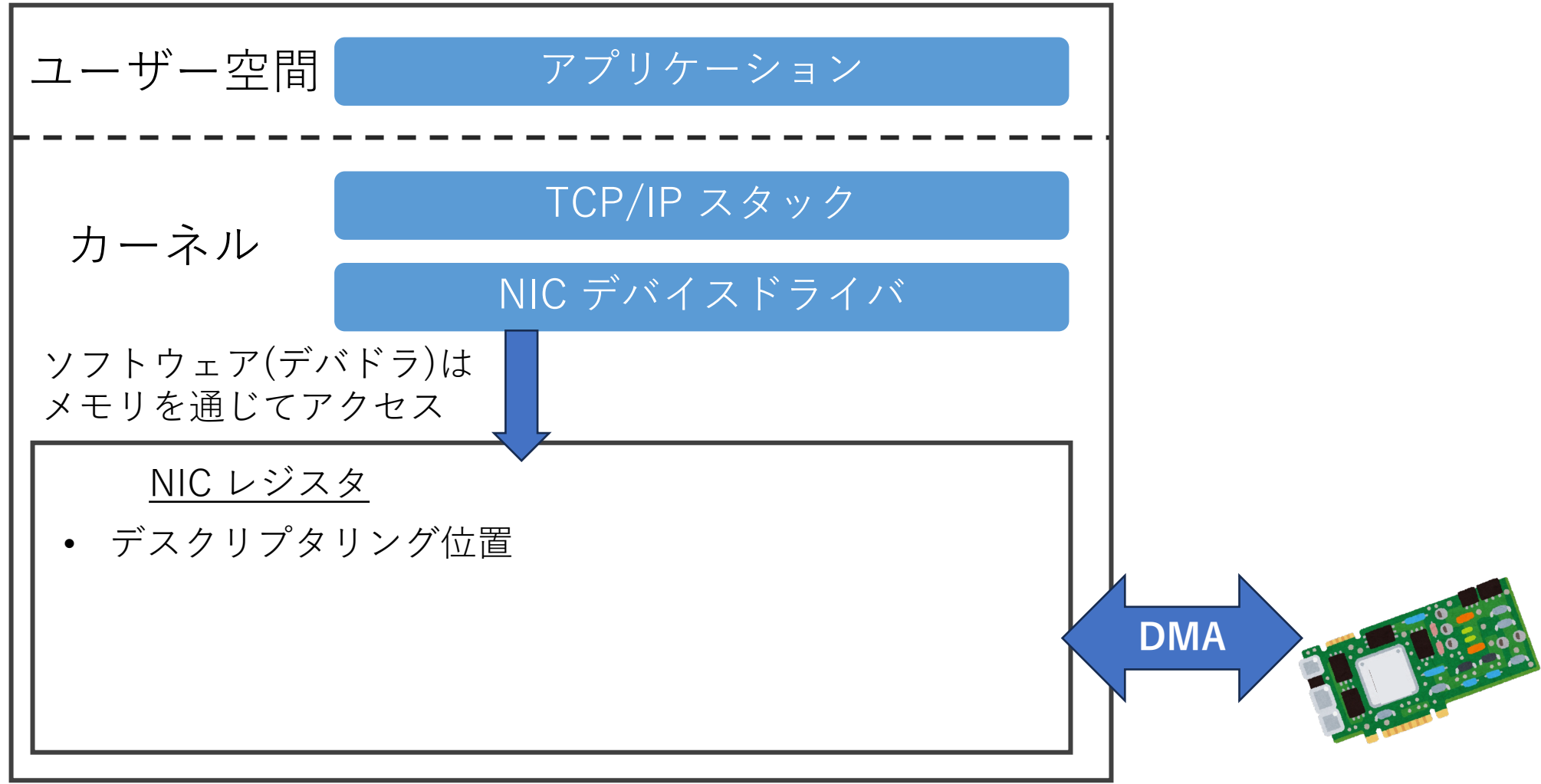
NIC と通信関連プログラムの構成



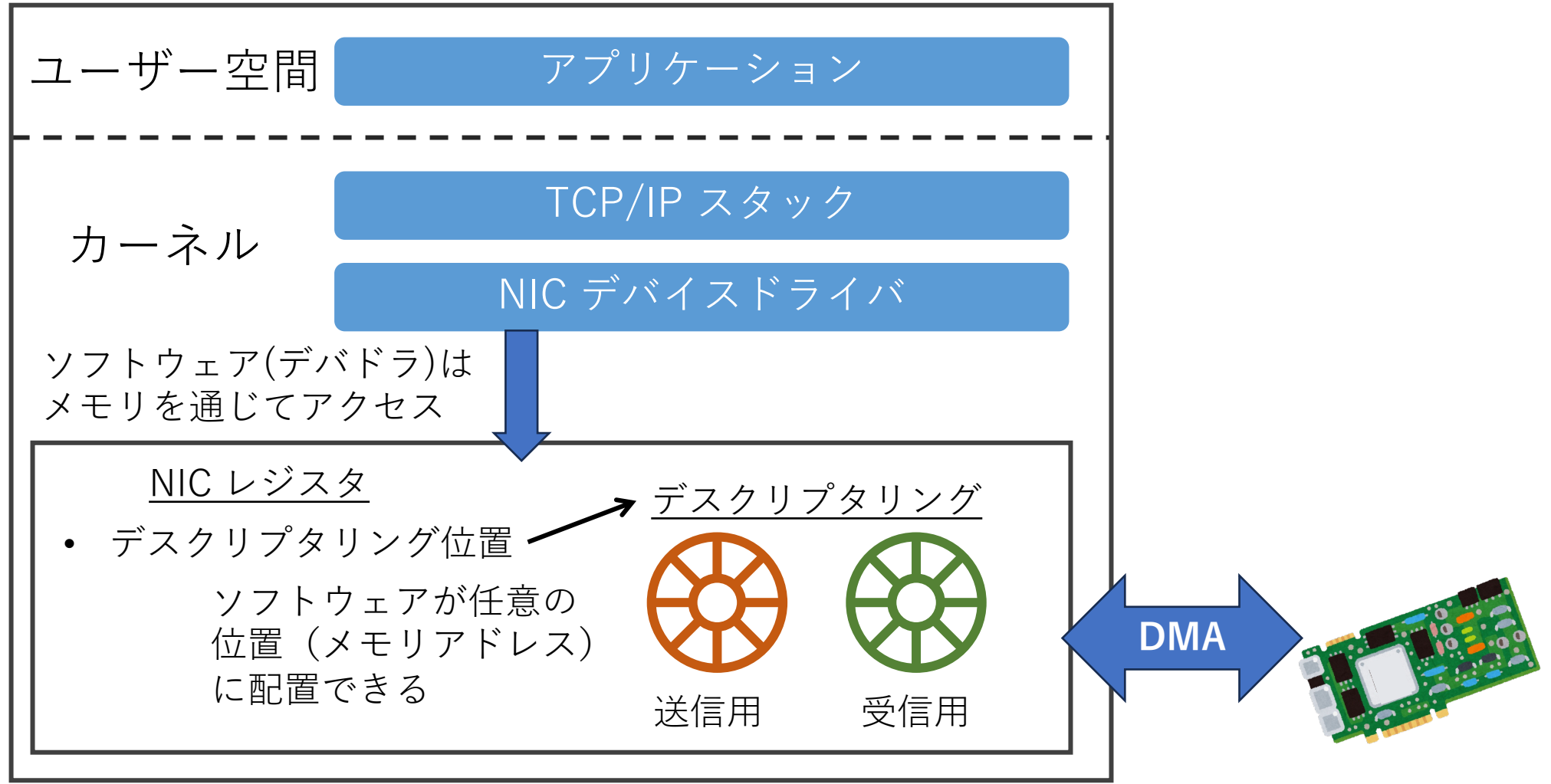
NIC と通信関連プログラムの構成



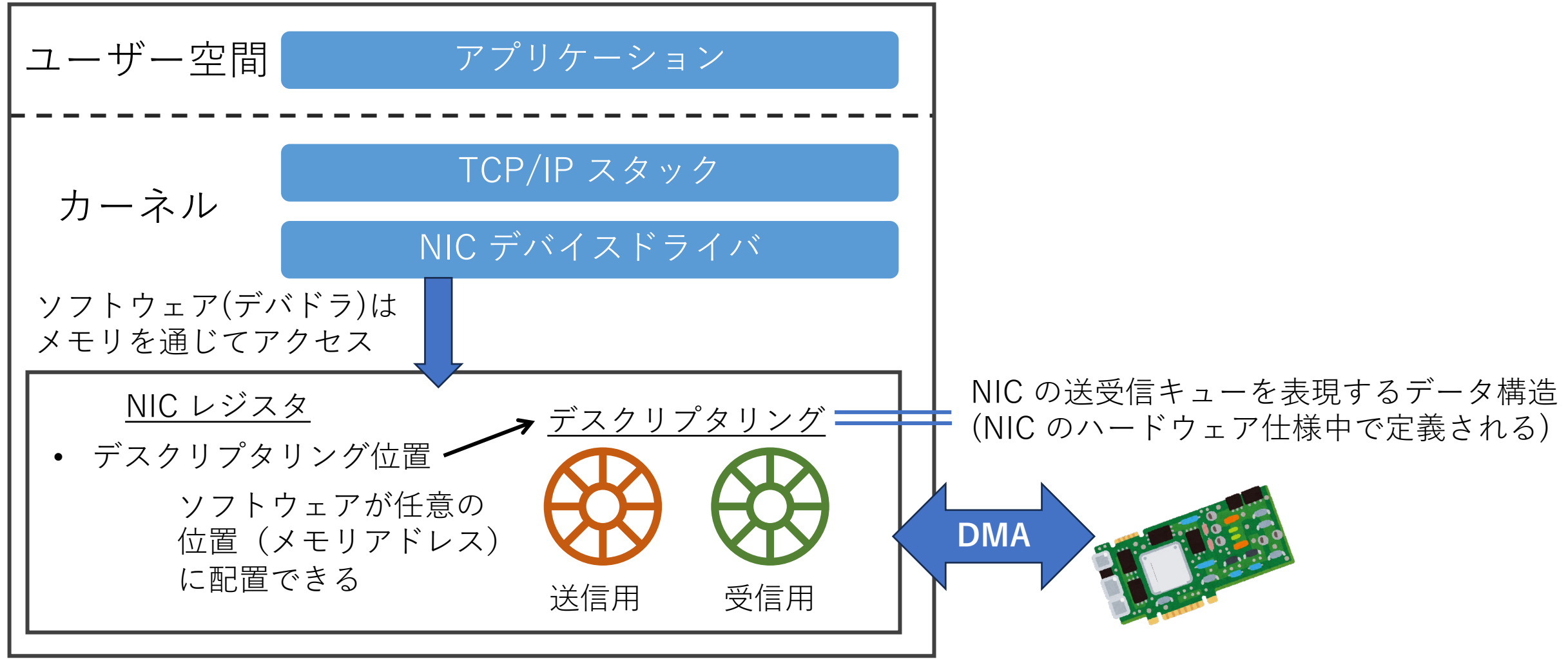
NIC と通信関連プログラムの構成



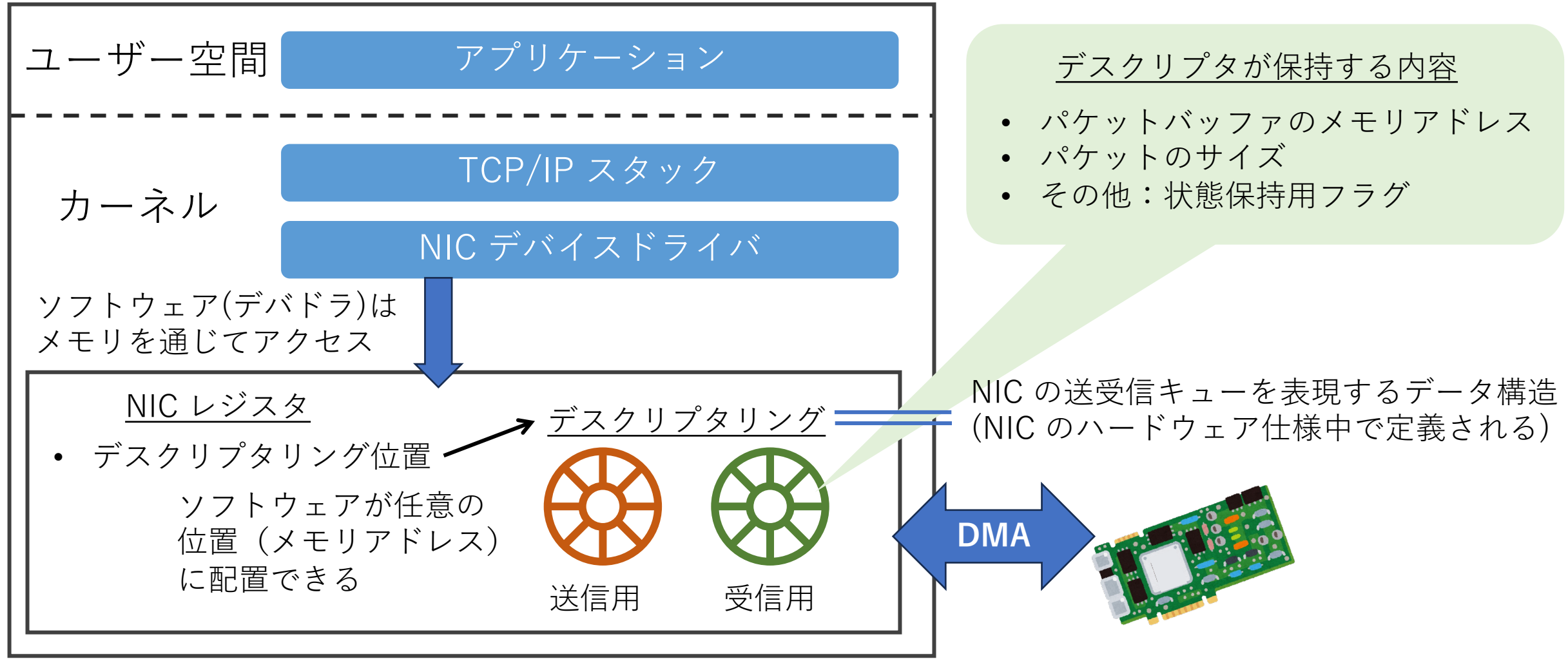
NIC と通信関連プログラムの構成



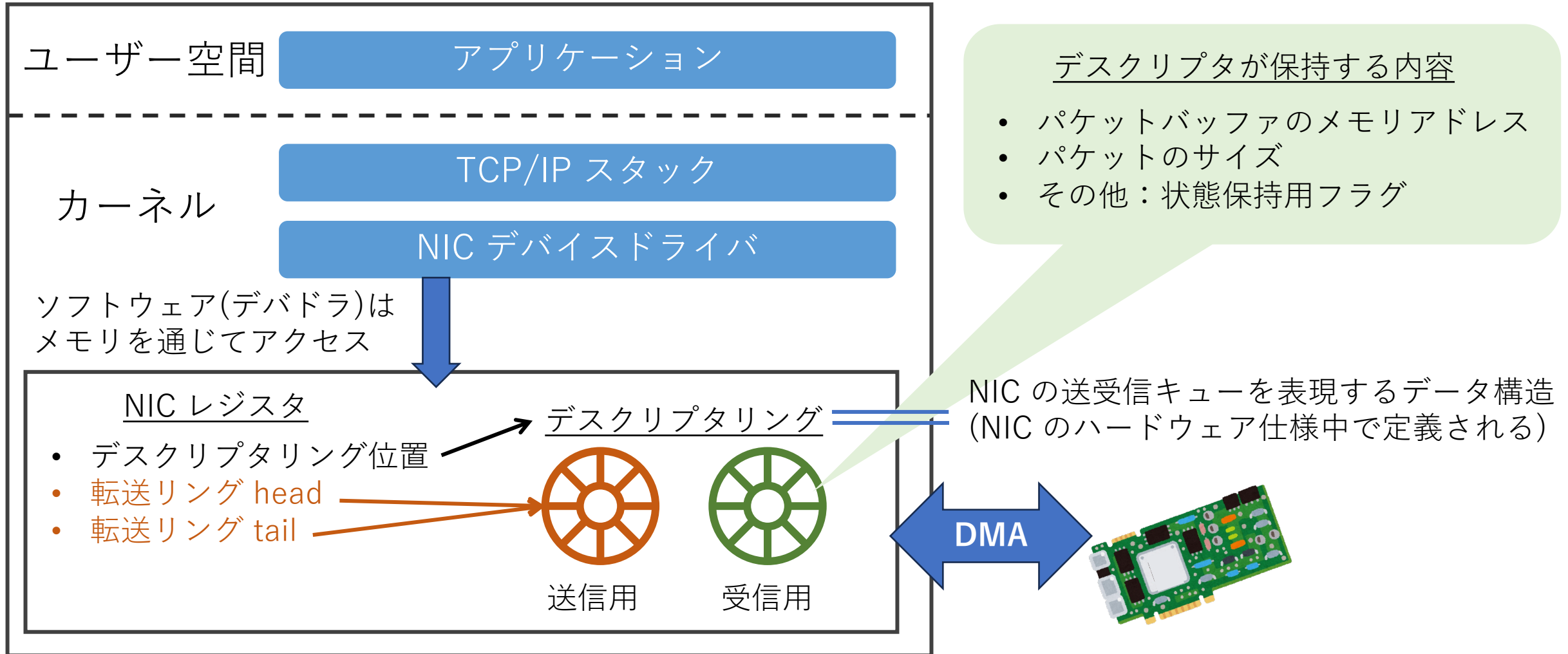
NIC と通信関連プログラムの構成



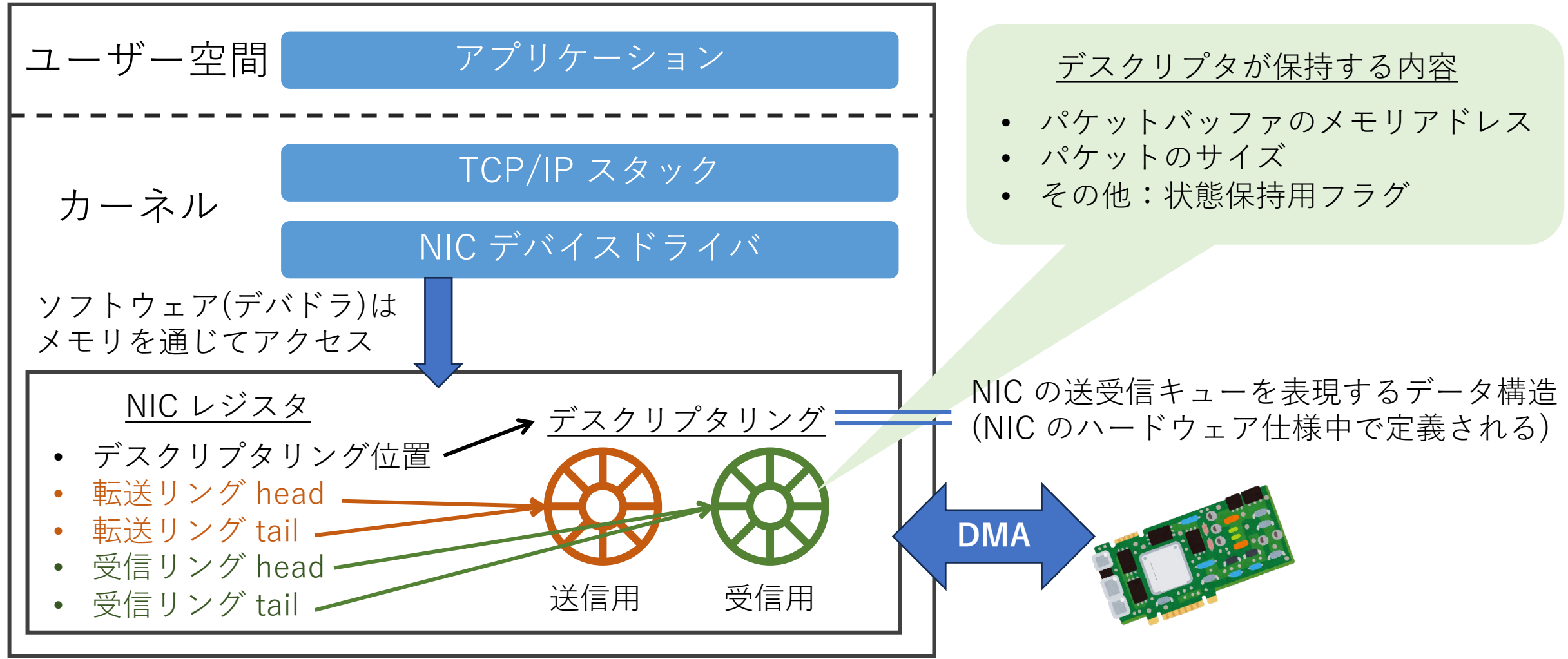
NIC と通信関連プログラムの構成



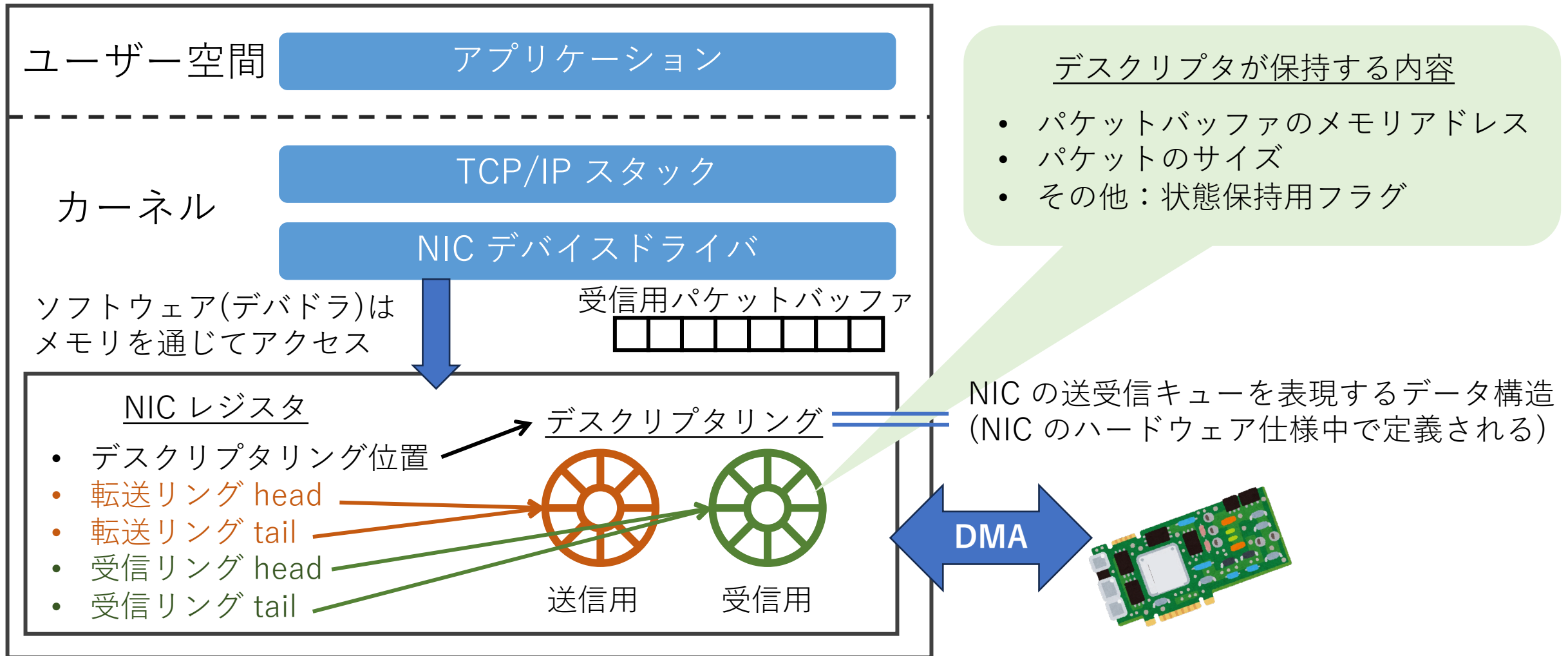
NIC と通信関連プログラムの構成



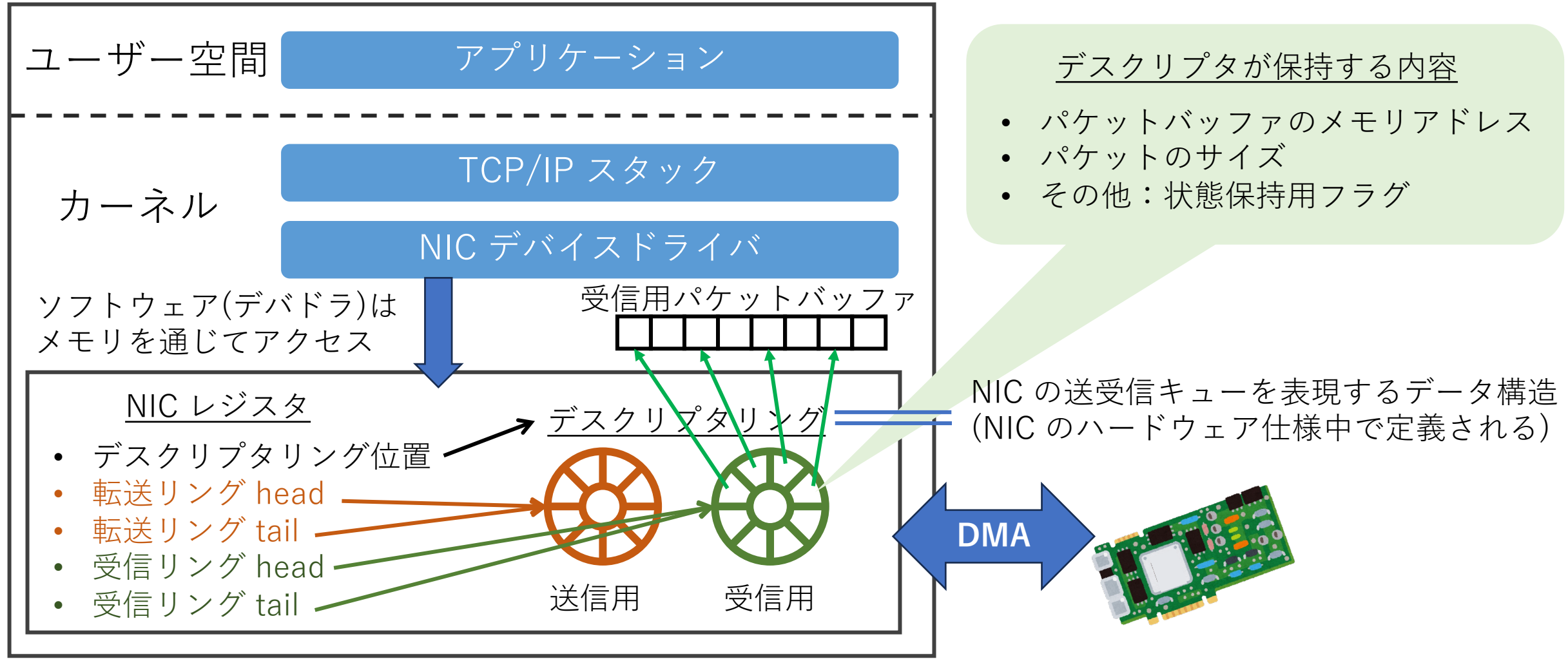
NIC と通信関連プログラムの構成



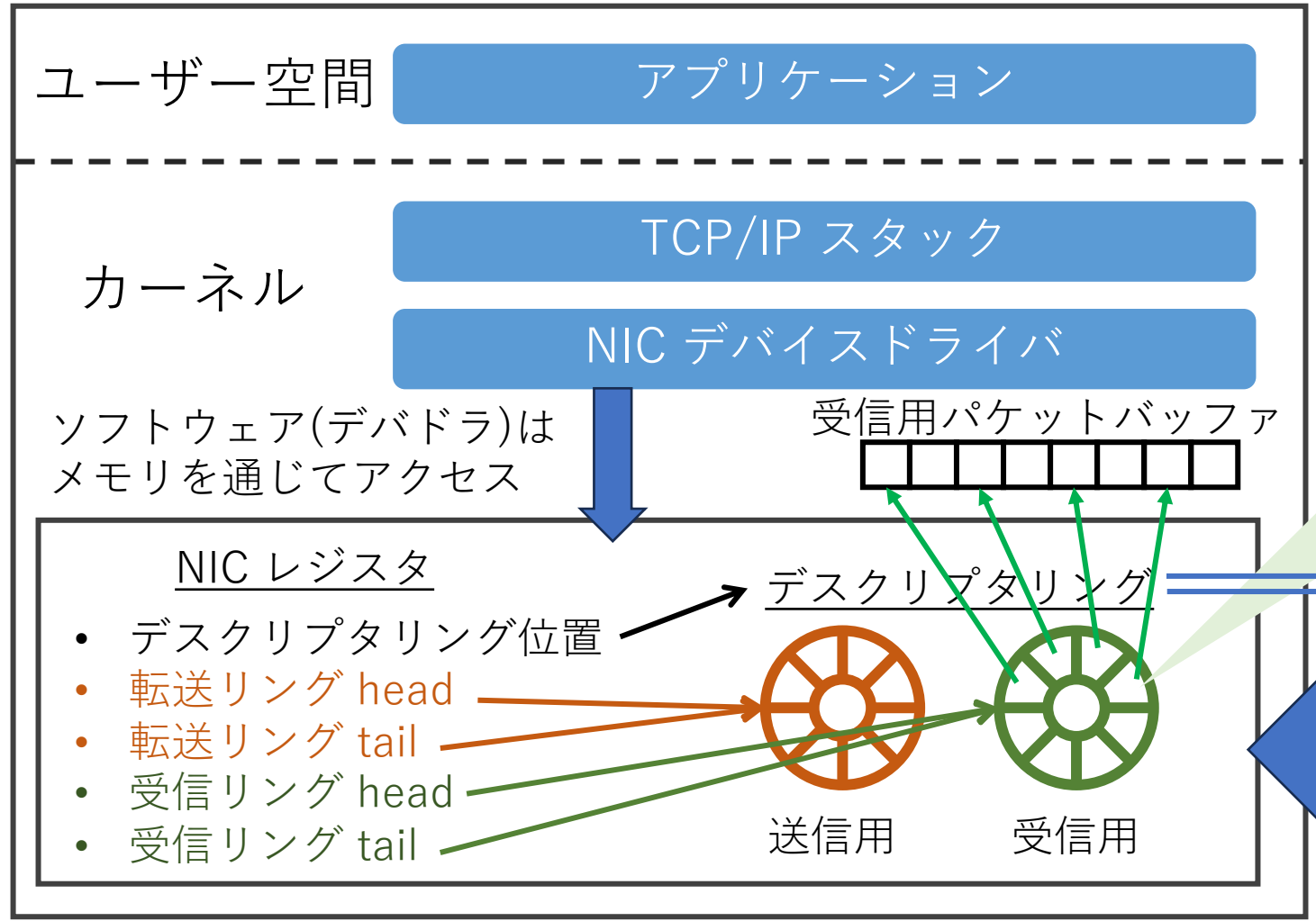
NIC と通信関連プログラムの構成



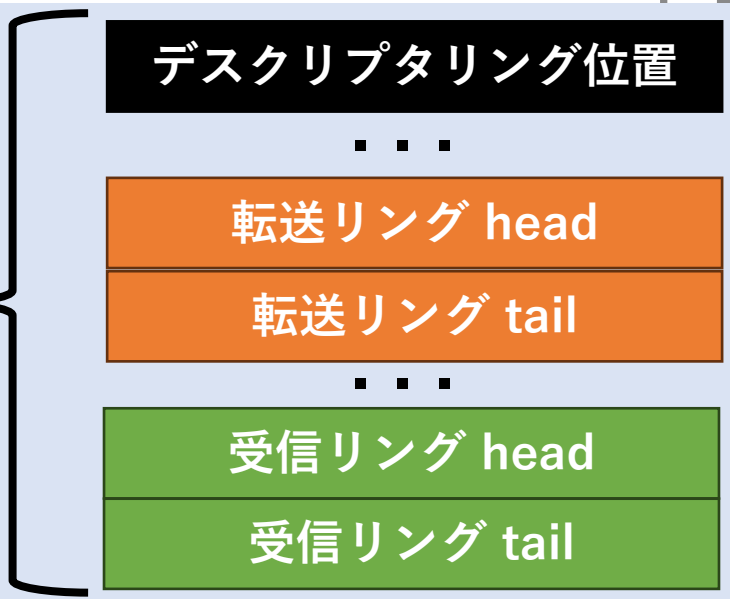
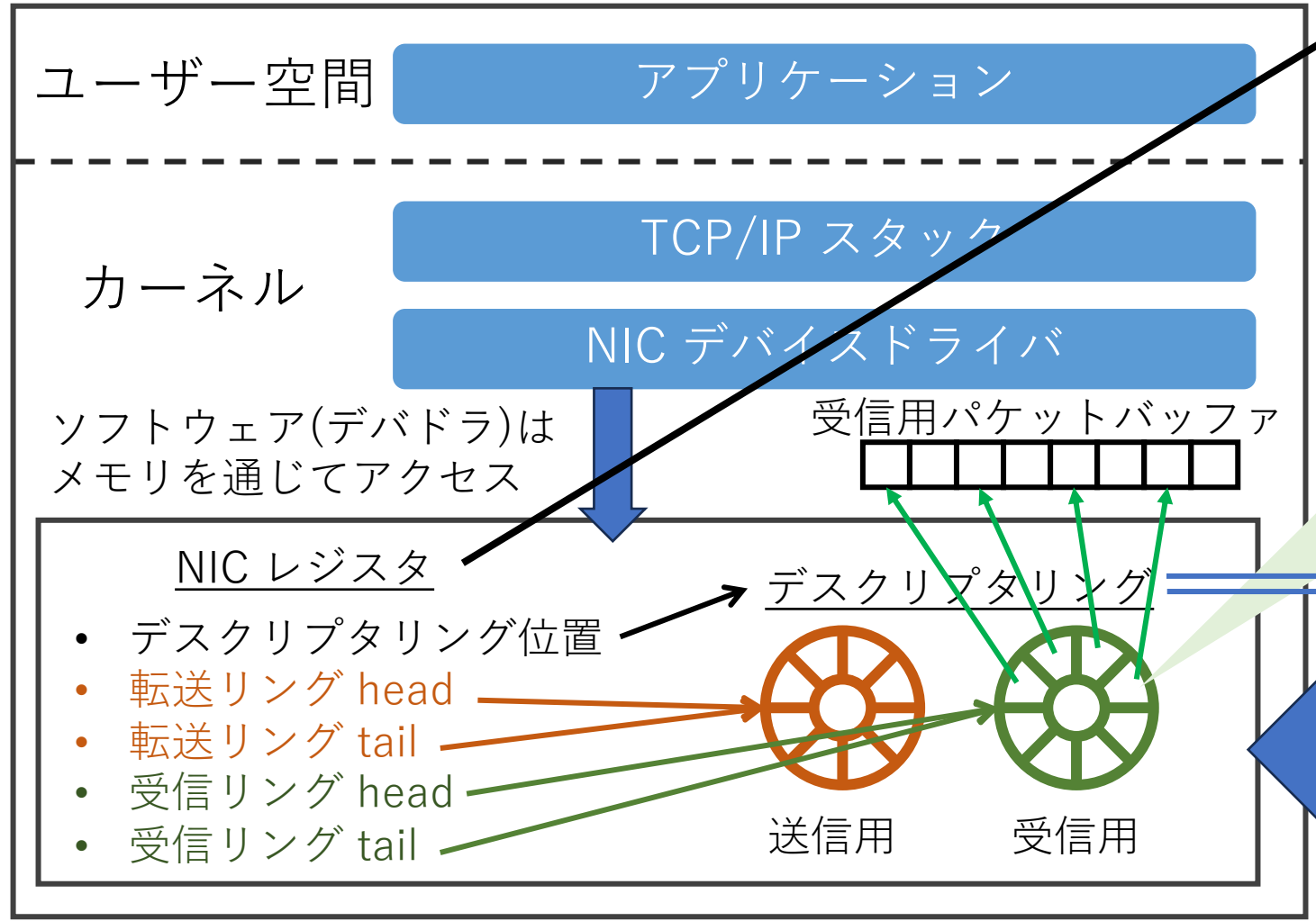
NIC と通信関連プログラムの構成



NIC と通信関連プログラム

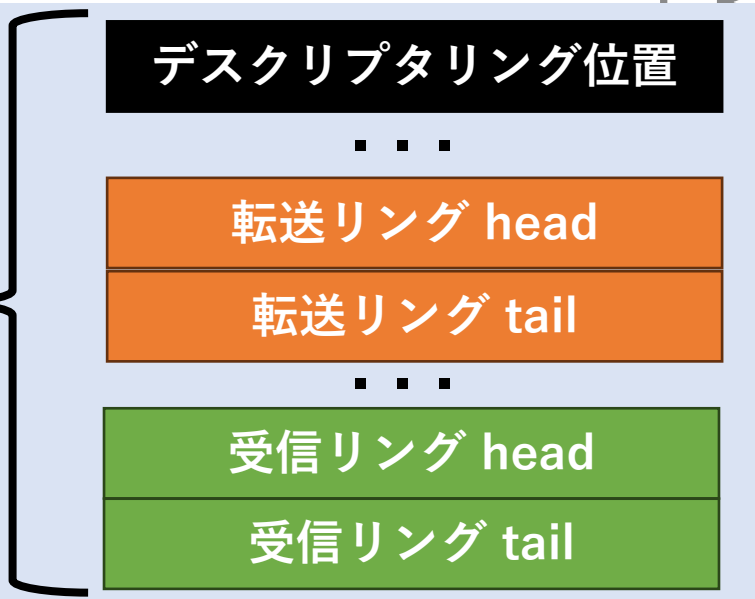
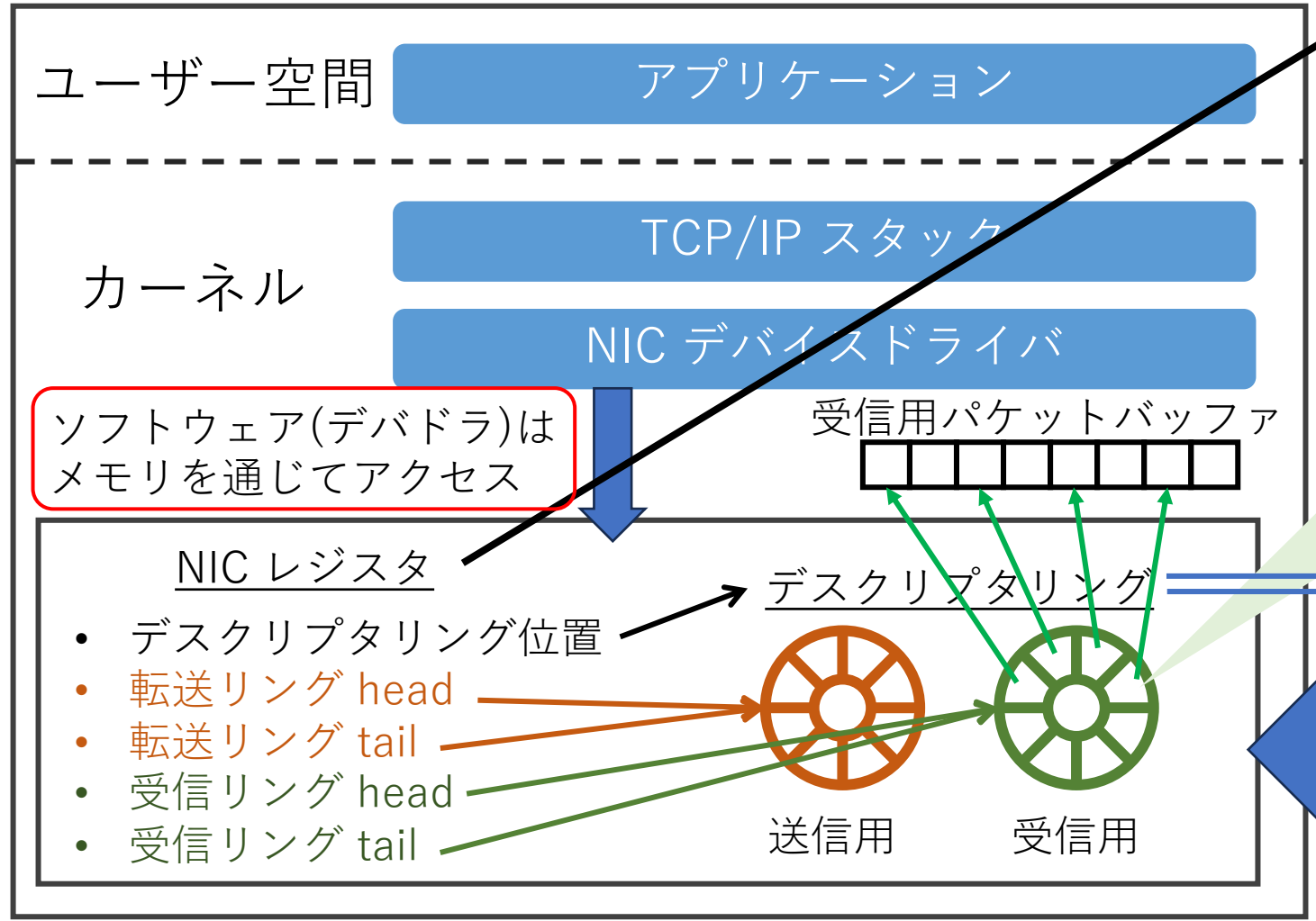


NIC と通信関連プログラム



メモリ上の概観

NIC と通信関連プログラム

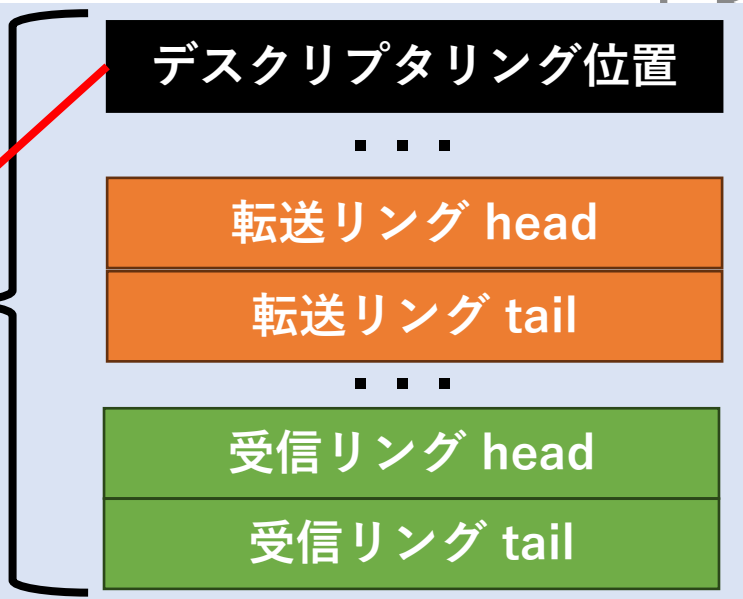
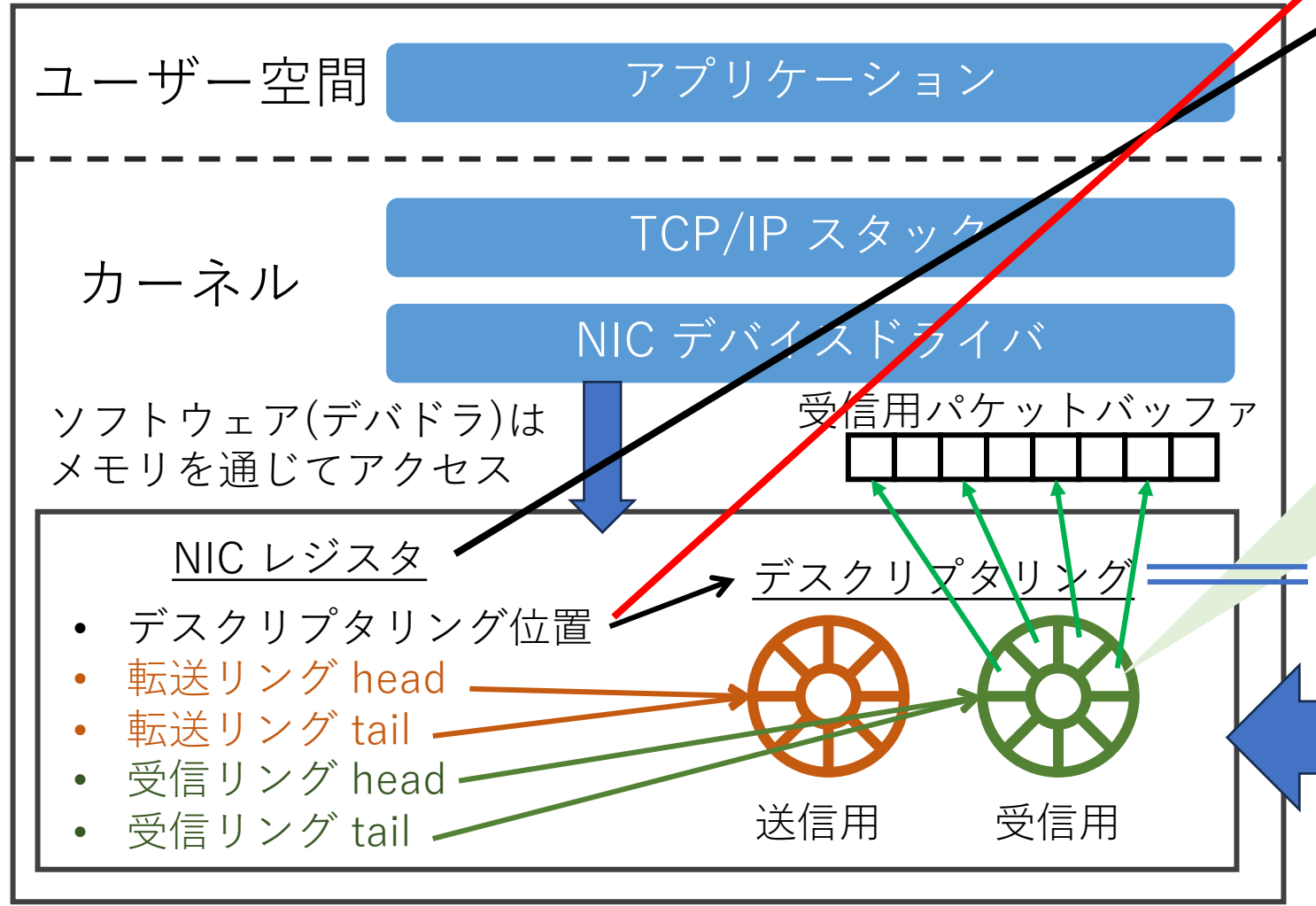


ソフトウェアはメモリを通じてNICのレジスタにアクセス可能

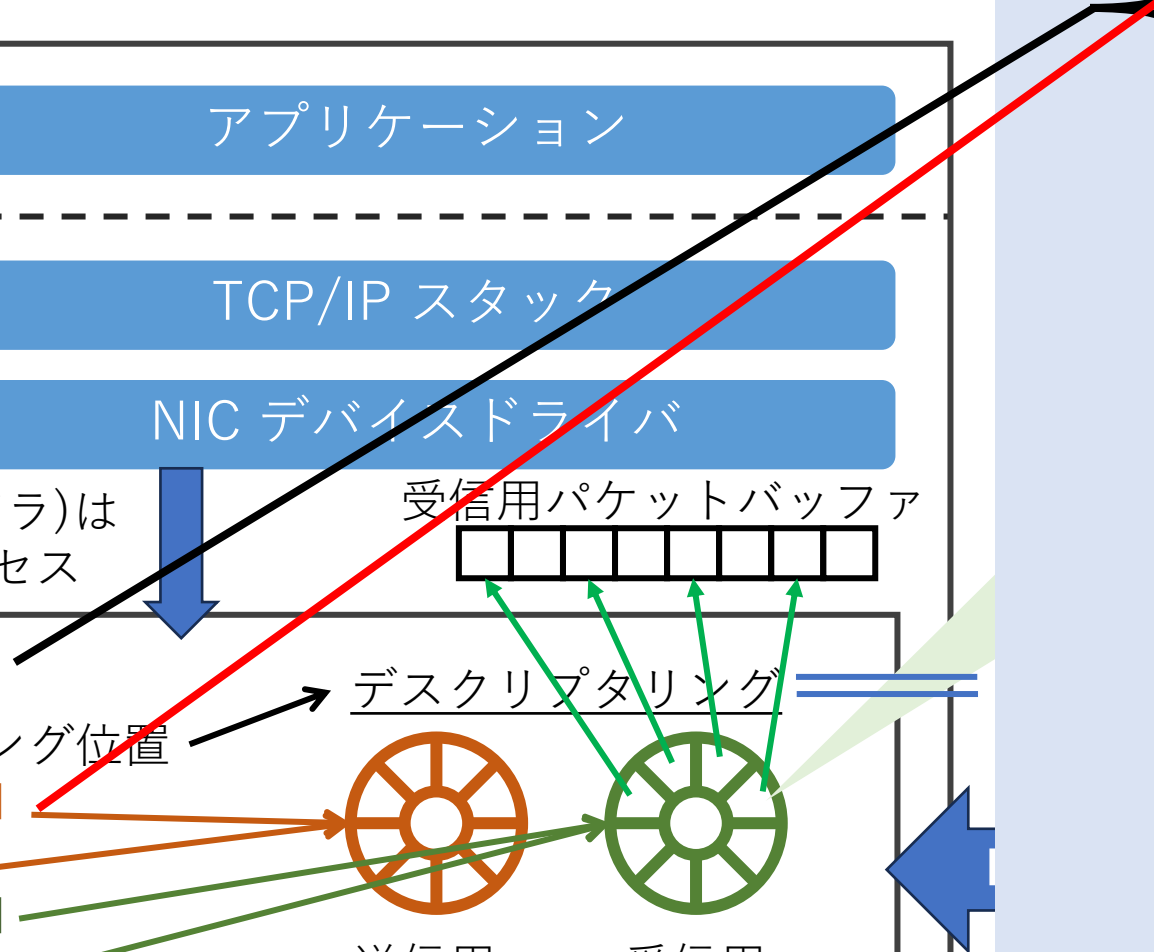
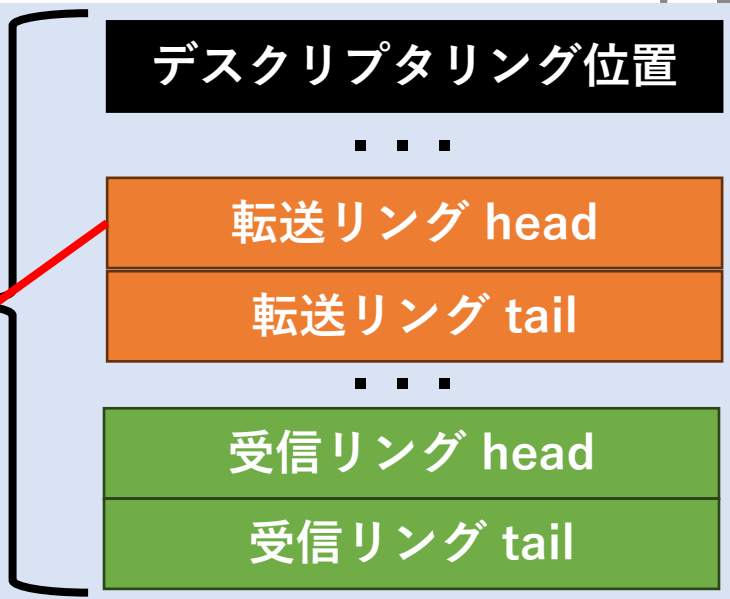
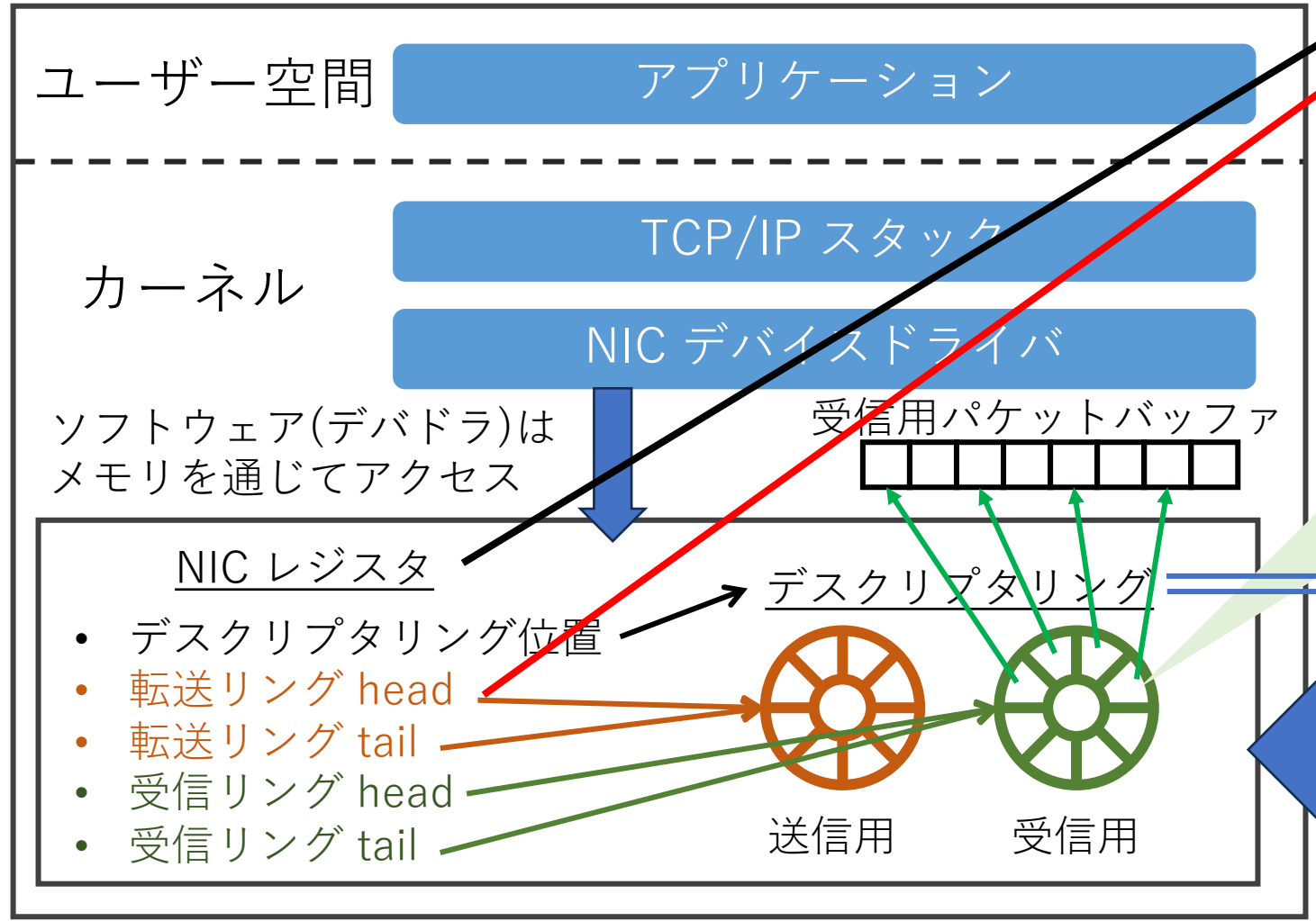
- **デスクリプタリング位置**
- **転送リング head**
- **転送リング tail**
- **受信リング head**
- **受信リング tail**

メモリ上の概観

NIC と通信関連プログラム

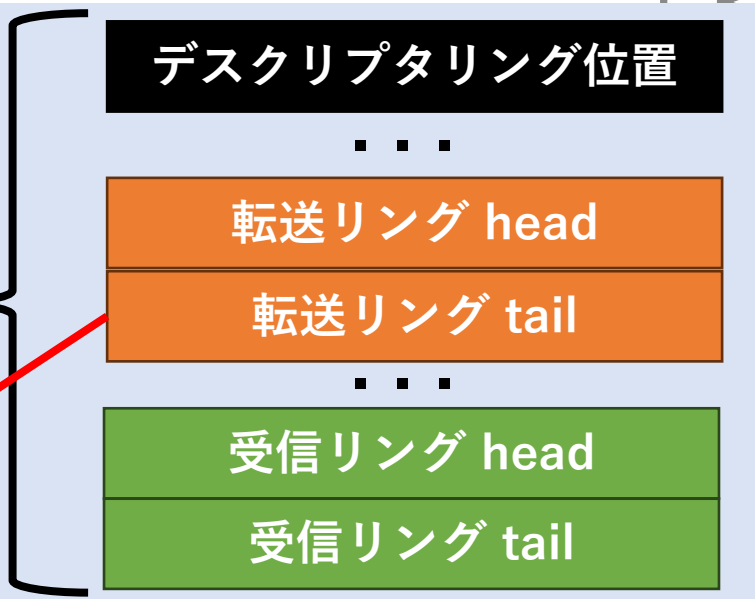
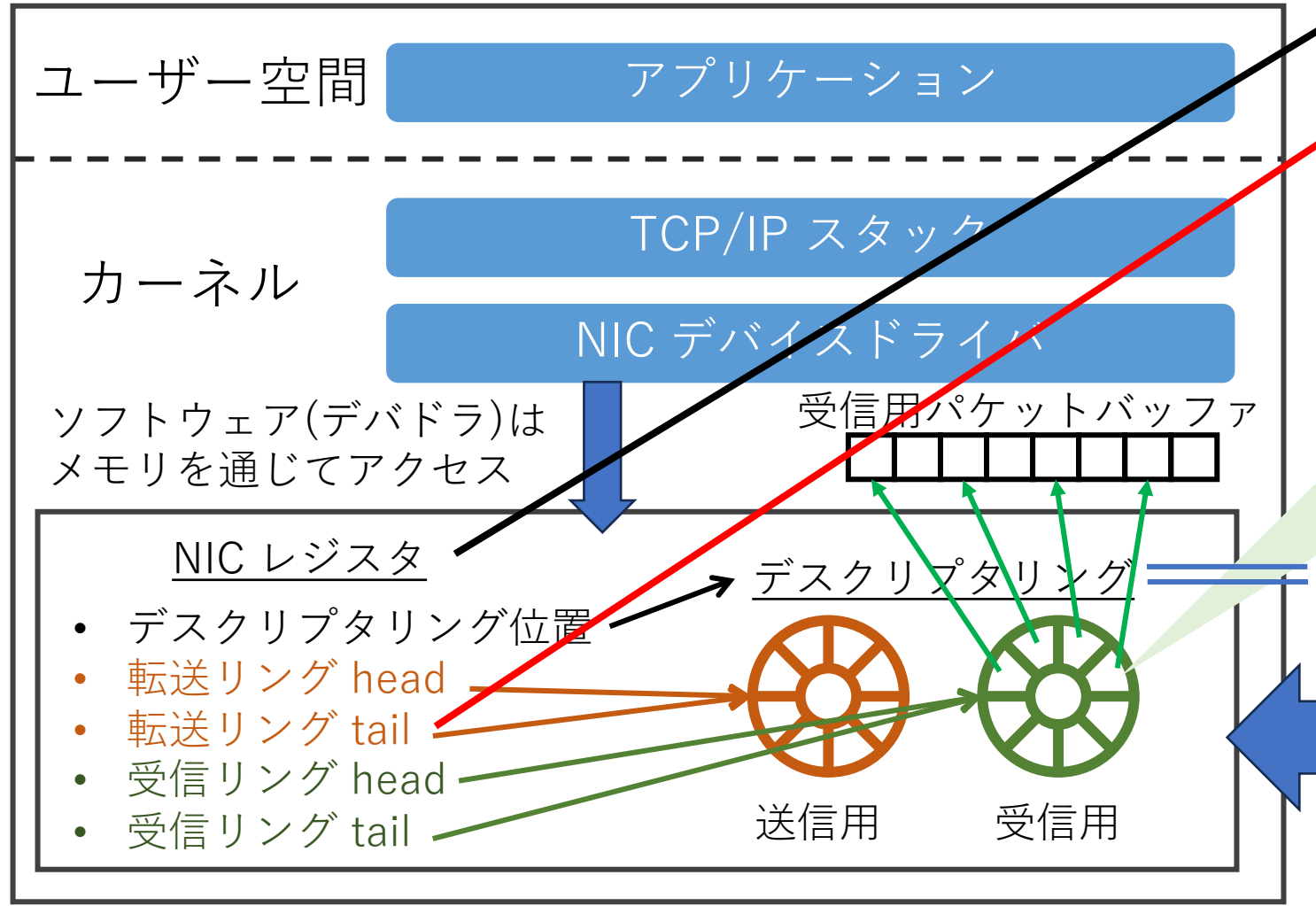


NIC と通信関連プログラム



メモリ上の概観

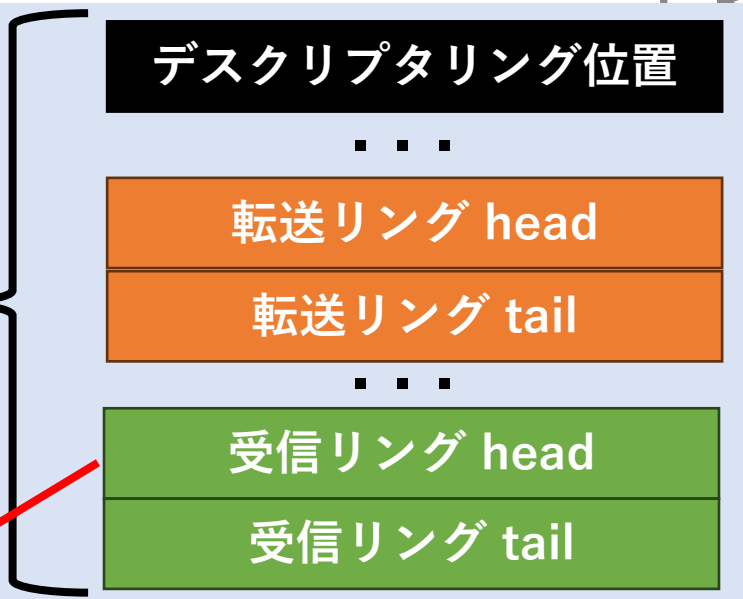
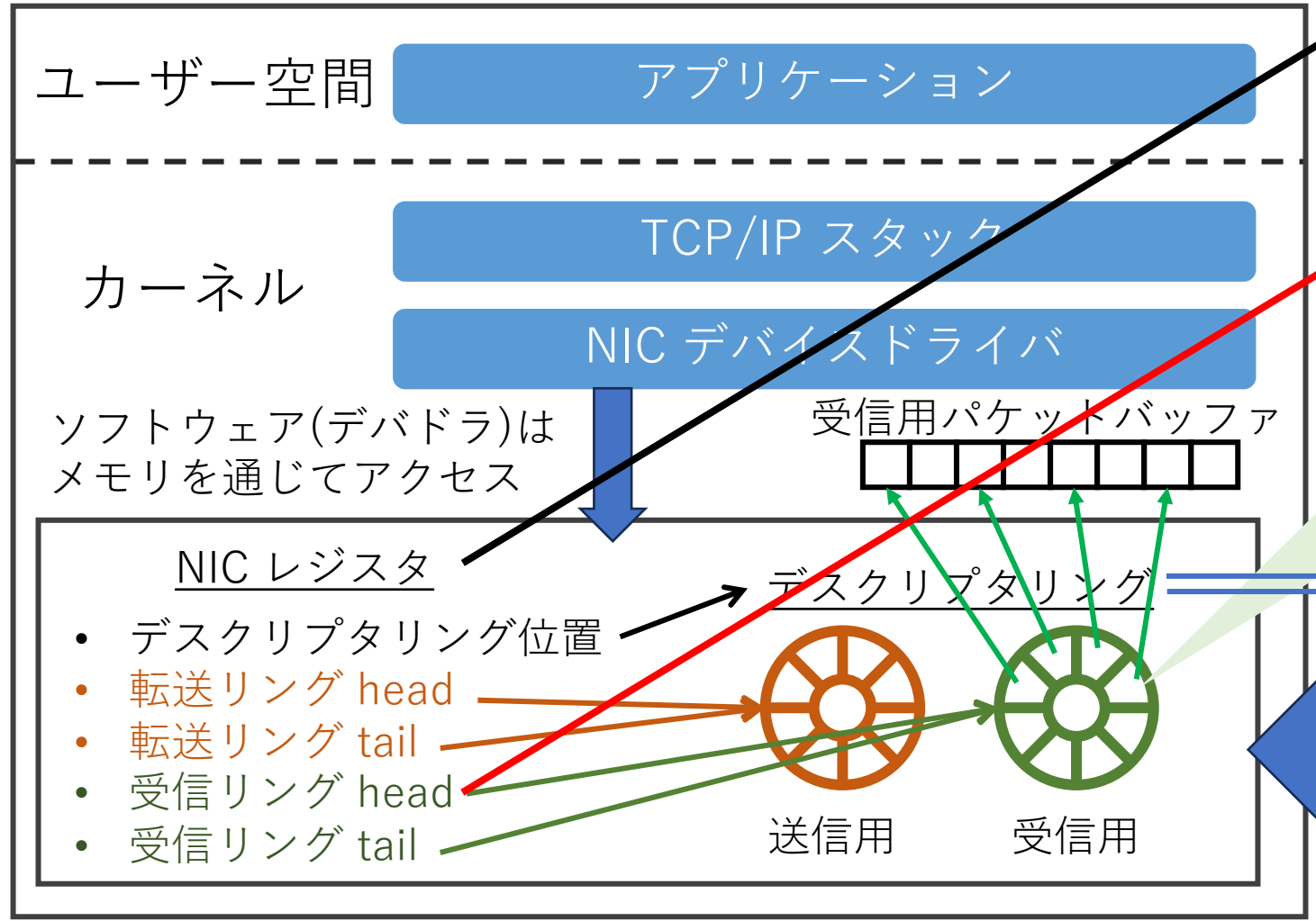
NIC と通信関連プログラム



- デスクリプタリング位置
- 転送リング head
- 転送リング tail
- 受信リング head
- 受信リング tail

メモリ上の概観

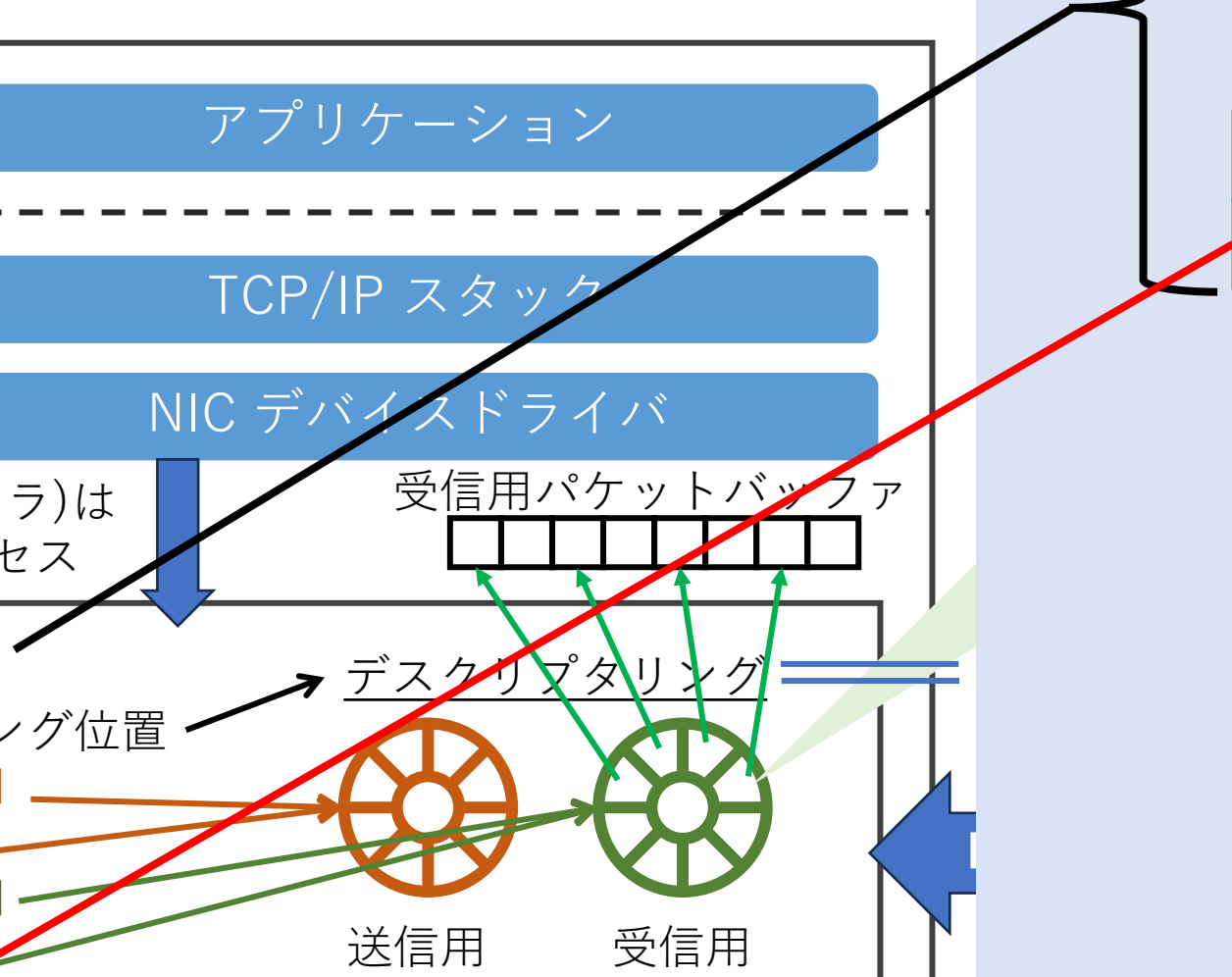
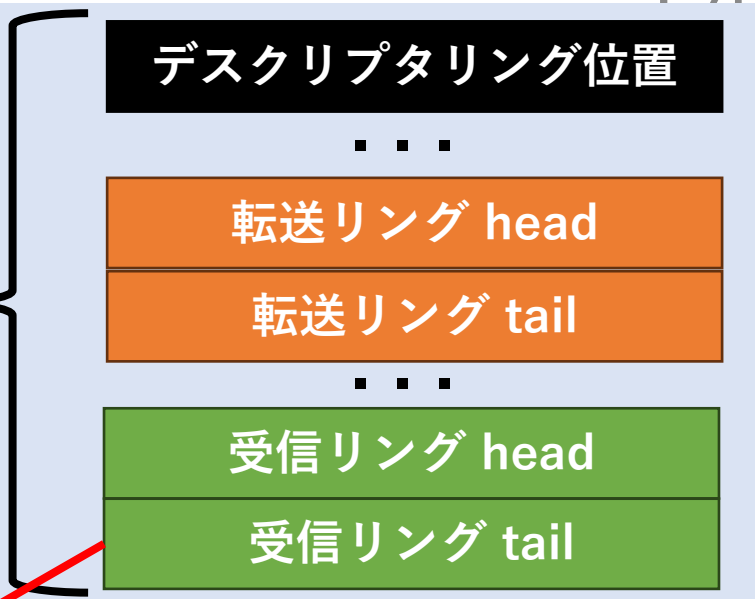
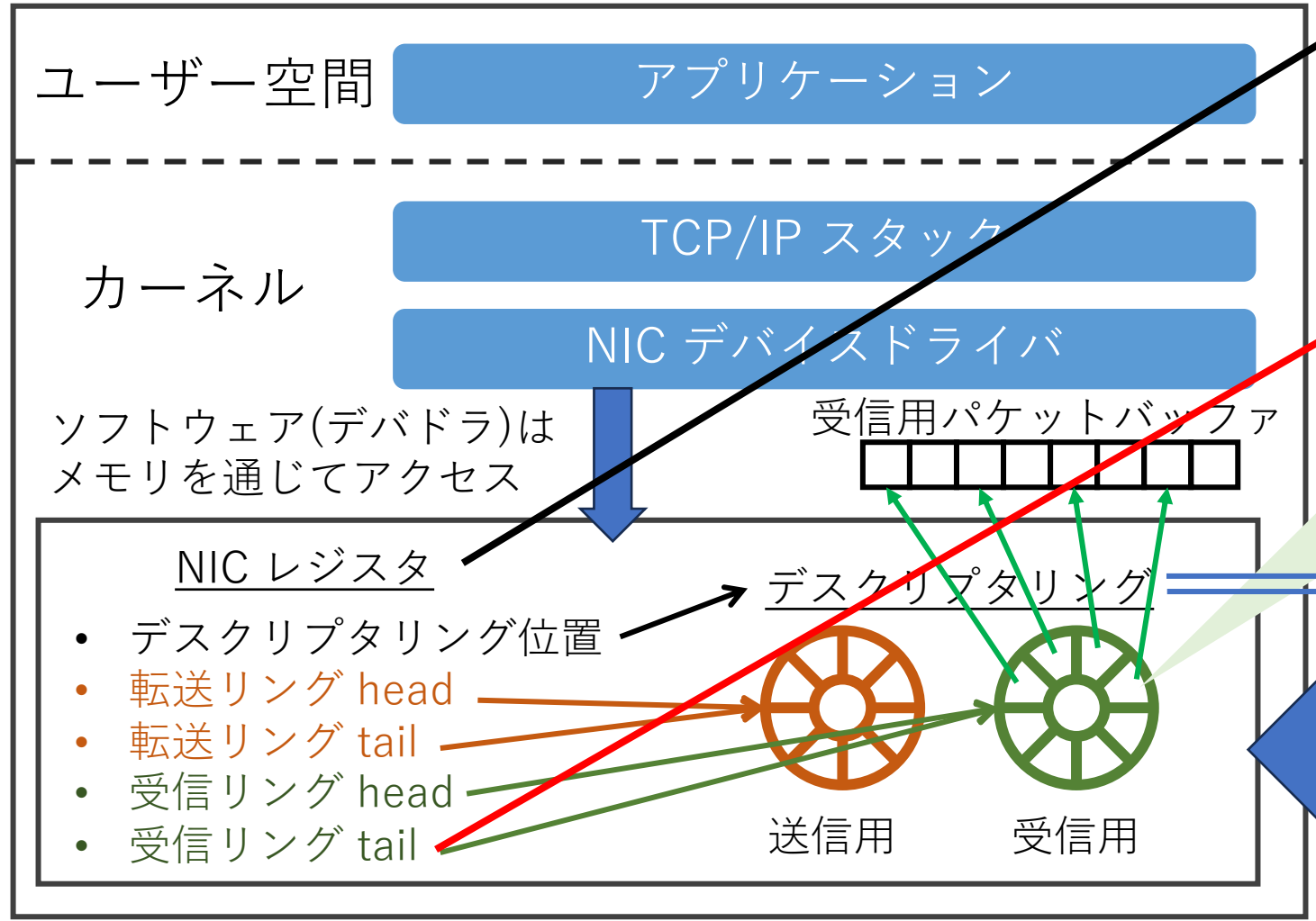
NIC と通信関連プログラム



- デスクリプタリング位置
- 転送リング head
- 転送リング tail
- 受信リング head
- 受信リング tail

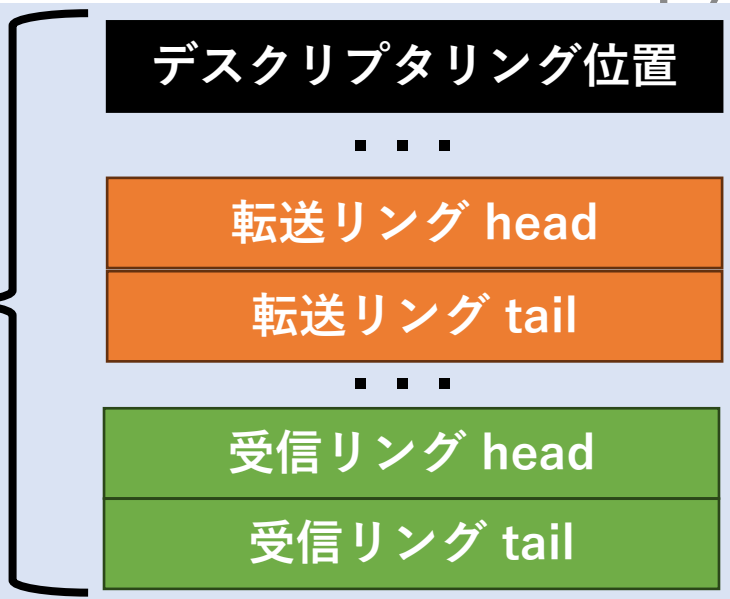
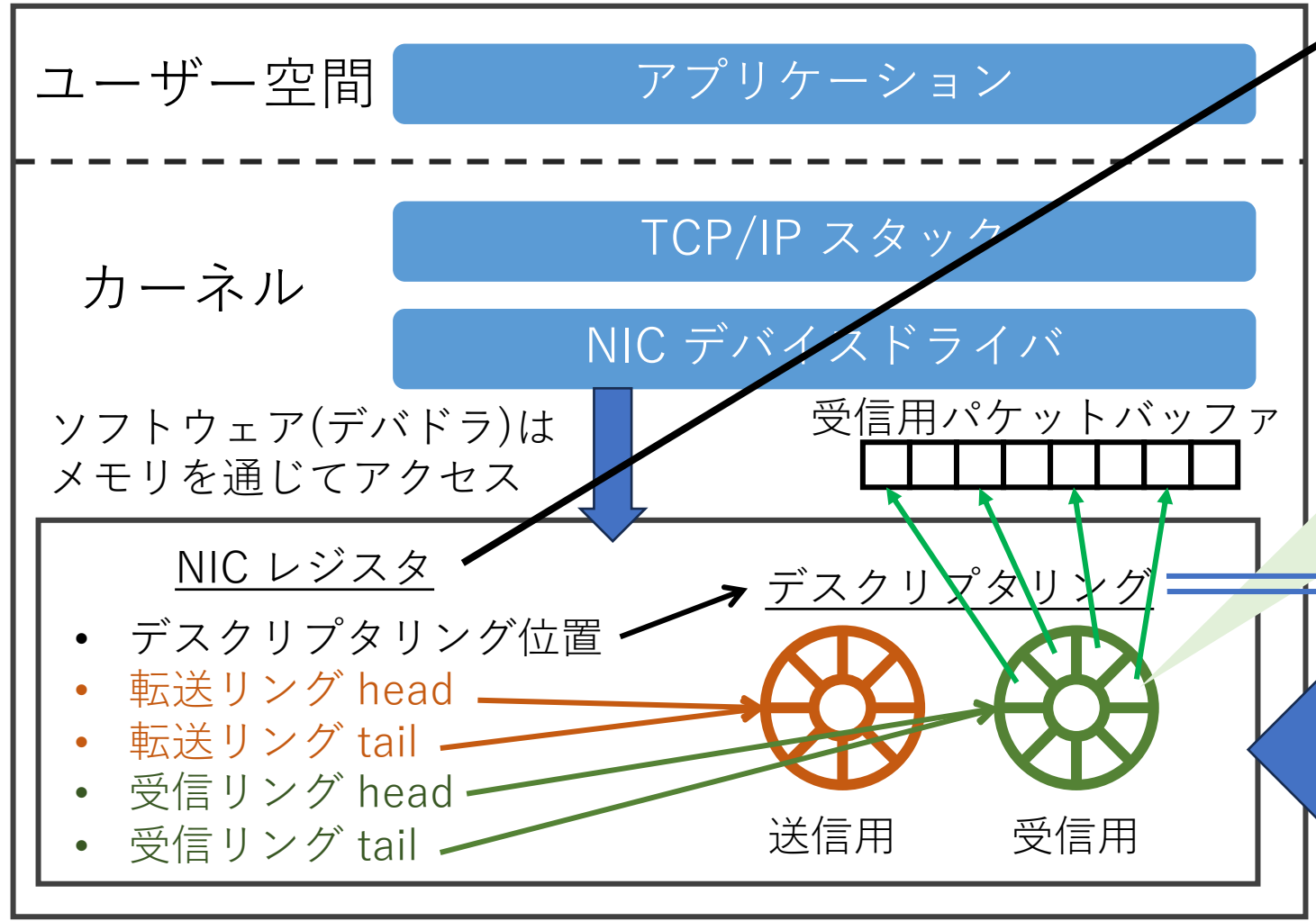
メモリ上の概観

NIC と通信関連プログラム

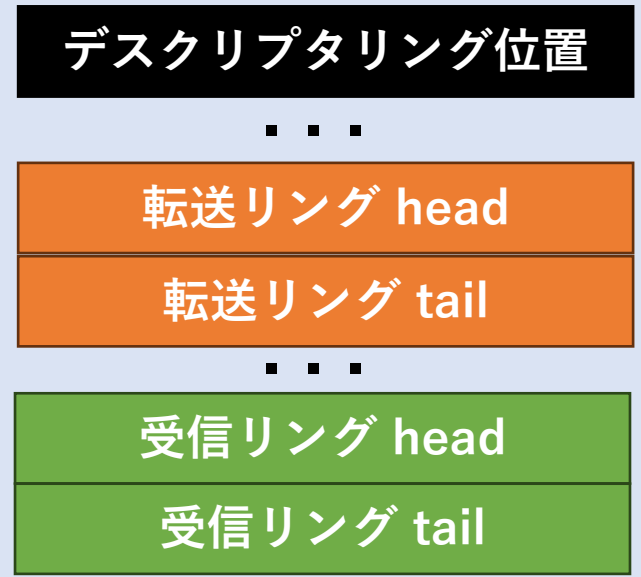
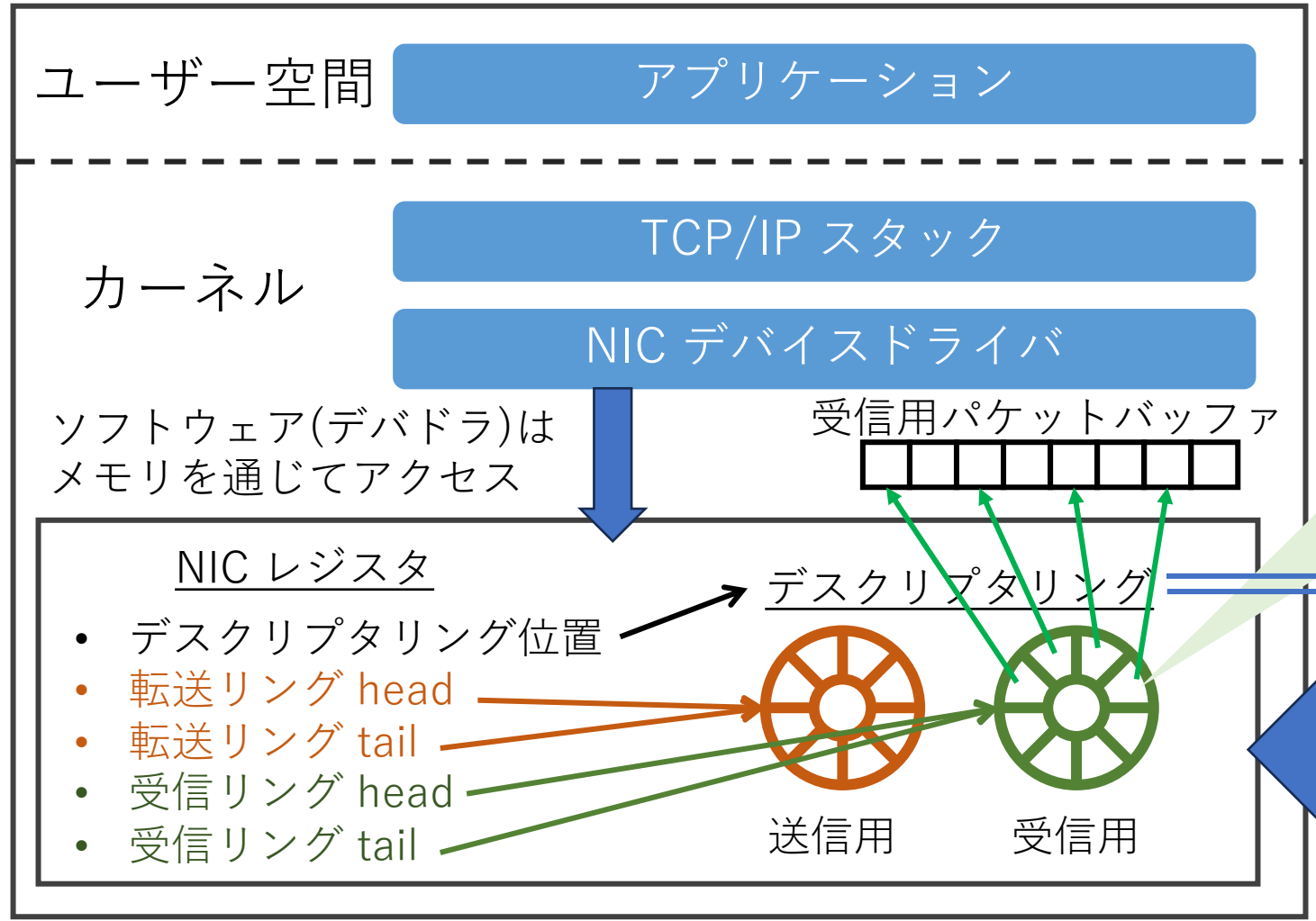


メモリ上の概観

NIC と通信関連プログラム



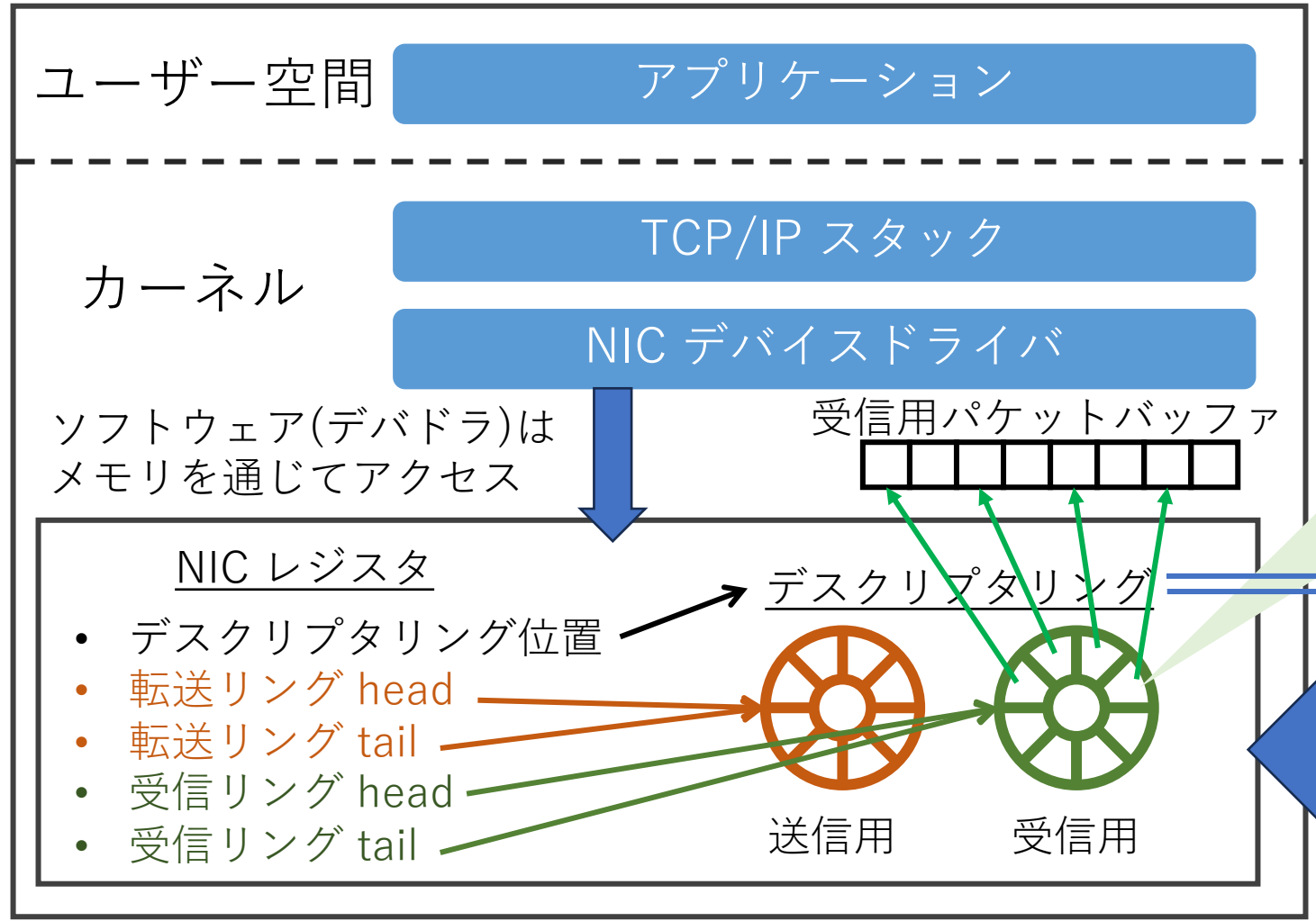
NIC と通信関連プログラム



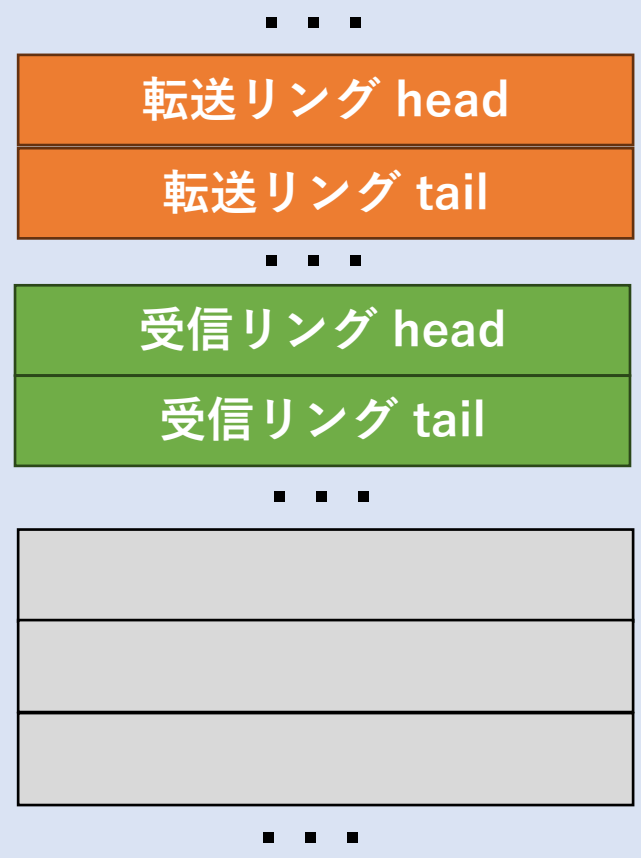
ソフトウェア (デバドラ) は初期設定として、まずデスクリプタリング用に連続的なメモリ領域を確保

メモリ上の概観

NIC と通信関連プログラム

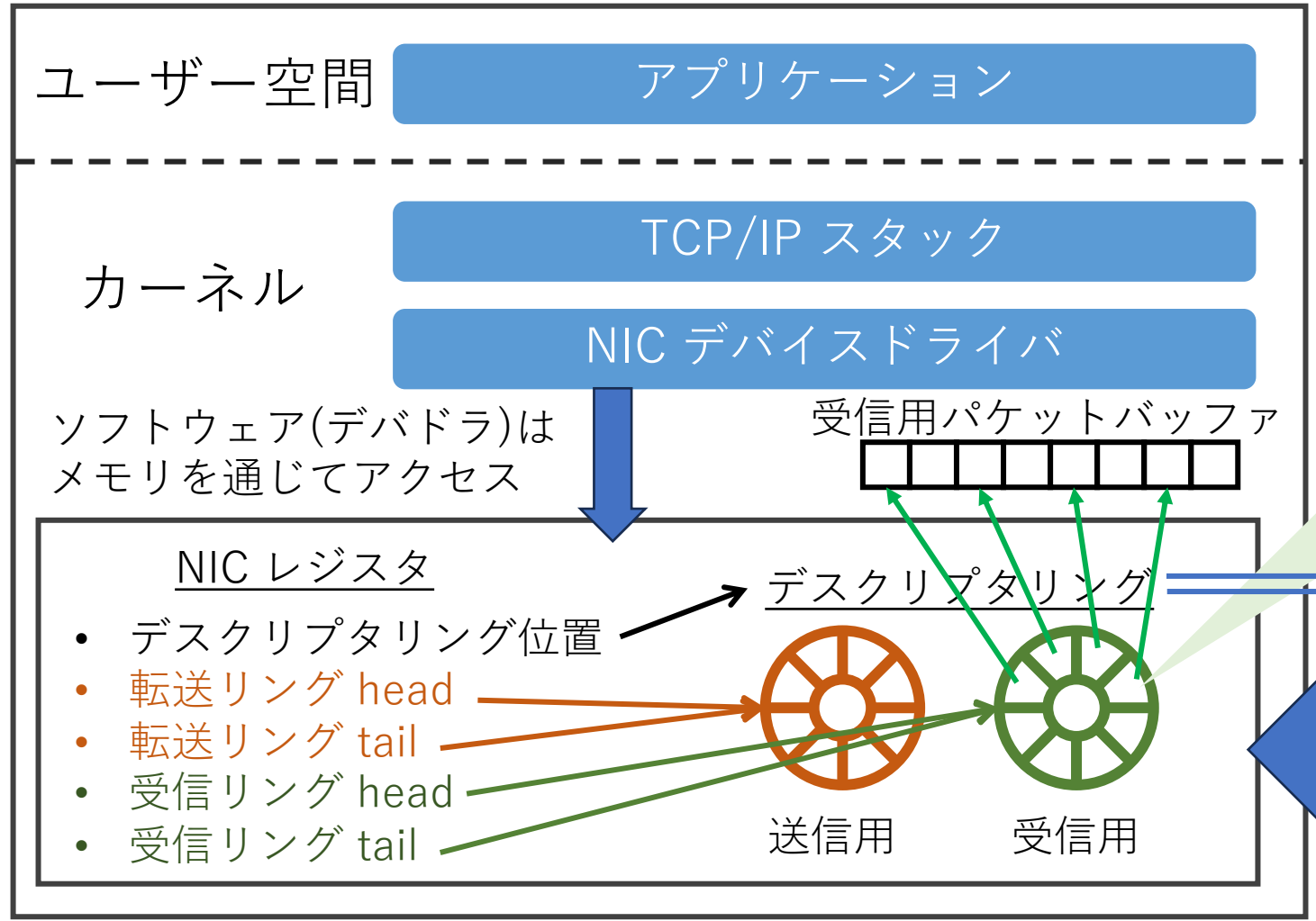


デスクリプタリング位置



ソフトウェア (デバドラ) は初期設定として、まずデスクリプタリング用に連続的なメモリ領域を確保

NIC と通信関連プログラム



デスクリプタリング位置

転送リング head

転送リング tail

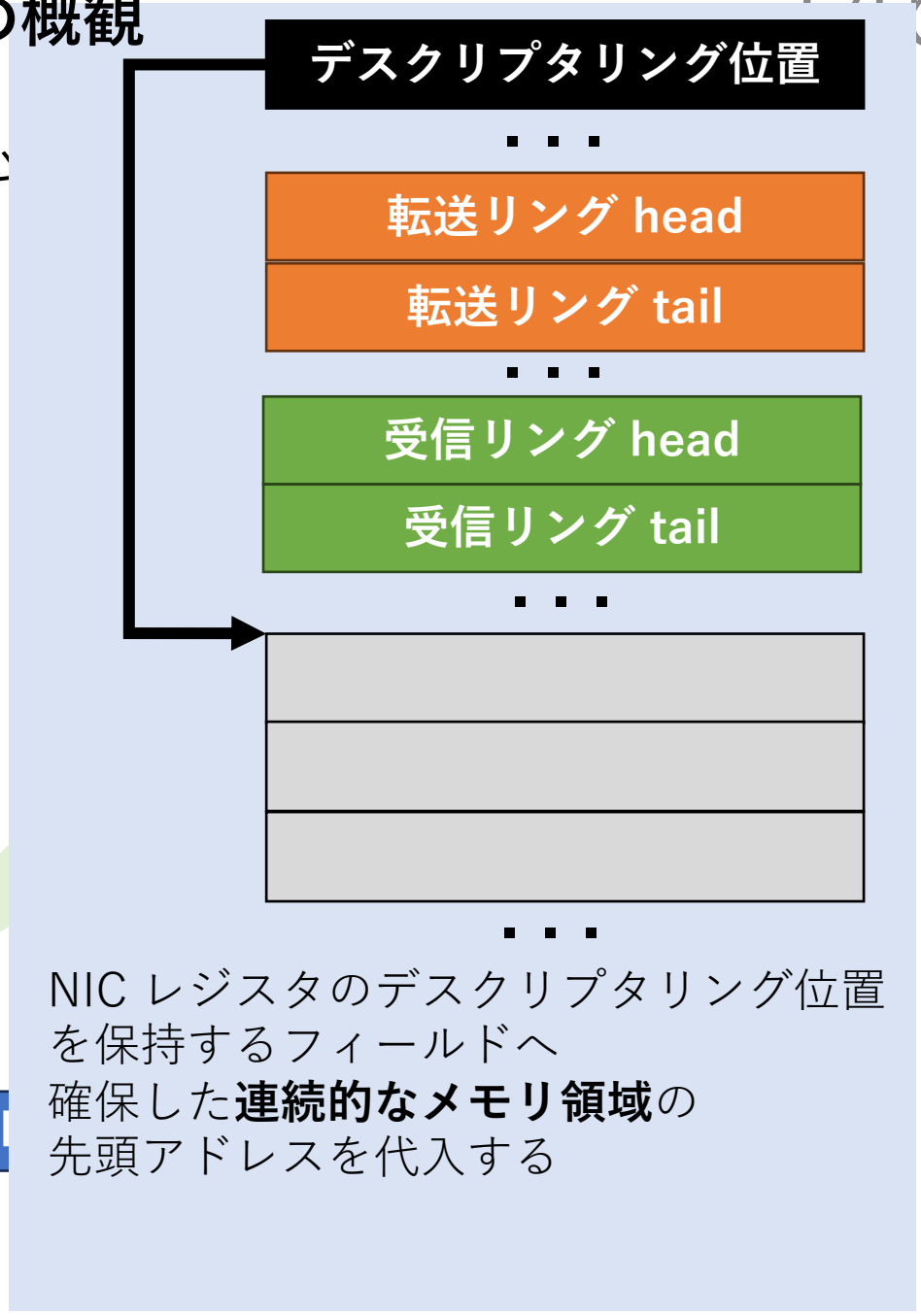
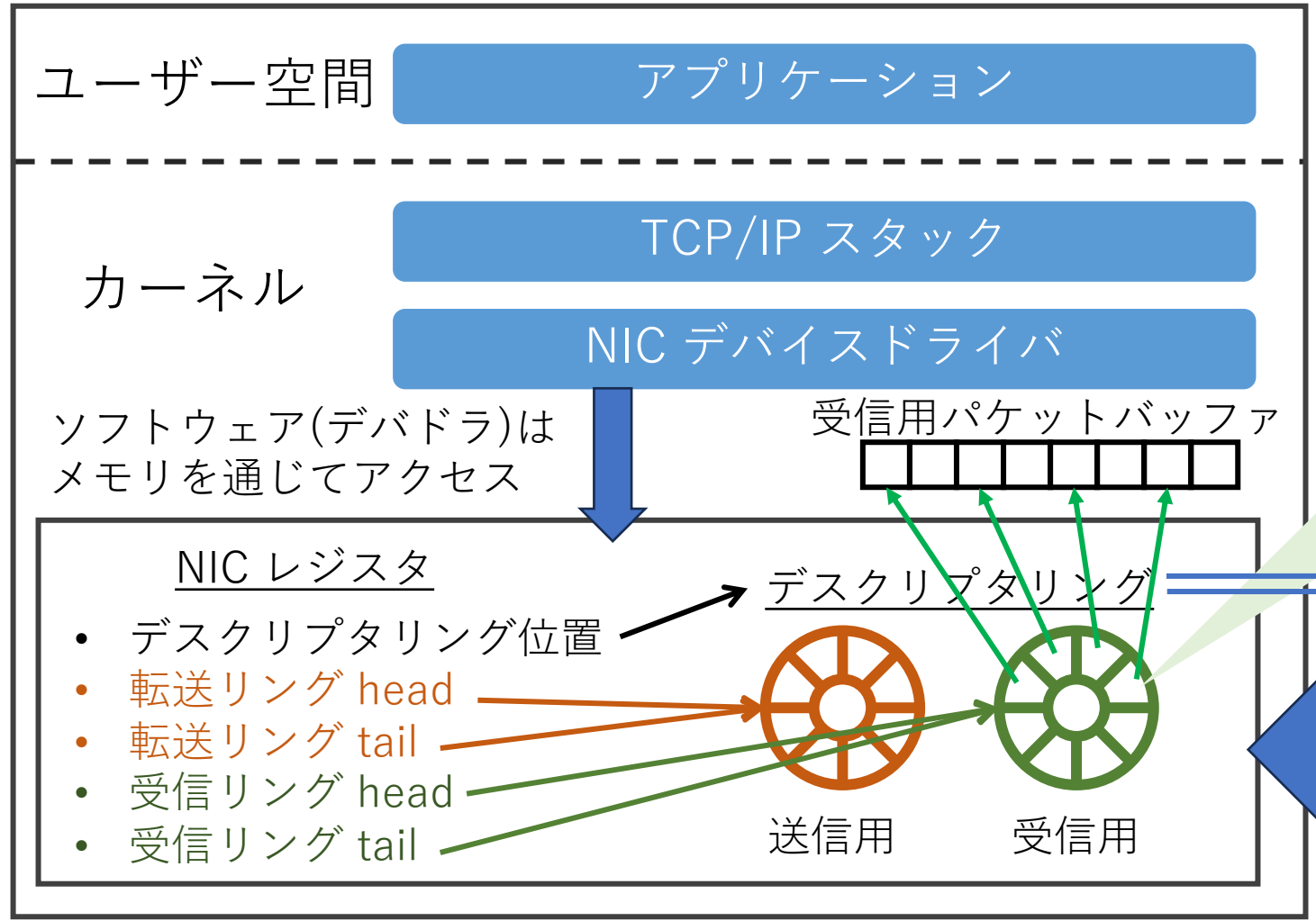
受信リング head

受信リング tail

NIC レジスタのデスクリプタリング位置を保持するフィールドへ確保した**連続的なメモリ領域**の先頭アドレスを代入する

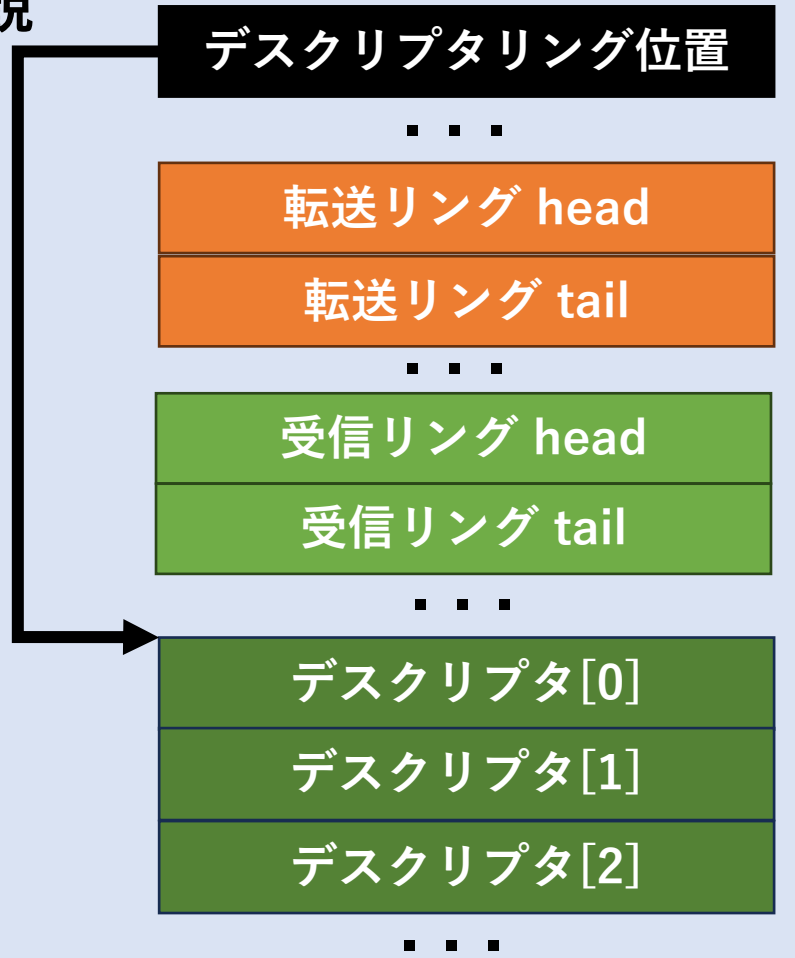
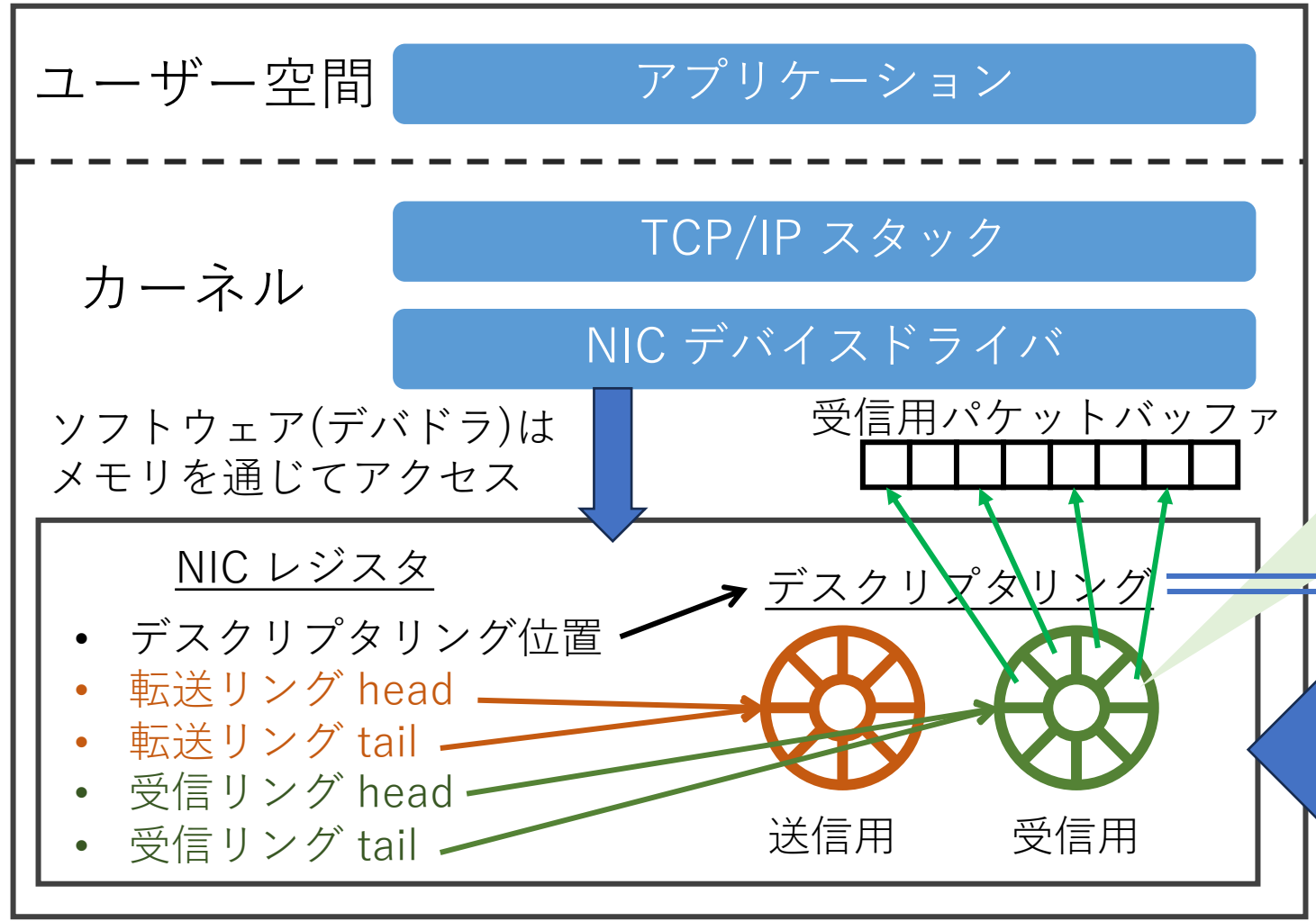
メモリ上の概観

NIC と通信関連プログラム



スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

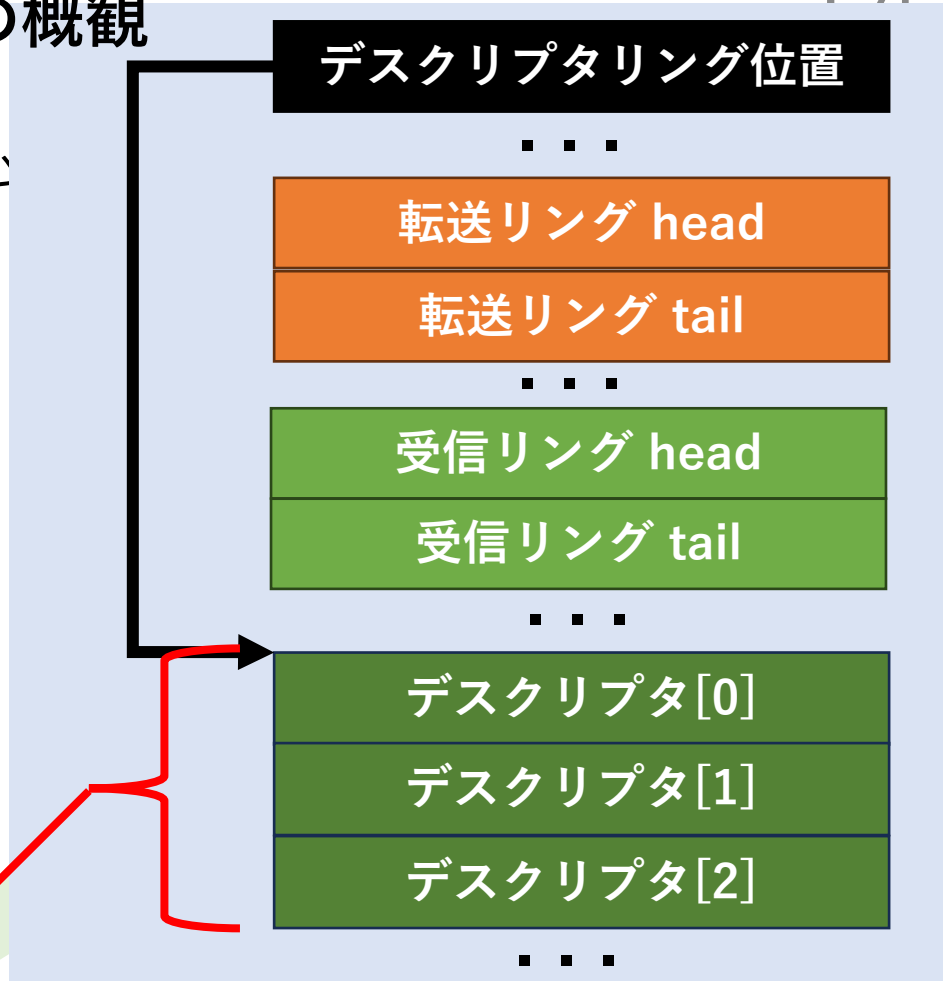
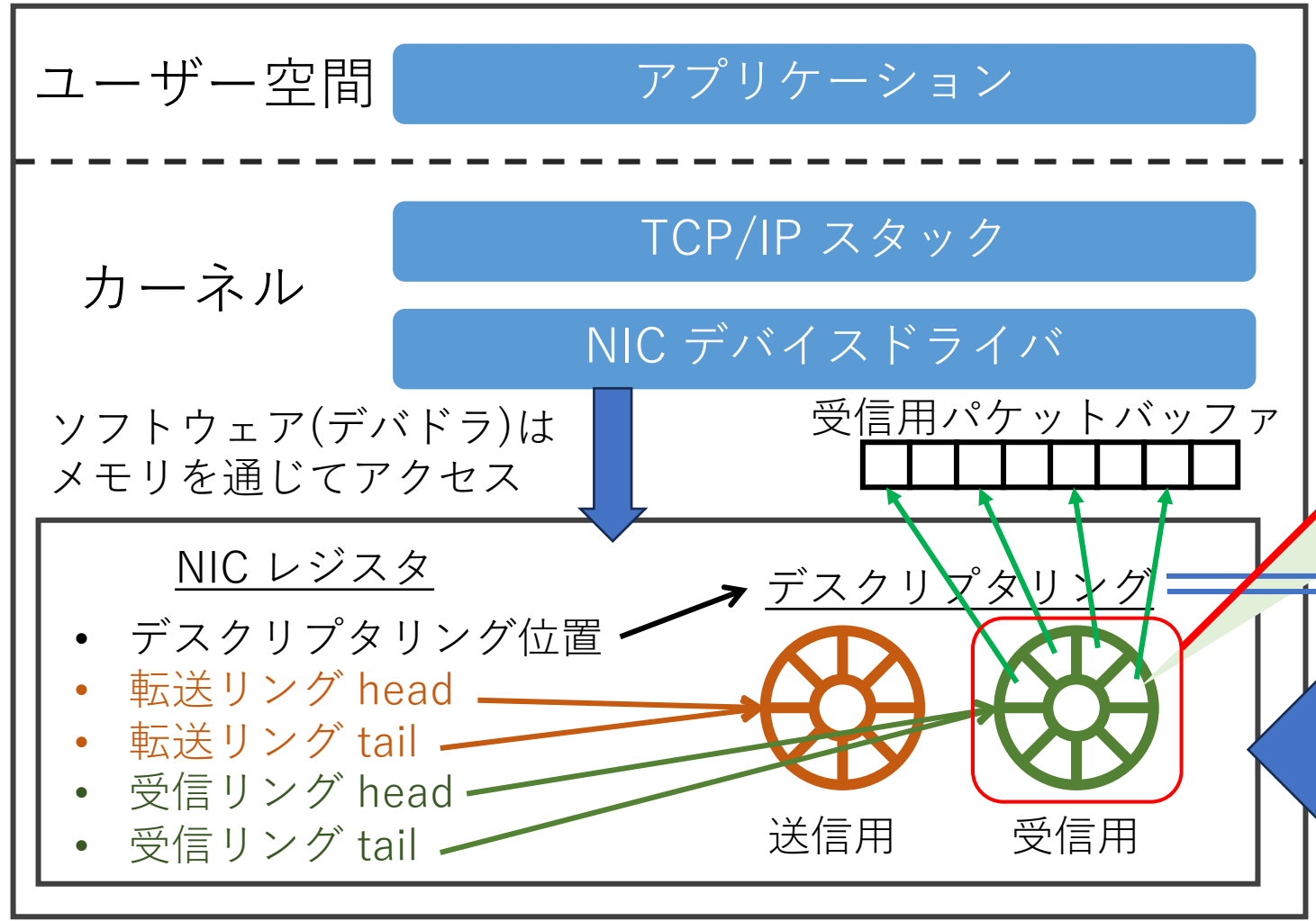
NIC と通信関連プログラム



これで、確保した**連続的なメモリ領域**がデスクリプタの配列としてNIC から認識される

スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

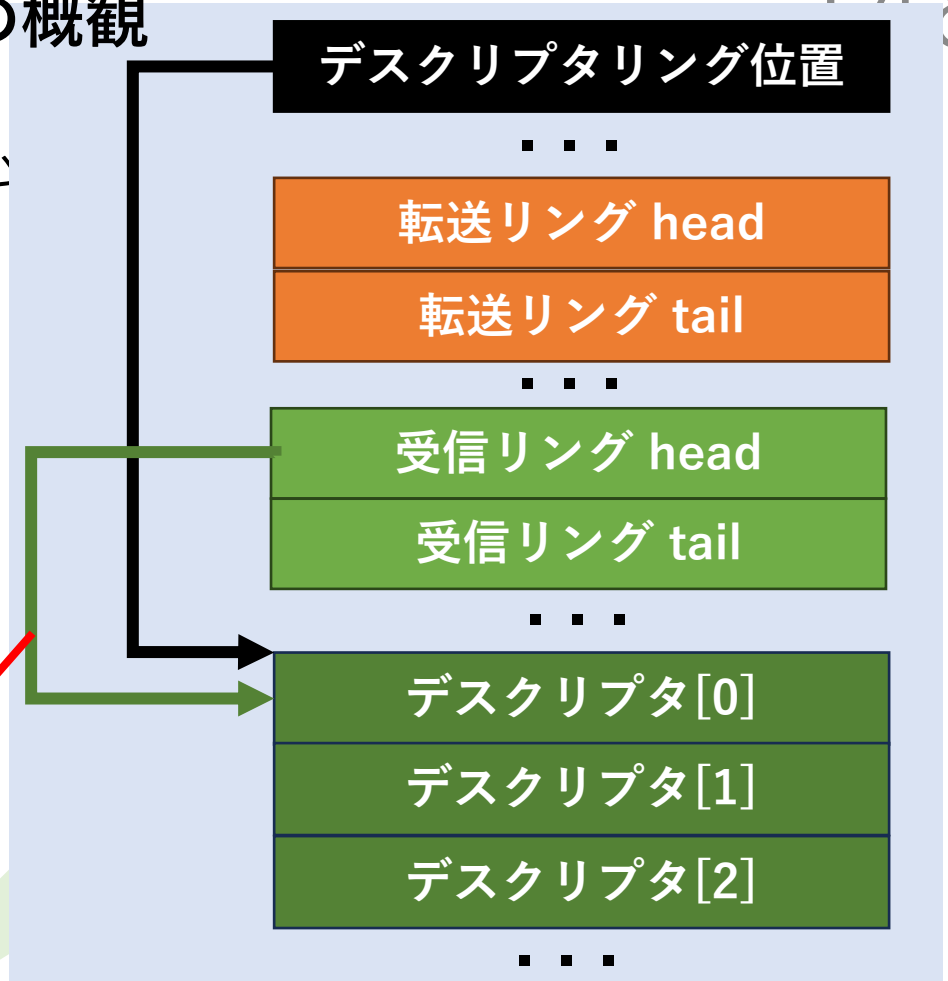
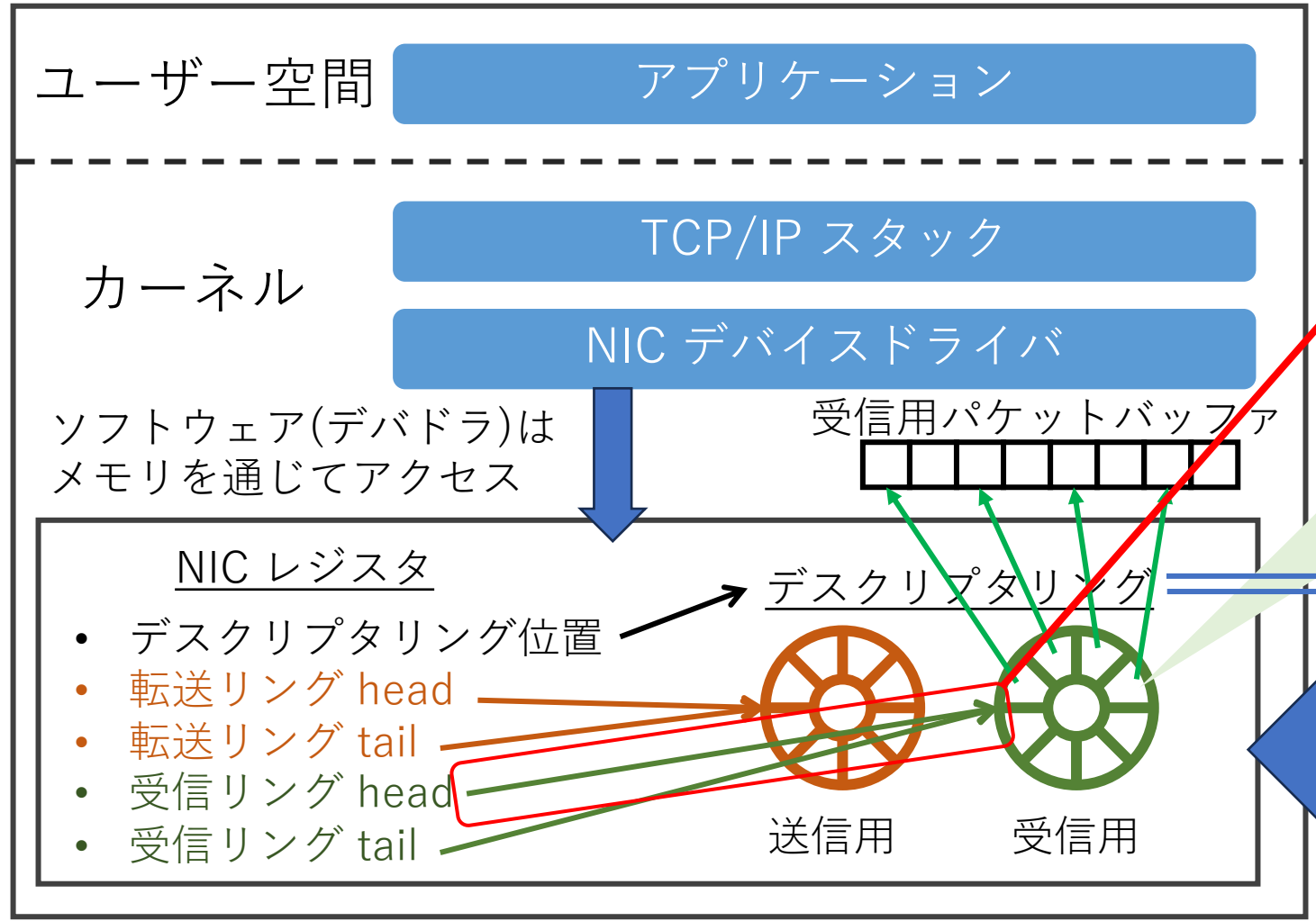
NIC と通信関連プログラム



これで、確保した**連続的なメモリ領域**がデスクリプタの配列として**NIC から認識される**

スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

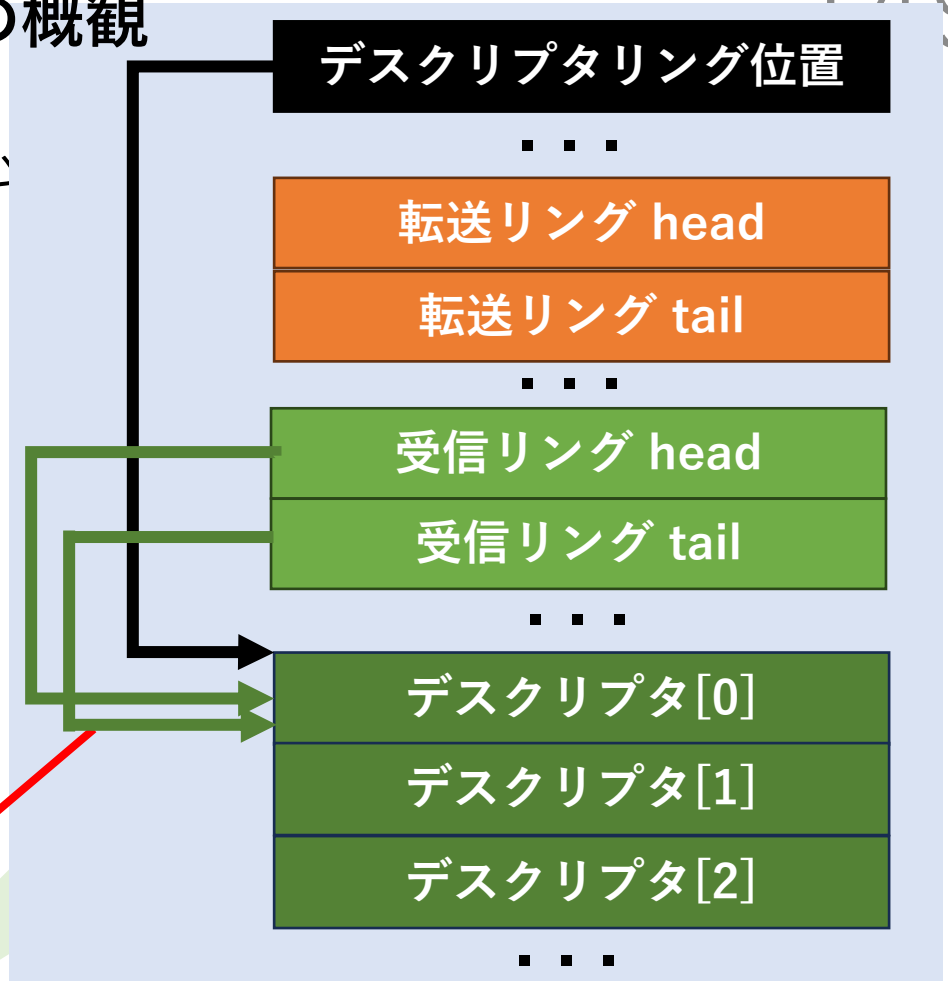
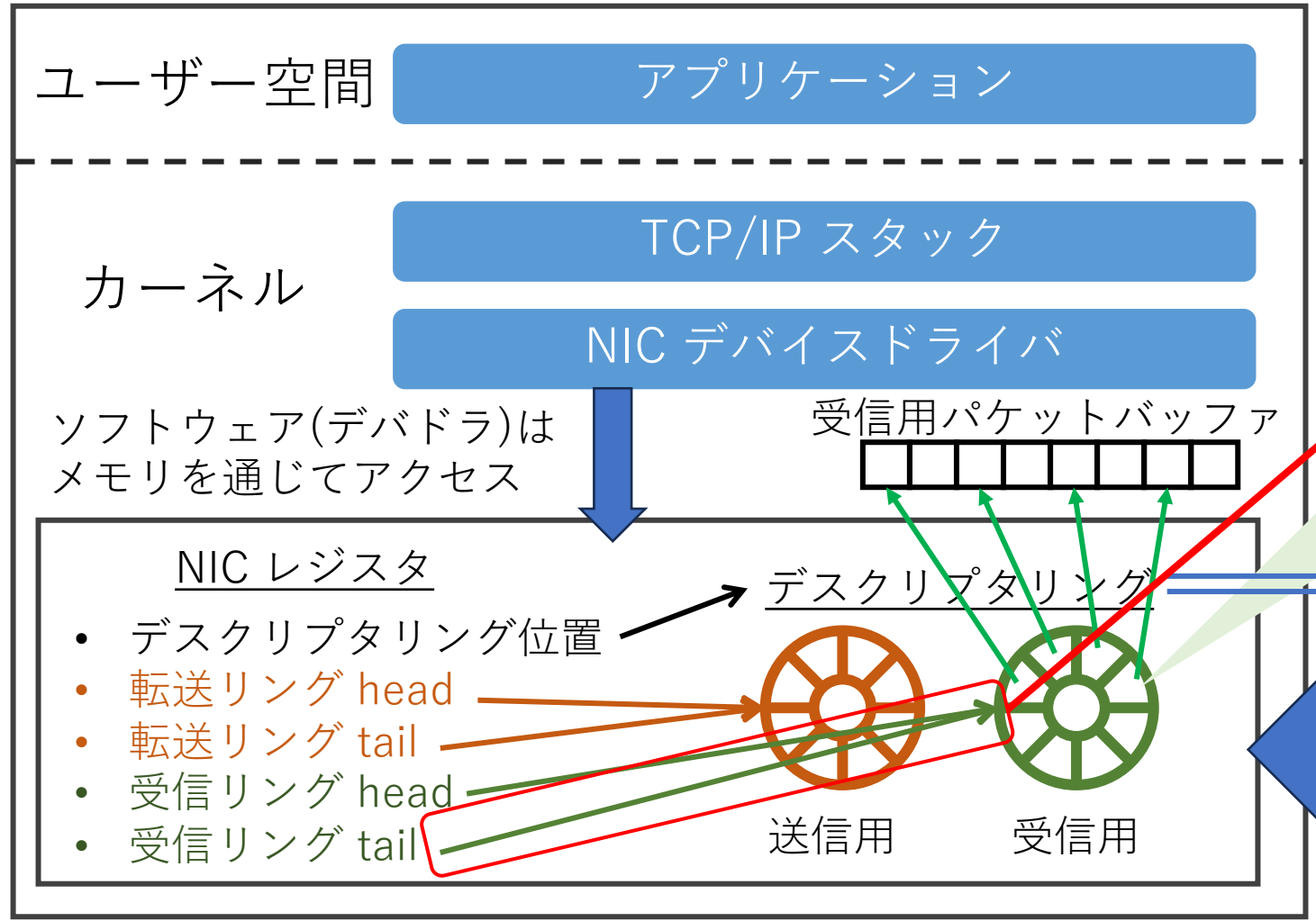
NIC と通信関連プログラム



NIC レジスタのうち、リングの head, tail を保持するレジスタは、デスクリプタ配列のインデックスを保持する

スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

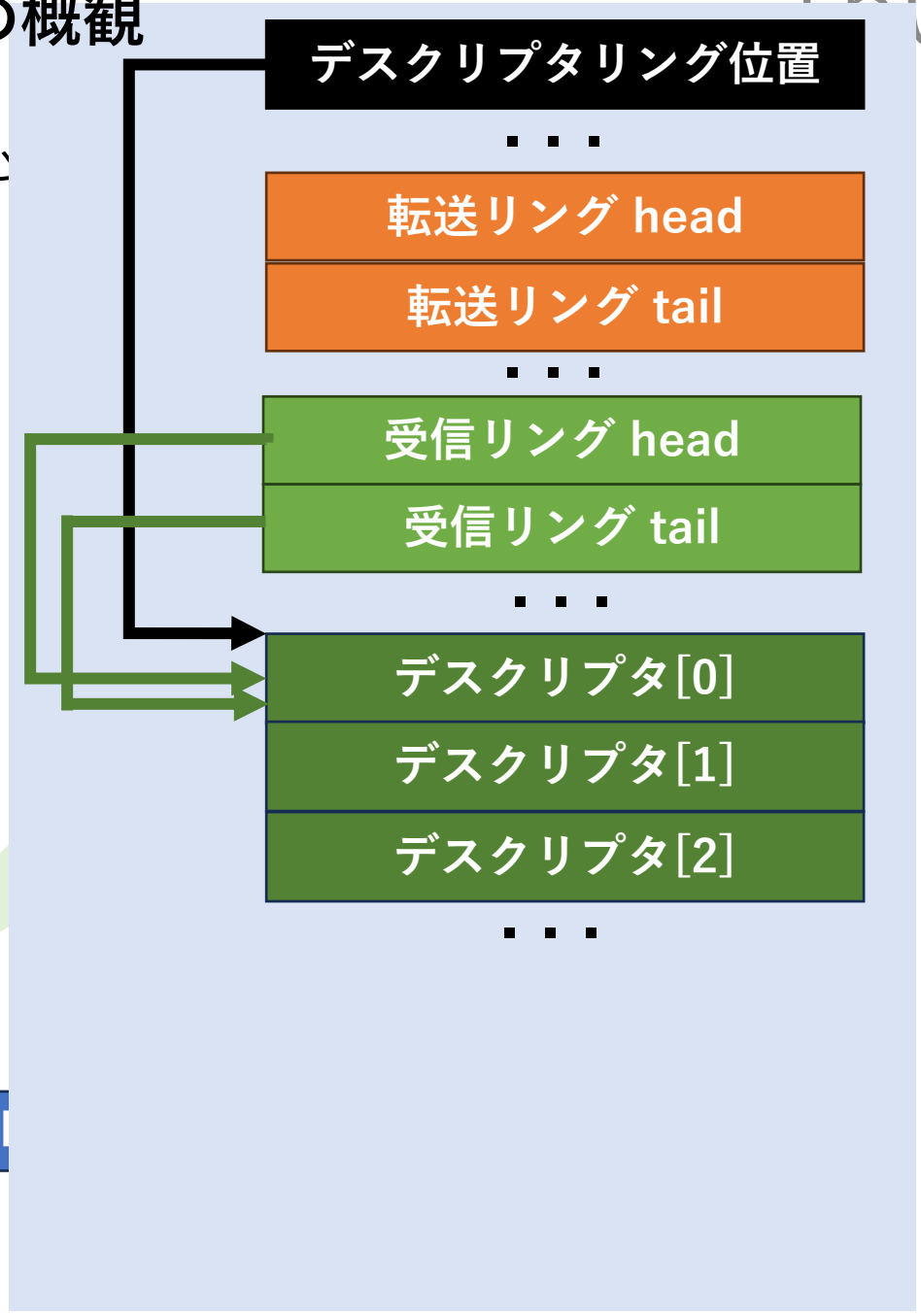
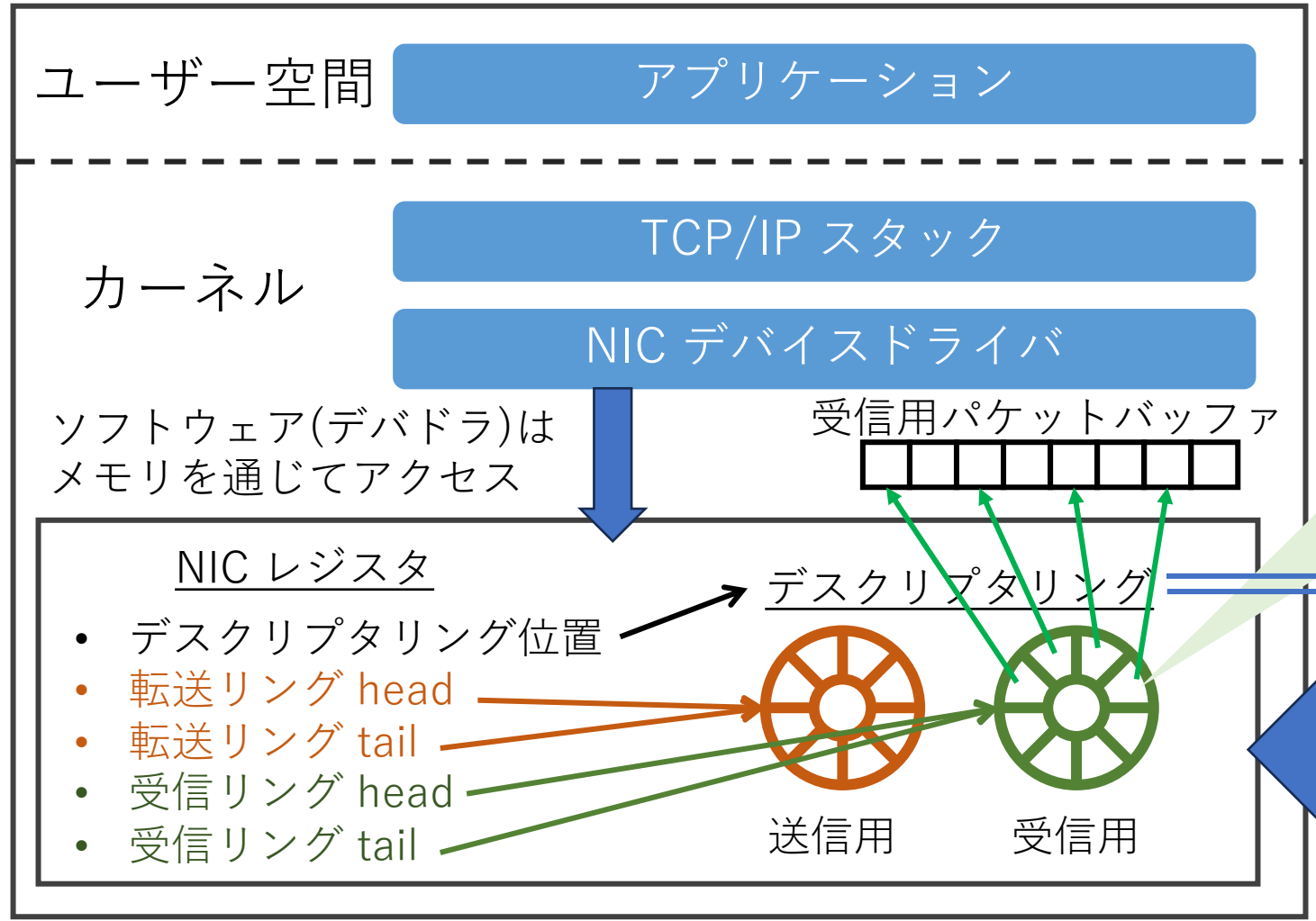
NIC と通信関連プログラム



NIC レジスタのうち、リングの head, tail を保持するレジスタは、デスクリプタ配列のインデックスを保持する

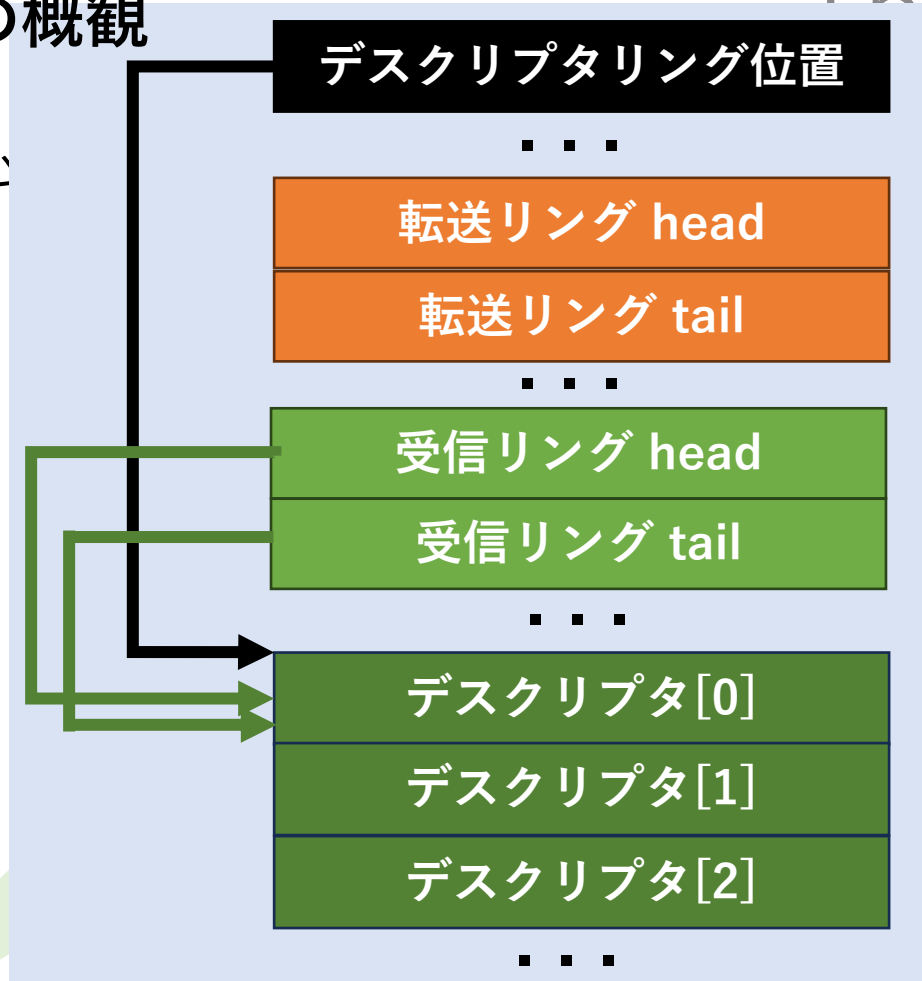
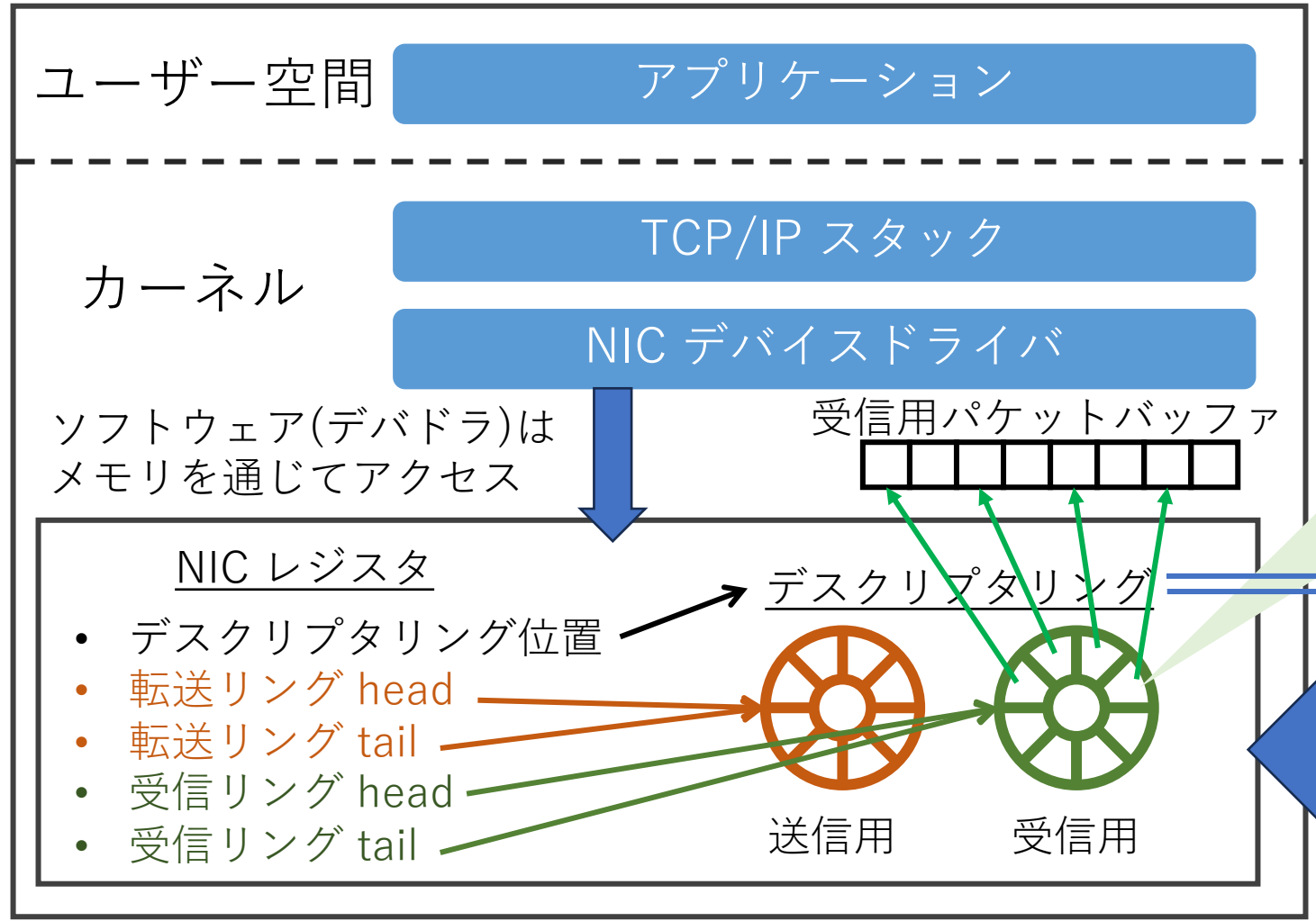
スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

NIC と通信関連プログラム



スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

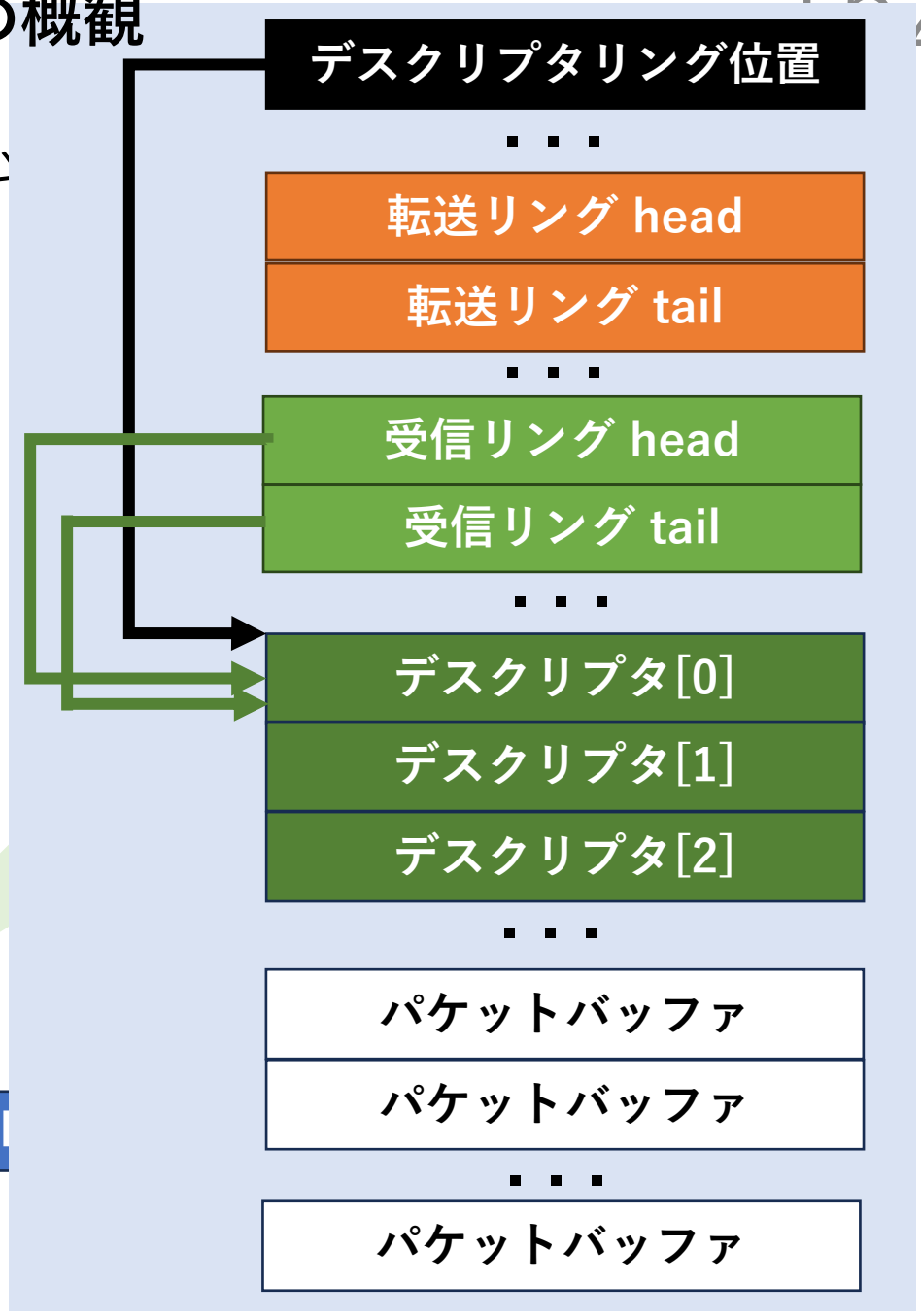
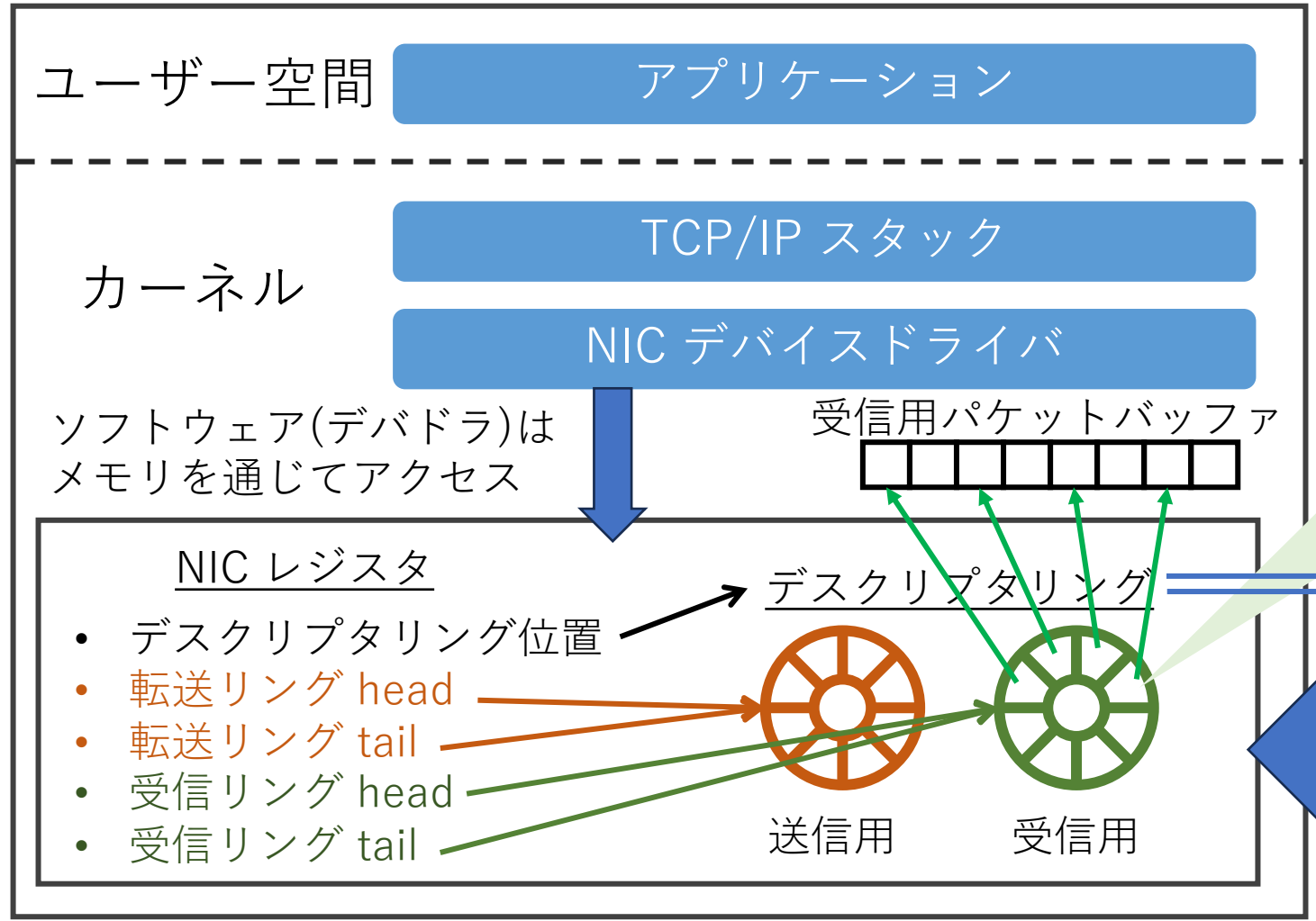
NIC と通信関連プログラム



次に、ソフトウェア (デバドラ) はパケットバッファ用メモリを確保する

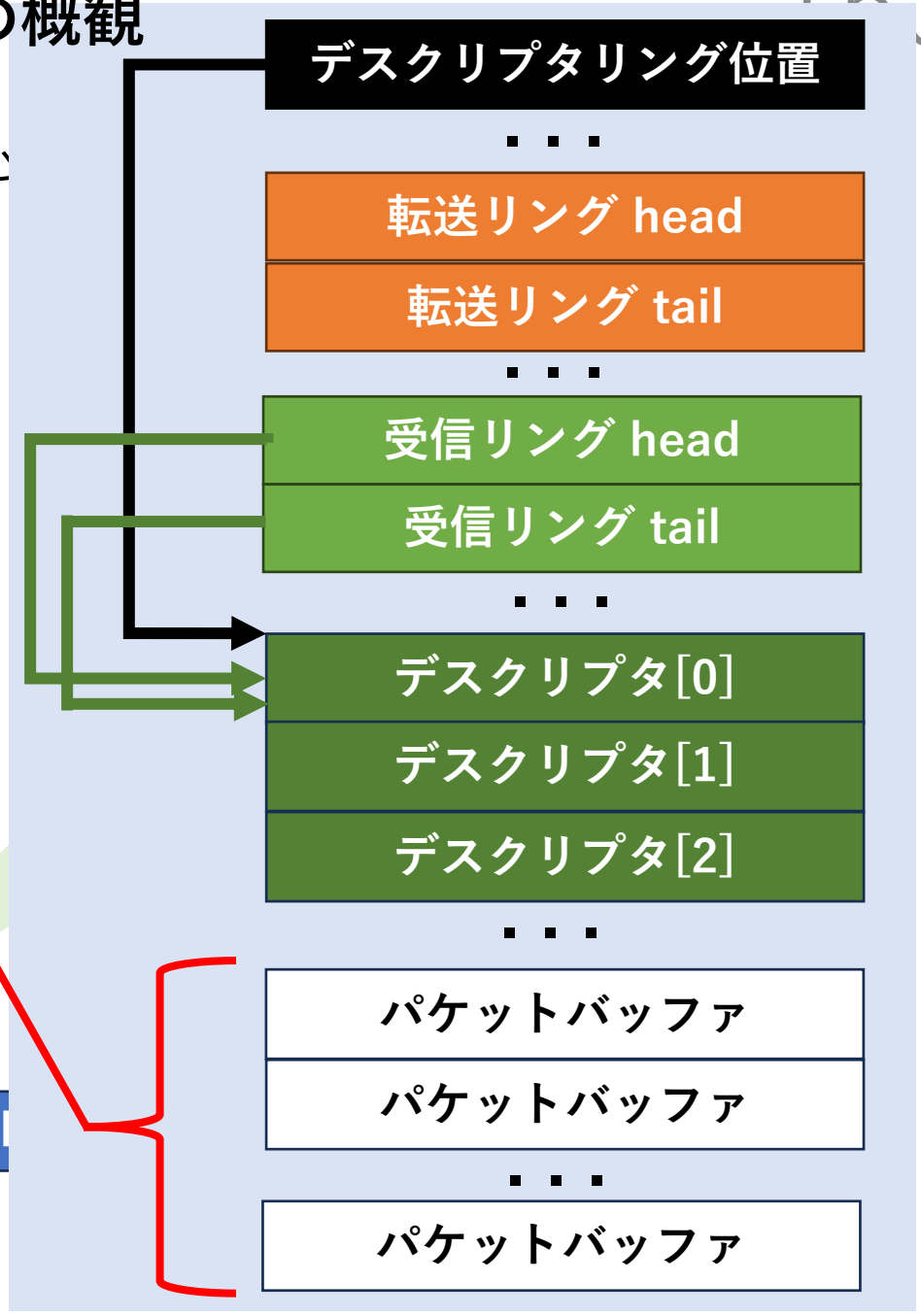
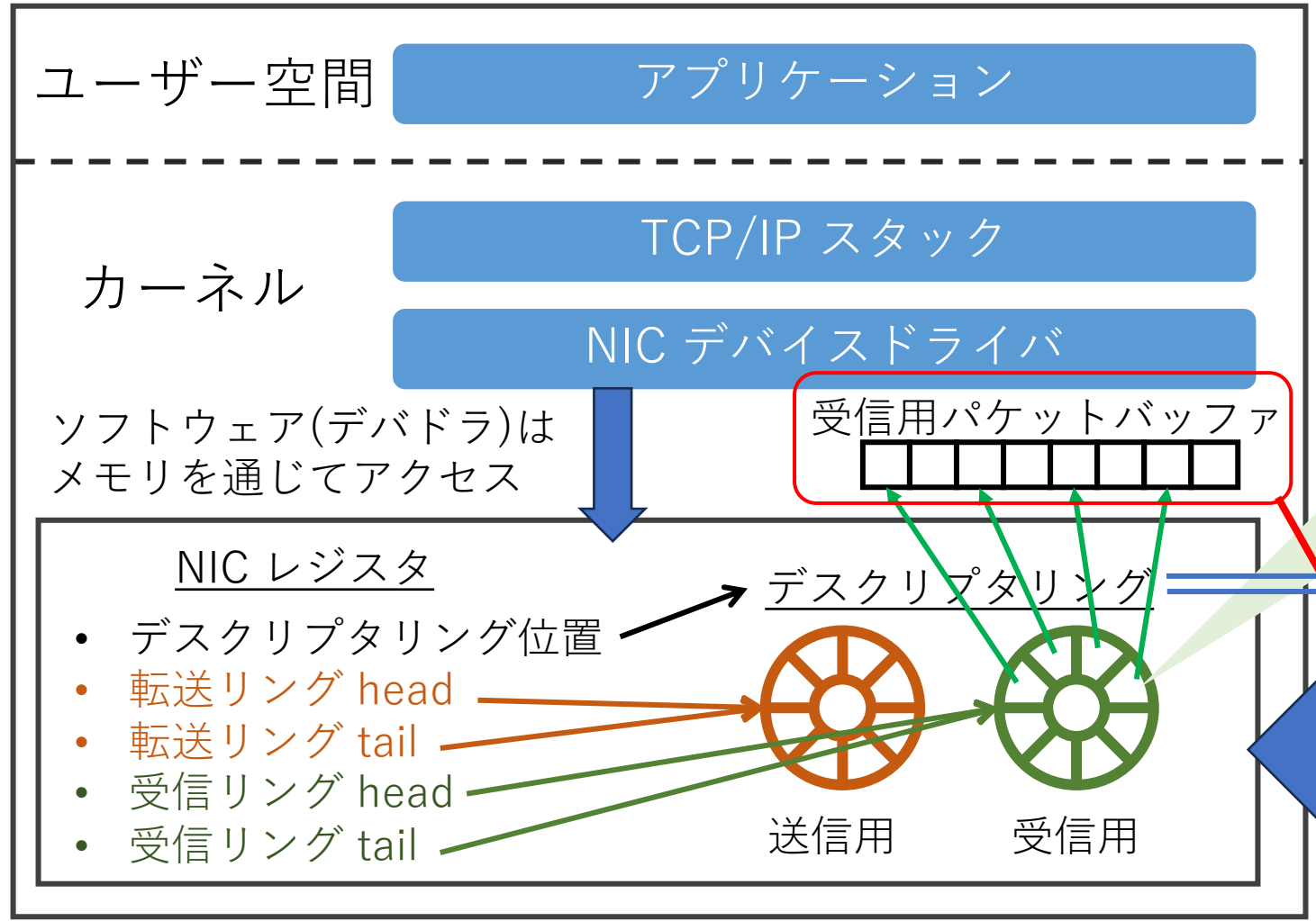
スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

NIC と通信関連プログラム



スペースの都合で転送用デスクリプタは省略しています **メモリ上の概観**

NIC と通信関連プログラム



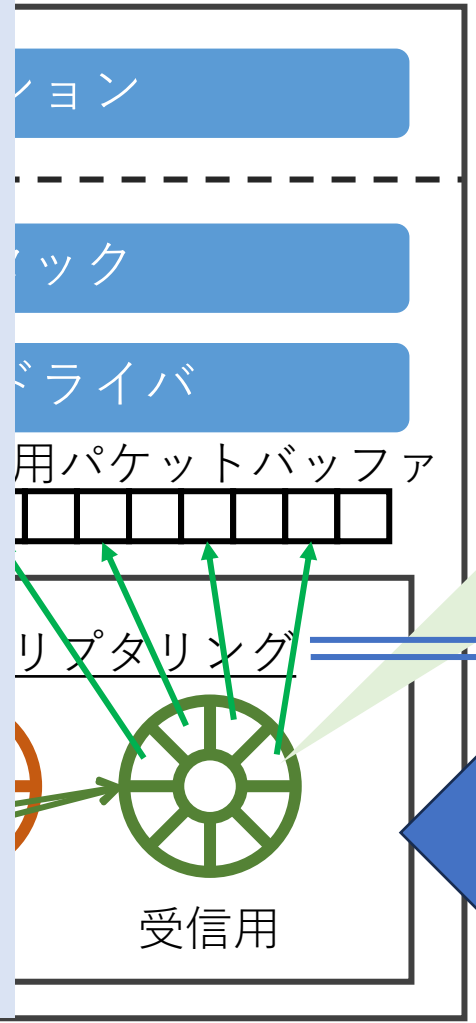
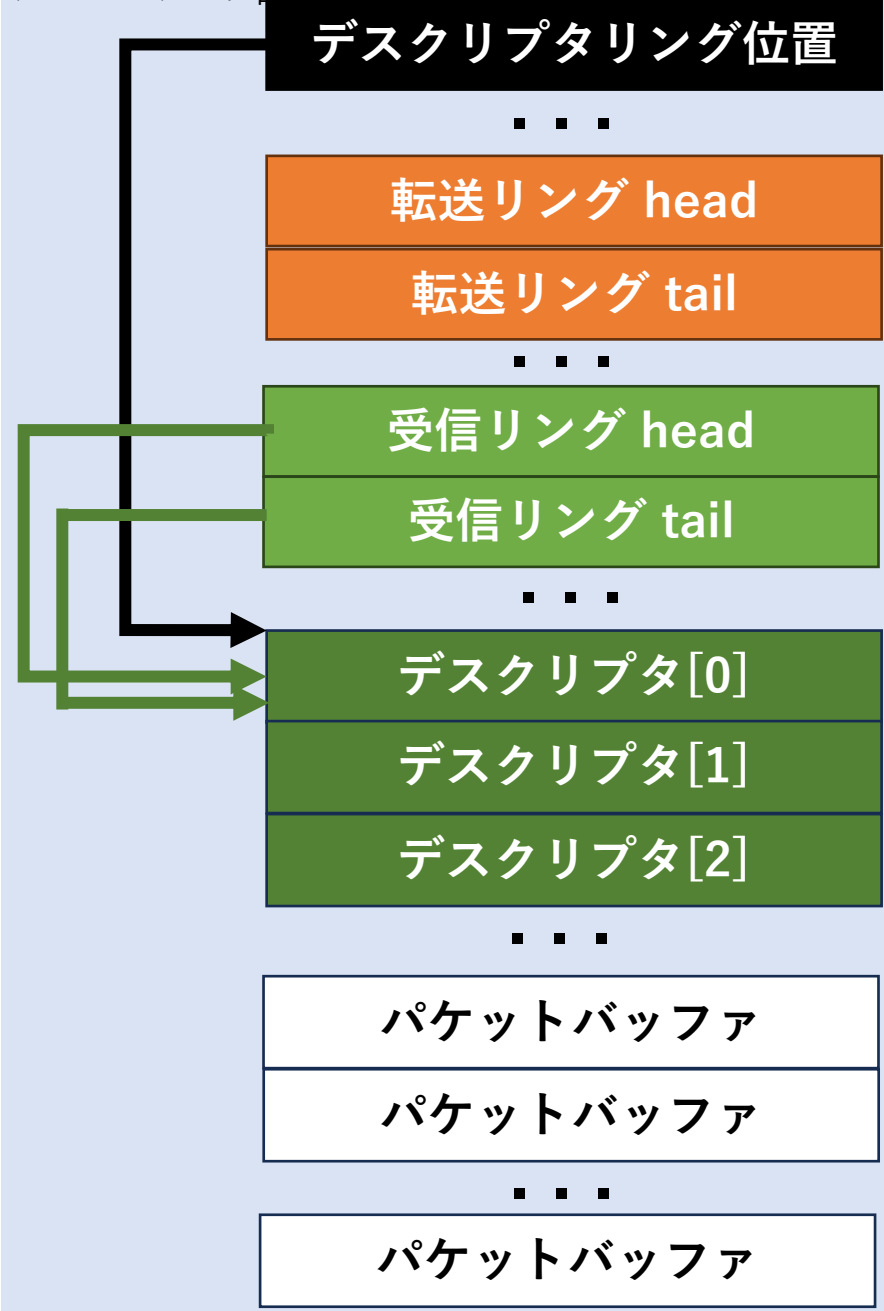
スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

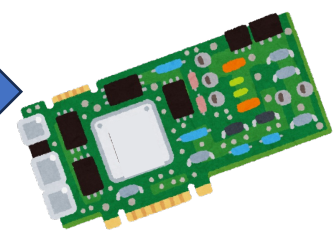
プログラムの構成

ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - パケットのサイズ
 - その他：状態保持用フラグ



NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)

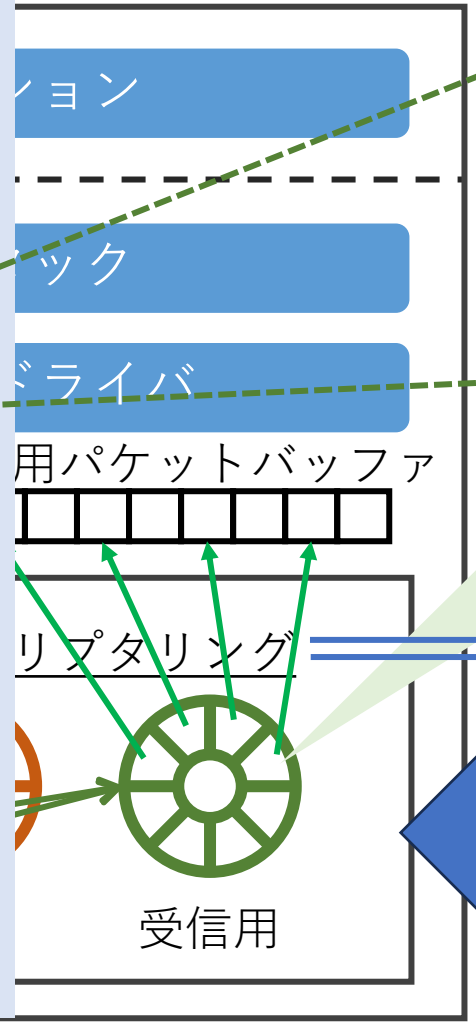
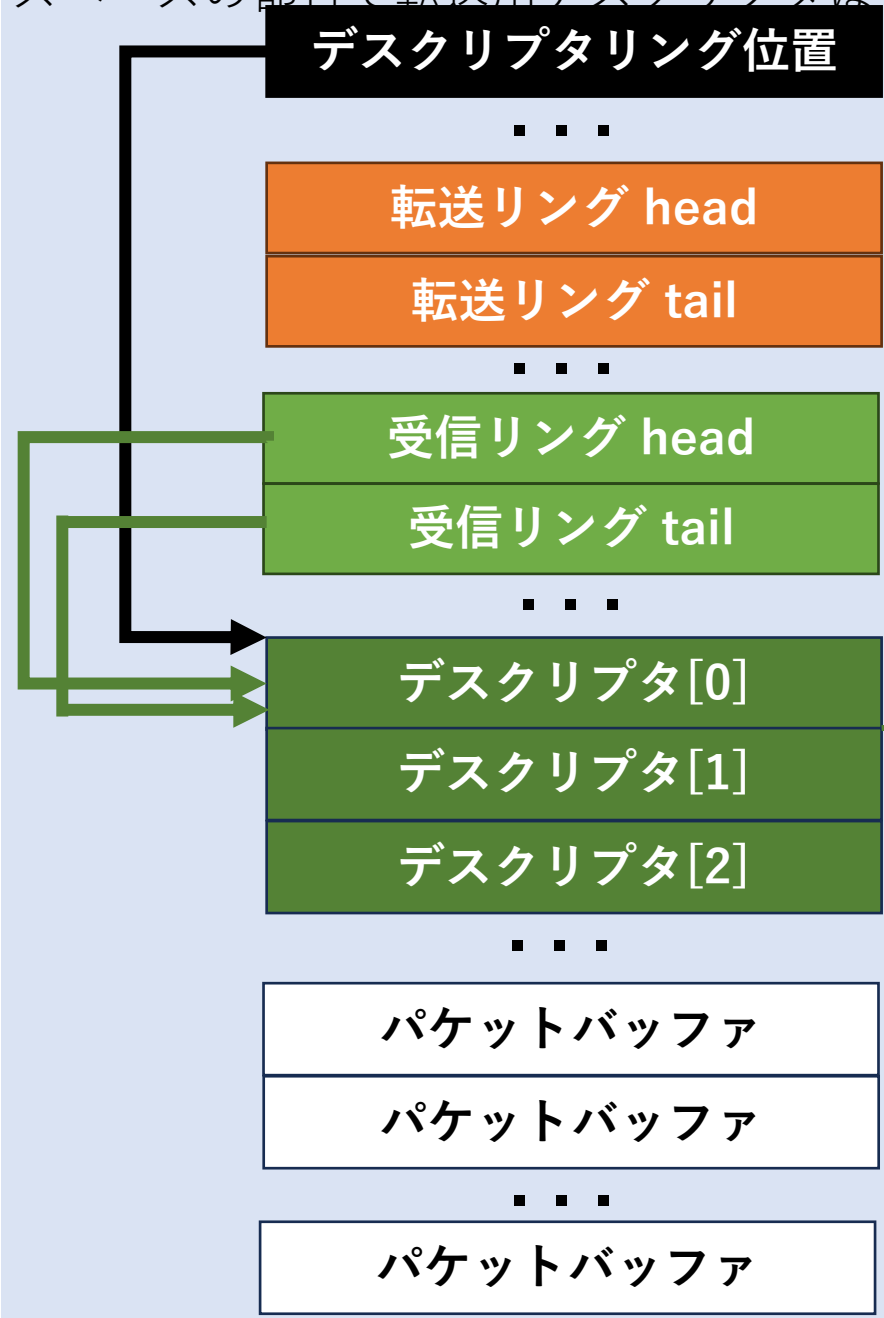


スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

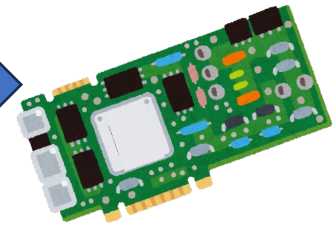
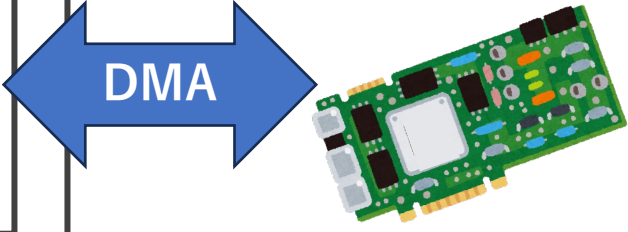
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う



デスクリプタが保持する内容

- バッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)

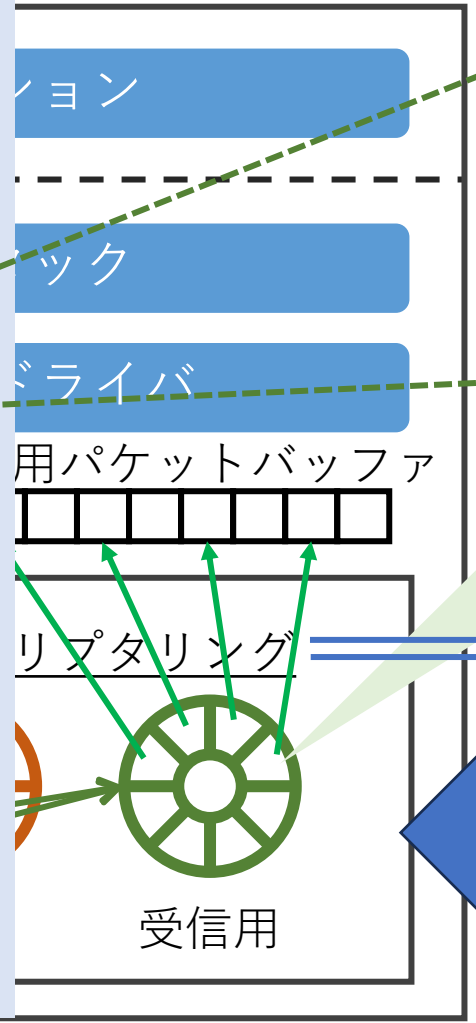
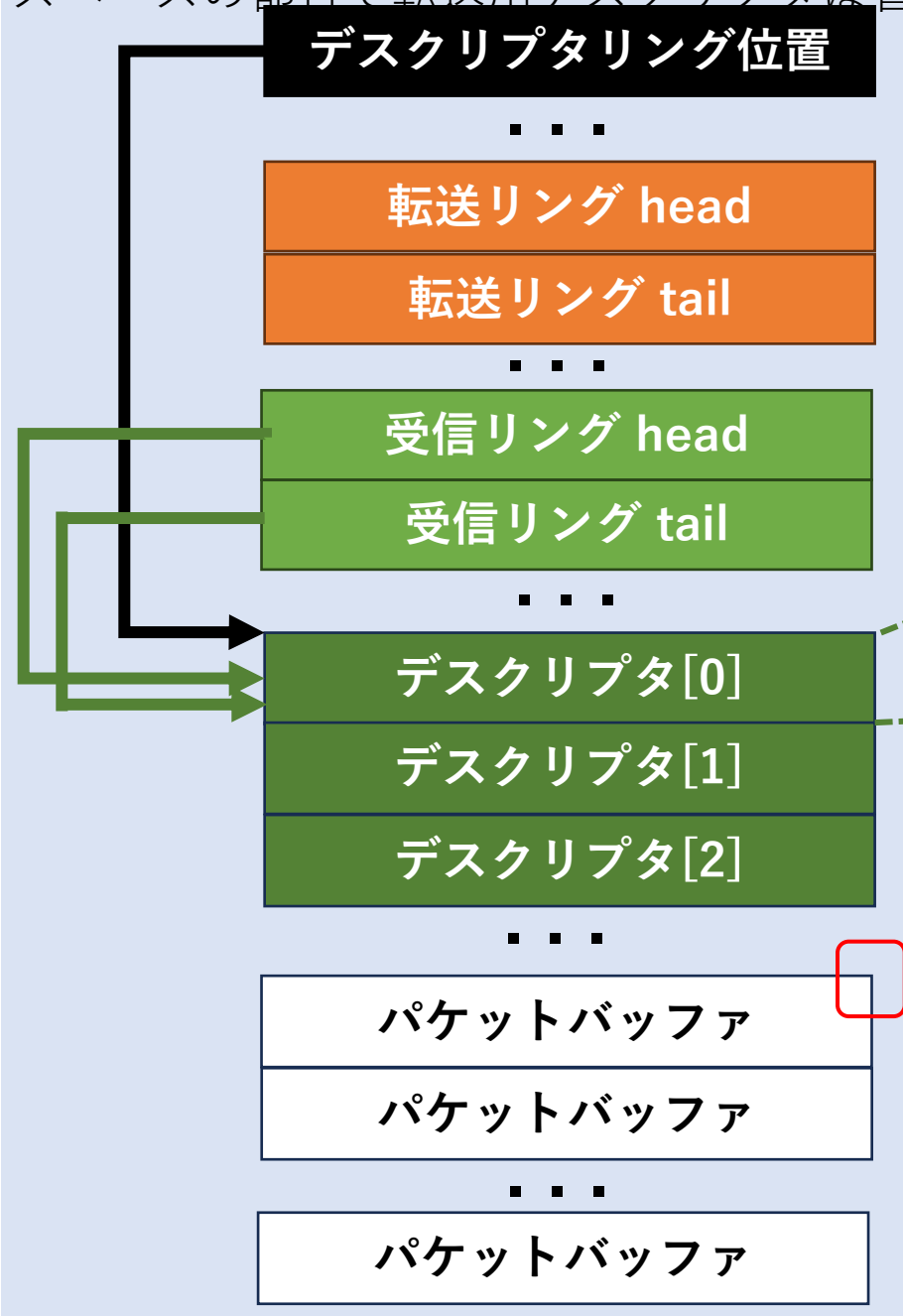


スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

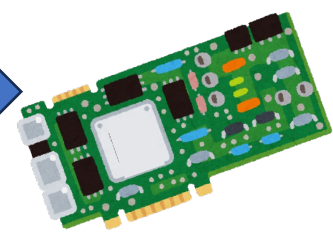
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う



デスクリプタが保持する内容

- バッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

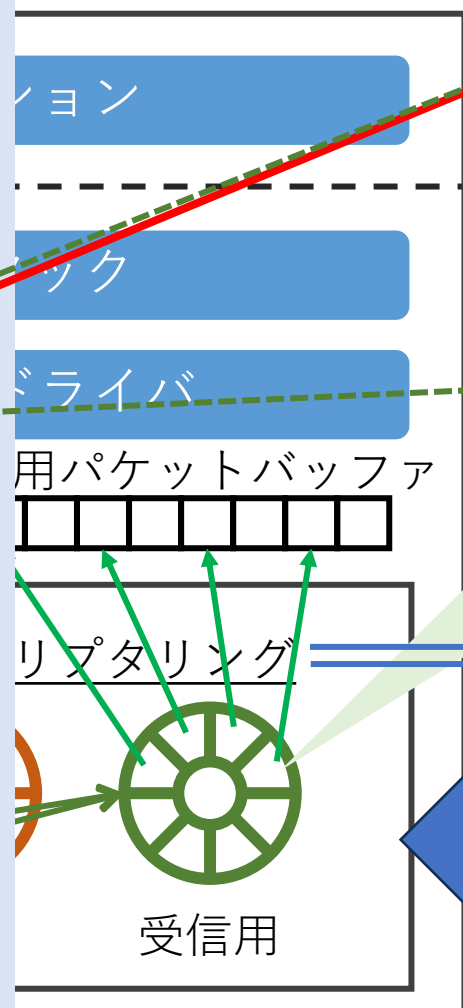
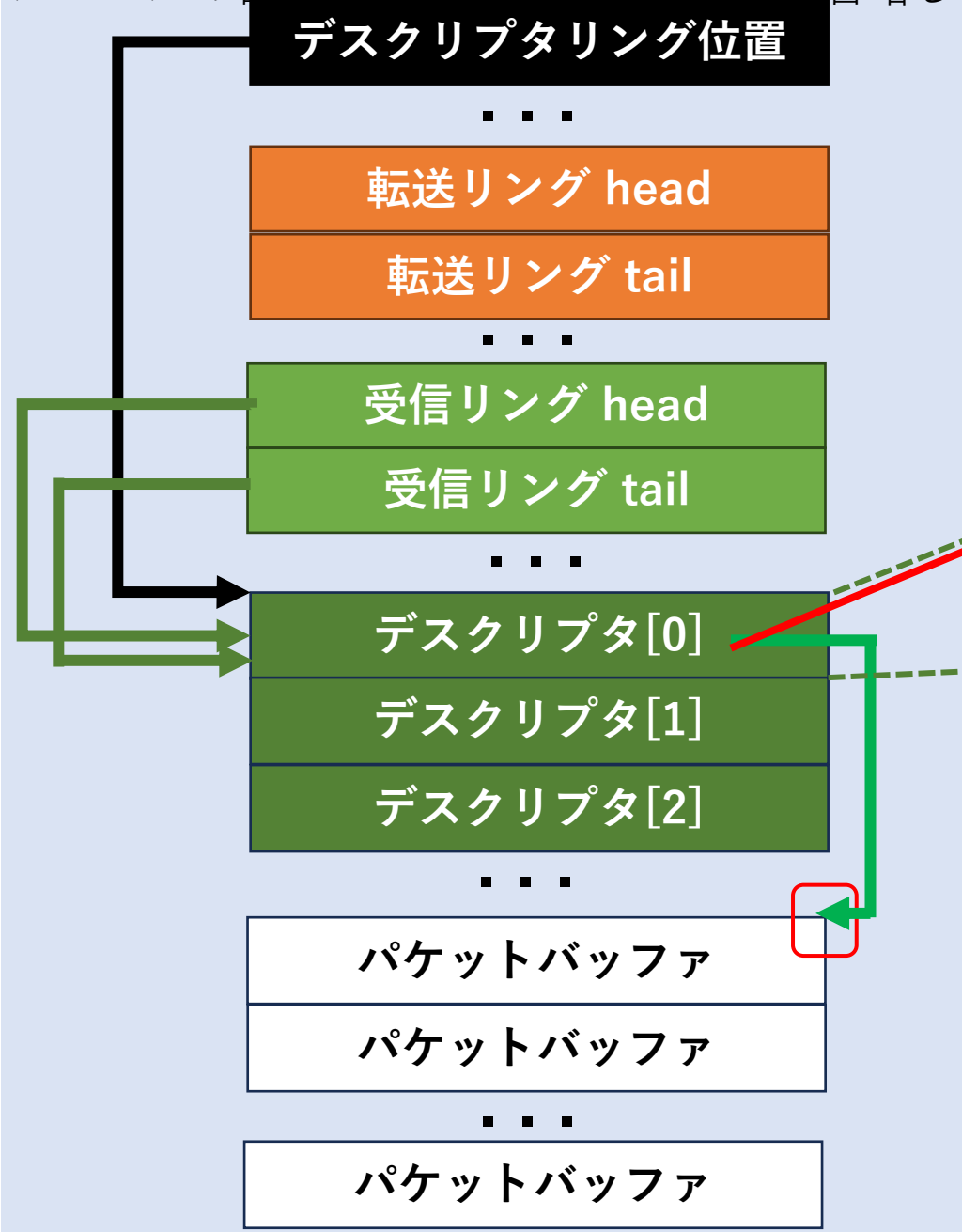
メモリ上の概観

プログラムの構成

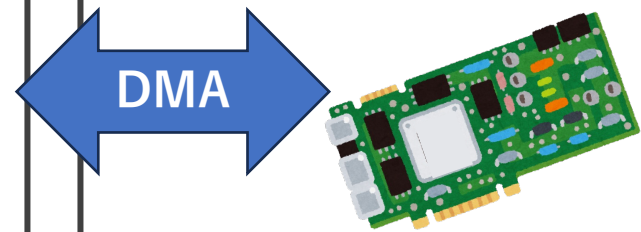
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ



NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

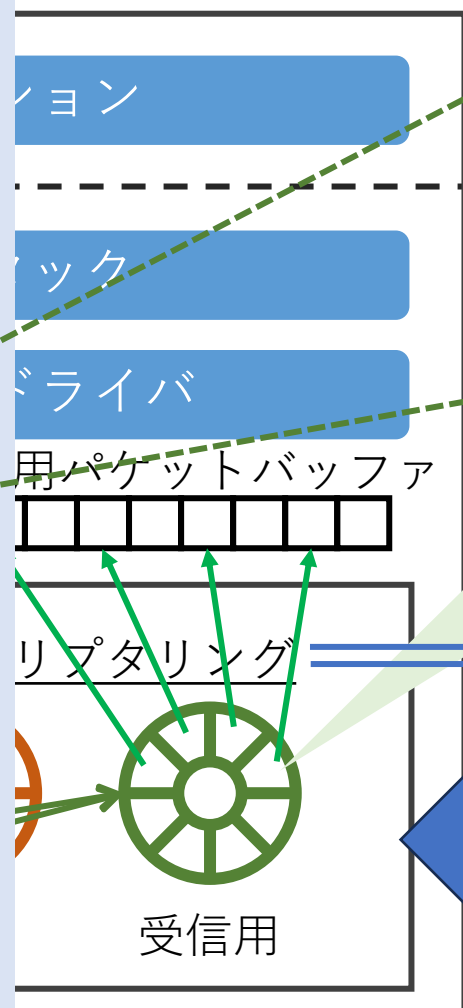
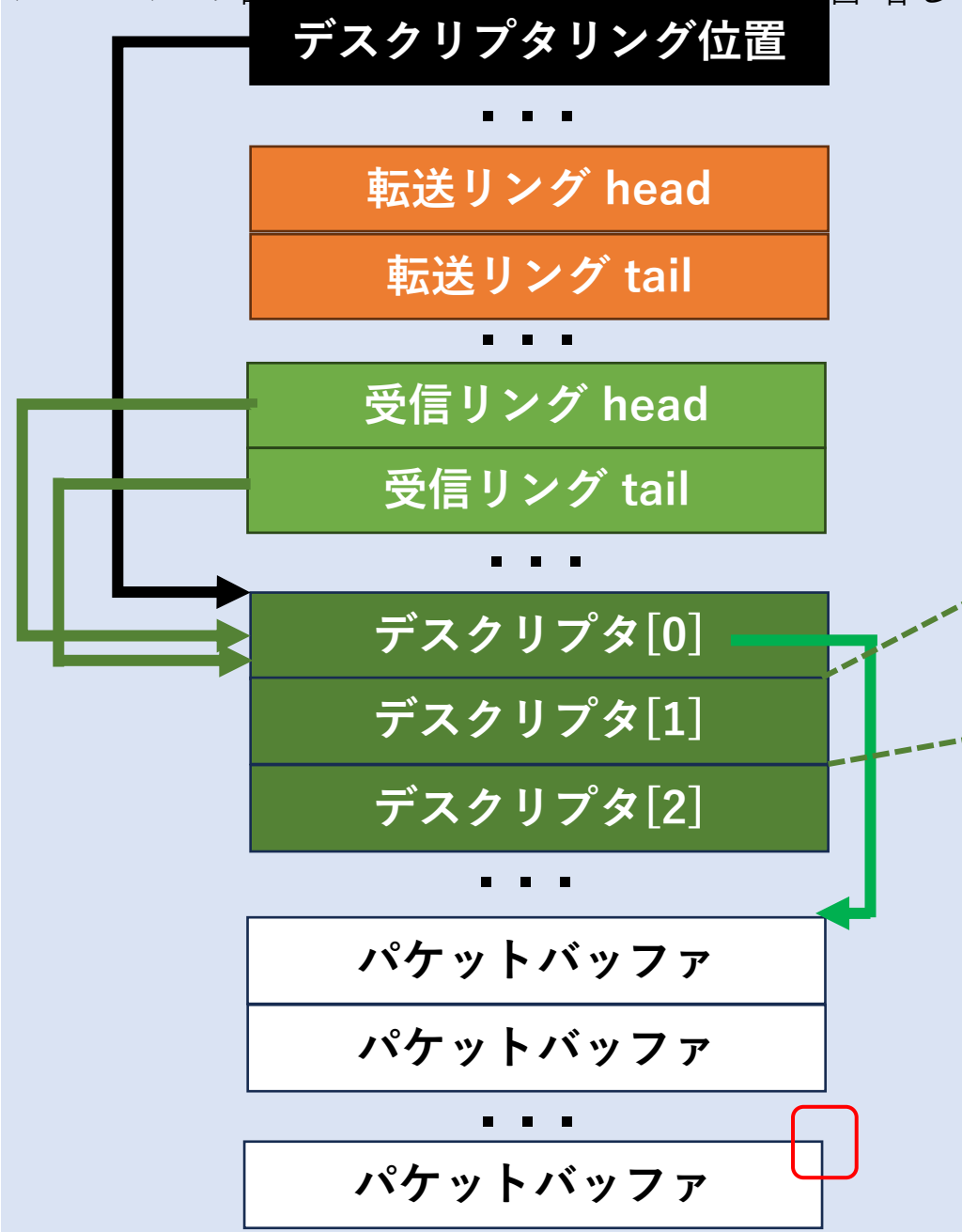
メモリ上の概観

プログラムの構成

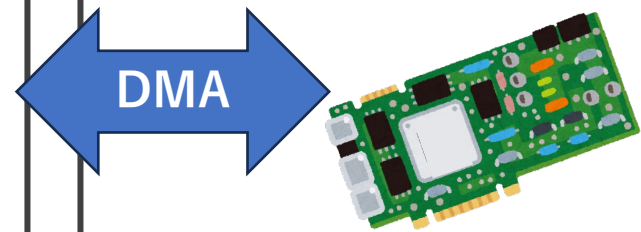
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ



NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

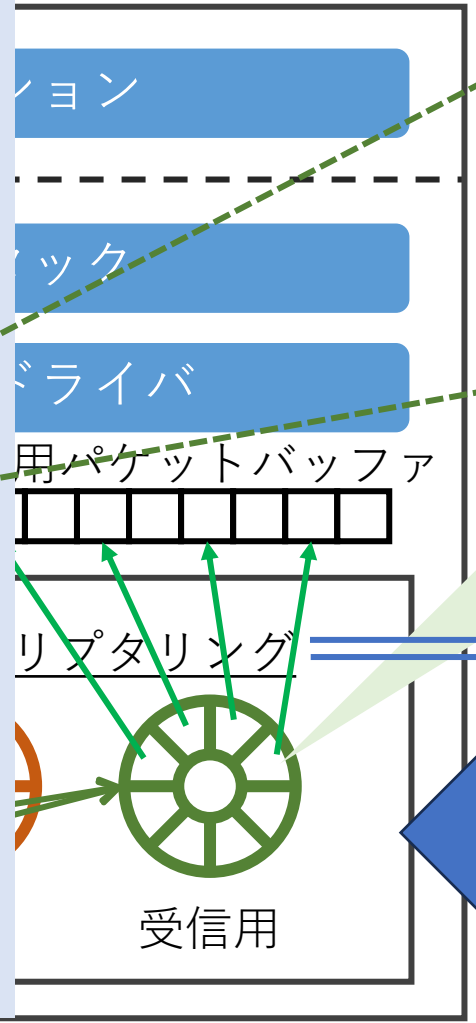
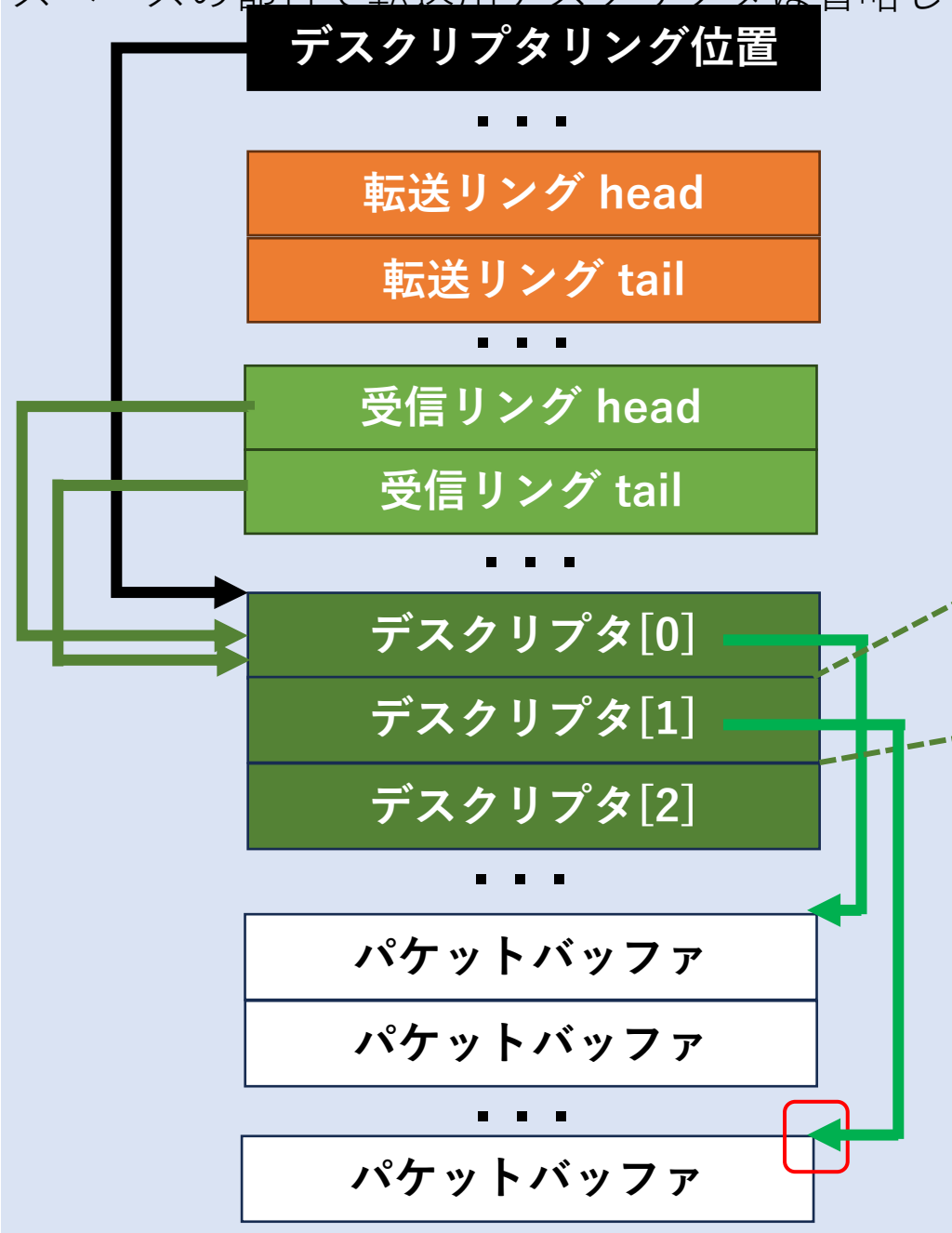
メモリ上の概観

プログラムの構成

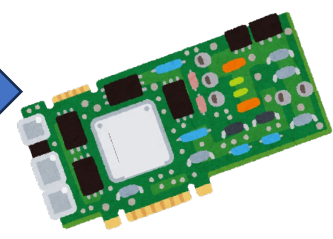
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ



NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

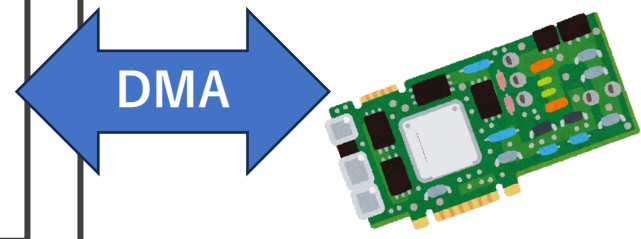
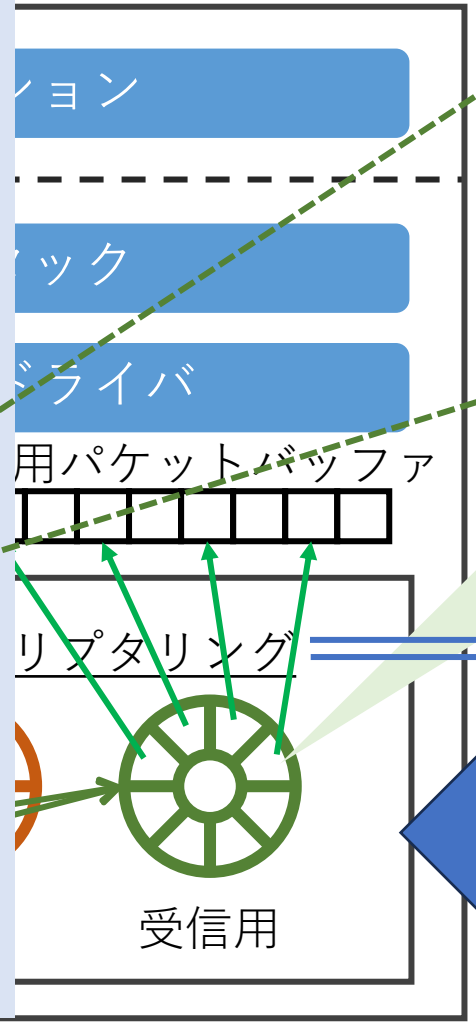
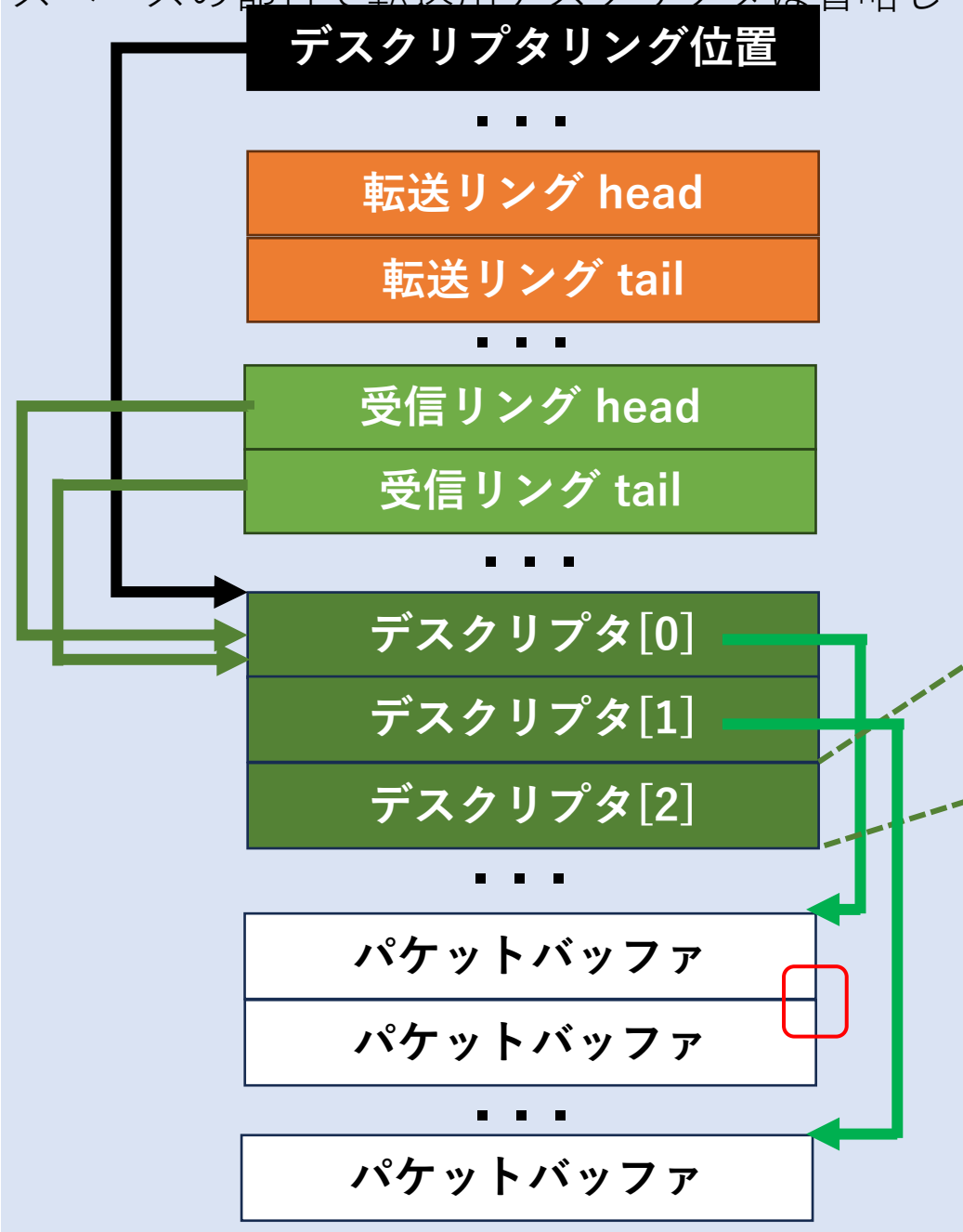
プログラムの構成

ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

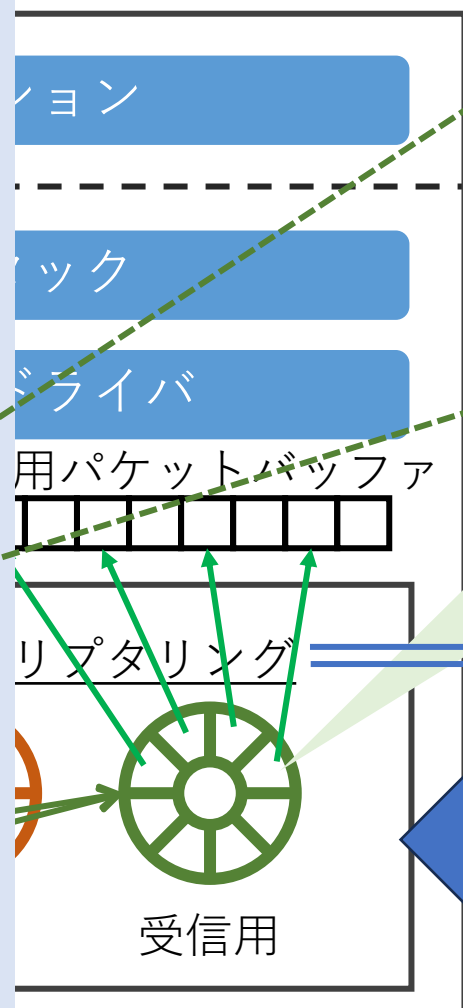
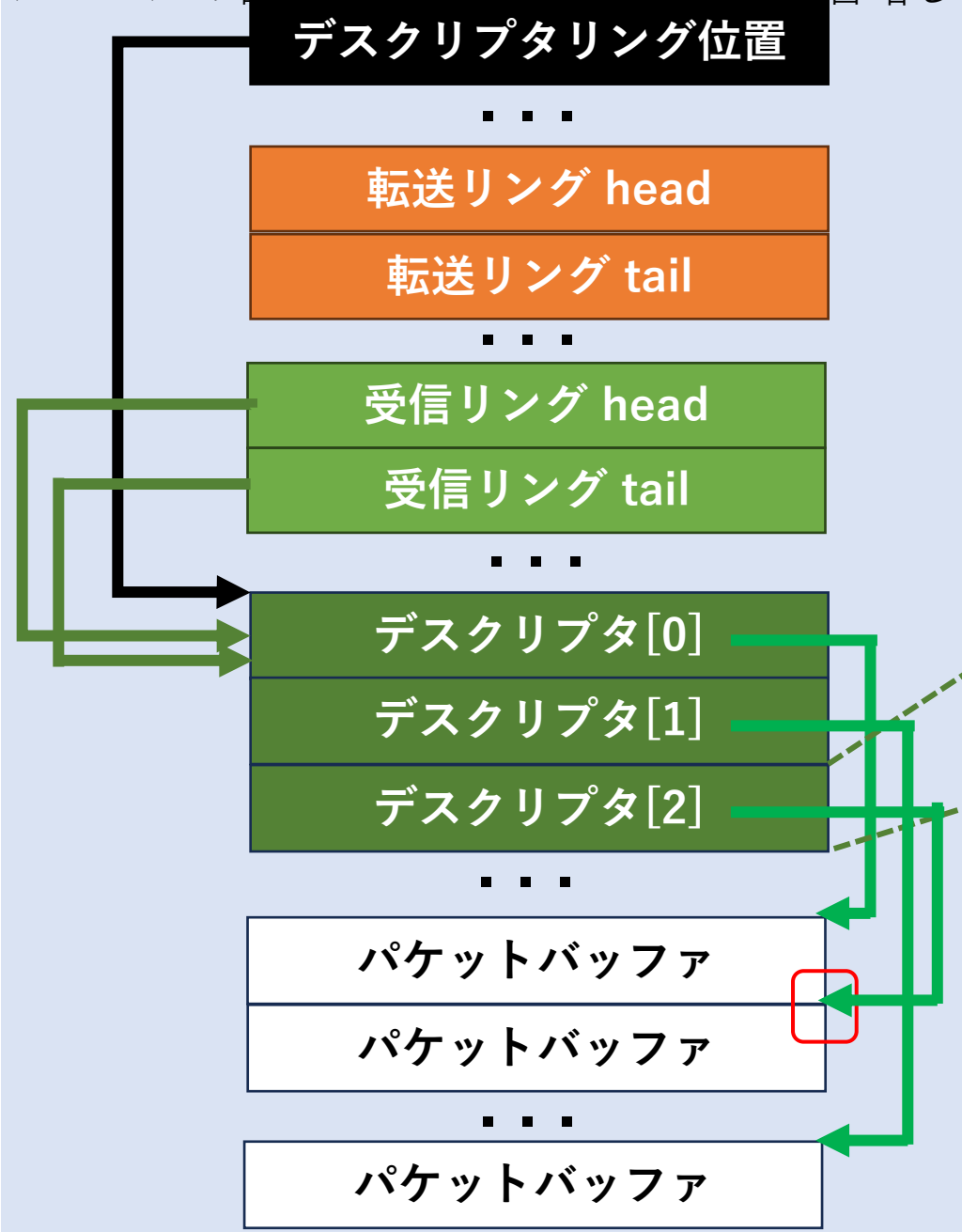
メモリ上の概観

プログラムの構成

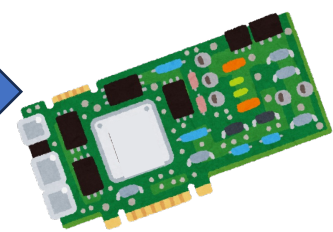
ソフトウェア（デバドラ）はデスクリプタのフィールドにバッファのアドレスを書き込むことでNICへの紐付けを行う

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ



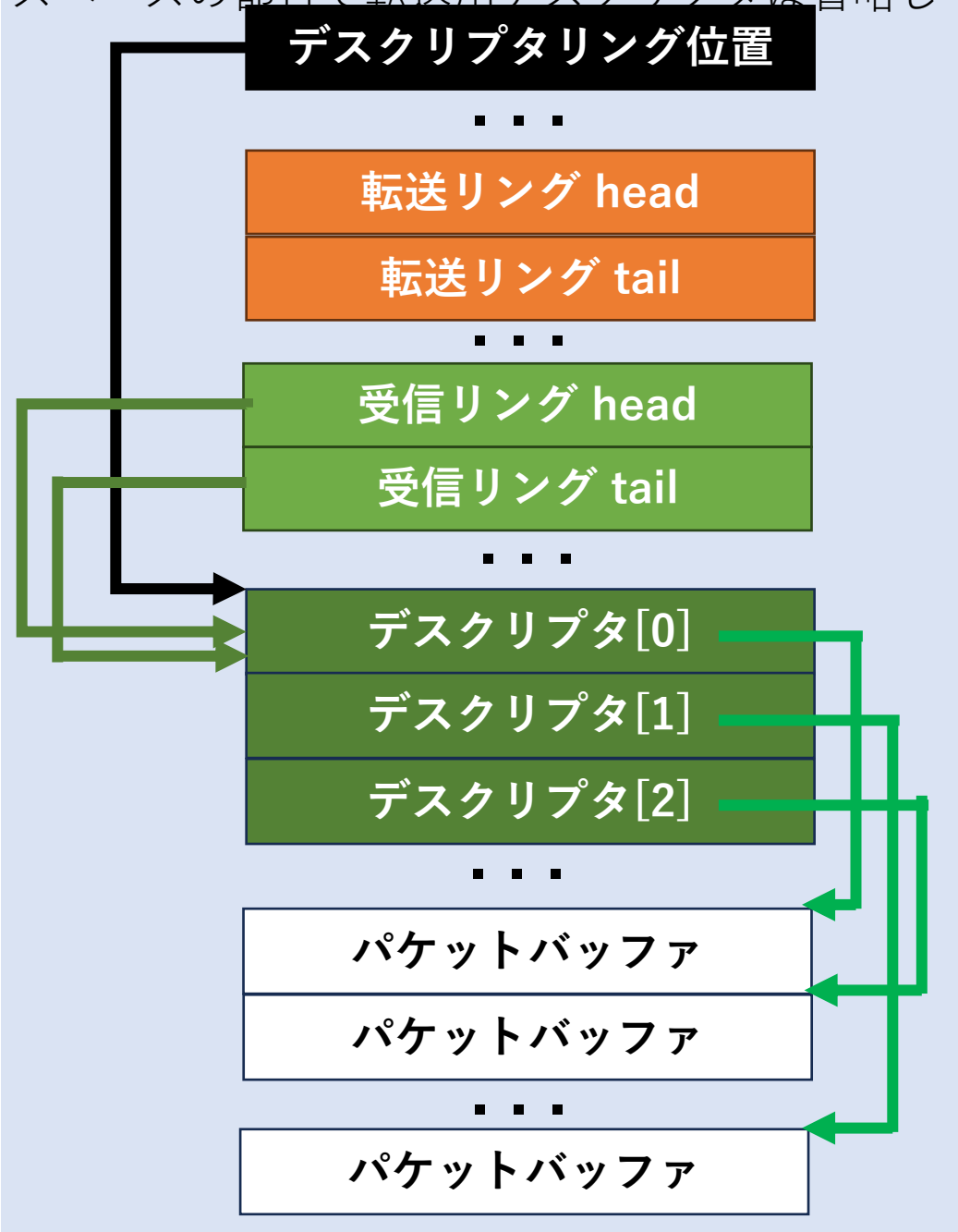
NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

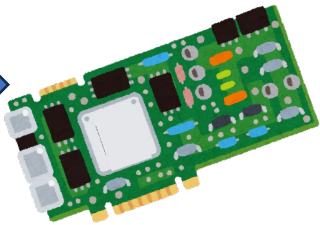
プログラムの構成



デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

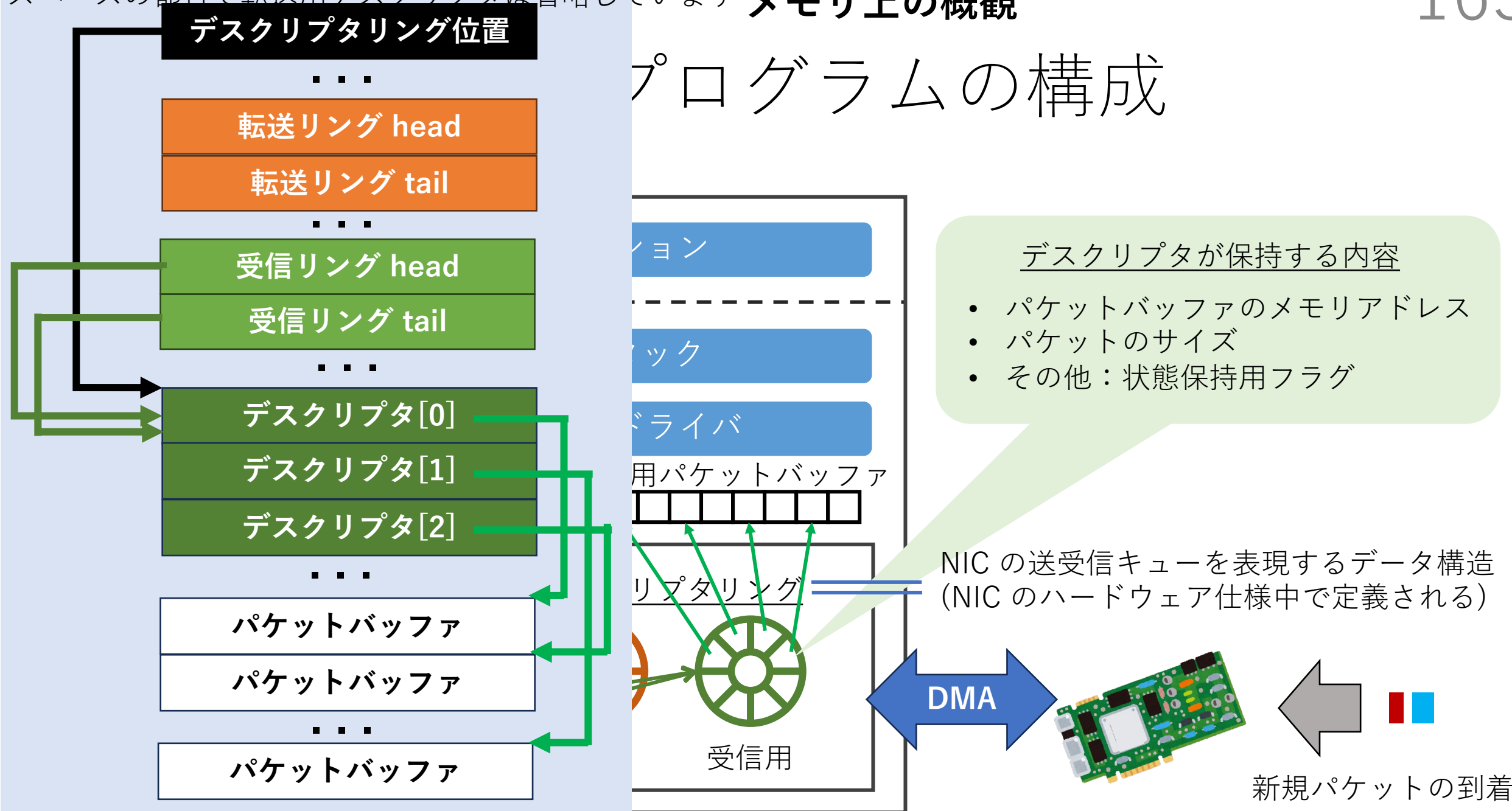
NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)



スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

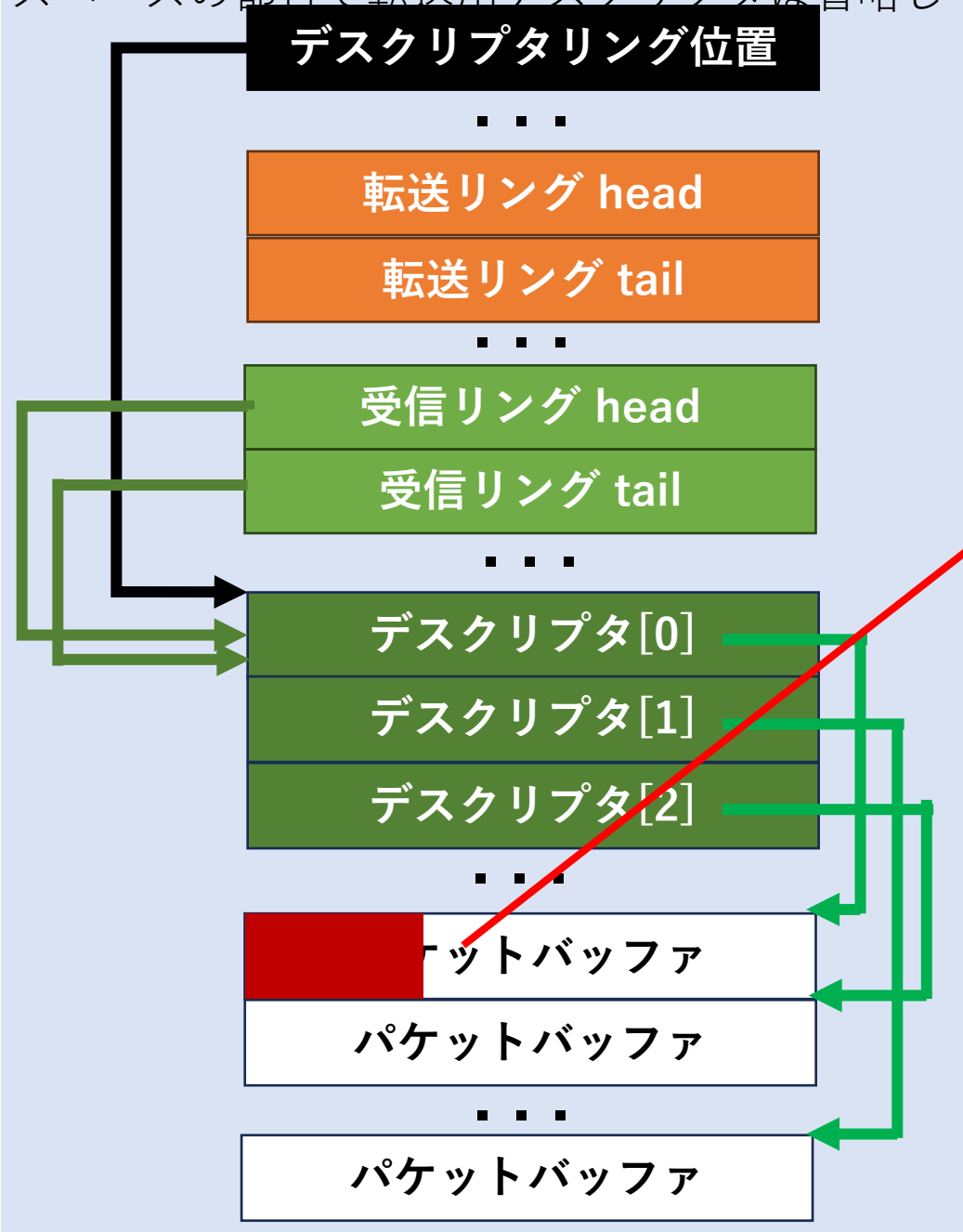
プログラムの構成



スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

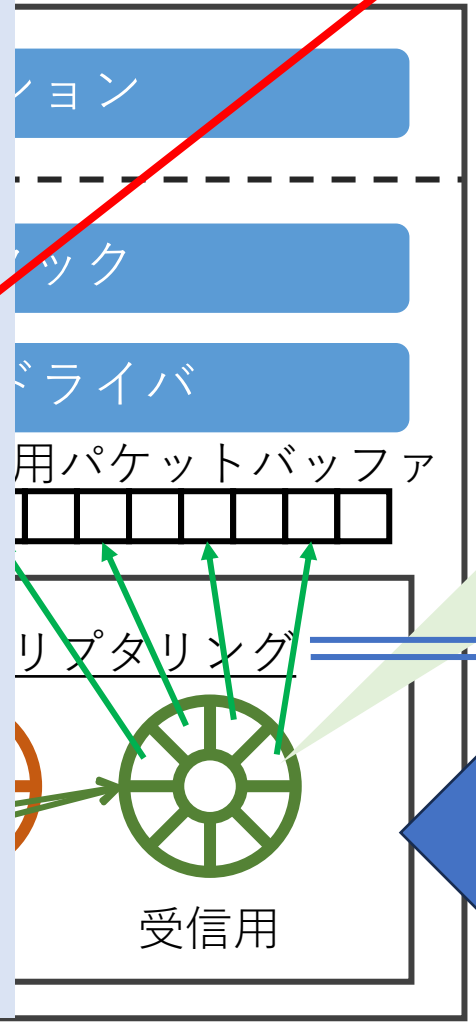
プログラムの構成



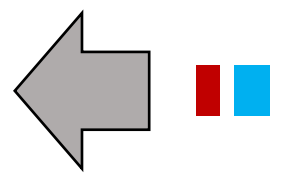
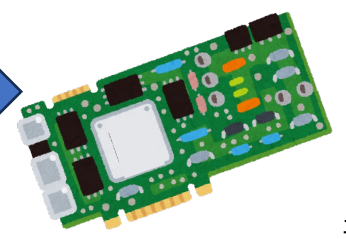
受信したデータはデスクリプタ経由で紐づけられたパケットバッファへ書き込まれる

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ



NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)



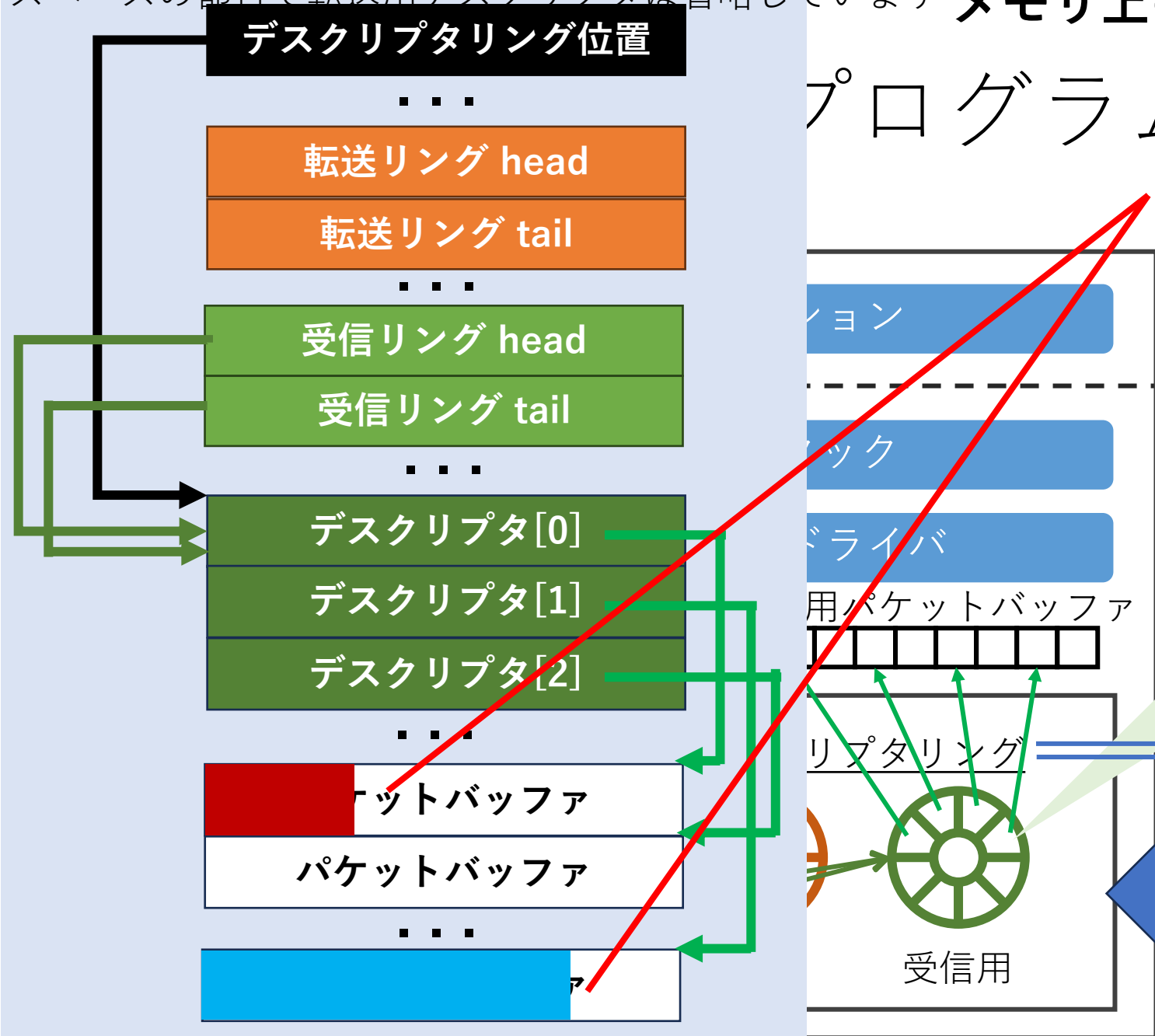
新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

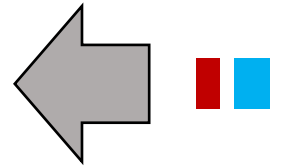
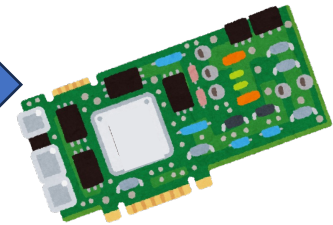
受信したデータはデスクリプタ経由で
紐づけられたパケットバッファへ書き込まれる



デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC の送受信キューを表現するデータ構造
(NIC のハードウェア仕様中で定義される)

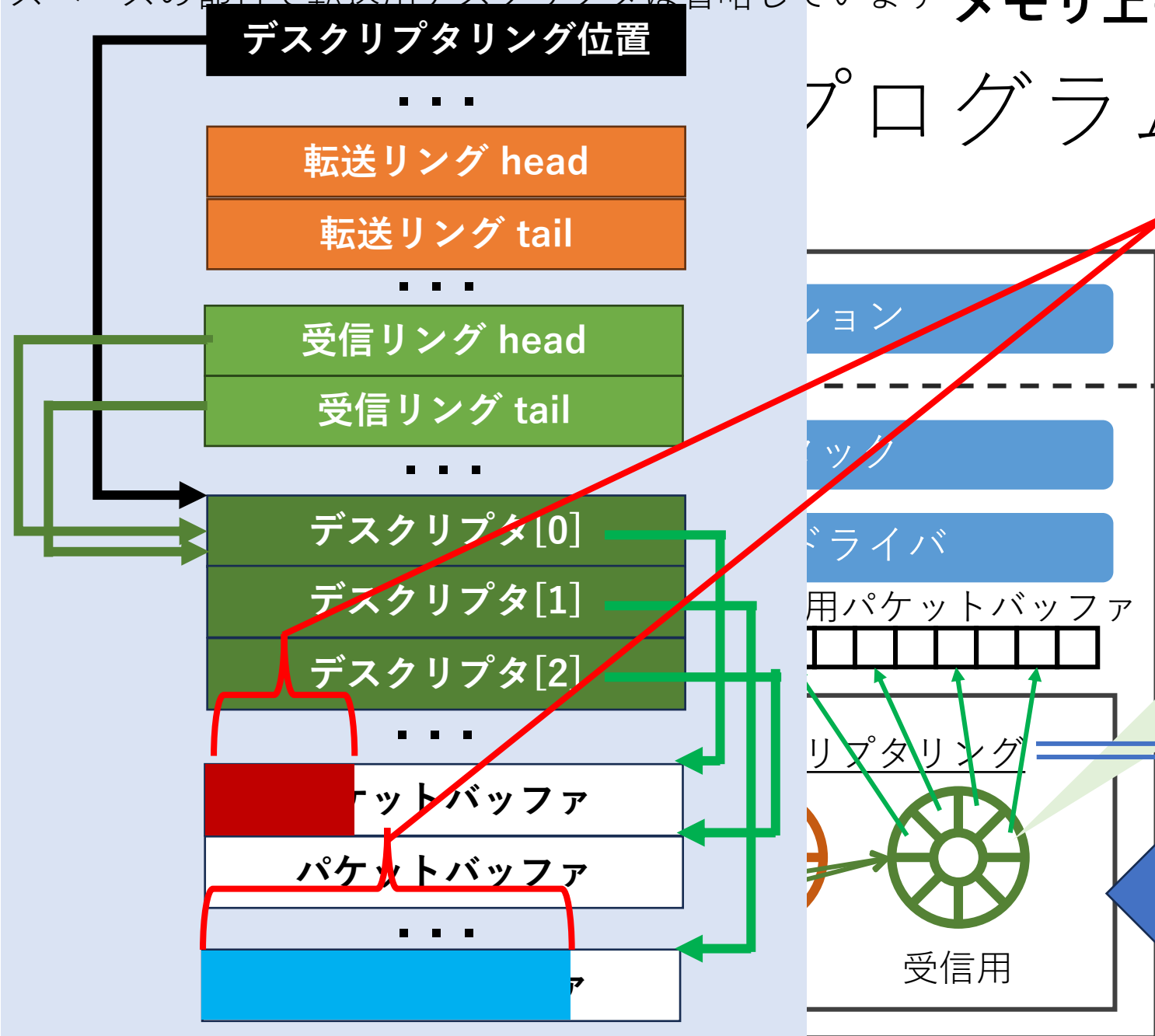


新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

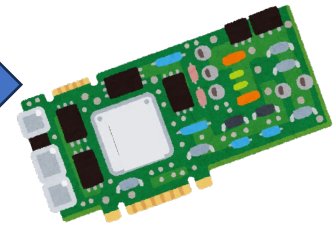
プログラムの構成



NICは受信したパケットのサイズを紐付けを行っているデスクリプタのフィールドに反映する

- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - パケットのサイズ
 - その他：状態保持用フラグ

NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)

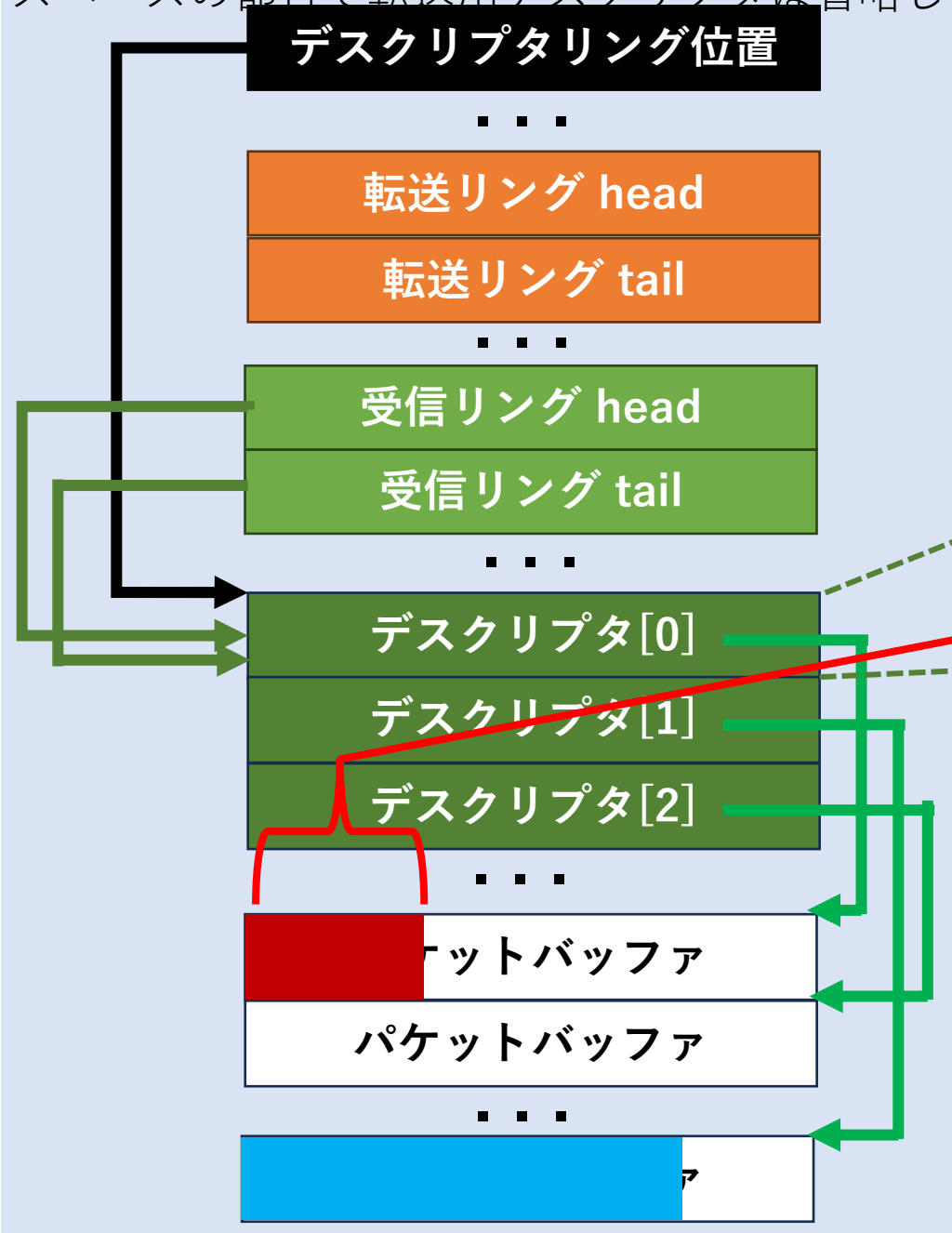


新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

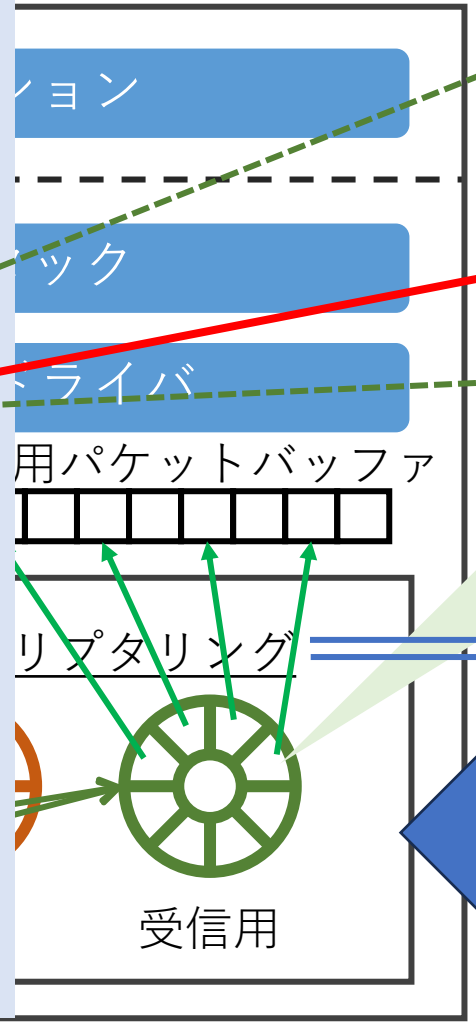
メモリ上の概観

プログラムの構成

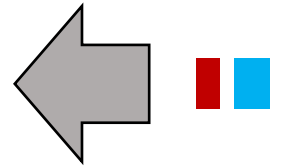
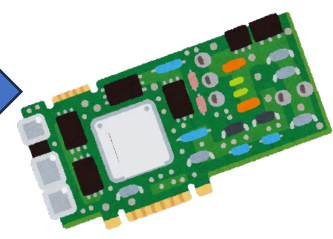
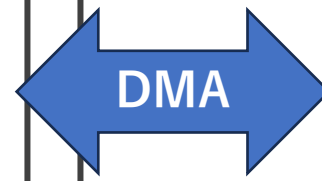


NICは受信したパケットのサイズを紐付けを行っているデスクリプタのフィールドに反映する

- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - **パケットのサイズ**
 - その他：状態保持用フラグ



NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

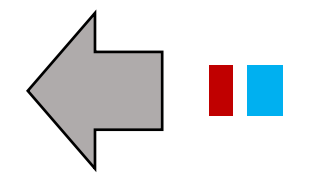
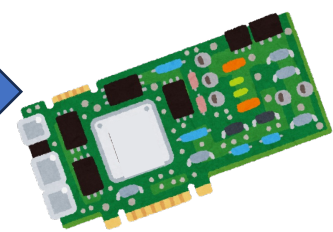
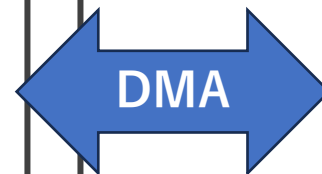
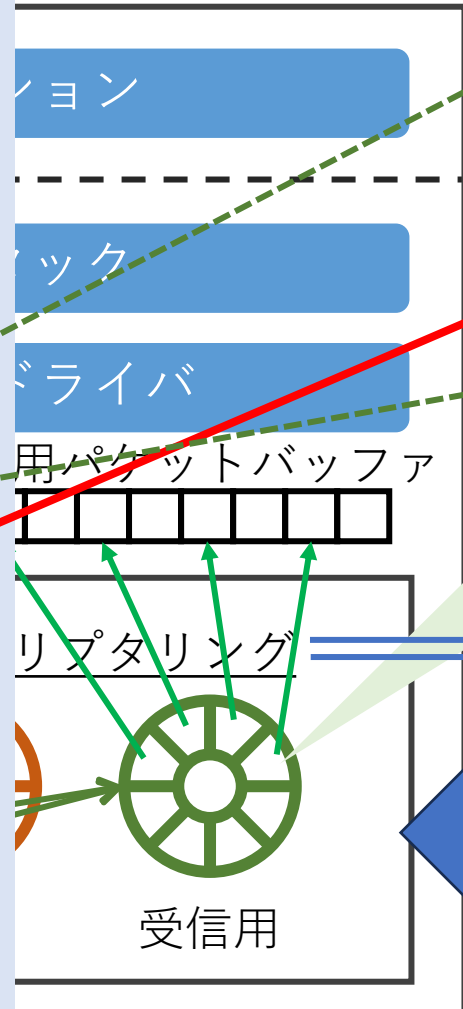
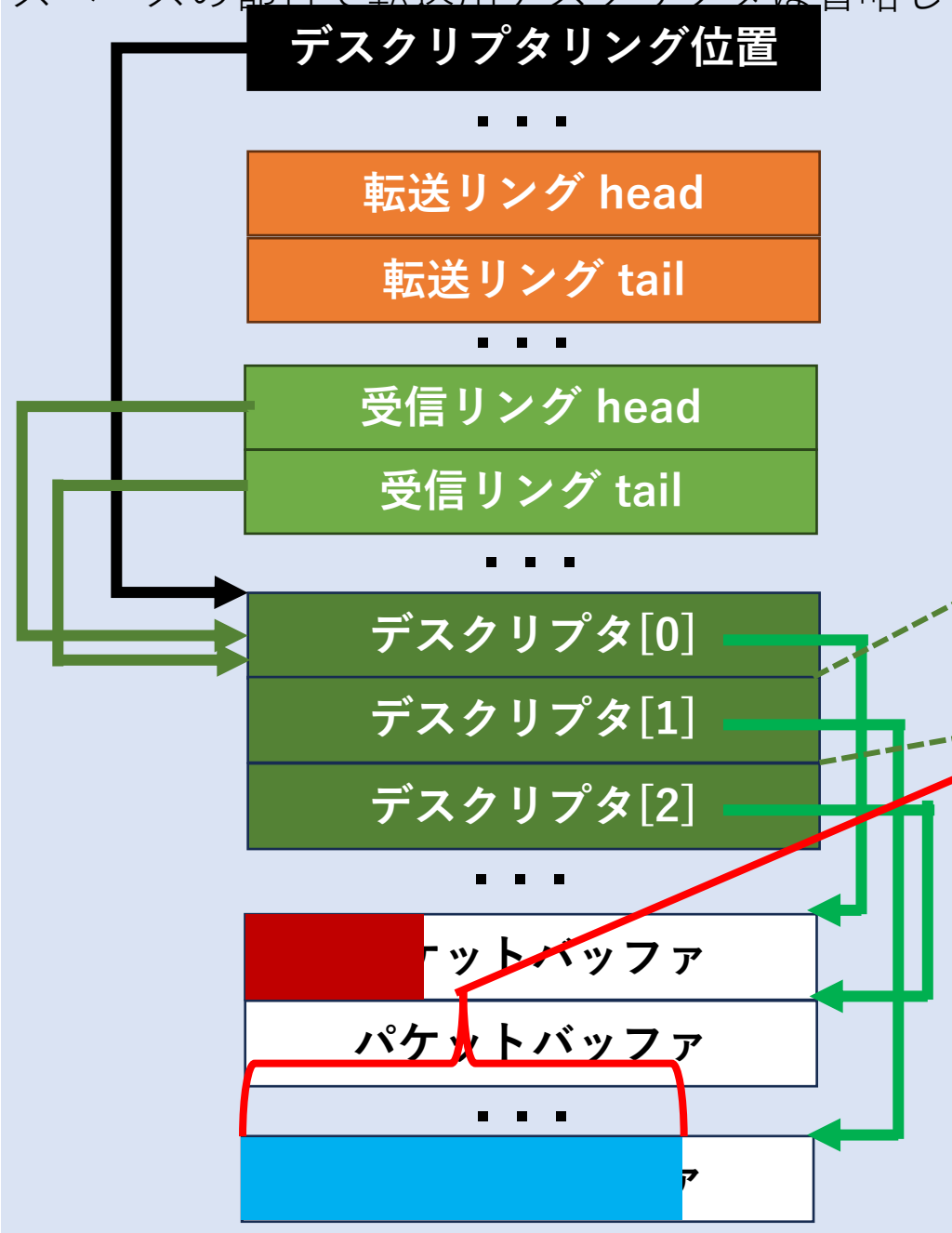
プログラムの構成

NICは受信したパケットのサイズを紐付けを行っているデスクリプタのフィールドに反映する

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- **パケットのサイズ**
- その他：状態保持用フラグ

NICの送受信キューを表現するデータ構造 (NICのハードウェア仕様中で定義される)



新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

その後、NICによって、レジスタの受信リング head の値が更新される

デスクリプタリング位置

転送リング head

転送リング tail

受信リング head

受信リング tail

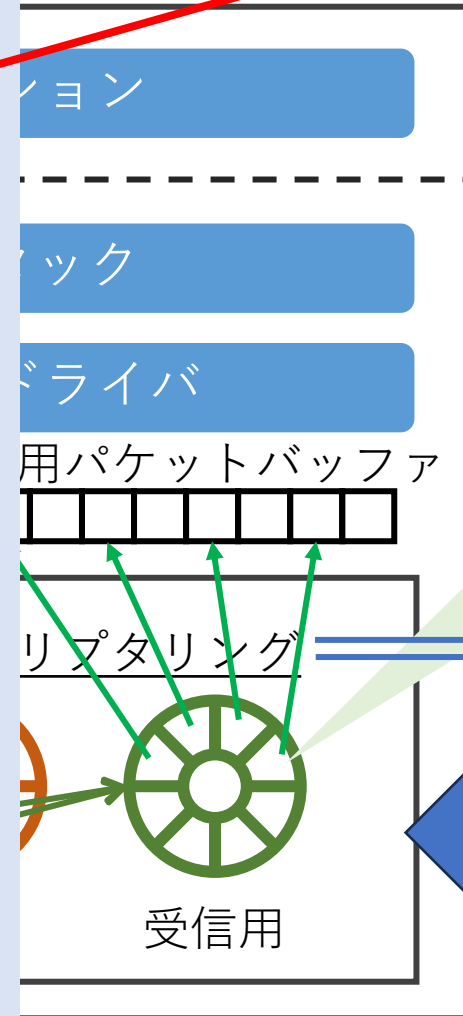
デスクリプタ[0]

デスクリプタ[1]

デスクリプタ[2]

ケットバッファ

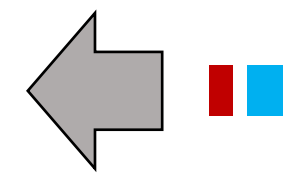
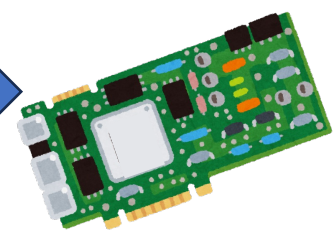
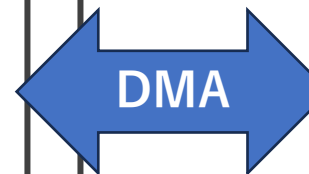
ケットバッファ



デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)

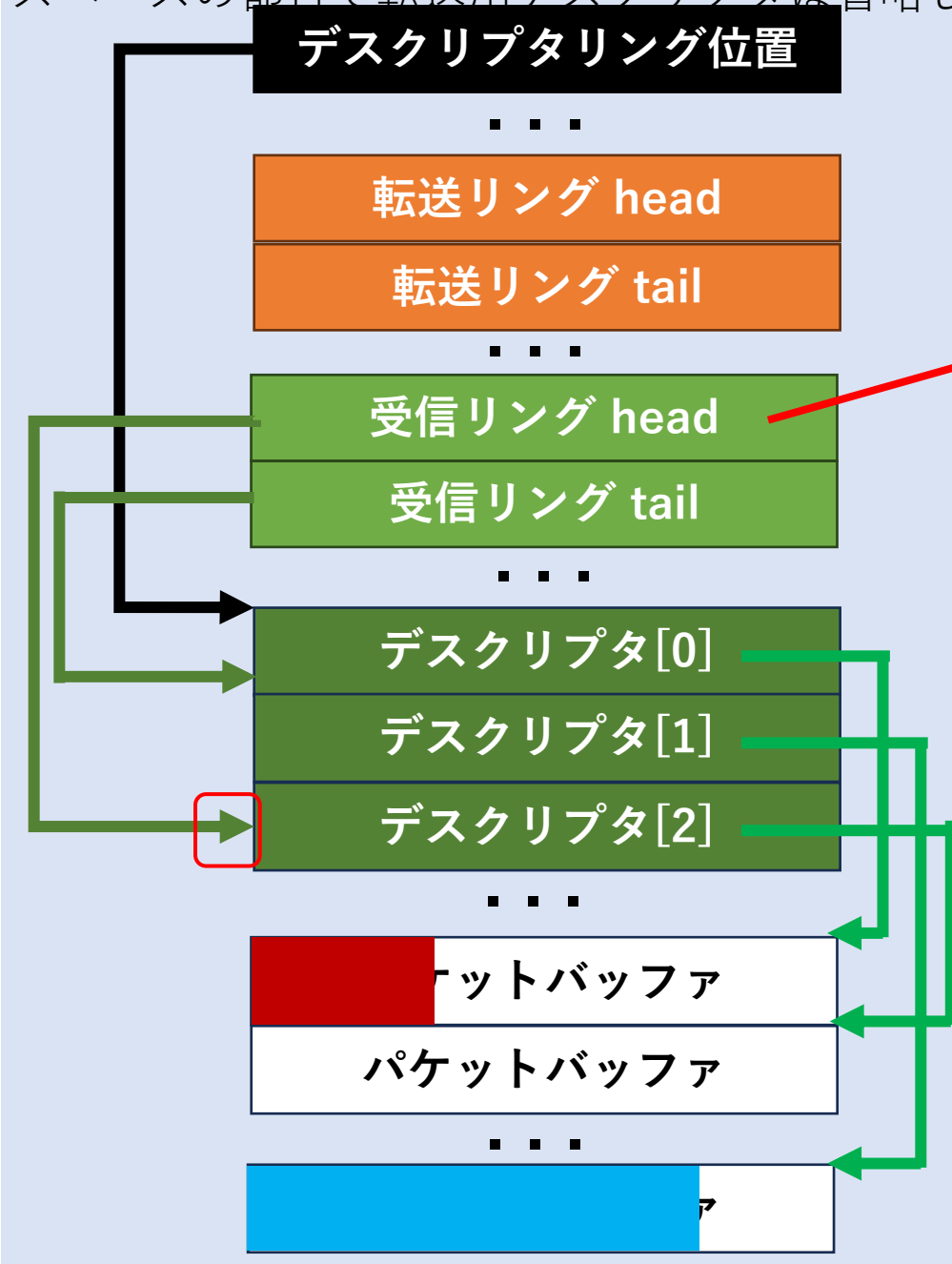


新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

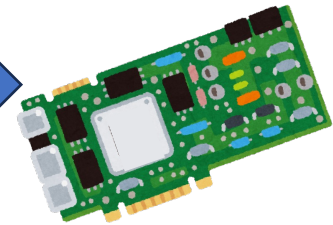


その後、NICによって、レジスタの受信リング head の値が更新される

- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - パケットのサイズ
 - その他：**状態保持用フラグ**

(受信リング head ではなく、状態保持用フラグに受信状況を設定するNICもあります)

NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)



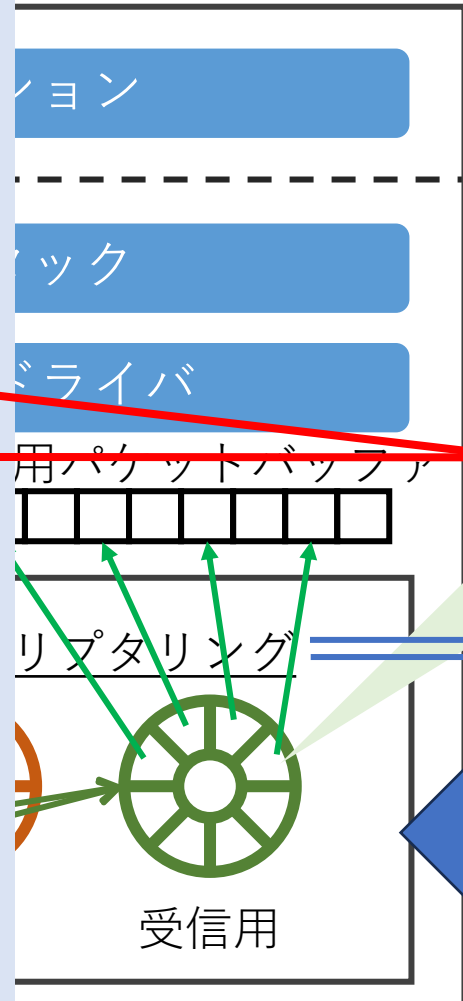
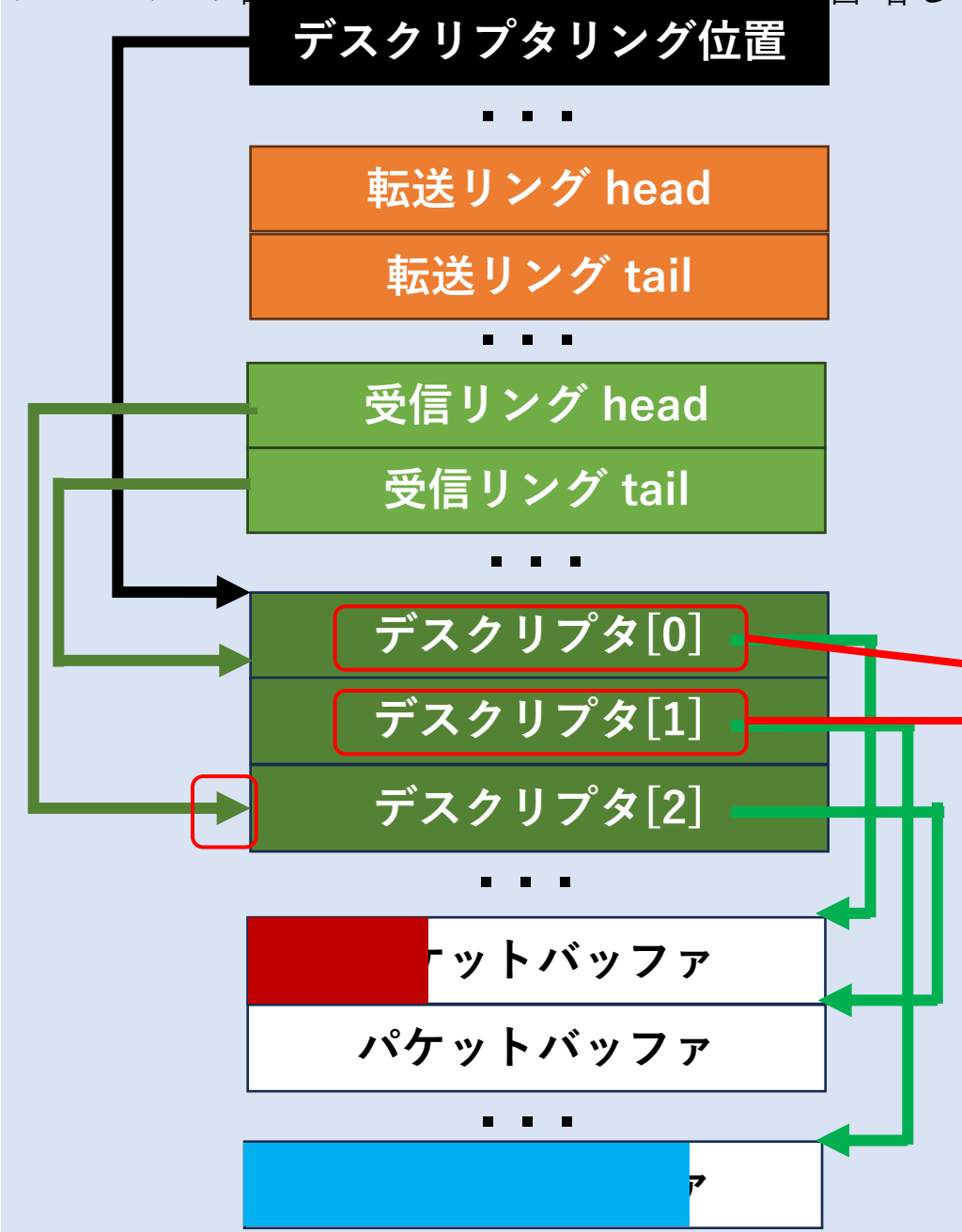
新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

その後、NICによって、レジスタの受信リング head の値が更新される

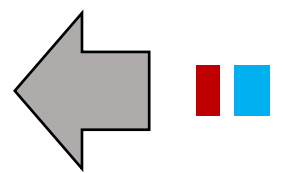
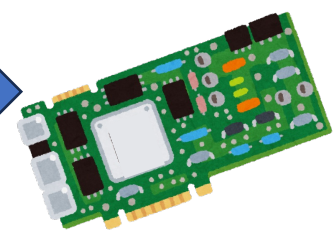


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

ソフトウェアは NIC レジスタの受信リング head を読み取り、デスクリプタ[0, 1] に紐づくバッファにデータが書き込まれたことを検知する

NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)

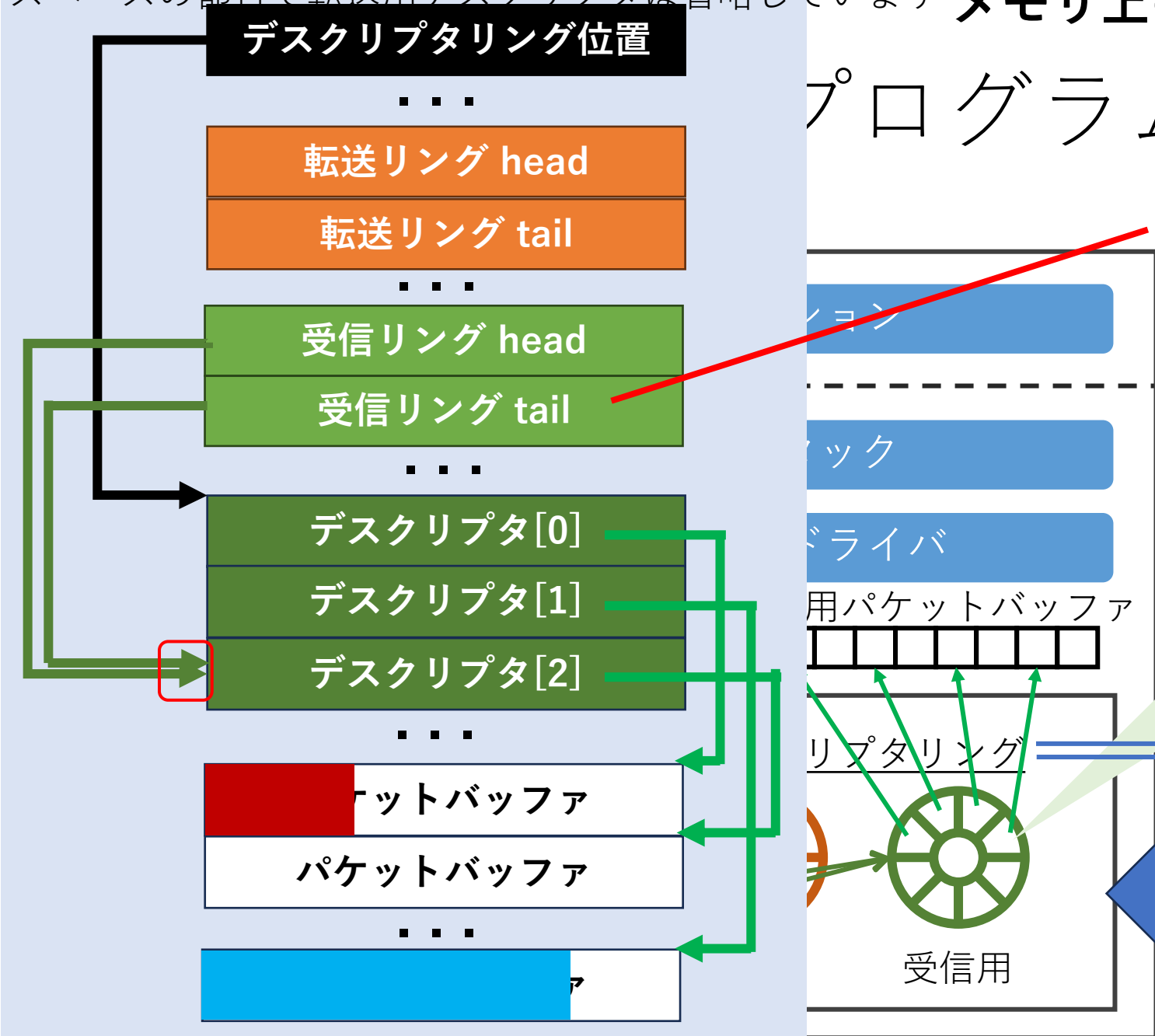


新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成



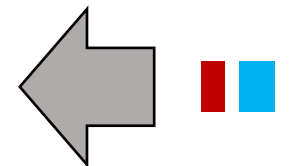
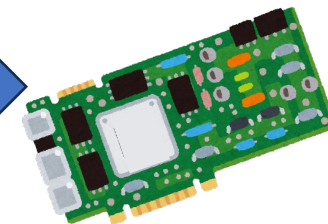
ソフトウェアは受信パケットを受け取った後レジスタの受信リング tail の値を更新して受信パケットを消費したことを NIC へ通知する

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

ソフトウェアは NIC レジスタの受信リング head を読み取り、デスクリプタ[0, 1] に紐づくバッファにデータが書き込まれたことを検知する

NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)



新規パケットの到着

スペースの都合で転送用デスクリプタは省略しています

メモリ上の概観

プログラムの構成

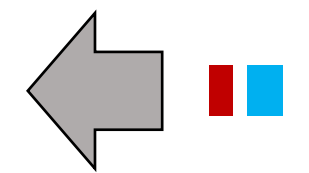
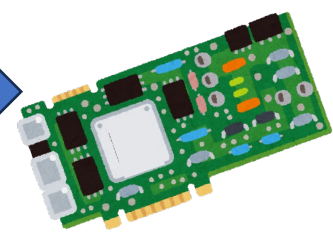
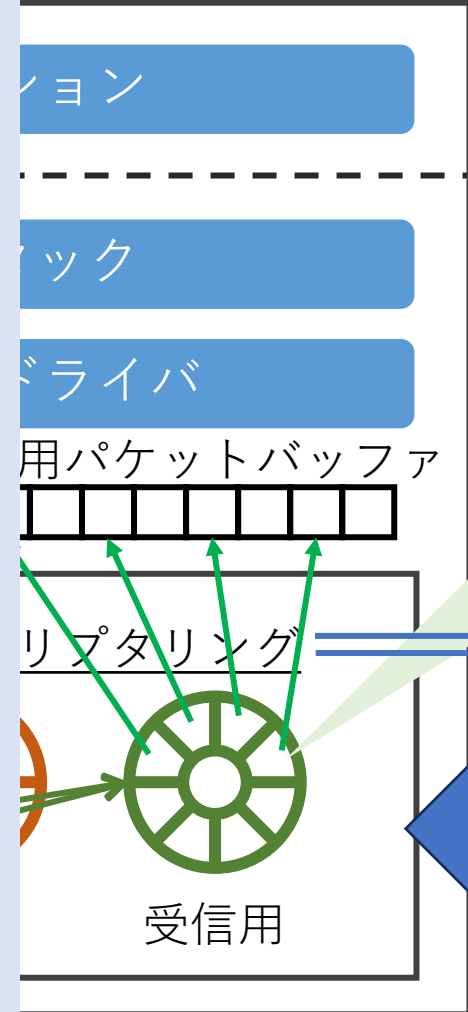
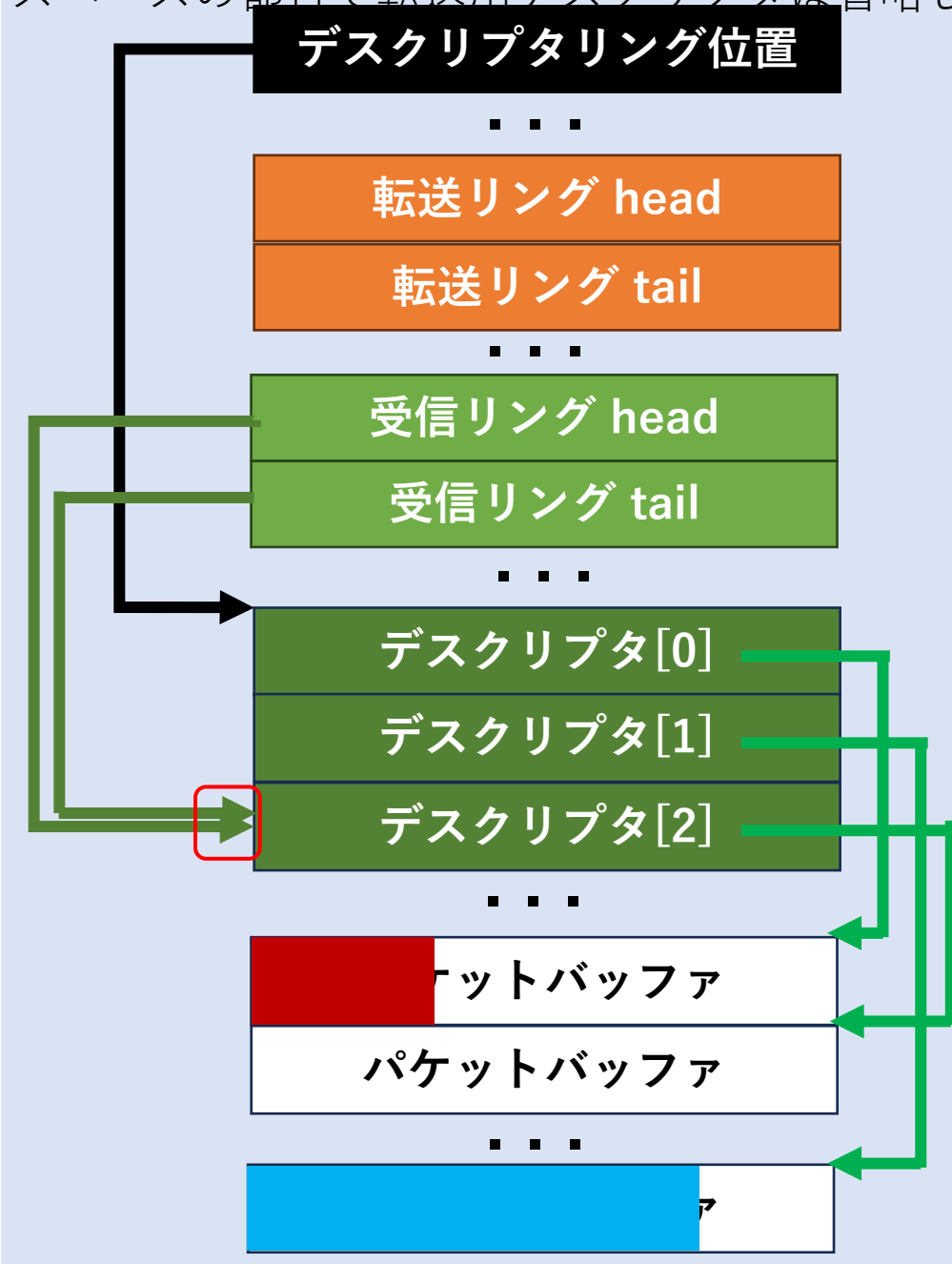
head がデスクリプタ配列の最後まで進められたら head はデスクリプタ[0] へ戻る (リングバッファとして機能する)

デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

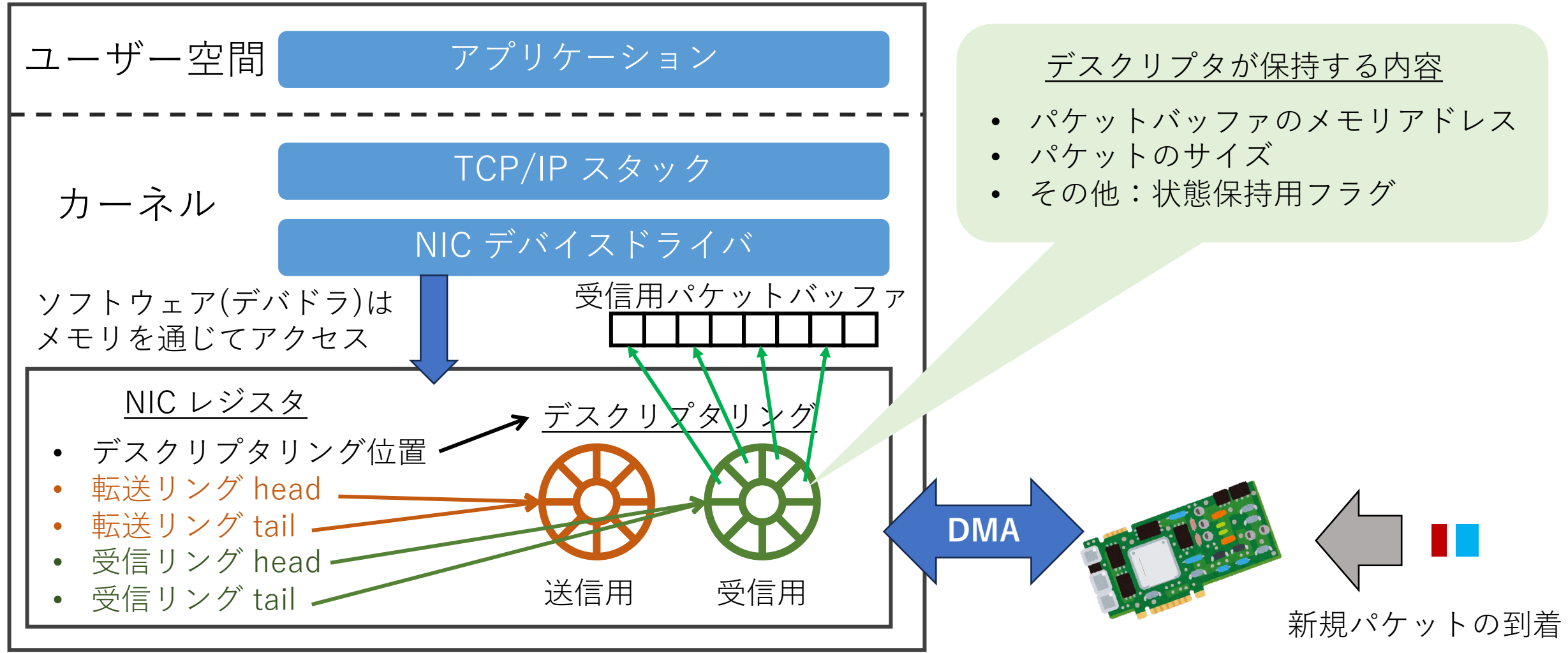
ソフトウェアは NIC レジスタの受信リング head を読み取り、デスクリプタ[0, 1] に紐づくバッファにデータが書き込まれたことを検知する

NIC の送受信キューを表現するデータ構造 (NIC のハードウェア仕様中で定義される)

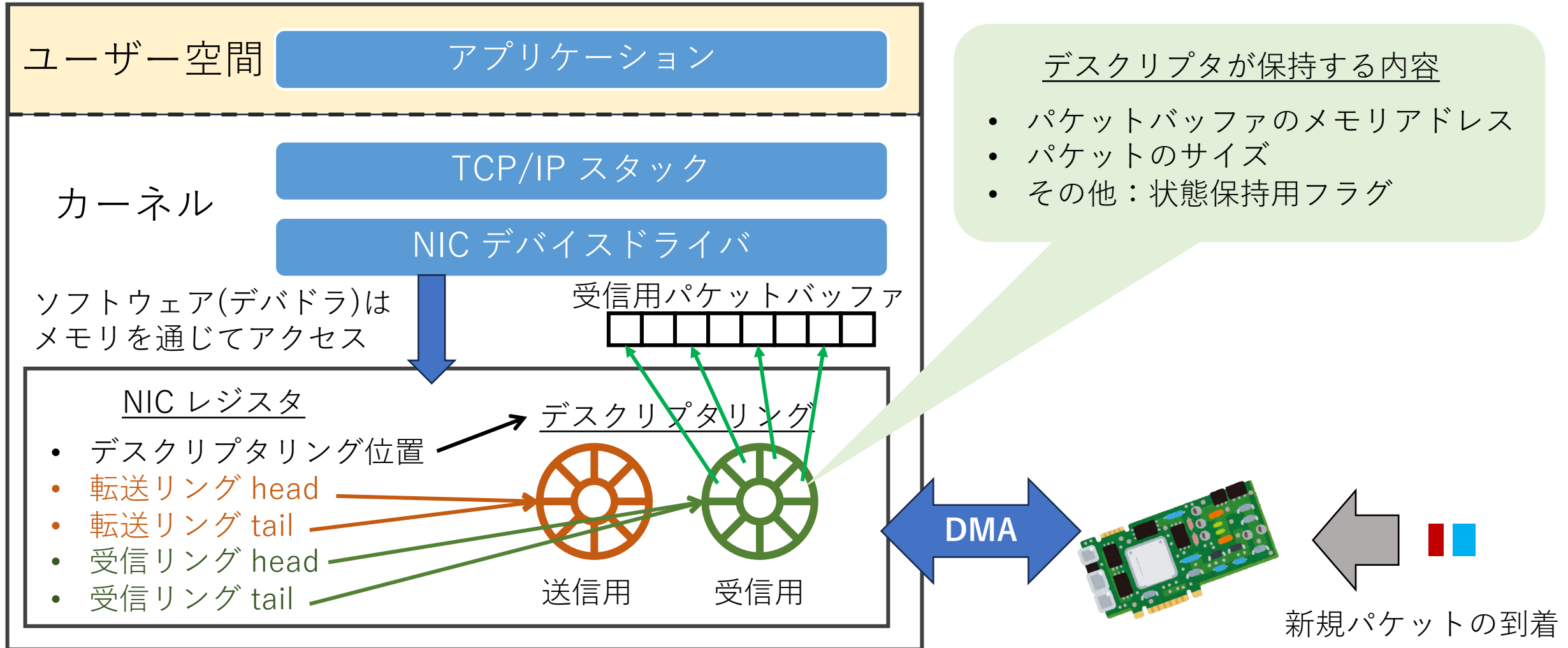


新規パケットの到着

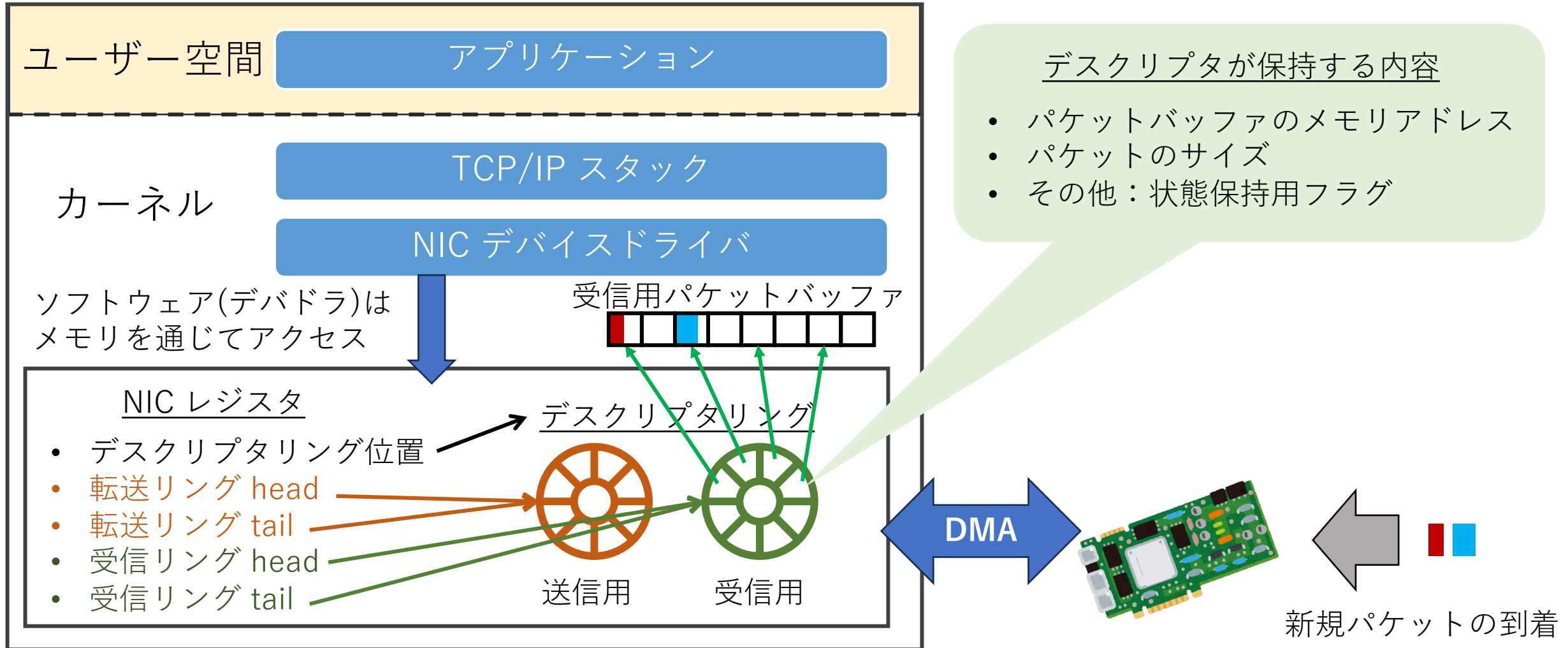
NIC と通信関連プログラムの構成



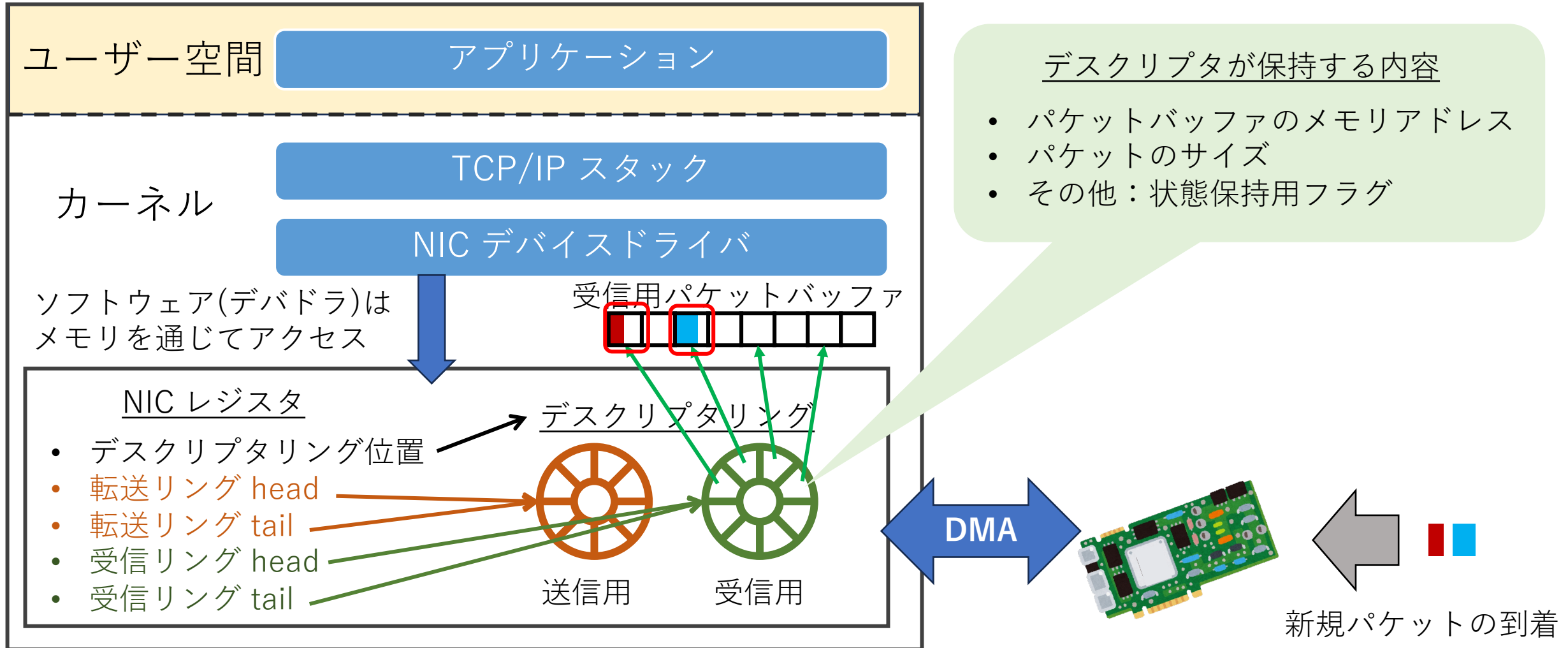
(ある程度) 一般的な受信処理の流れ



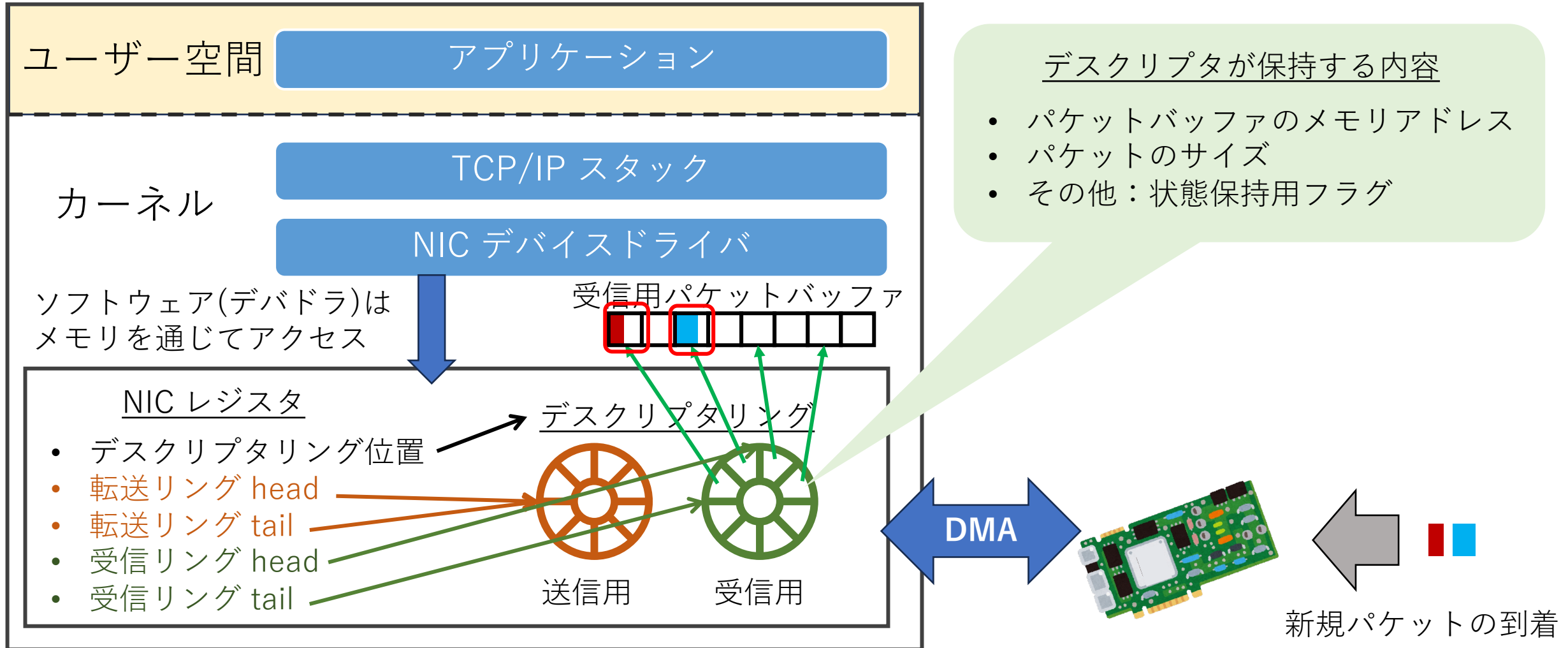
(ある程度) 一般的な受信処理の流れ



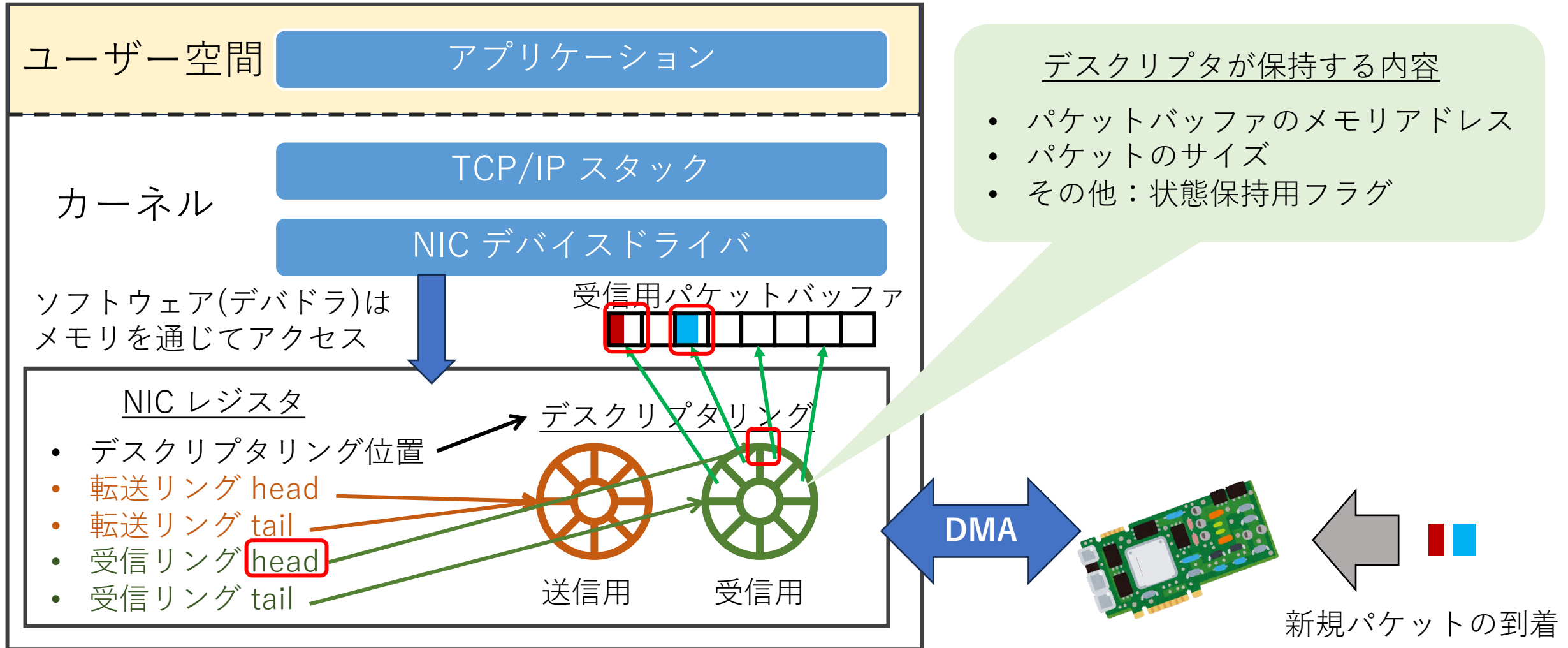
(ある程度) 一般的な受信処理の流れ



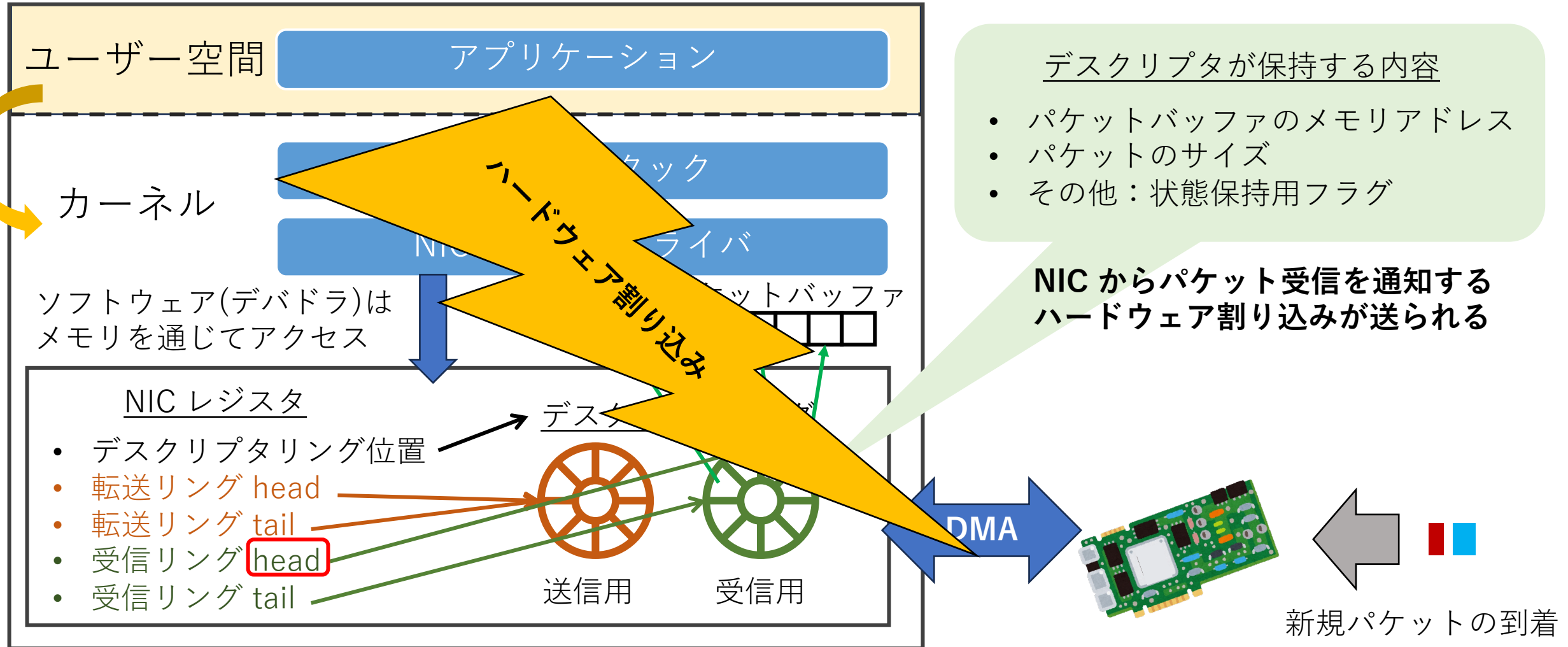
(ある程度) 一般的な受信処理の流れ



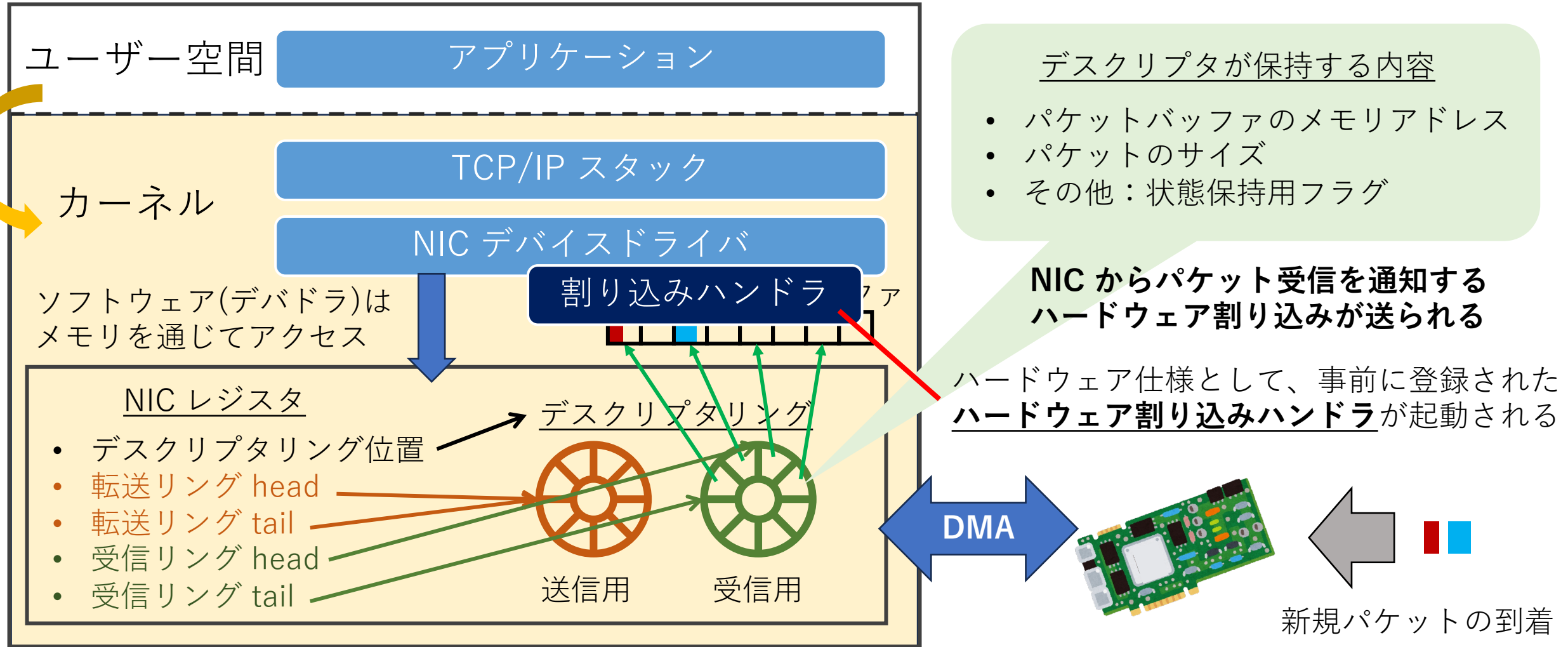
(ある程度) 一般的な受信処理の流れ



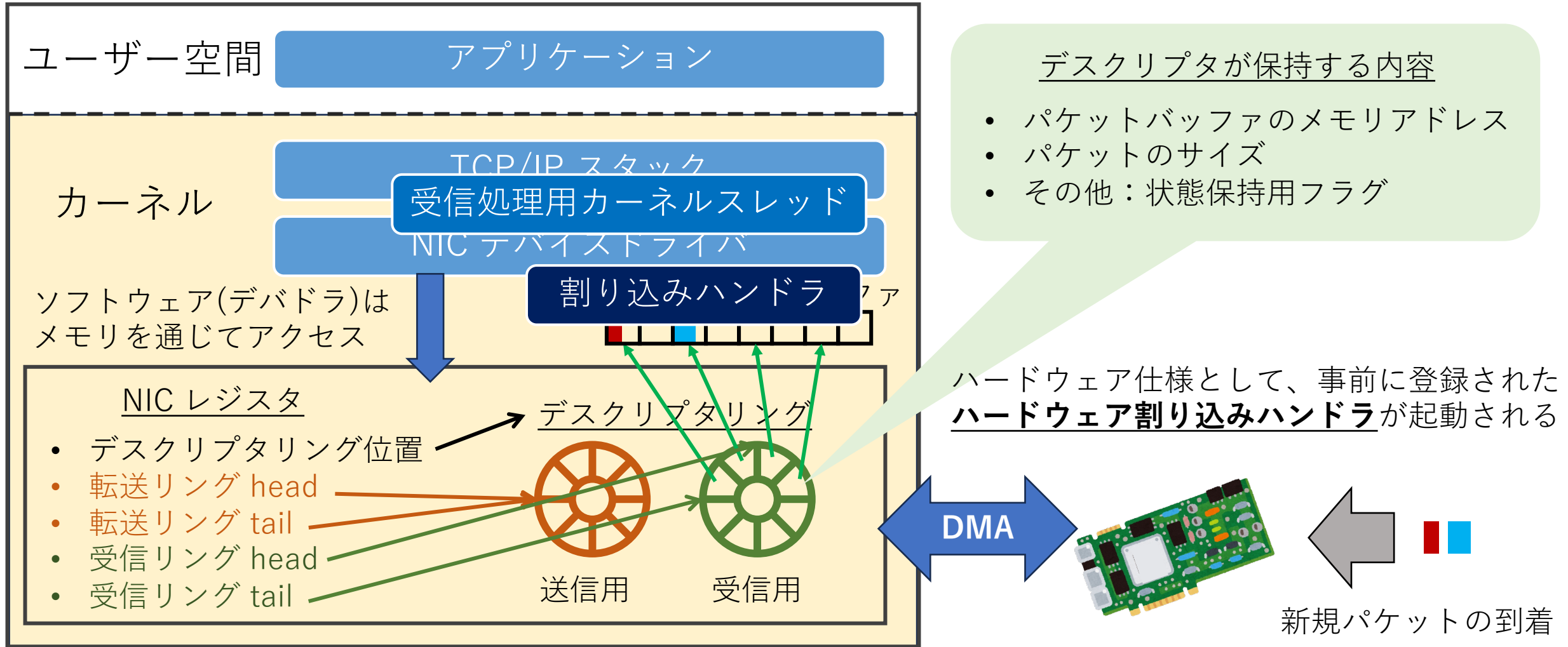
(ある程度) 一般的な受信処理の流れ



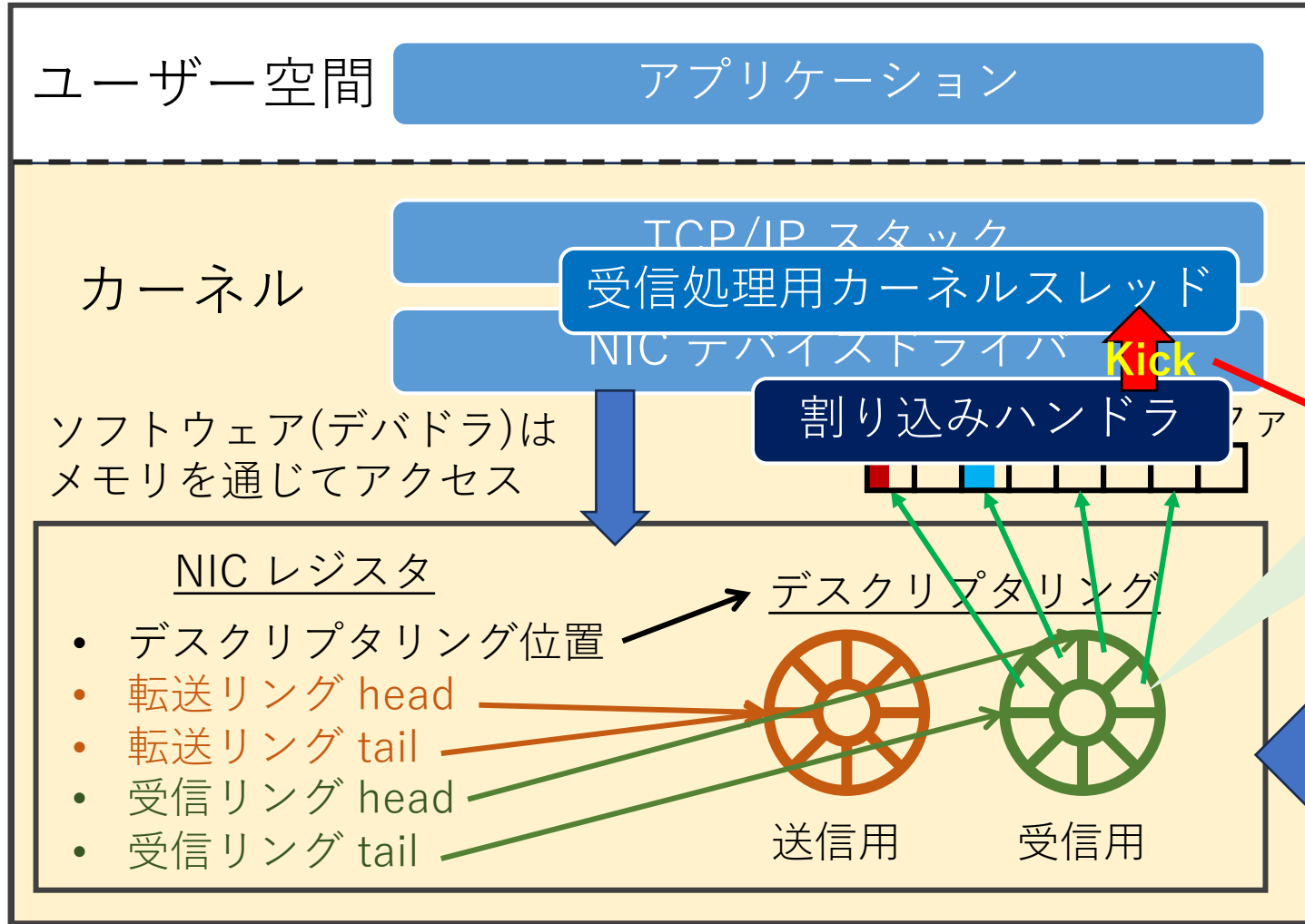
(ある程度) 一般的な受信処理の流れ



(ある程度) 一般的な受信処理の流れ



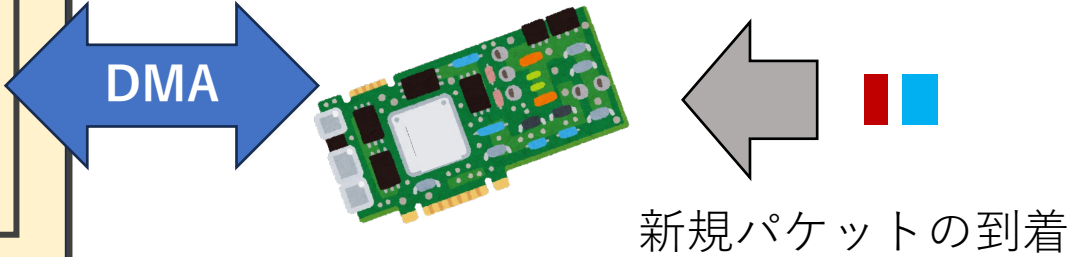
(ある程度) 一般的な受信処理の流れ



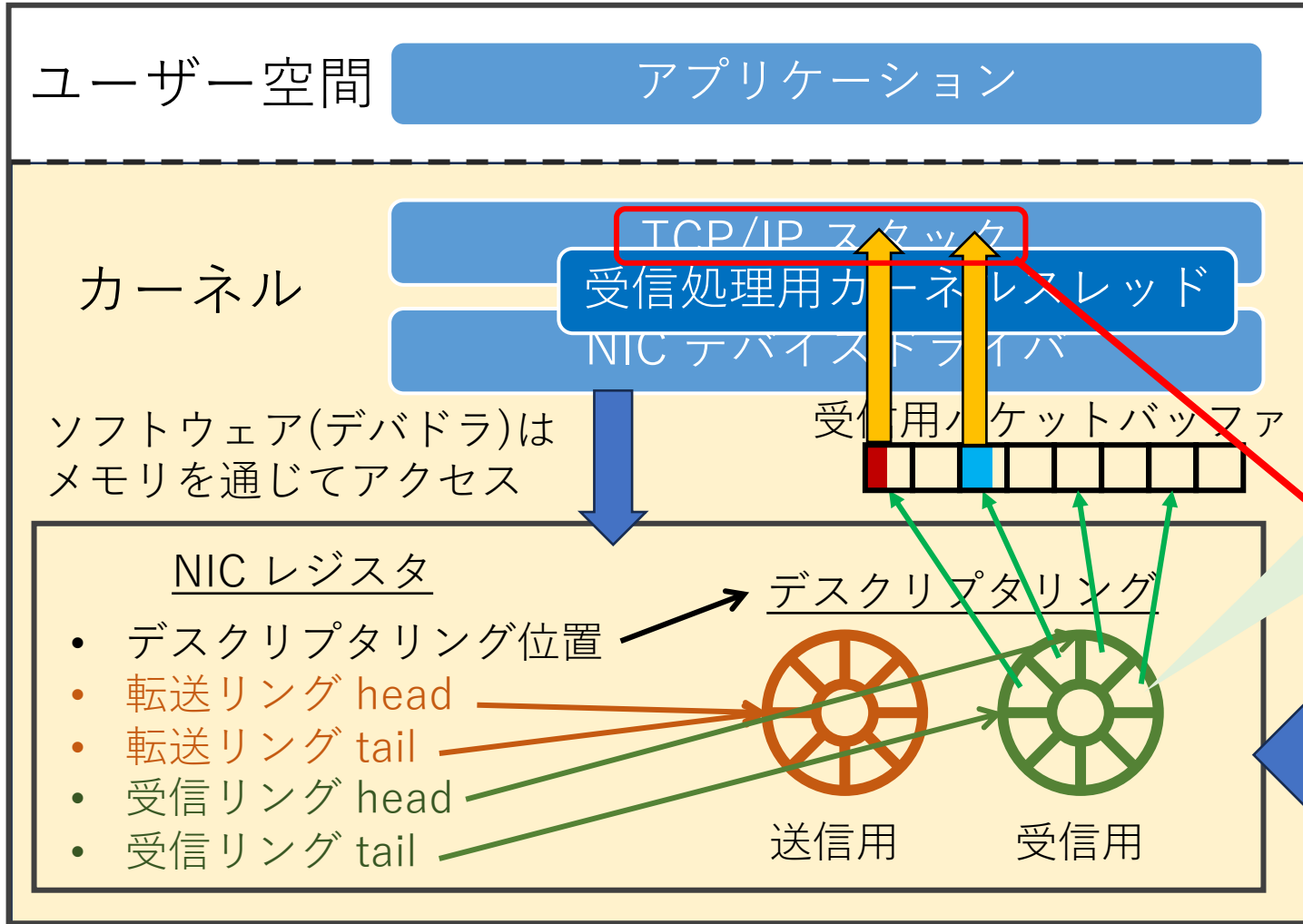
- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - パケットのサイズ
 - その他：状態保持用フラグ

ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

ハードウェア仕様として、事前に登録されたハードウェア割り込みハンドラが起動される



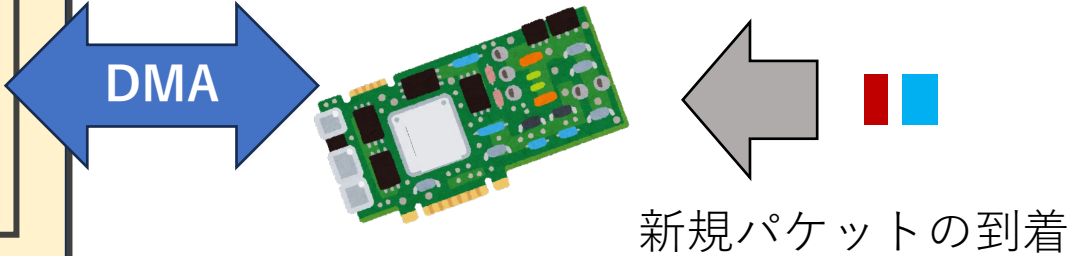
(ある程度) 一般的な受信処理の流れ



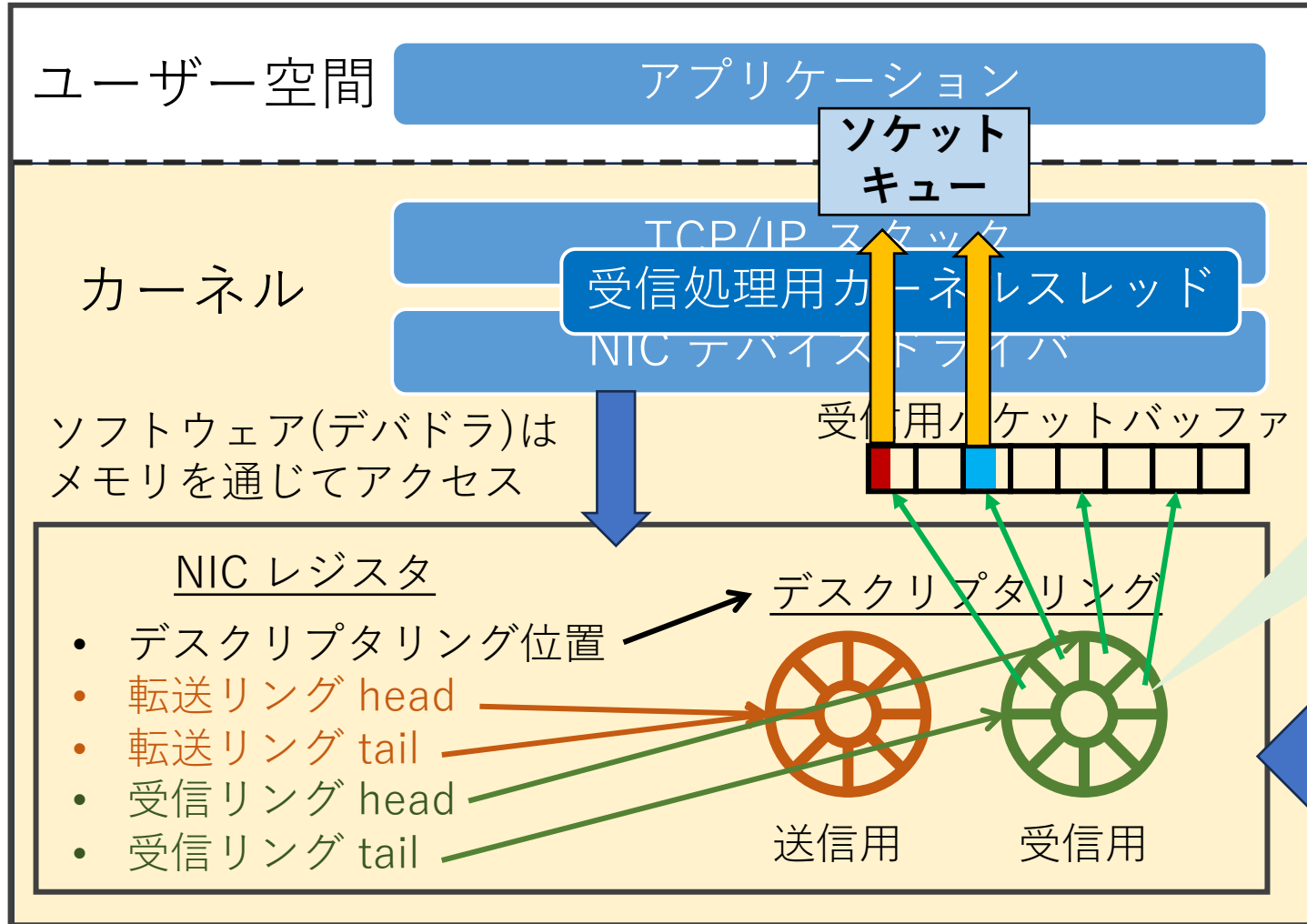
- デスクリプタが保持する内容
- パケットバッファのメモリアドレス
 - パケットのサイズ
 - その他：状態保持用フラグ

ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

このカーネルスレッドがパケットバッファからデータを取り出し TCP/IP 処理を実行



(ある程度) 一般的な受信処理の流れ

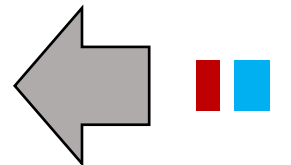
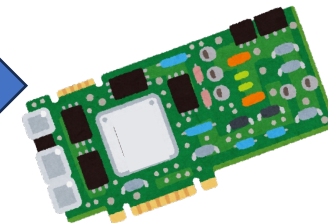


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

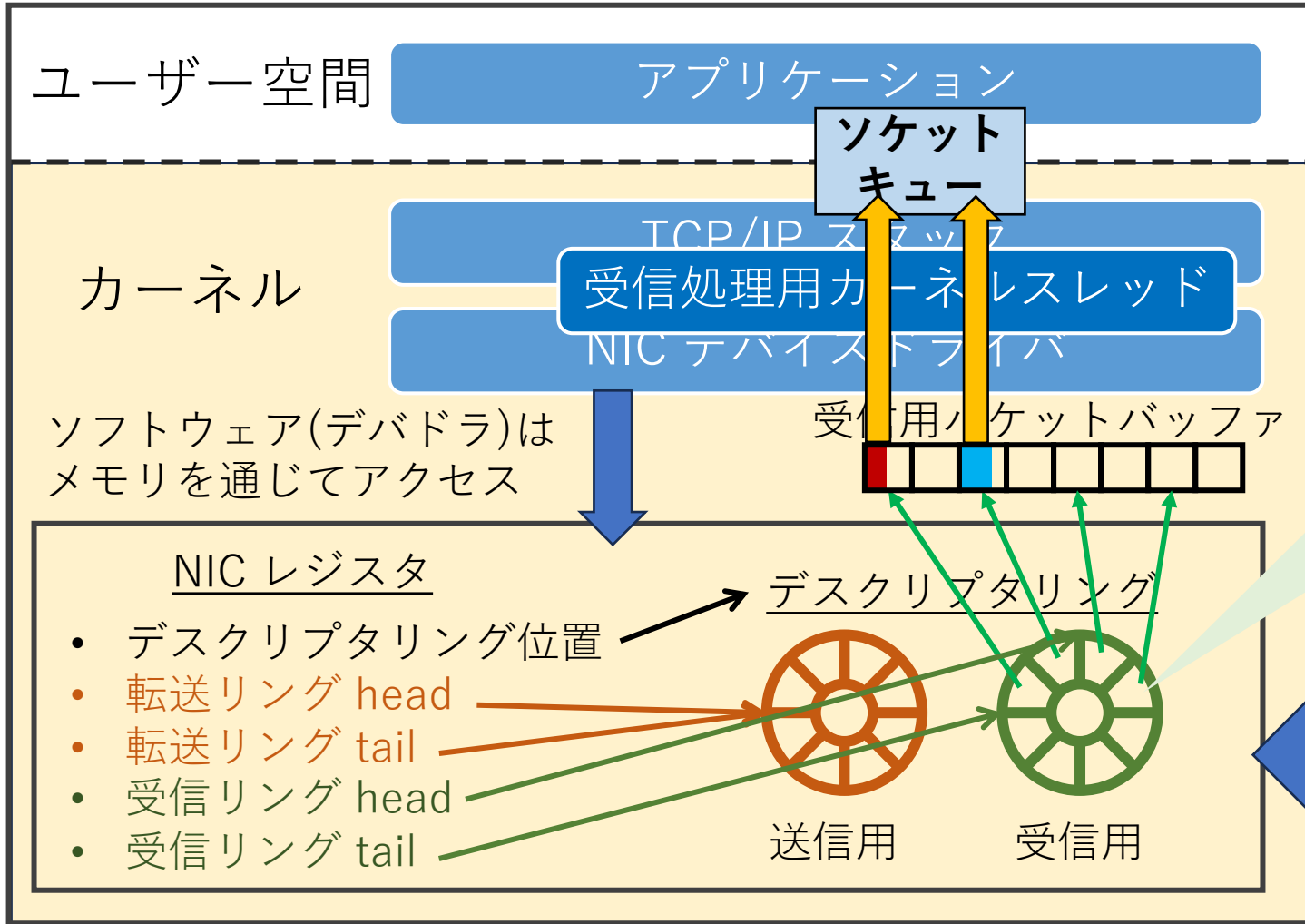
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

このカーネルスレッドがパケットバッファからデータを取り出し TCP/IP 処理を実行



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

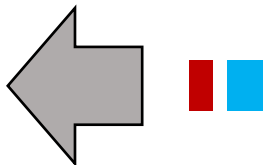
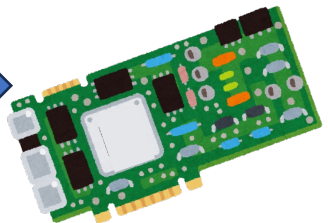


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

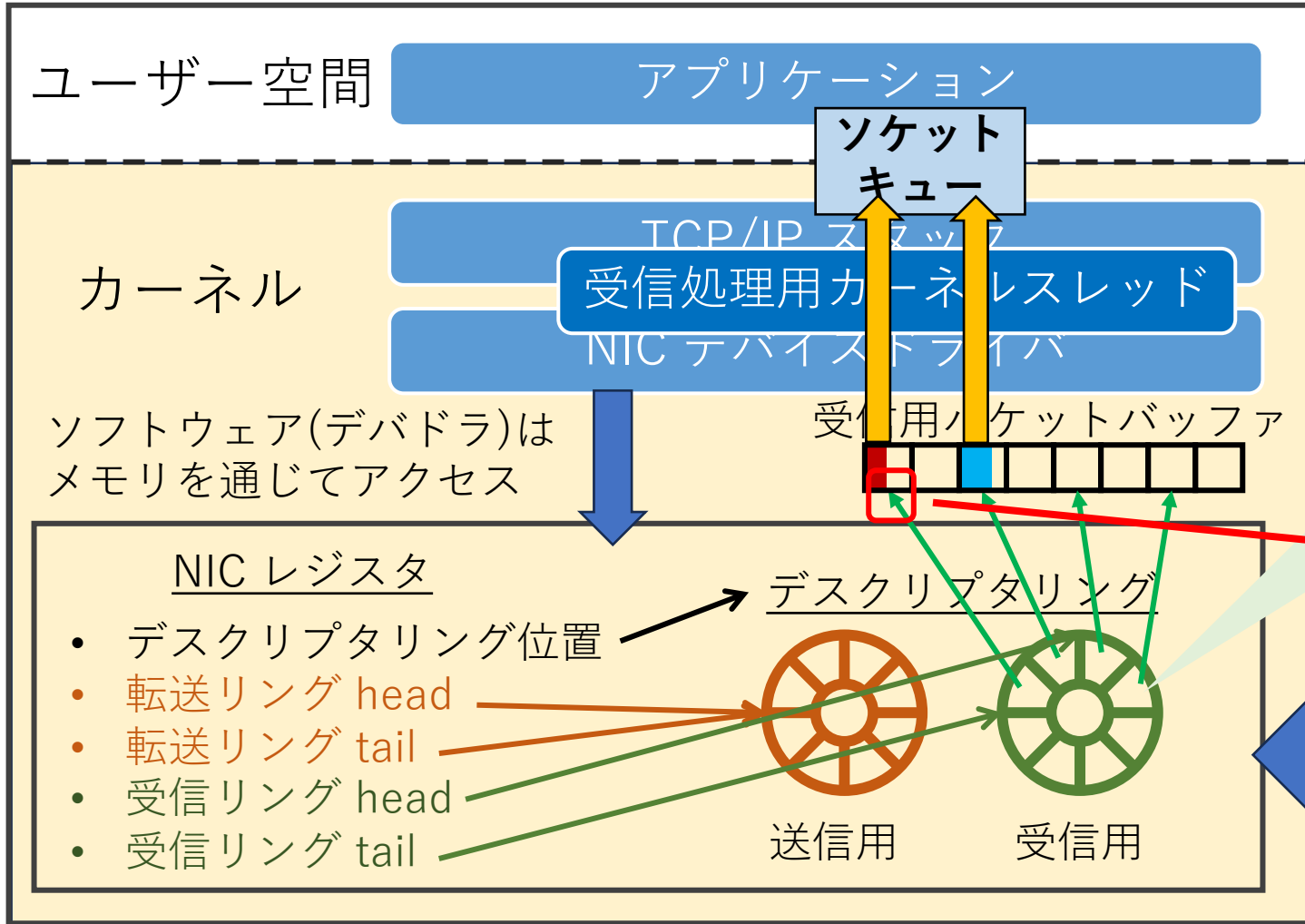
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

このカーネルスレッドがパケットバッファからデータを取り出し TCP/IP 処理を実行 + ソケットのキューへデータを紐付け



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

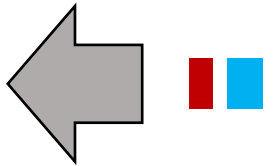
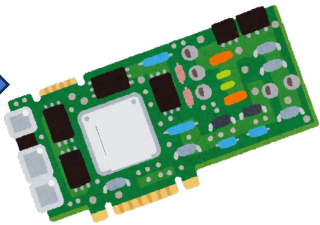


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

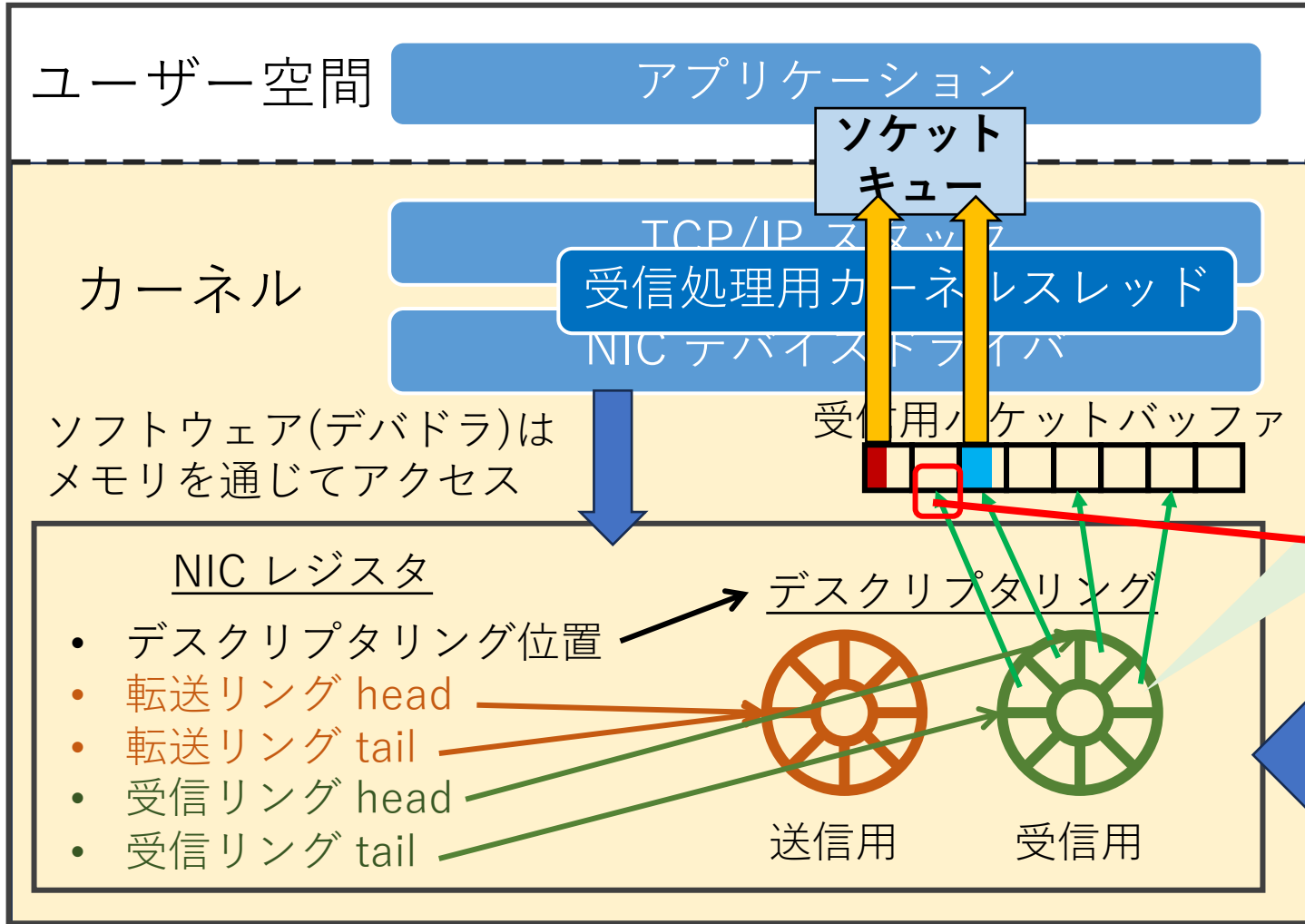
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

受信用ソケットバッファと NIC の紐付きを解除 + 新しいバッファを紐づける



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

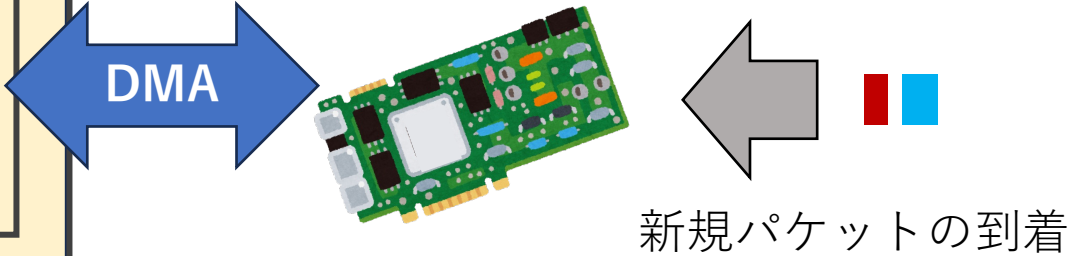


デスクリプタが保持する内容

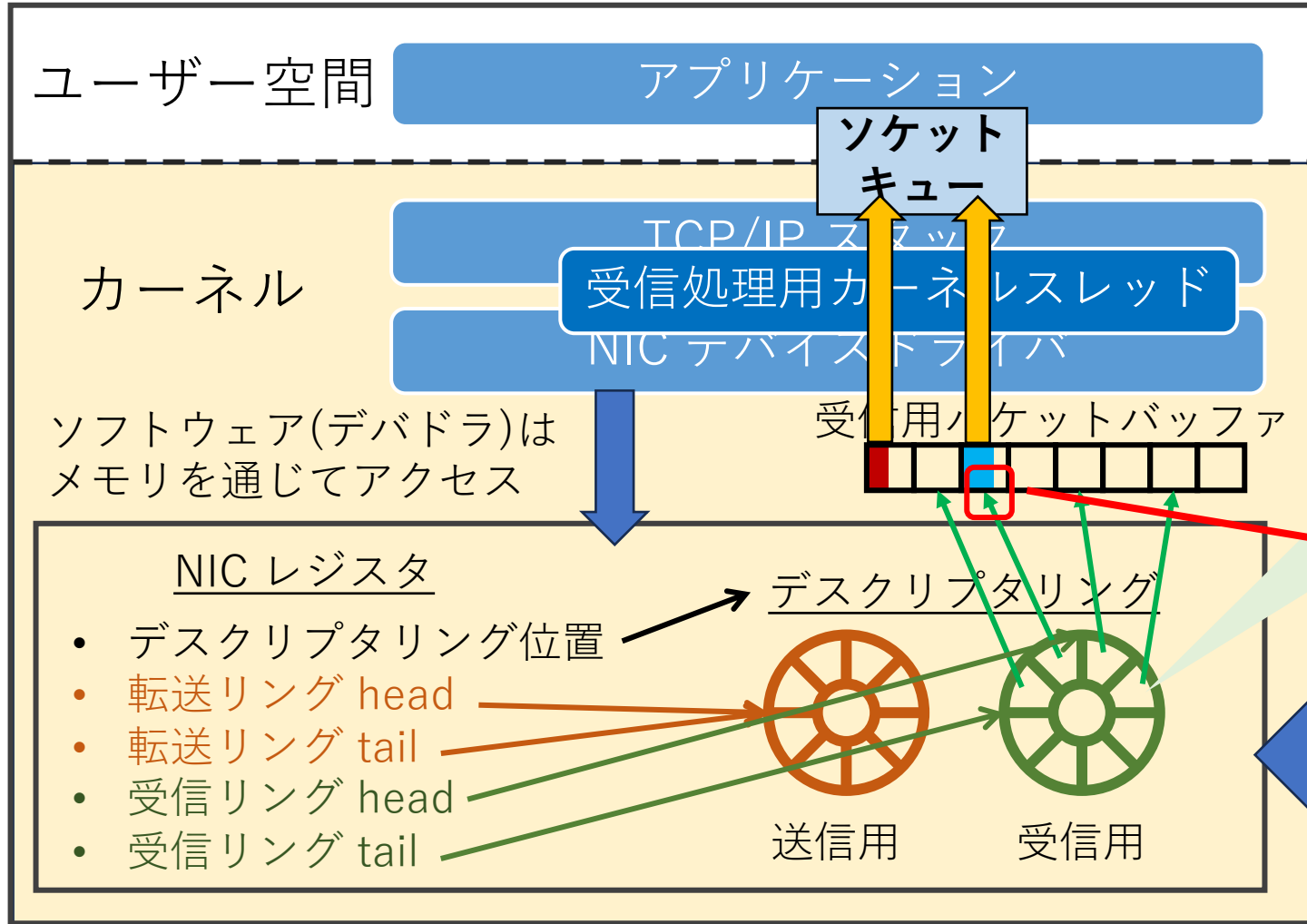
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

受信用パケットバッファと NIC の紐付きを解除 + 新しいバッファを紐づける



(ある程度) 一般的な受信処理の流れ

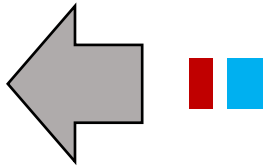
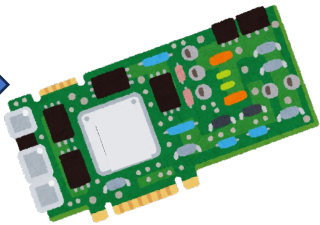


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

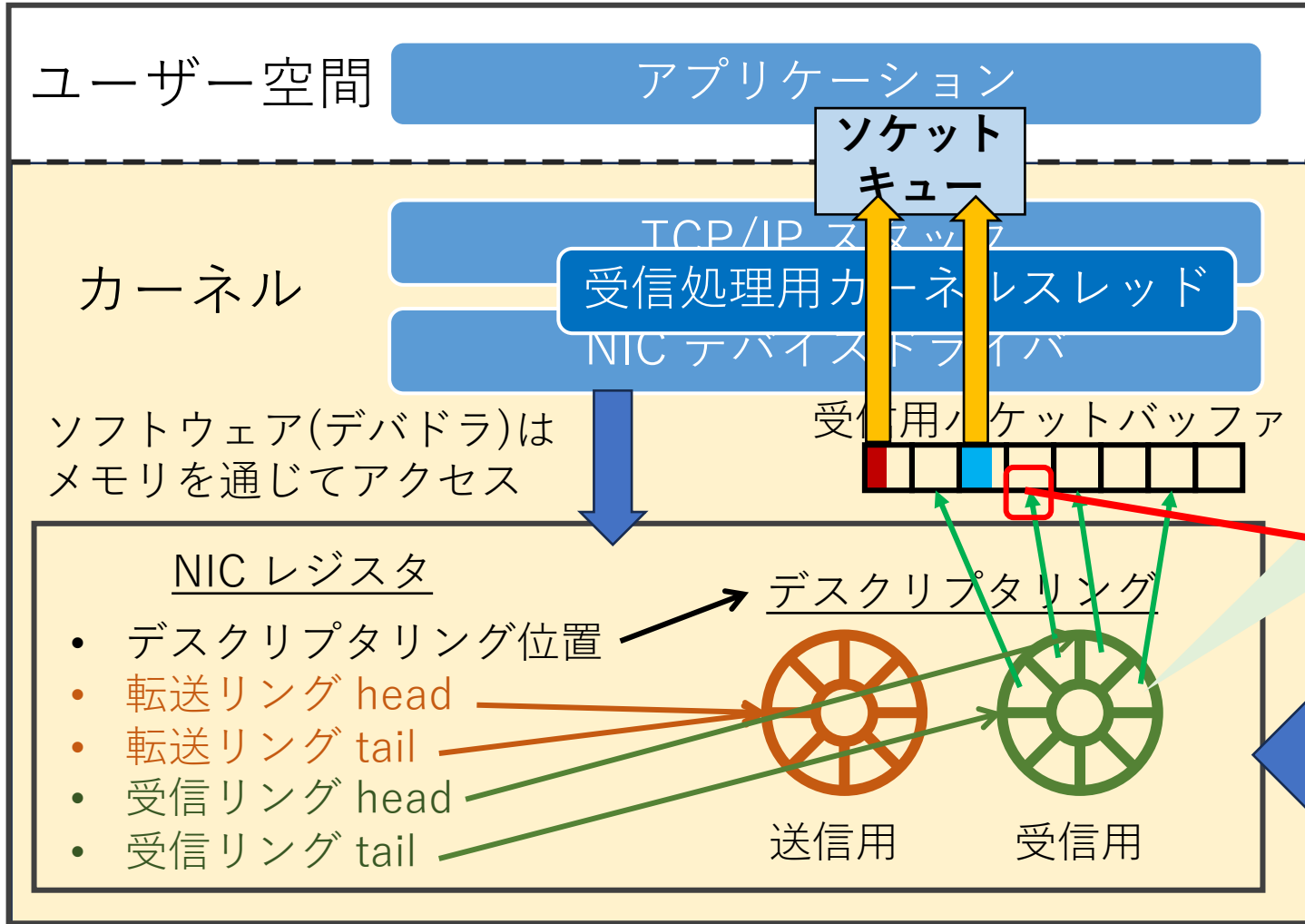
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

受信用ソケットバッファと NIC の紐付きを解除 + 新しいバッファを紐づける



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

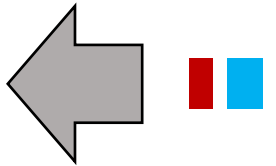
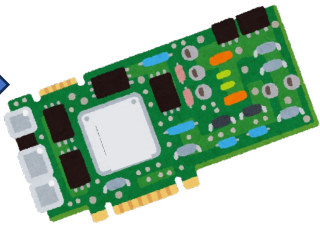
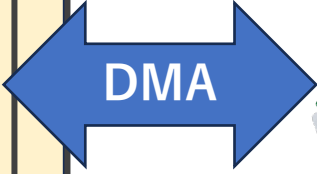


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

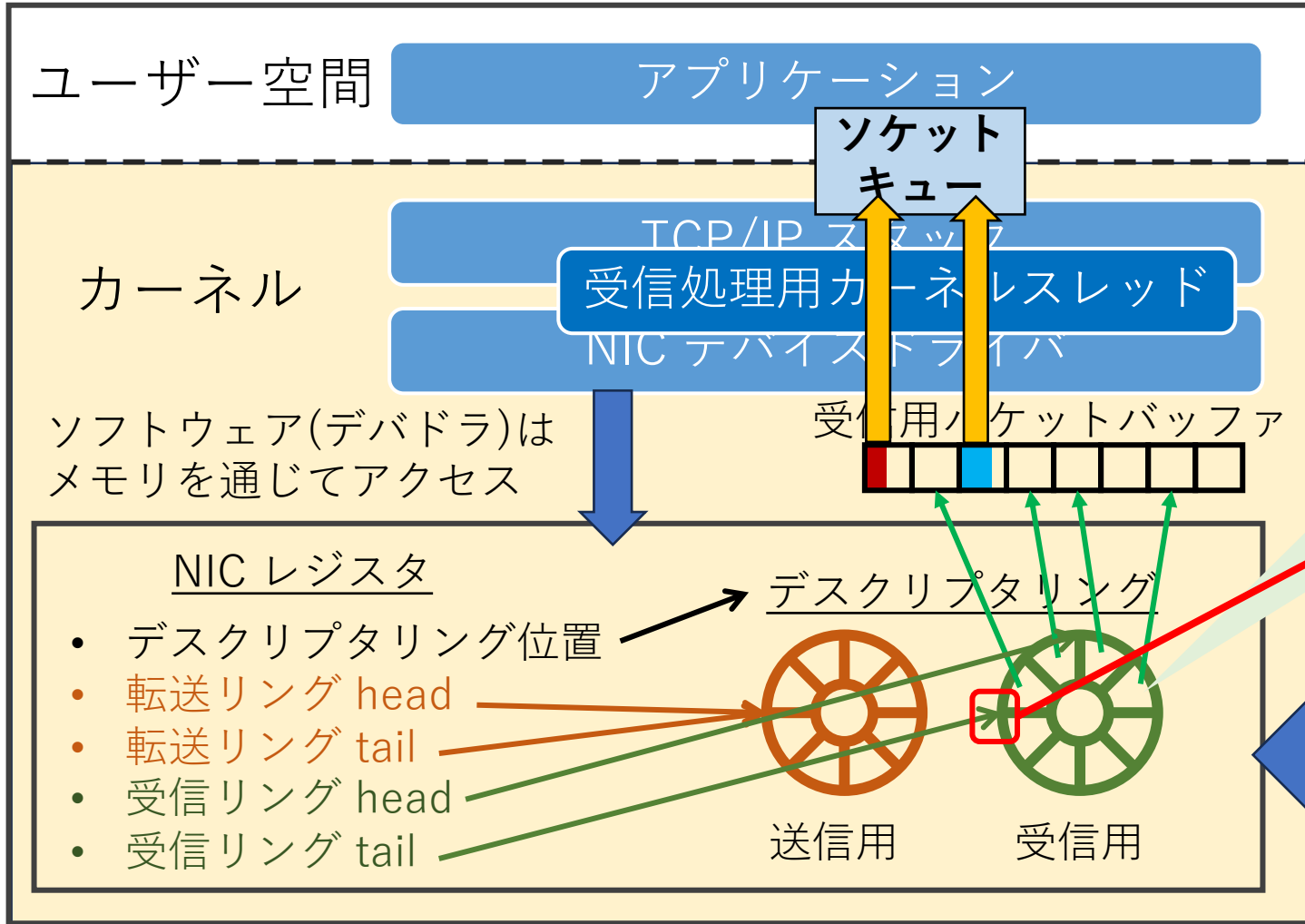
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

受信用ソケットバッファと NIC の紐付きを解除 + 新しいバッファを紐づける



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

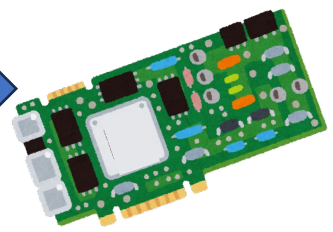


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

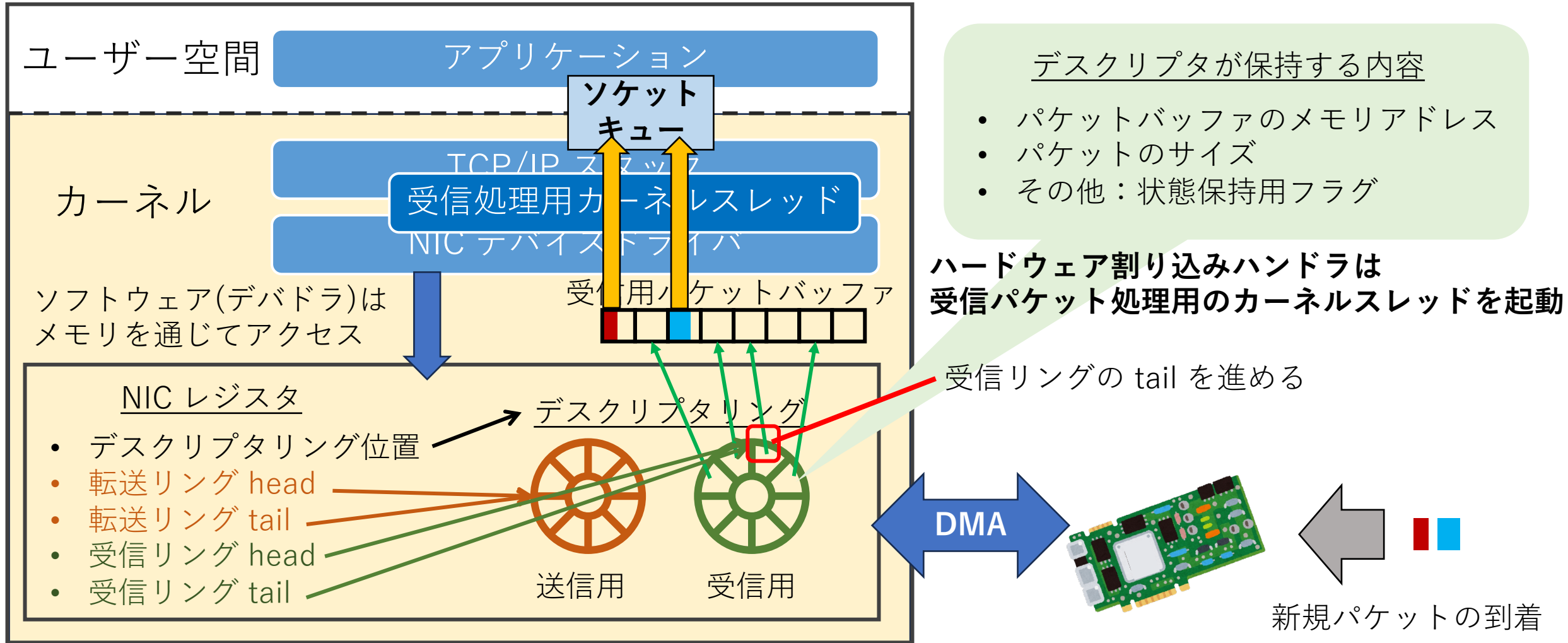
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

受信リングの tail を進める

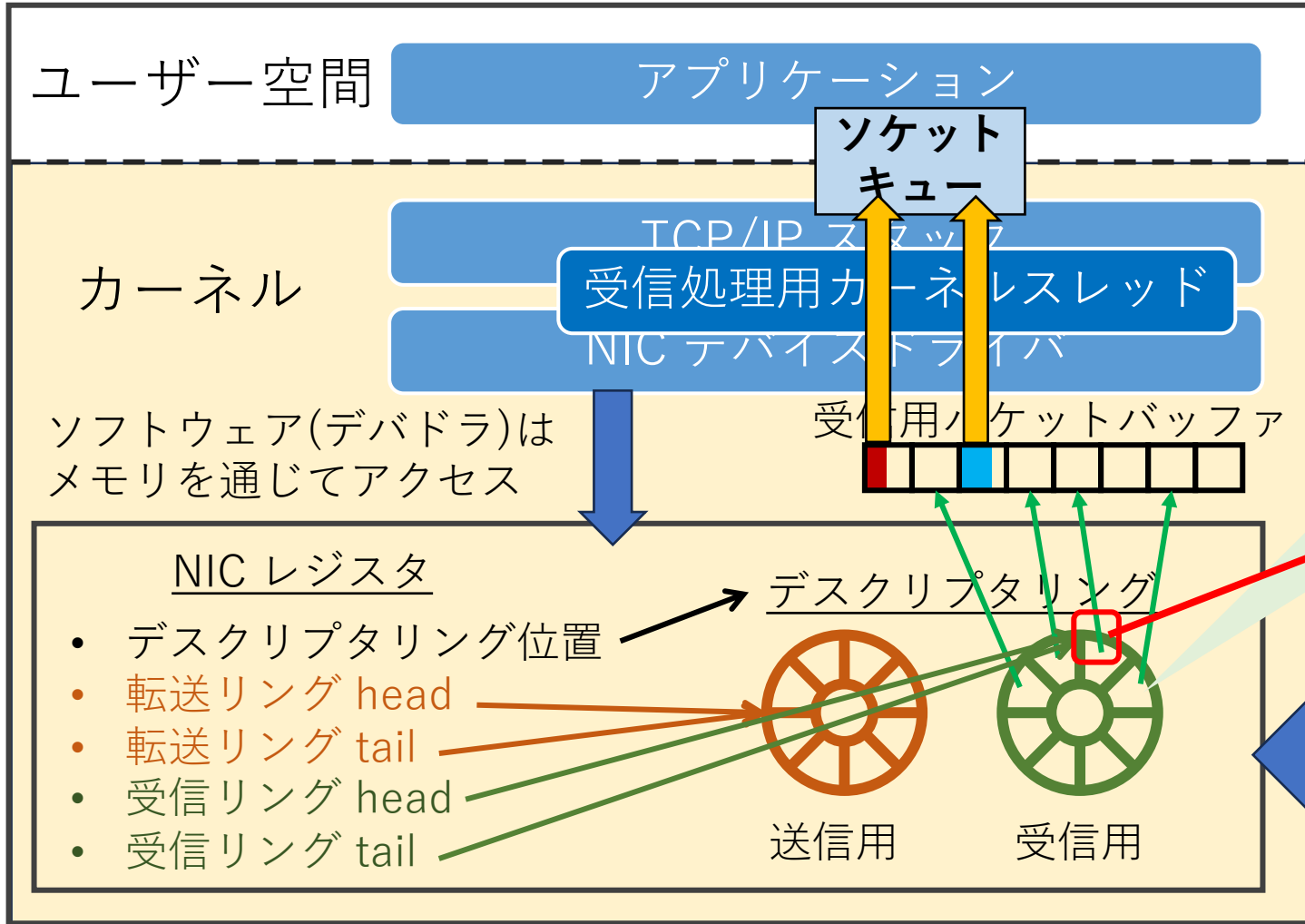


新規パケットの到着

(ある程度) 一般的な受信処理の流れ



(ある程度) 一般的な受信処理の流れ

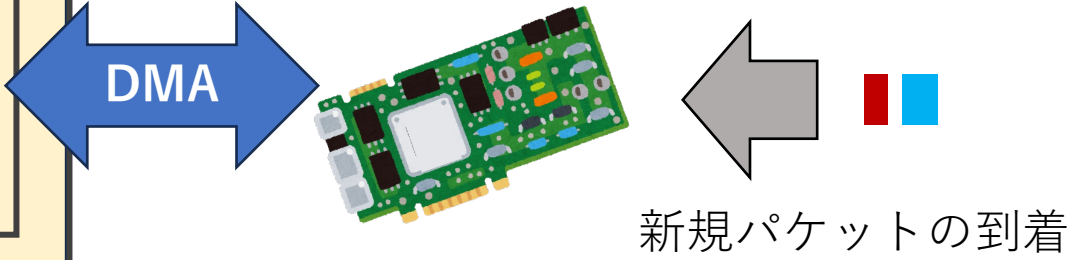


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

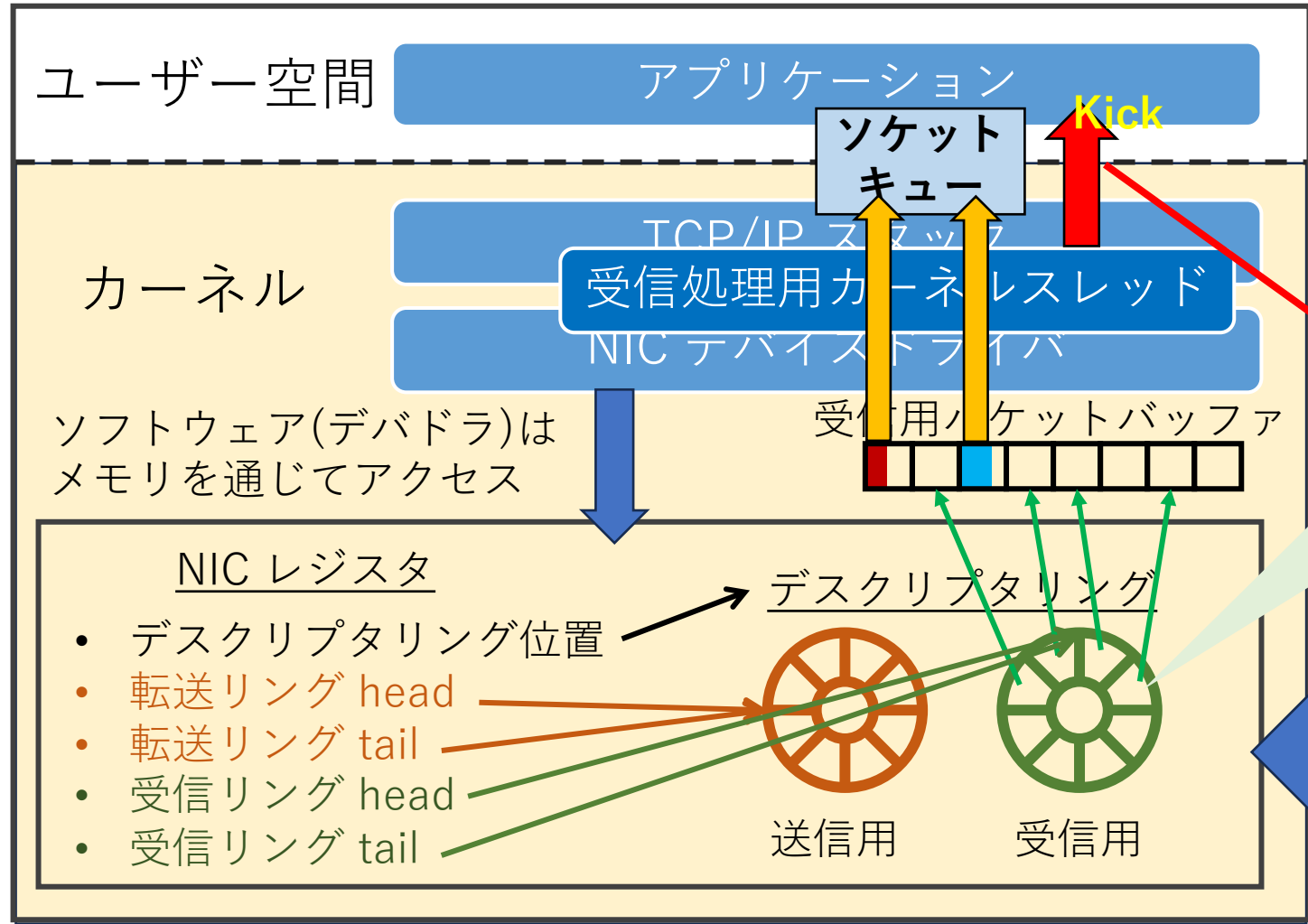
ハードウェア割り込みハンドラは受信パケット処理用のカーネルスレッドを起動

パケットは受信リングの tail より先へは head を進めないで新規パケット受信のためには tail の更新が必要



(ある程度) 一般的な受信処理の流れ

read(), select(), poll() 等でブロックされていれば、ブロックが解除される

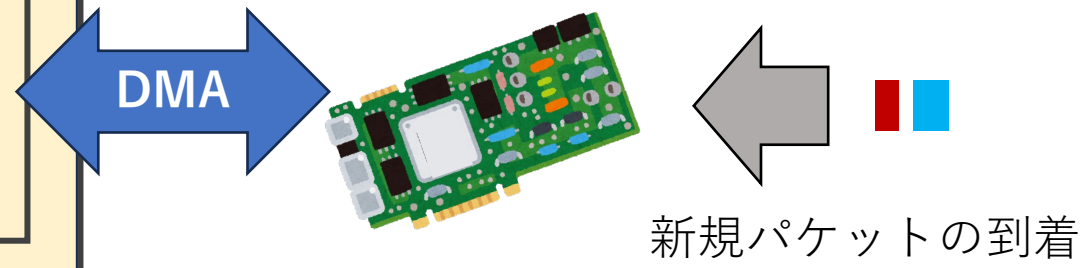


デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

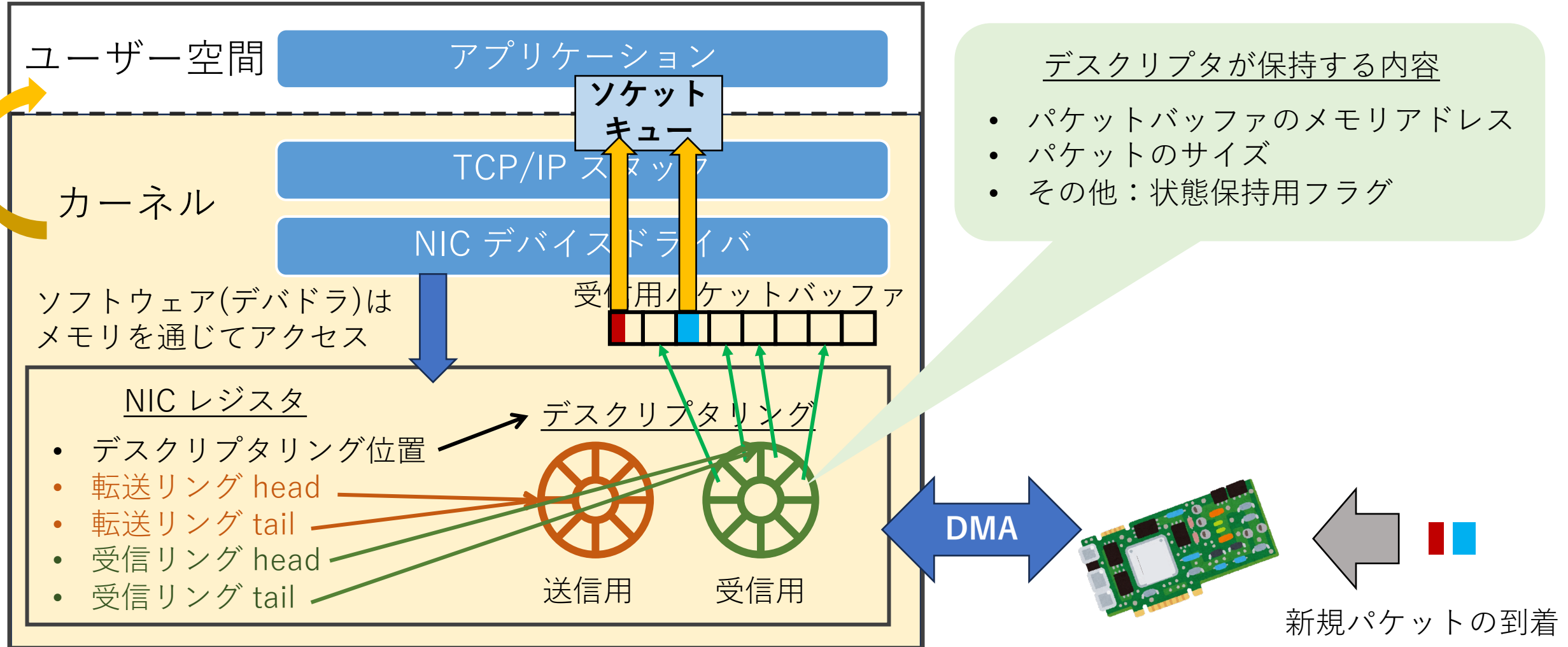
受信パケット処理用のカーネルスレッドがアプリケーションスレッドへ通知を送る

- NIC レジスタ
- デスクリプタリング位置
 - 転送リング head
 - 転送リング tail
 - 受信リング head
 - 受信リング tail



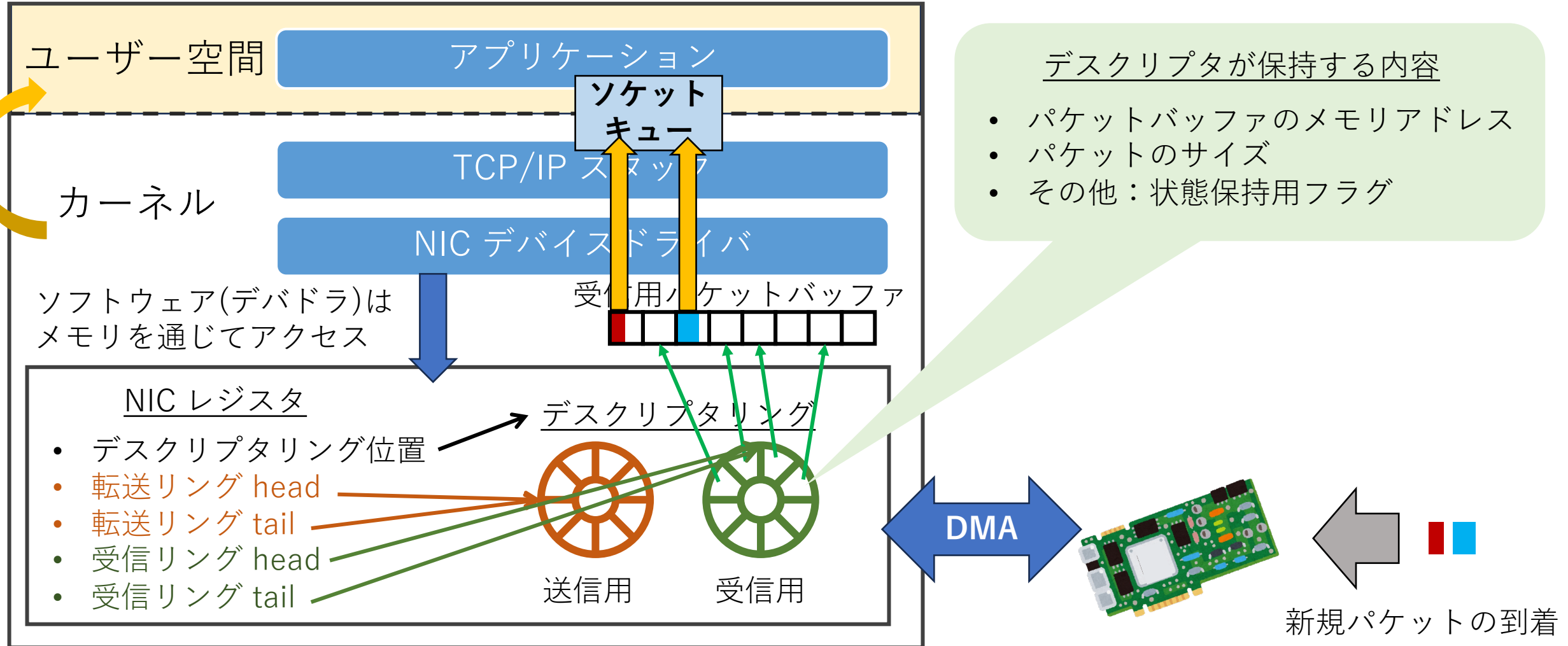
(ある程度) 一般的な受信処理の流れ

read(), select(), poll() 等でブロックされていれば、ブロックが解除される

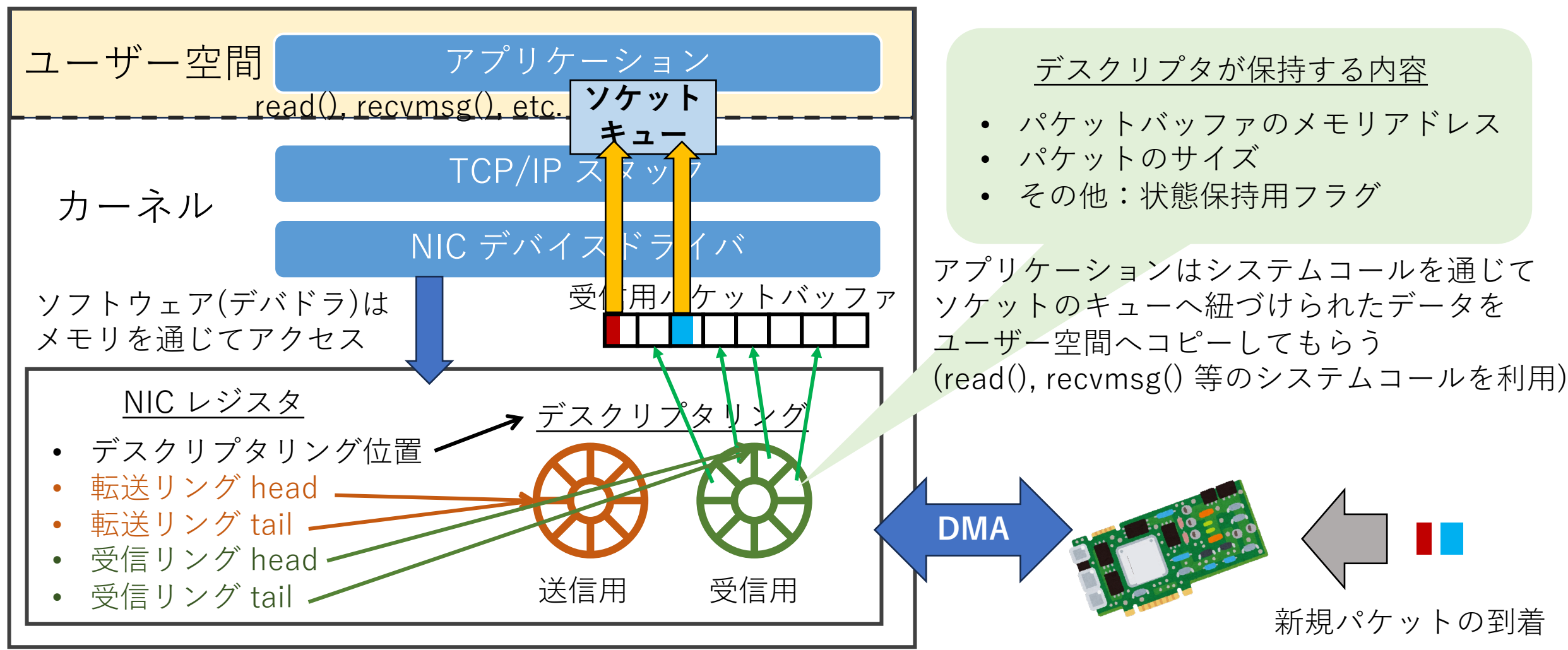


(ある程度) 一般的な受信処理の流れ

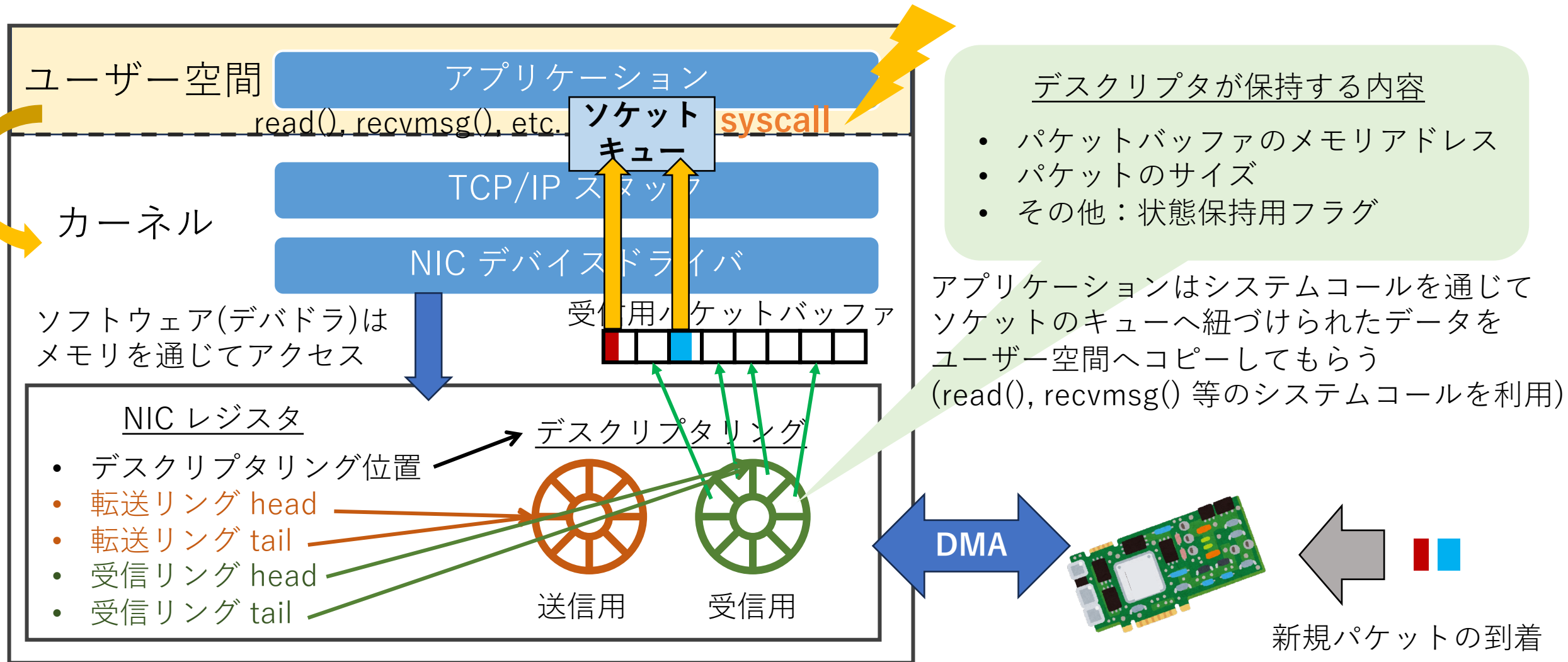
read(), select(), poll() 等でブロックされていれば、ブロックが解除される



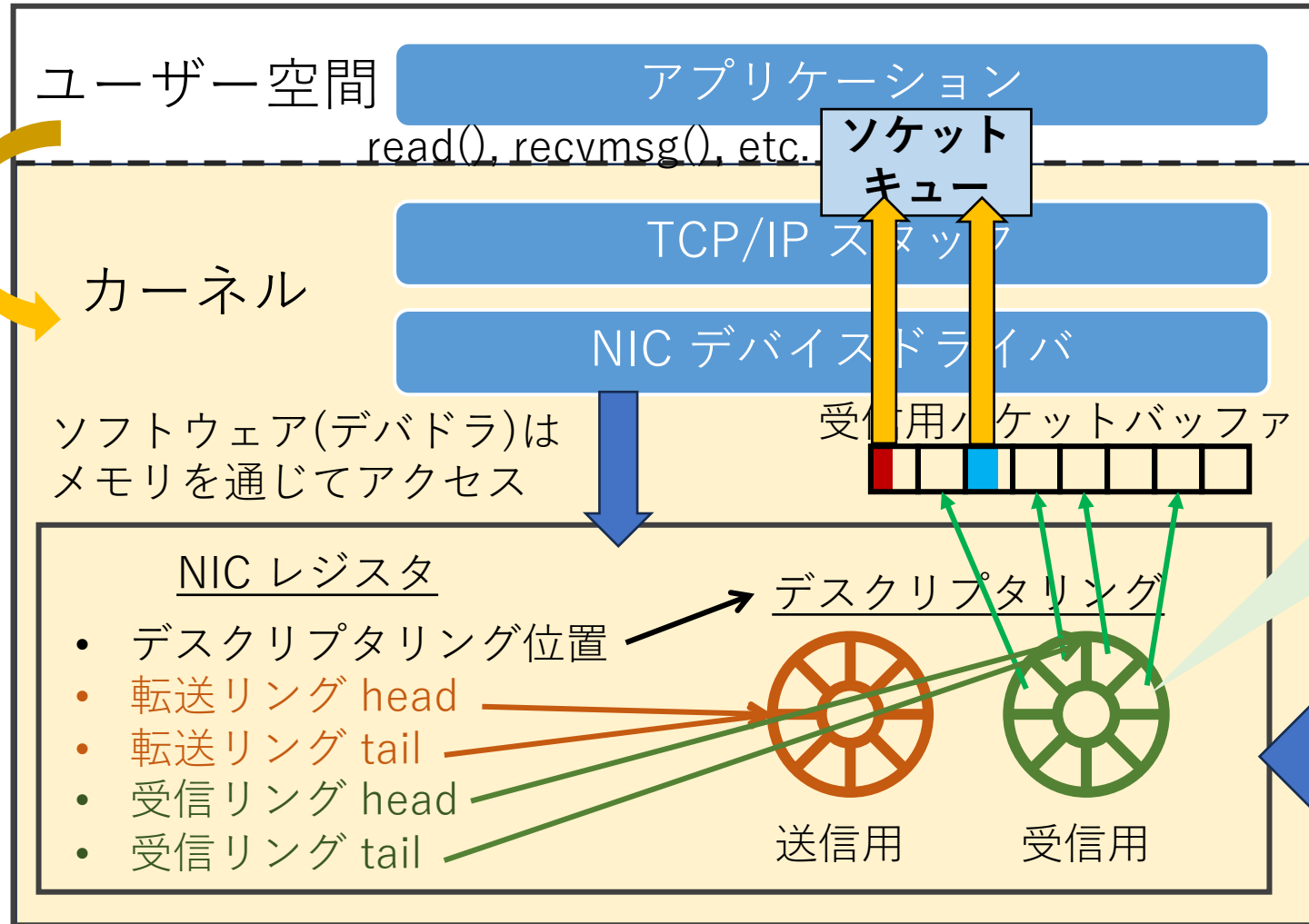
(ある程度) 一般的な受信処理の流れ



(ある程度) 一般的な受信処理の流れ



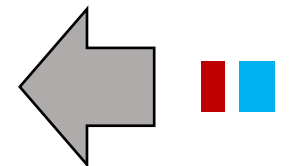
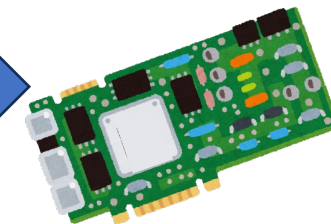
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

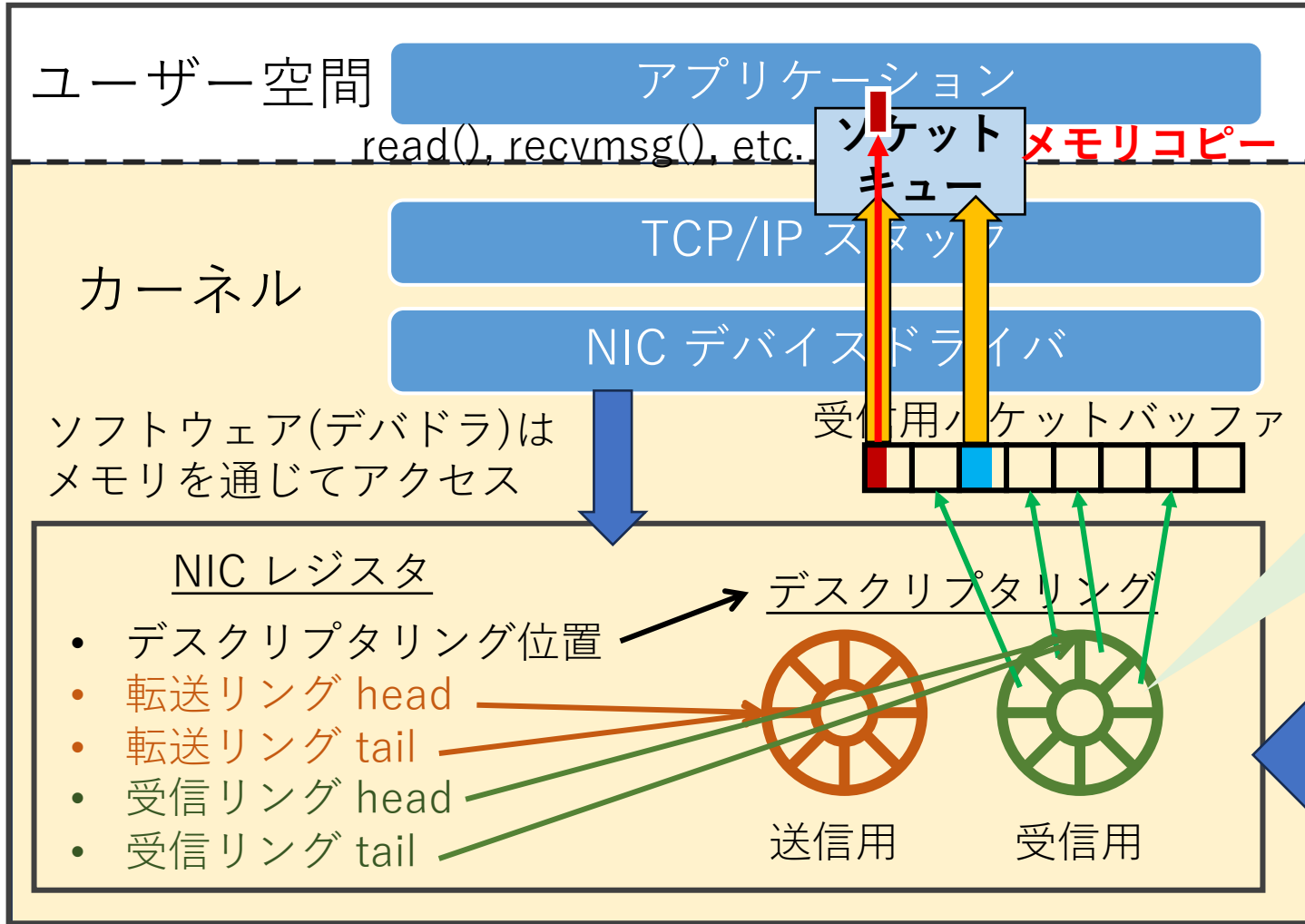
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)



新規パケットの到着

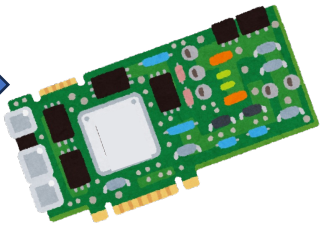
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

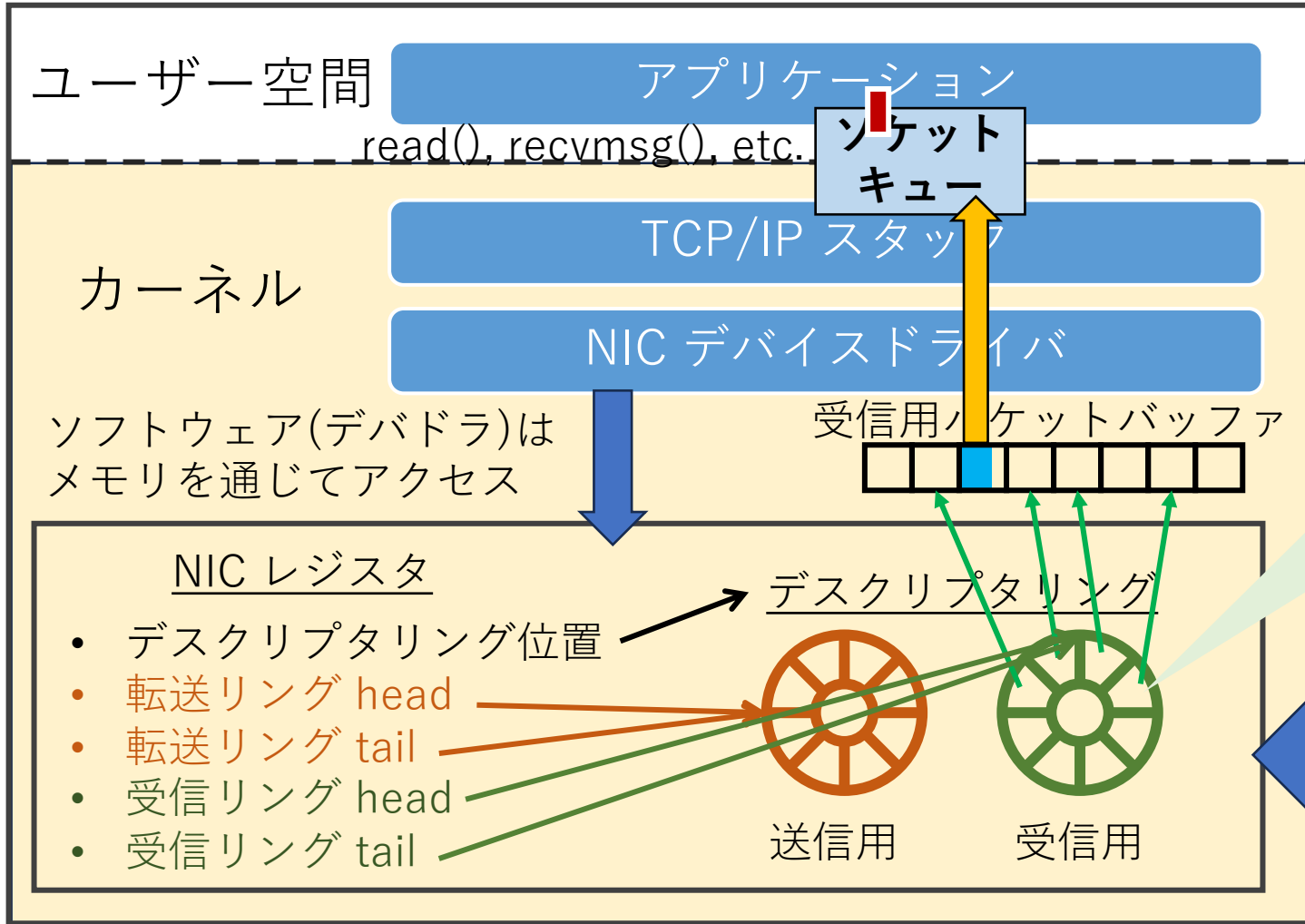
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他: 状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (read(), recvmsg() 等のシステムコールを利用)



← 新規パケットの到着

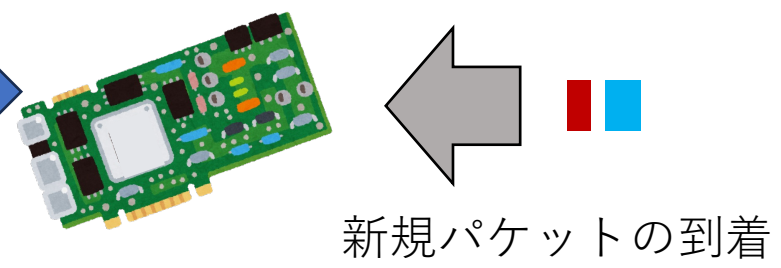
(ある程度) 一般的な受信処理の流れ



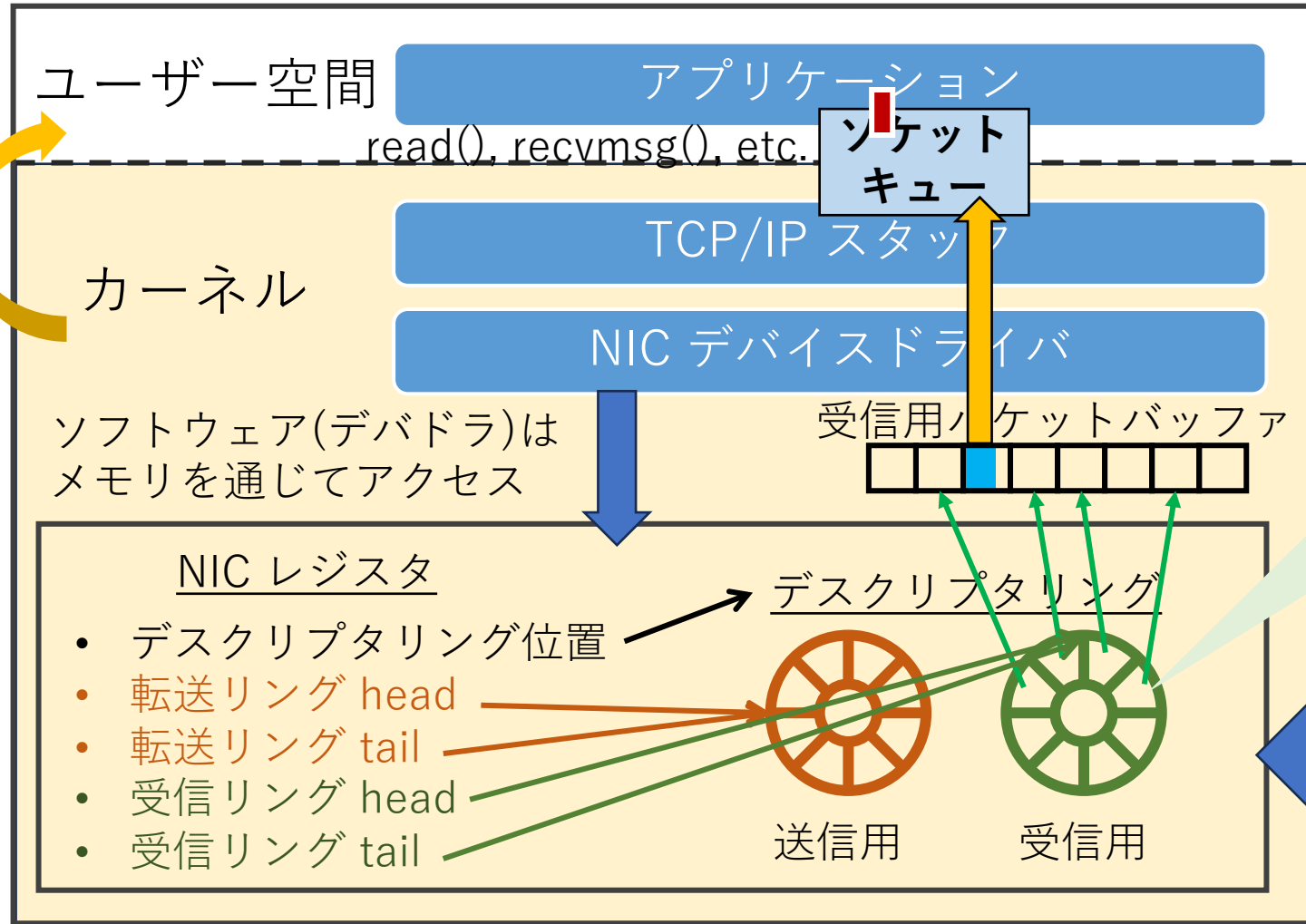
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (read(), recvmsg() 等のシステムコールを利用)



(ある程度) 一般的な受信処理の流れ



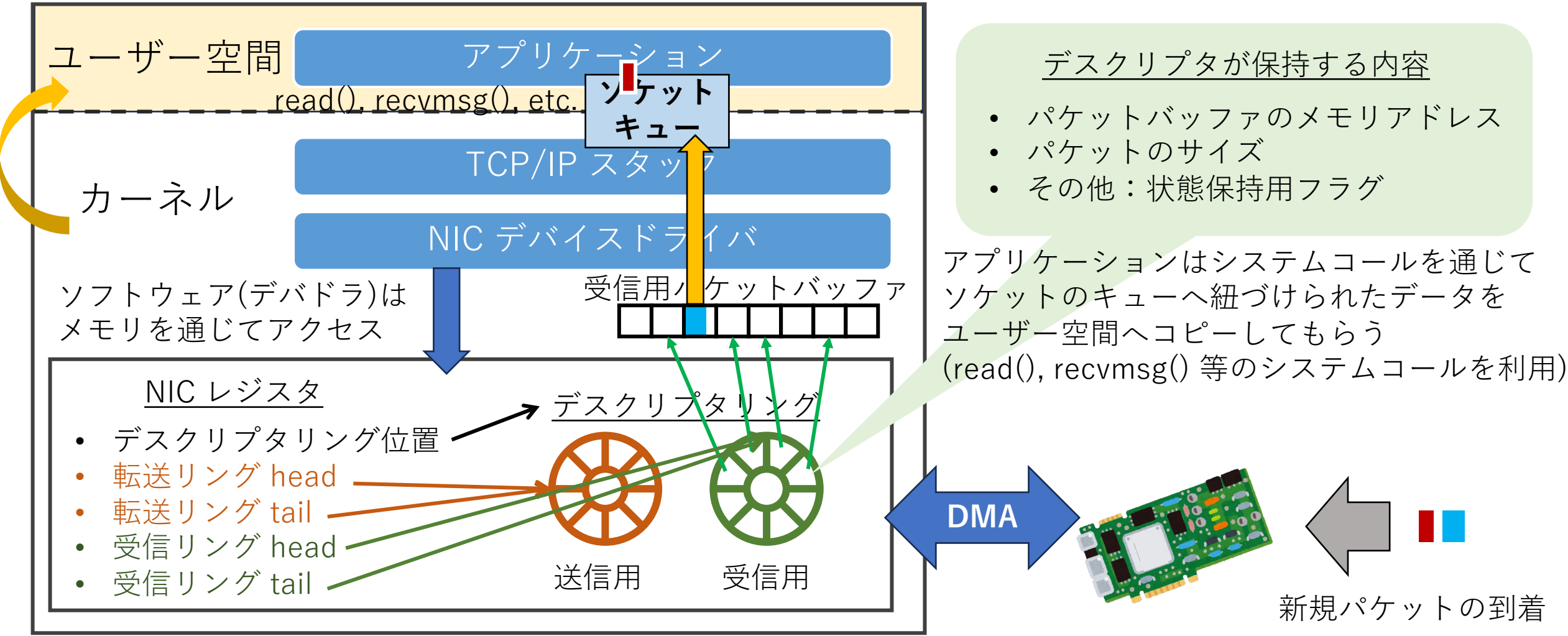
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

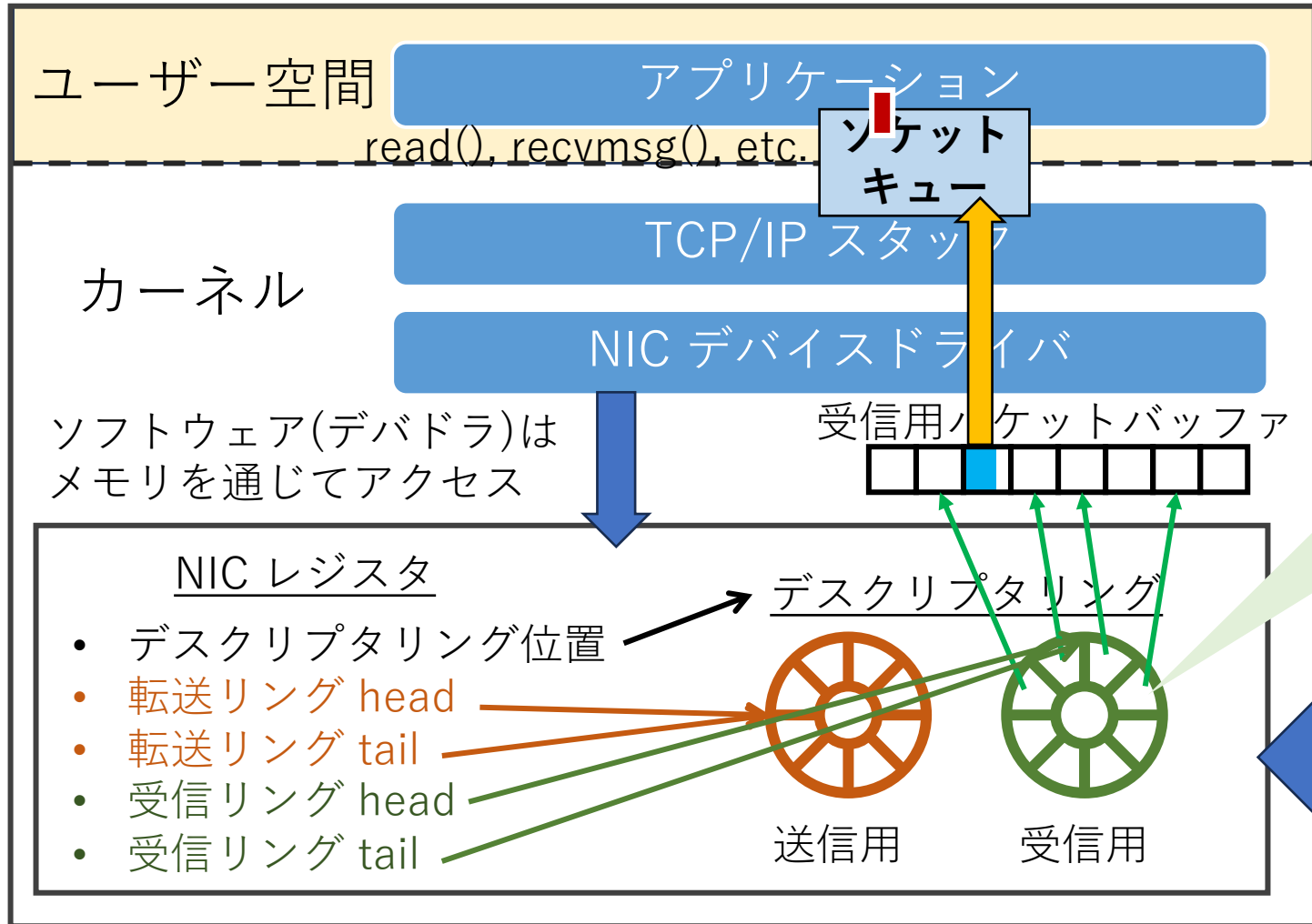
アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)

新規パケットの到着

(ある程度) 一般的な受信処理の流れ



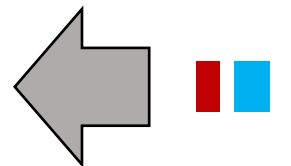
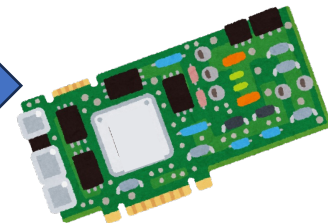
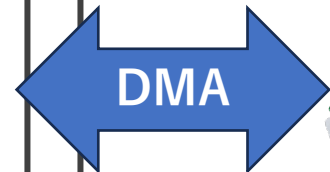
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

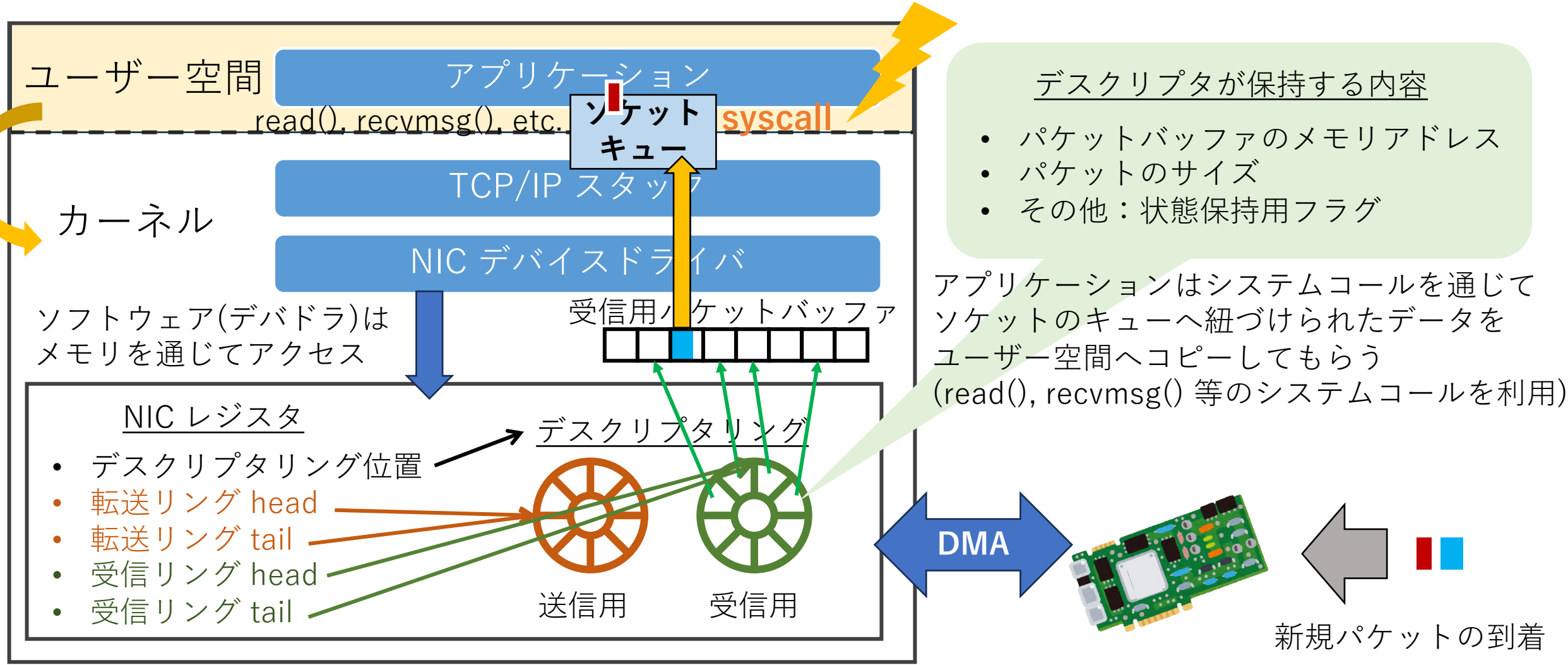
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)

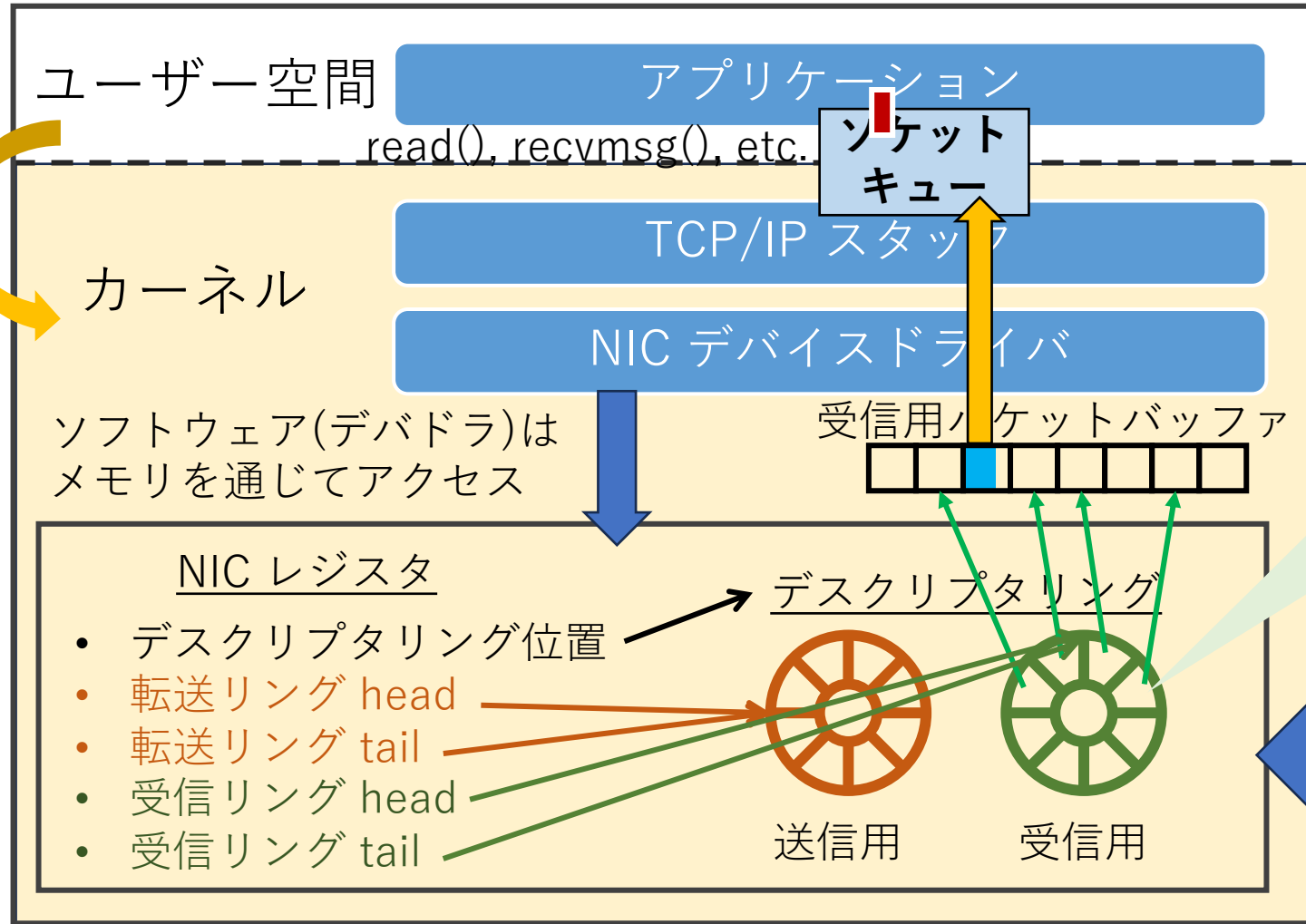


新規パケットの到着

(ある程度) 一般的な受信処理の流れ



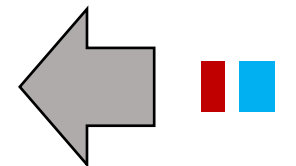
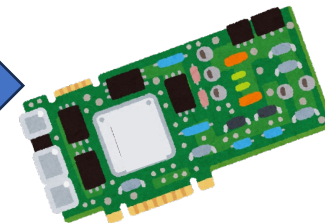
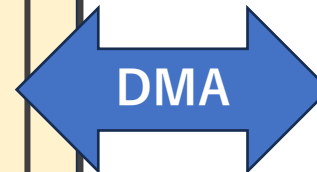
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

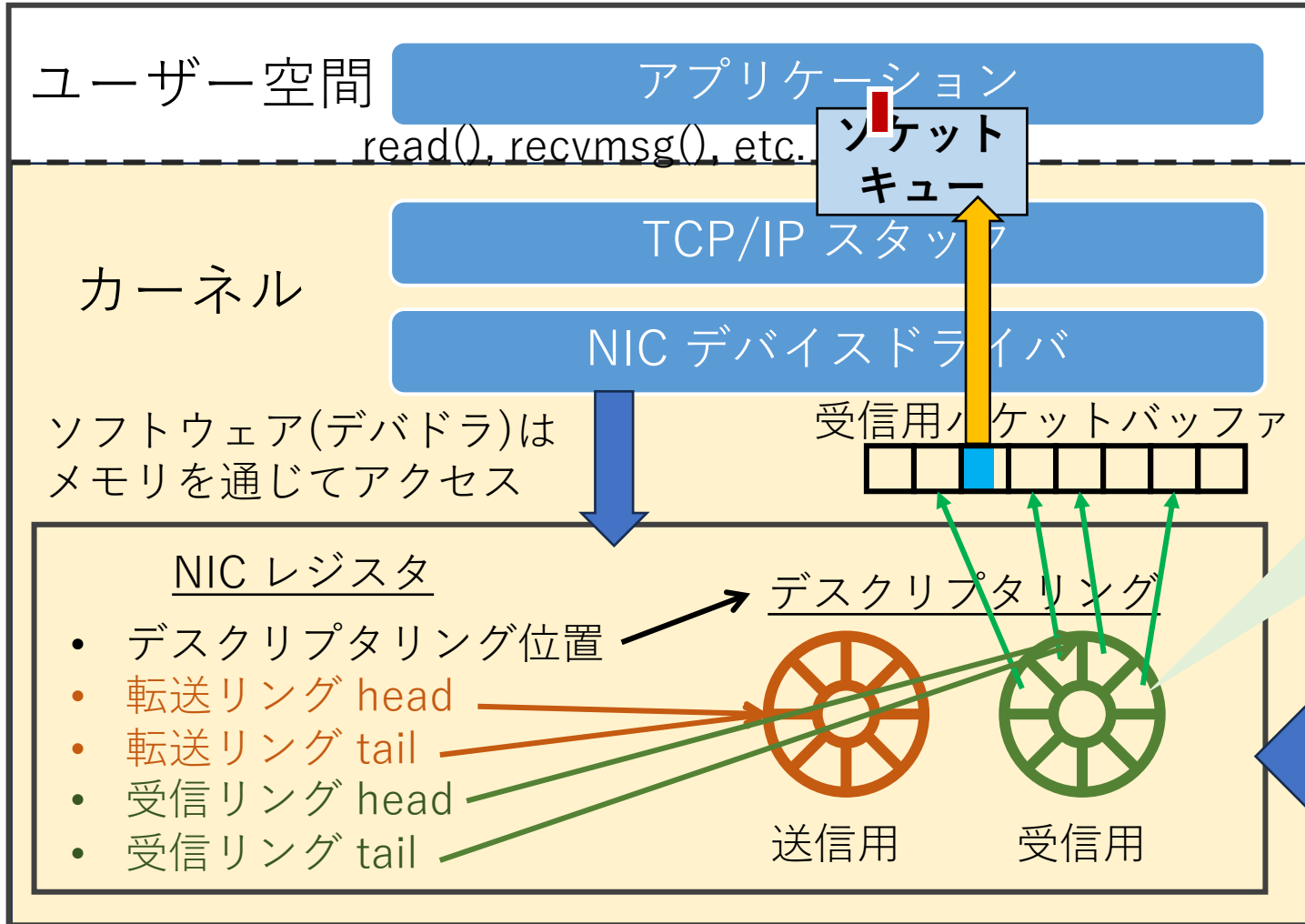
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)



新規パケットの到着

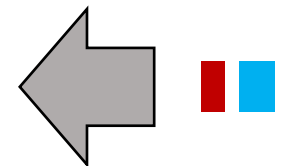
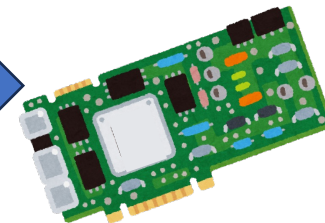
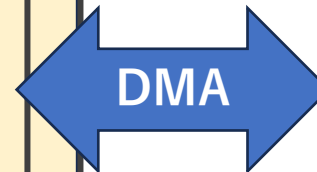
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

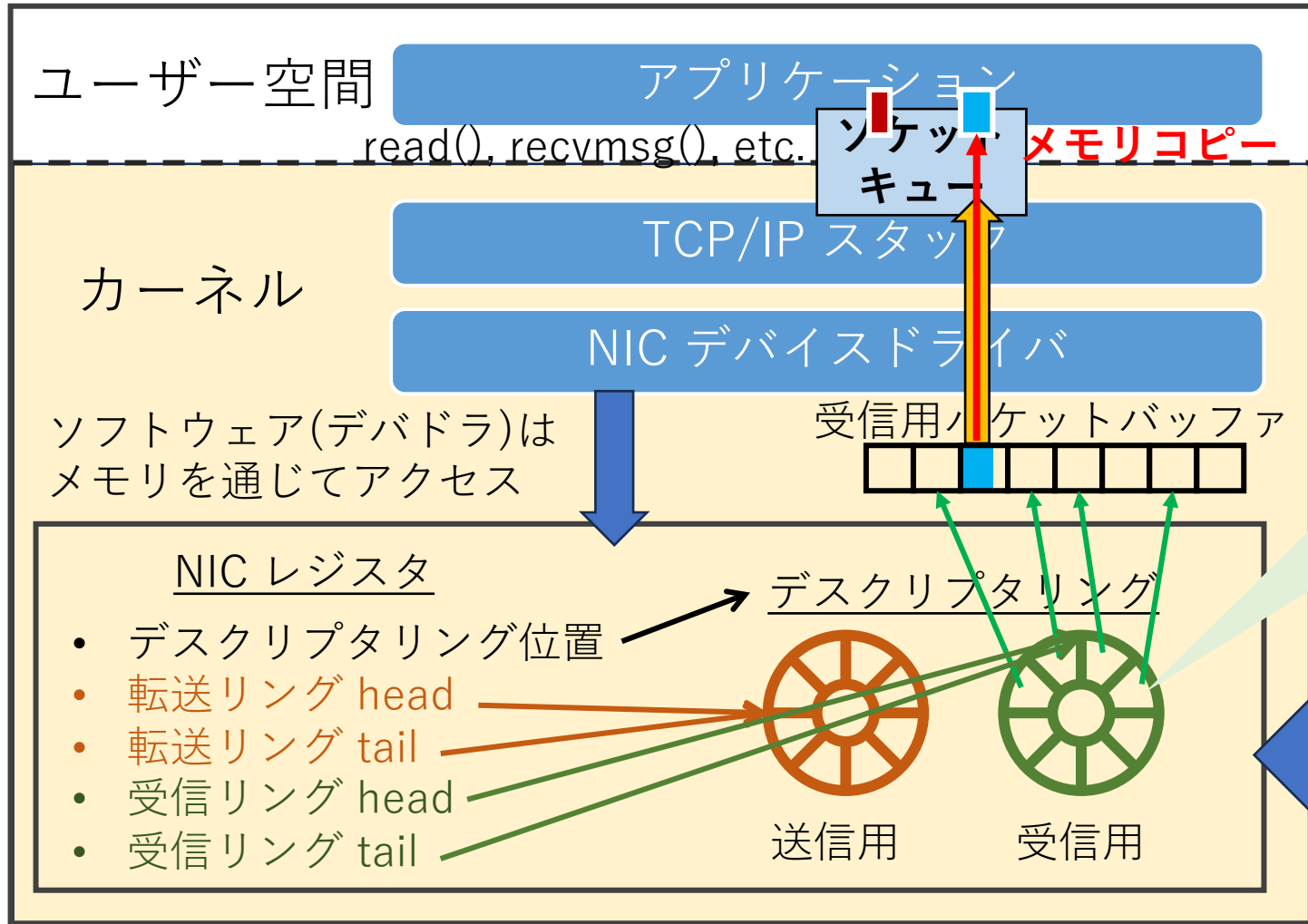
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)



新規パケットの到着

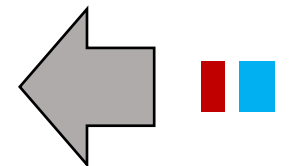
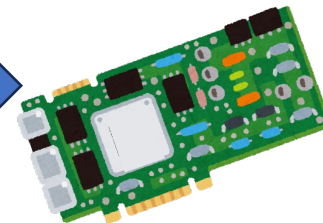
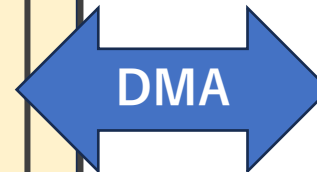
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

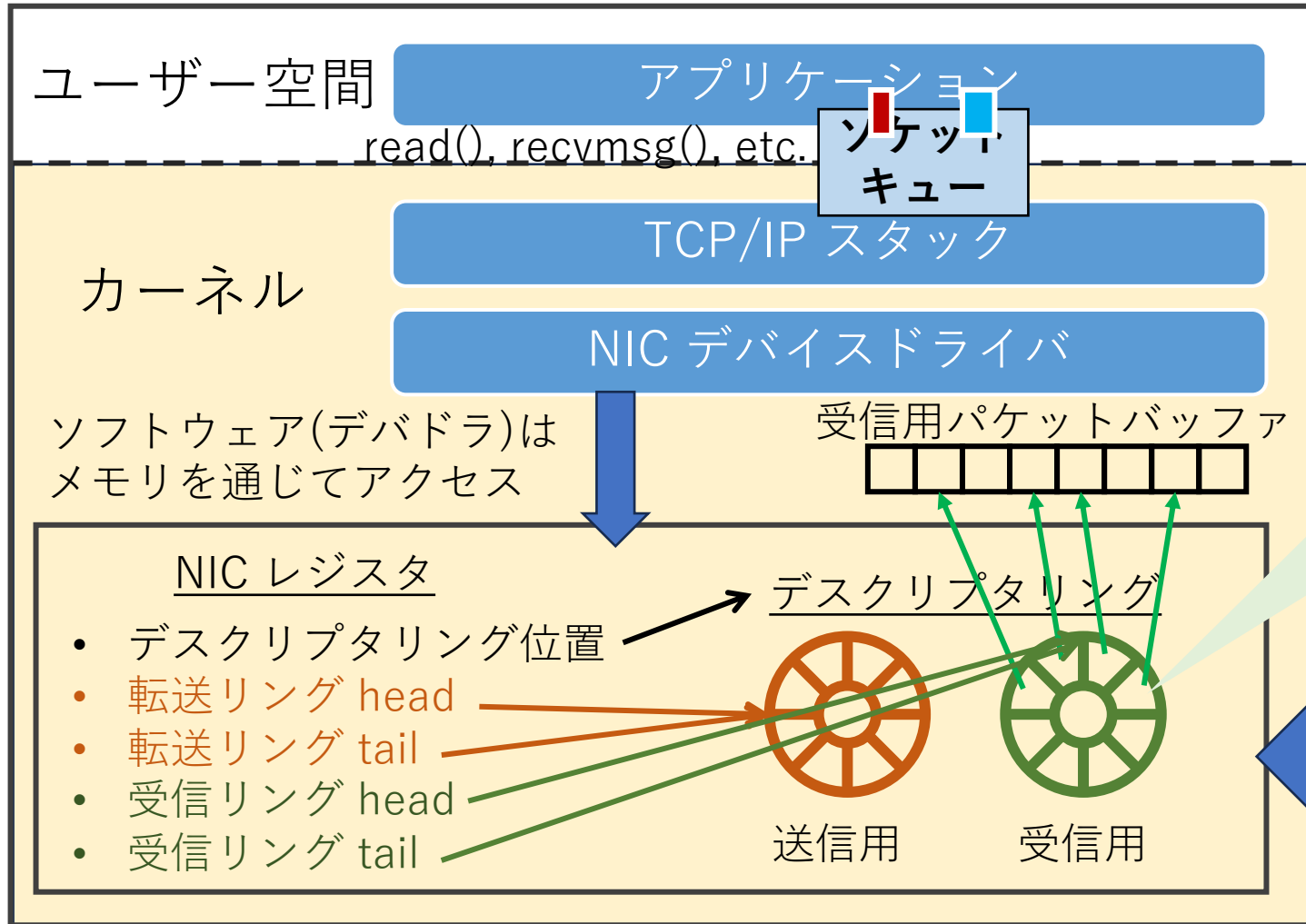
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)



新規パケットの到着

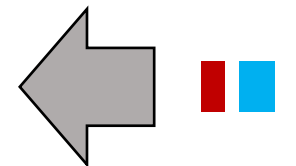
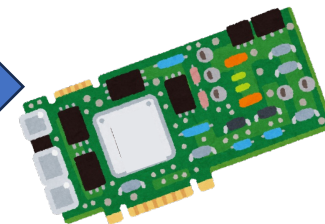
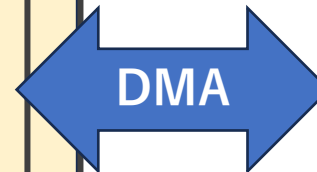
(ある程度) 一般的な受信処理の流れ



デスクリプタが保持する内容

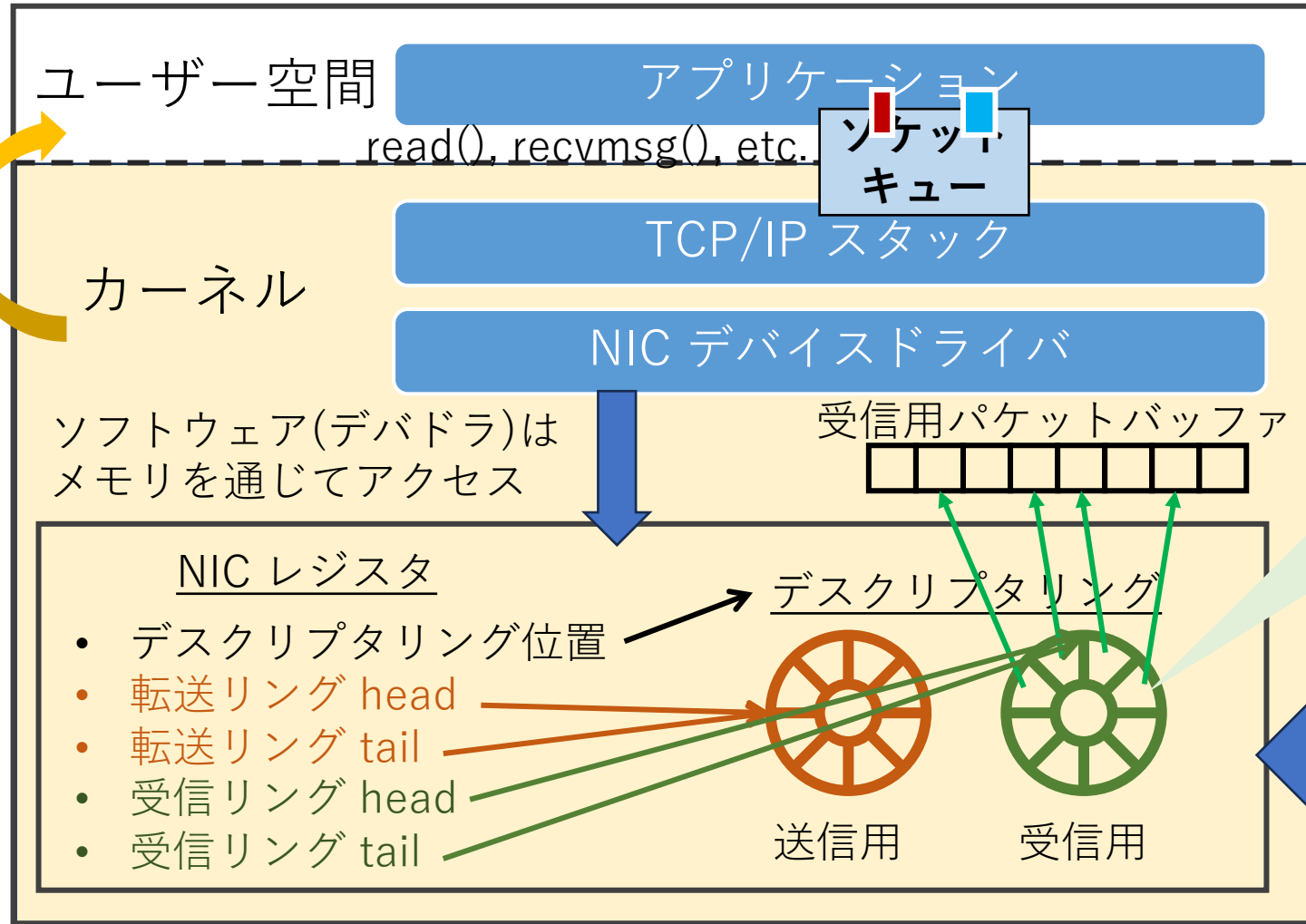
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (`read()`, `recvmsg()` 等のシステムコールを利用)



新規パケットの到着

(ある程度) 一般的な受信処理の流れ

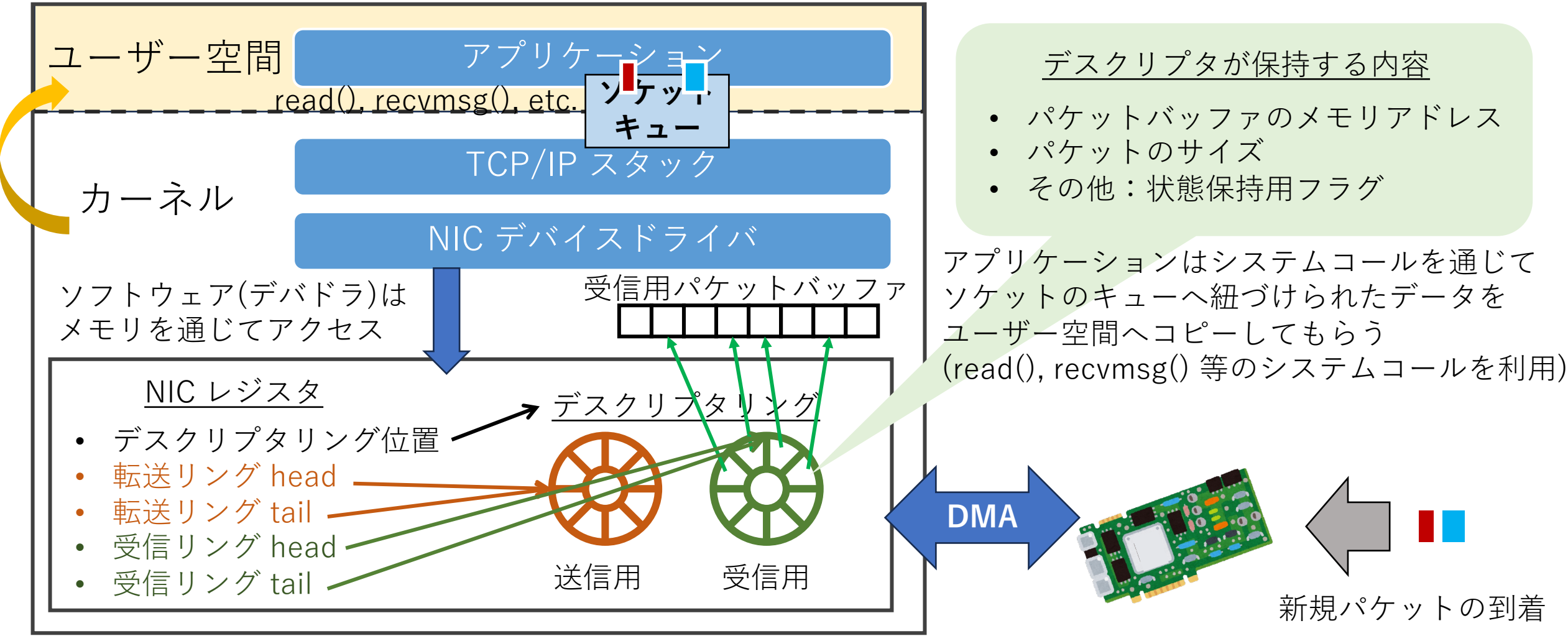


デスクリプタが保持する内容

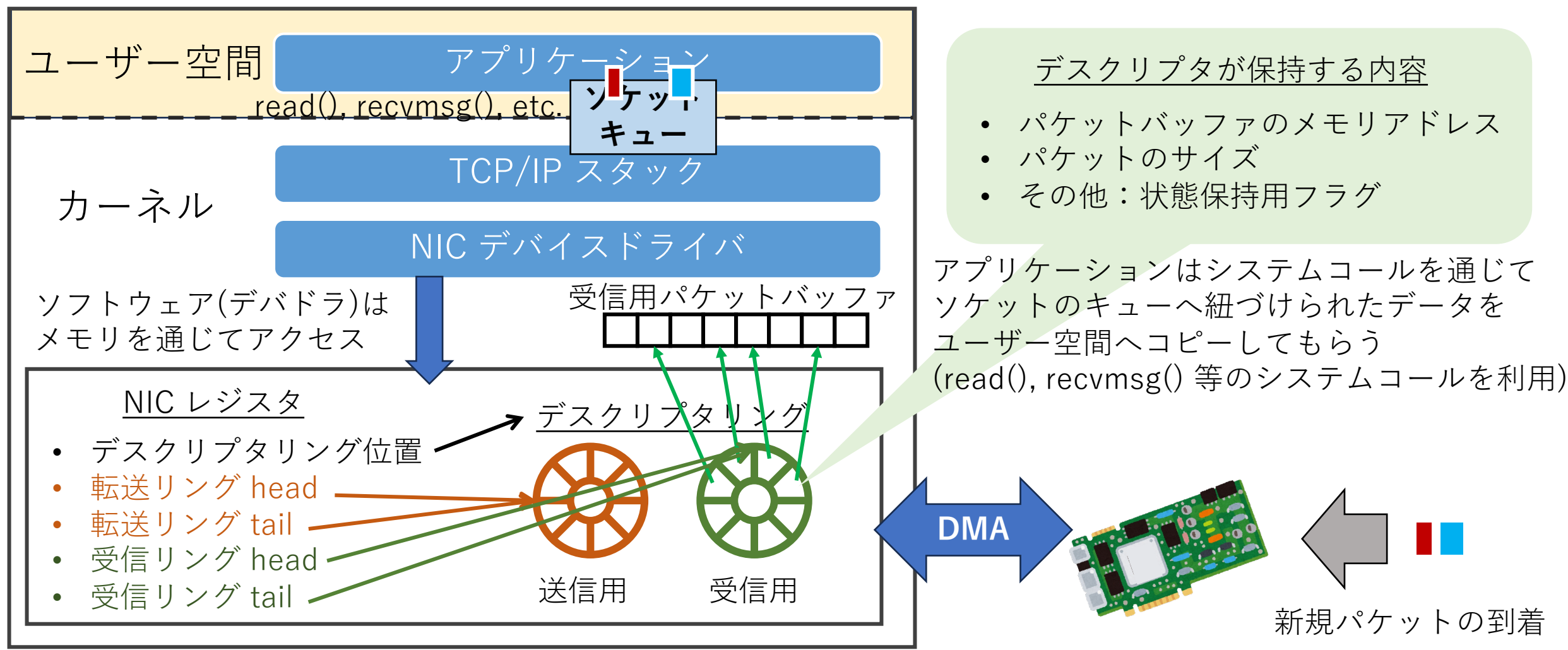
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてソケットのキューへ紐づけられたデータをユーザー空間へコピーしてもらう (read(), recvmsg() 等のシステムコールを利用)

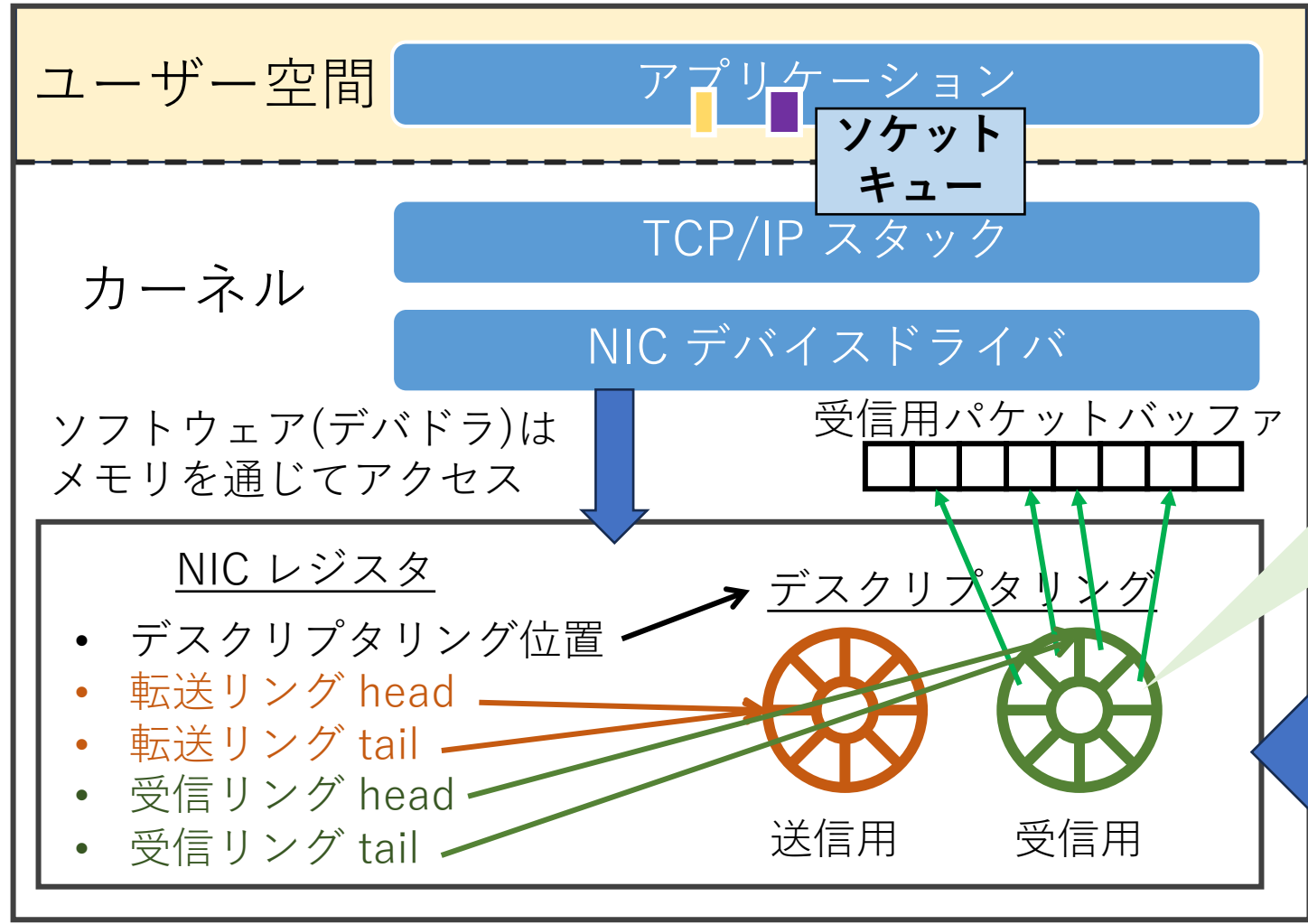
(ある程度) 一般的な受信処理の流れ



(ある程度) 一般的な受信処理の流れ



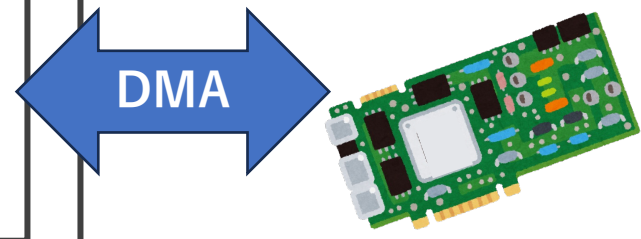
(ある程度) 一般的な送信処理の流れ



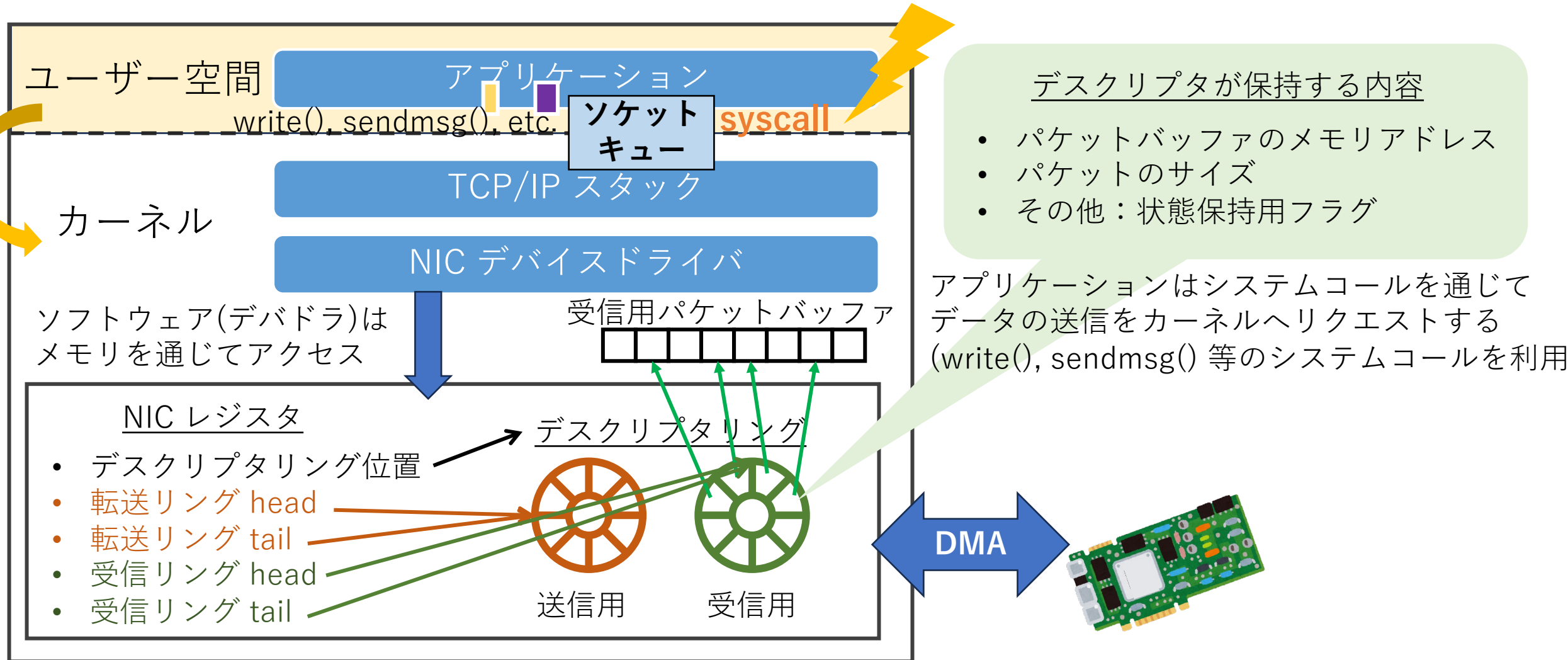
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

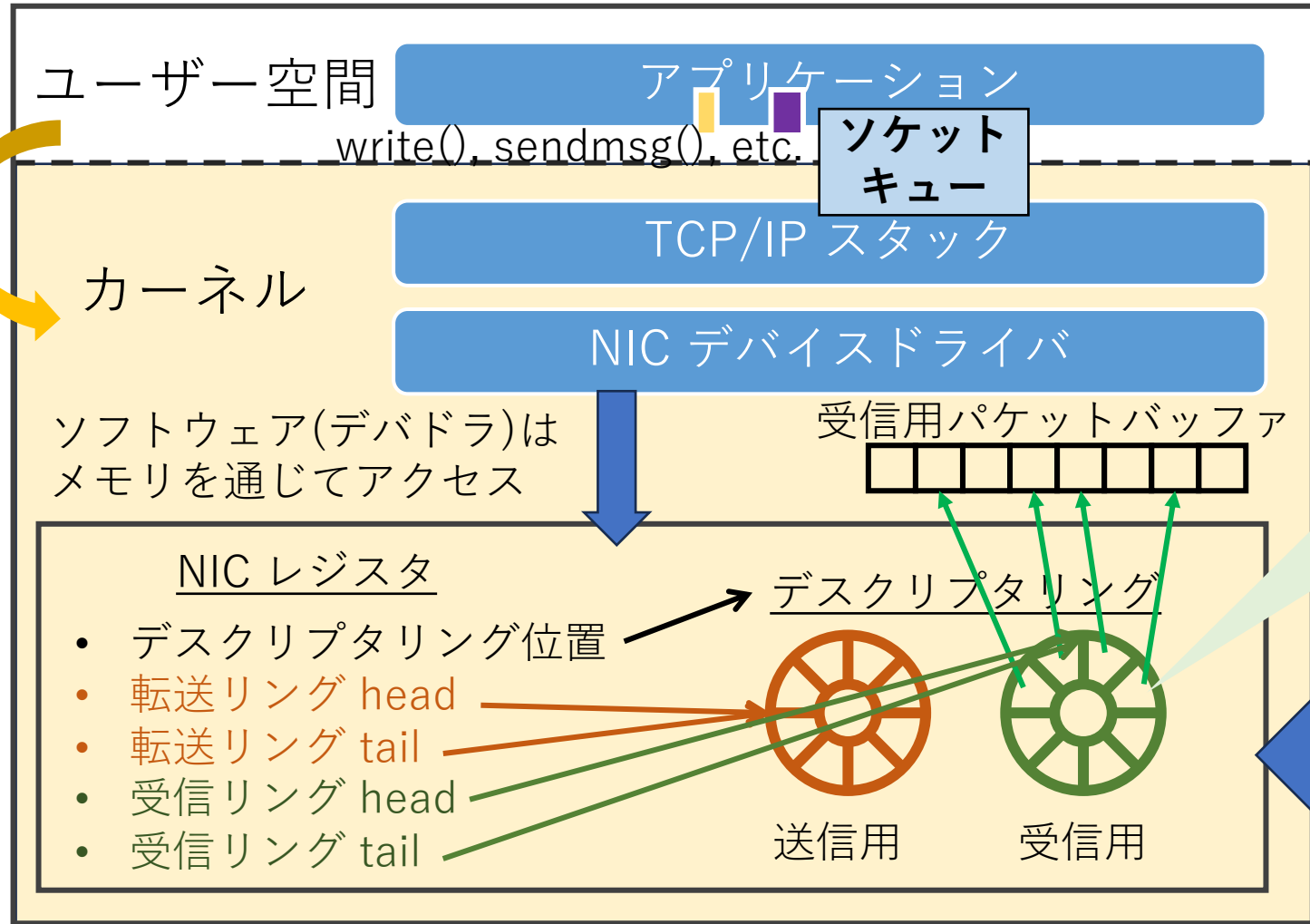
アプリケーションはシステムコールを通じてデータの送信をカーネルへリクエストする (write(), sendmsg() 等のシステムコールを利用)



(ある程度) 一般的な送信処理の流れ



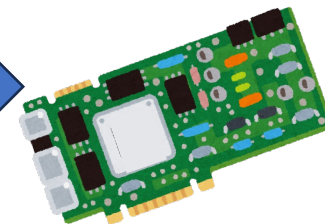
(ある程度) 一般的な送信処理の流れ



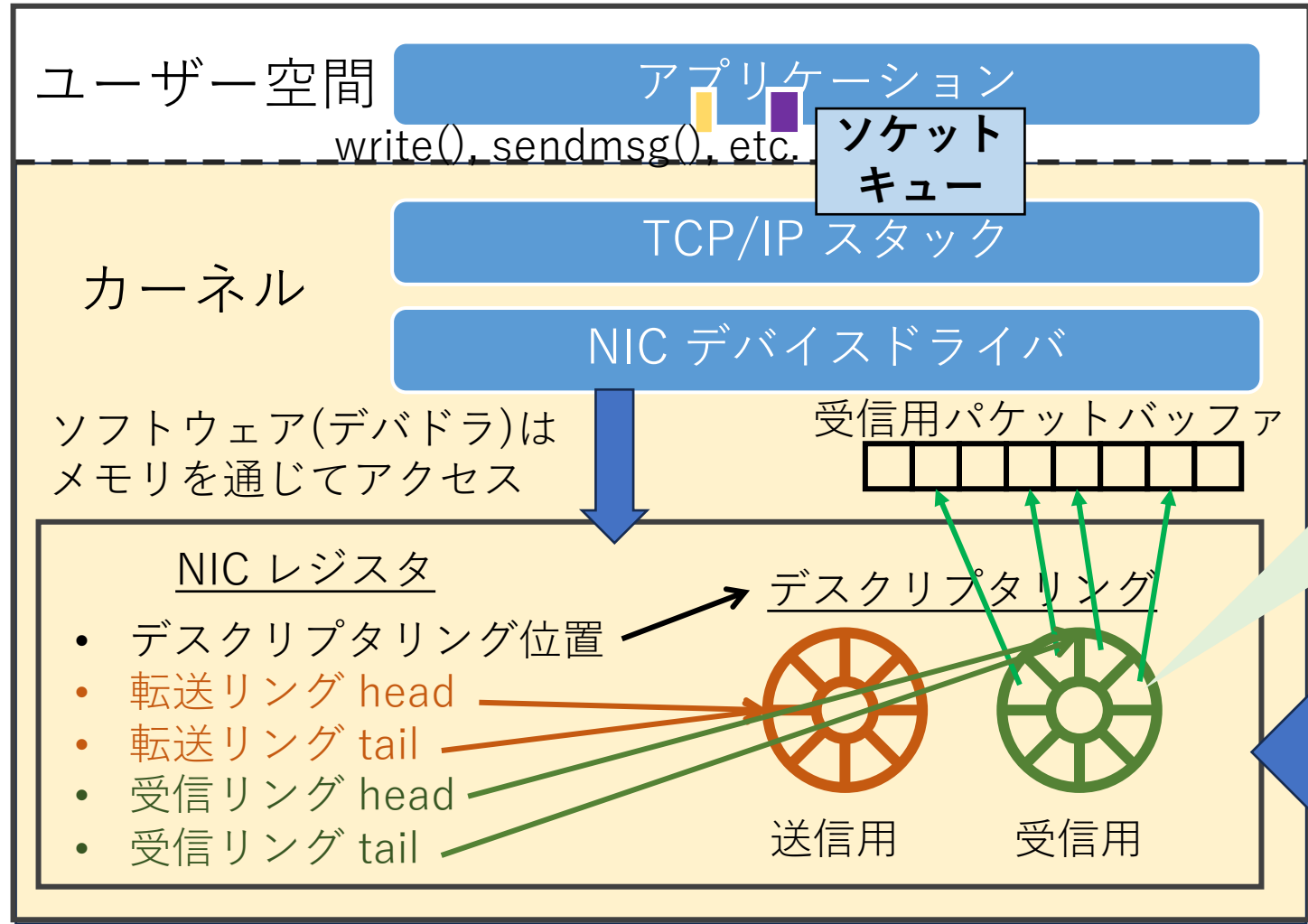
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

アプリケーションはシステムコールを通じてデータの送信をカーネルへリクエストする (`write()`, `sendmsg()` 等のシステムコールを利用)



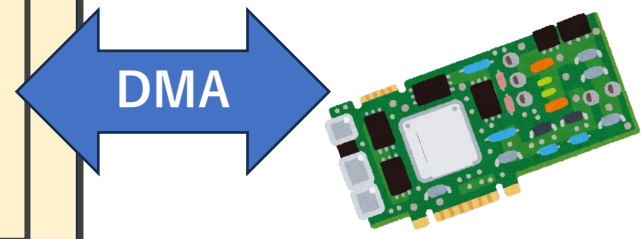
(ある程度) 一般的な送信処理の流れ



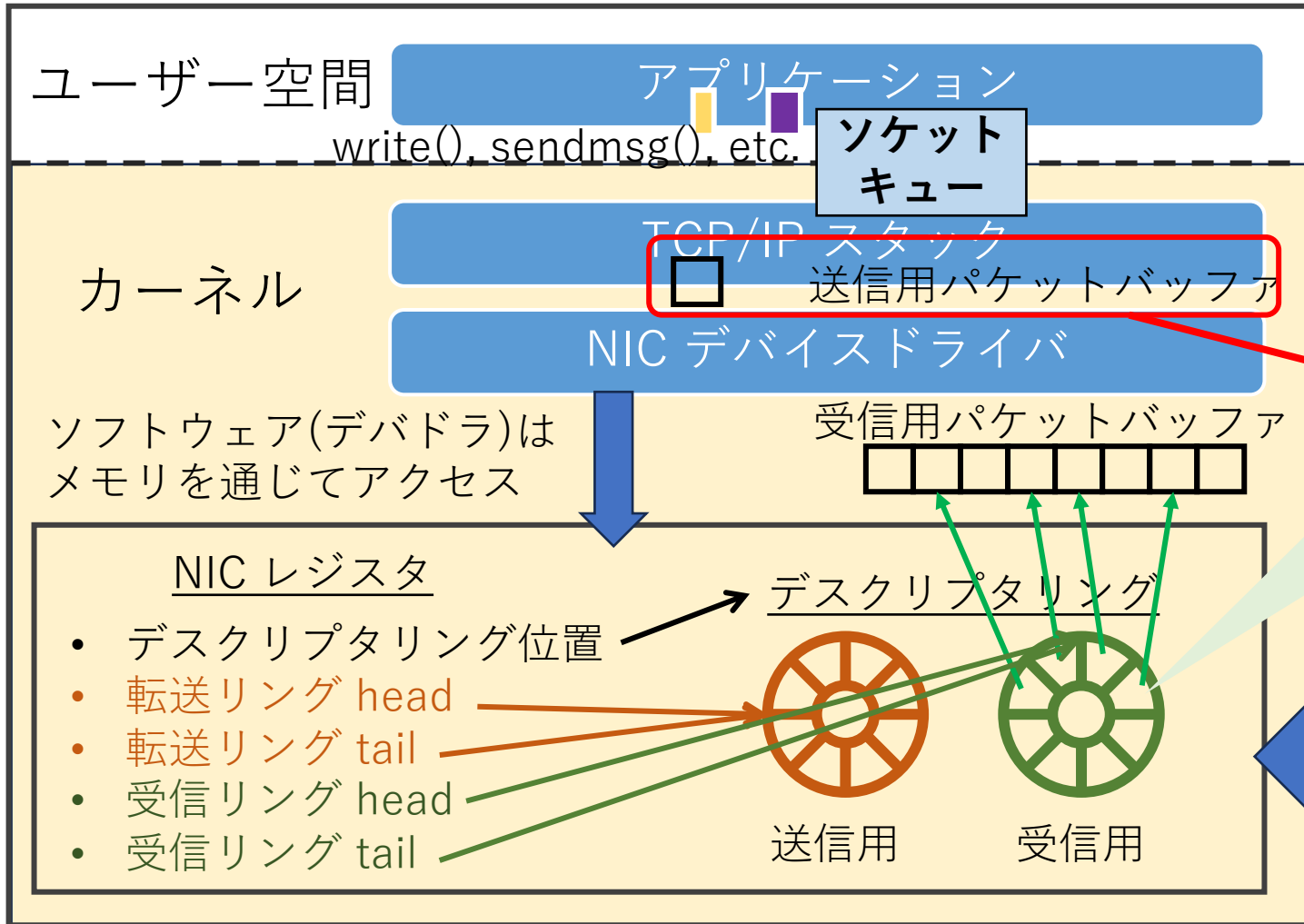
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

カーネルは送信用パッケージバッファを確保



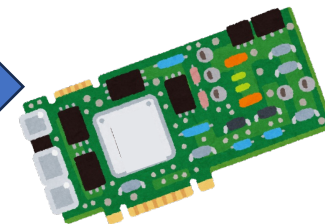
(ある程度) 一般的な送信処理の流れ



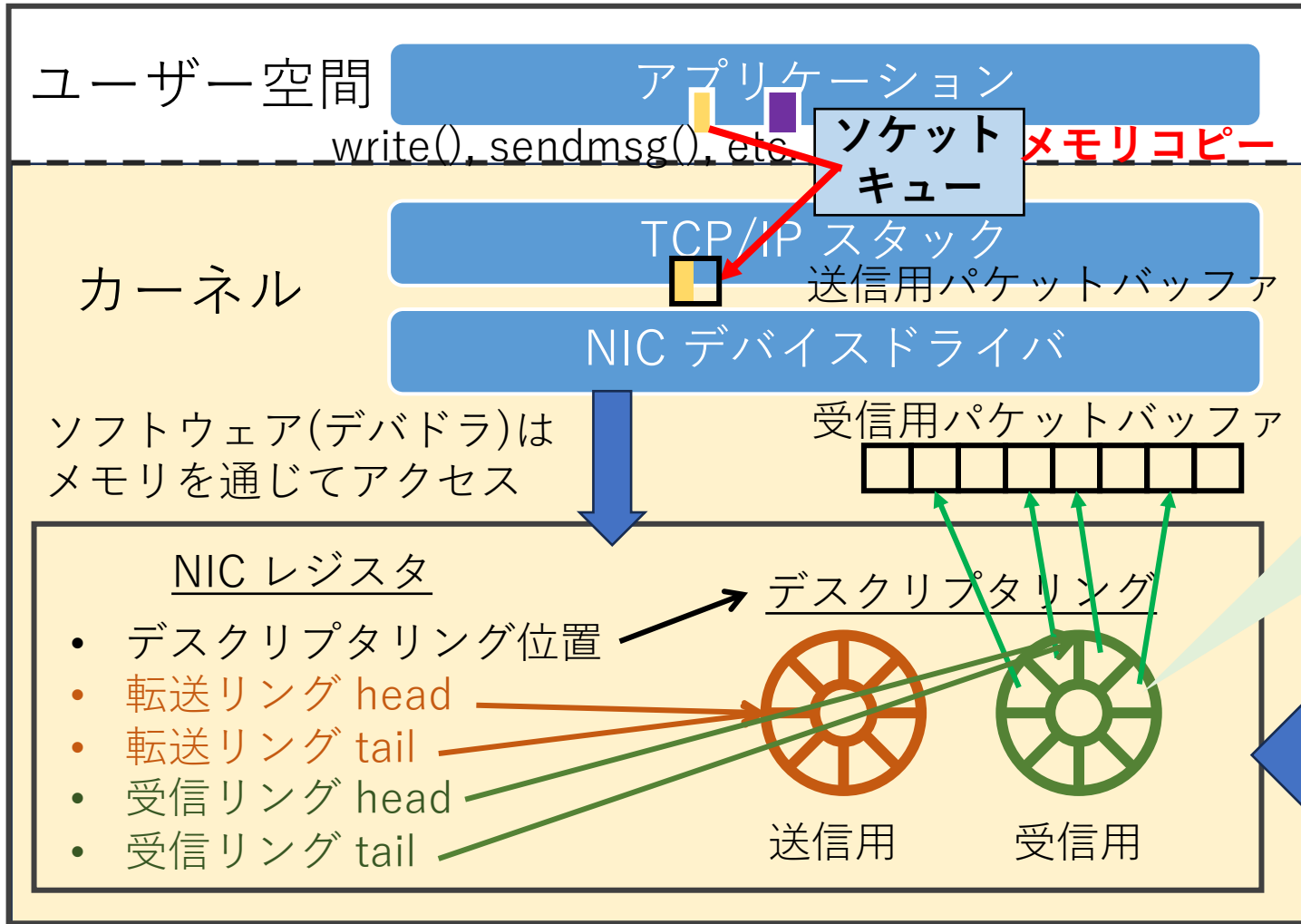
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

カーネルは送信用バッファを確保



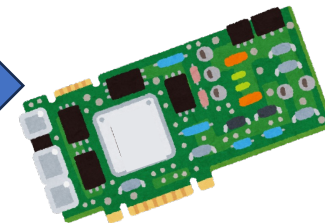
(ある程度) 一般的な送信処理の流れ



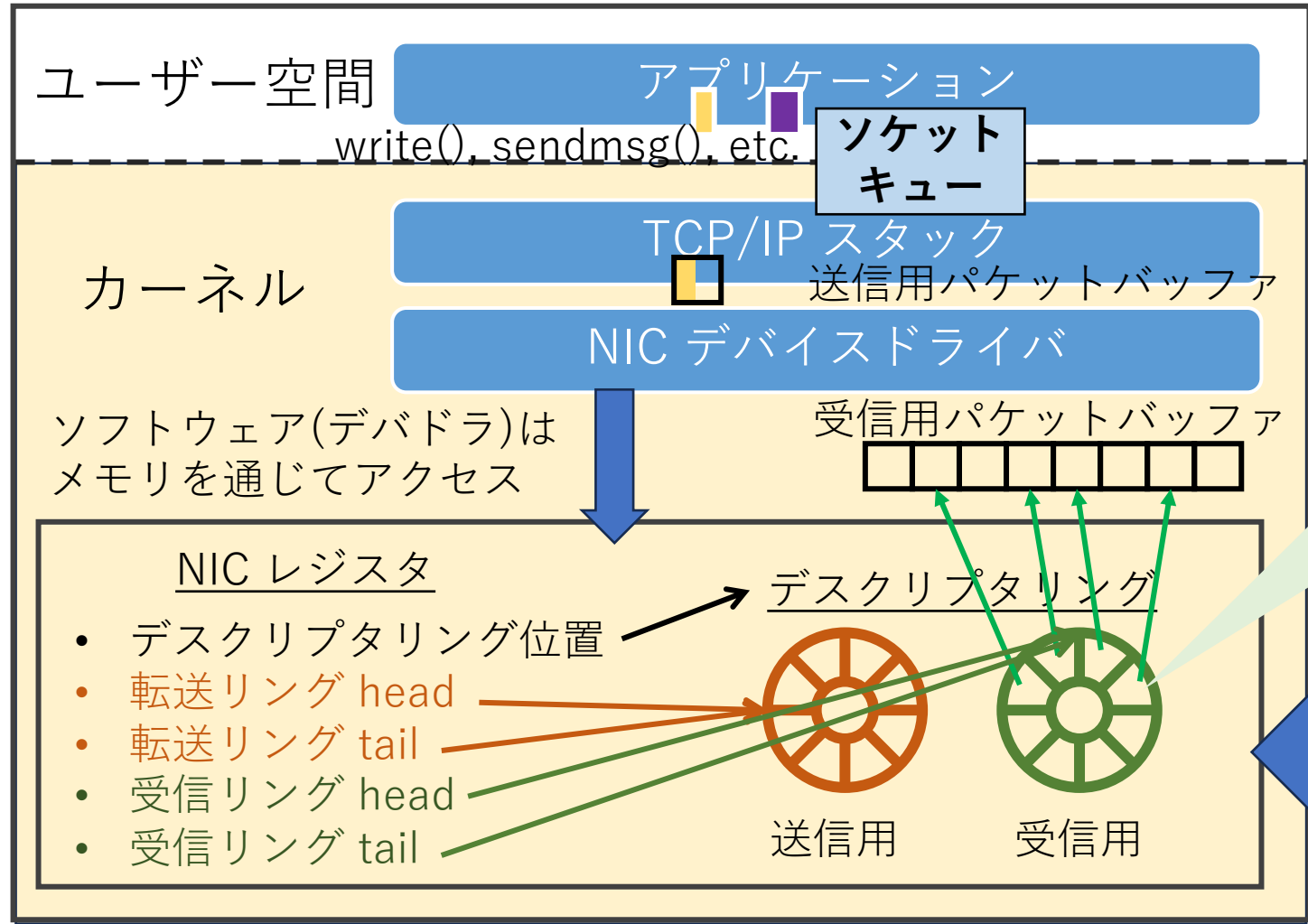
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

ユーザー空間からデータをカーネル空間で確保した送信用バッファへコピー



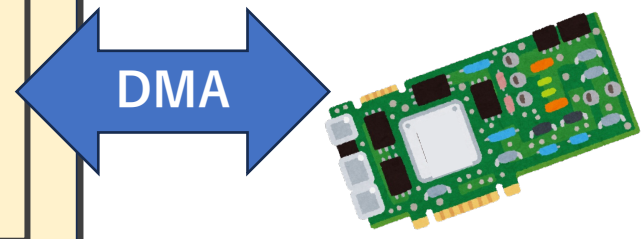
(ある程度) 一般的な送信処理の流れ



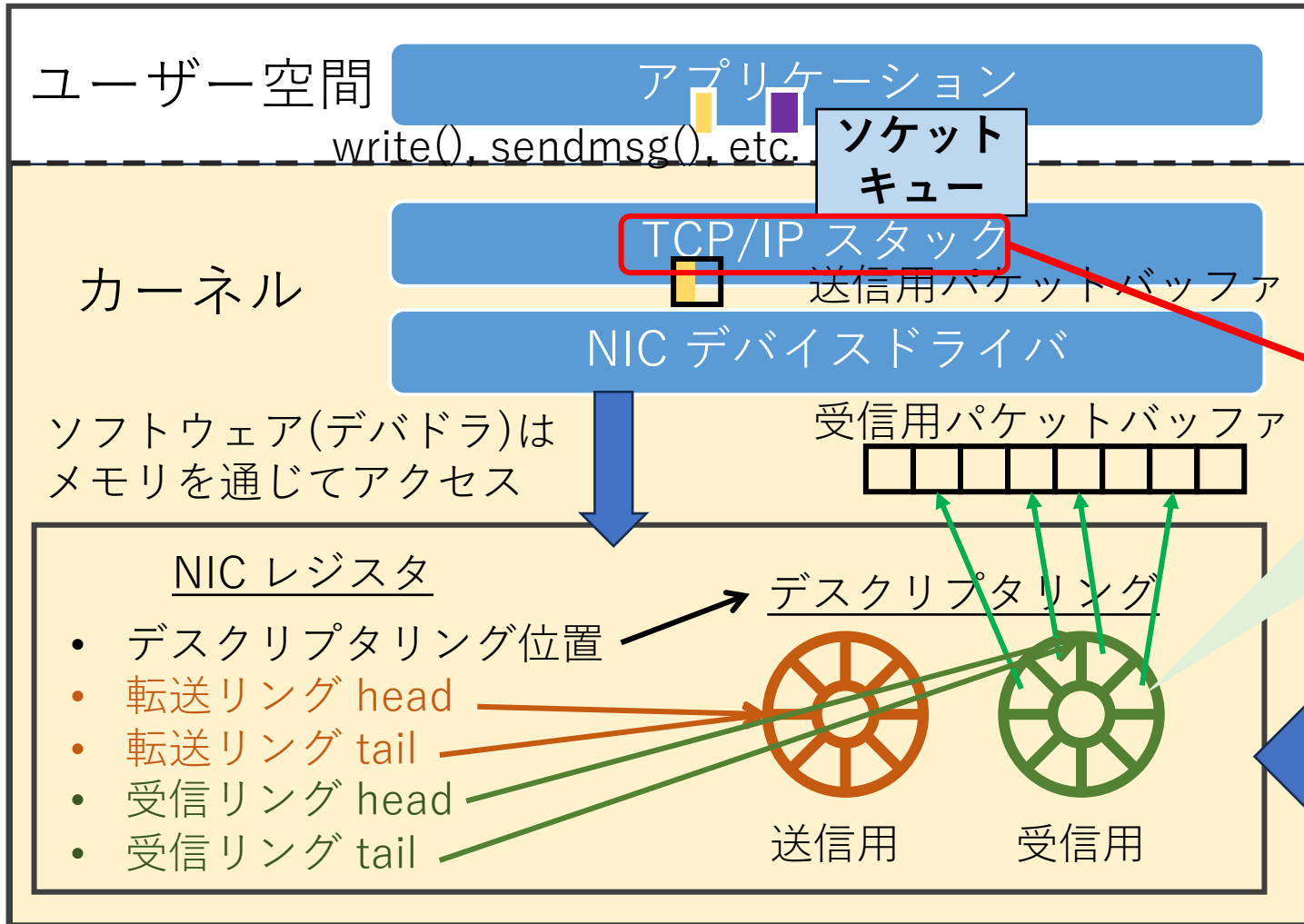
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

ユーザー空間からデータをカーネル空間で確保した送信用バッファへコピー



(ある程度) 一般的な送信処理の流れ



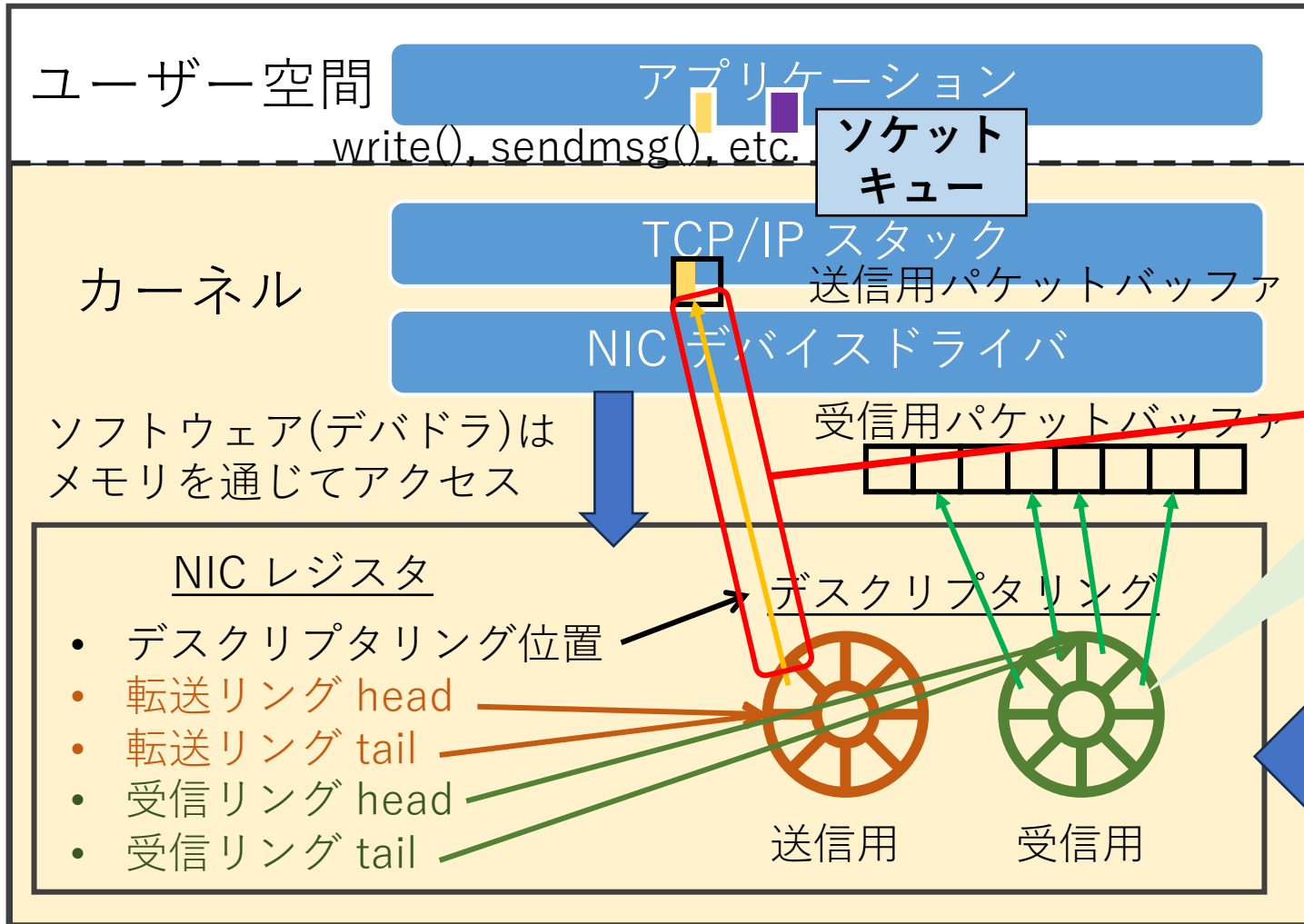
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

TCP/IP スタックの処理を実行
(ヘッダの付与)

DMA

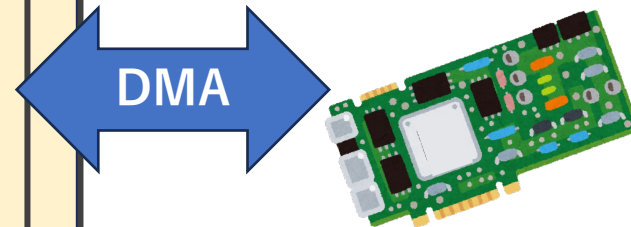
(ある程度) 一般的な送信処理の流れ



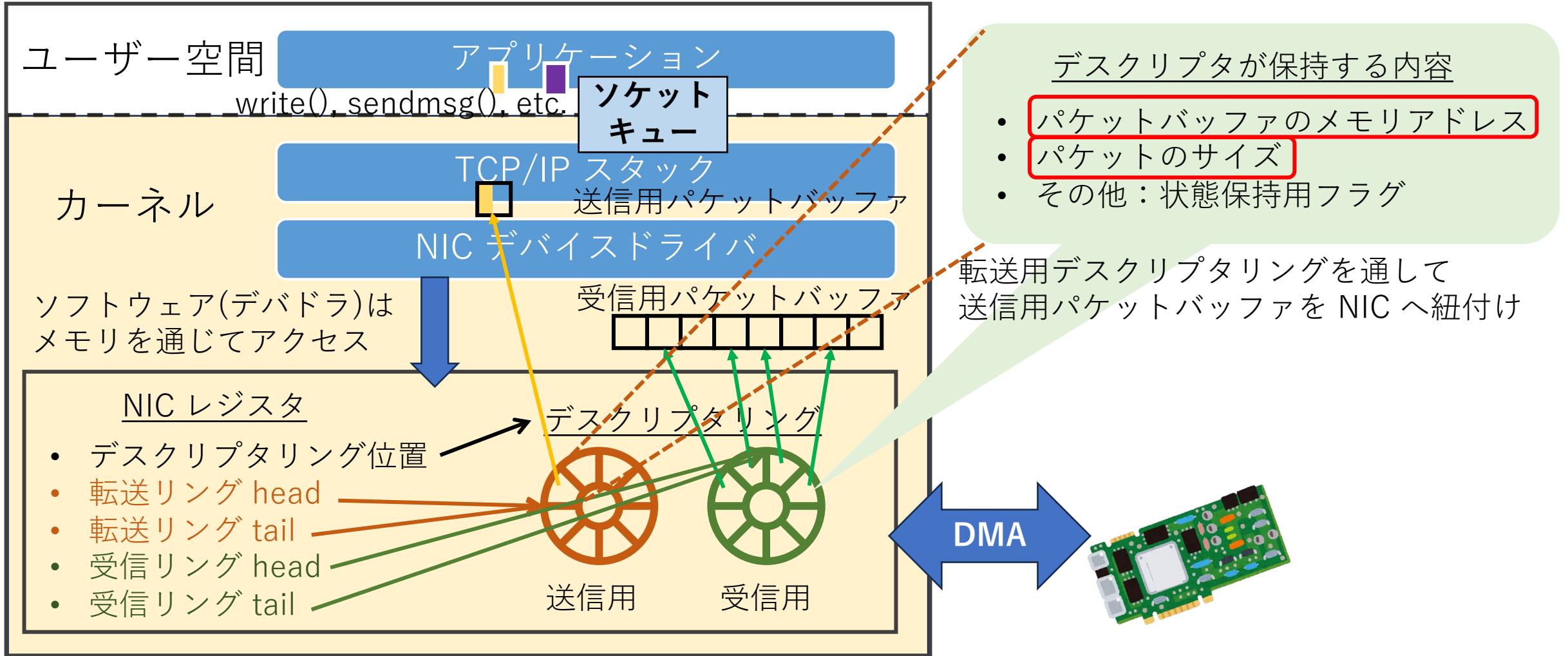
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

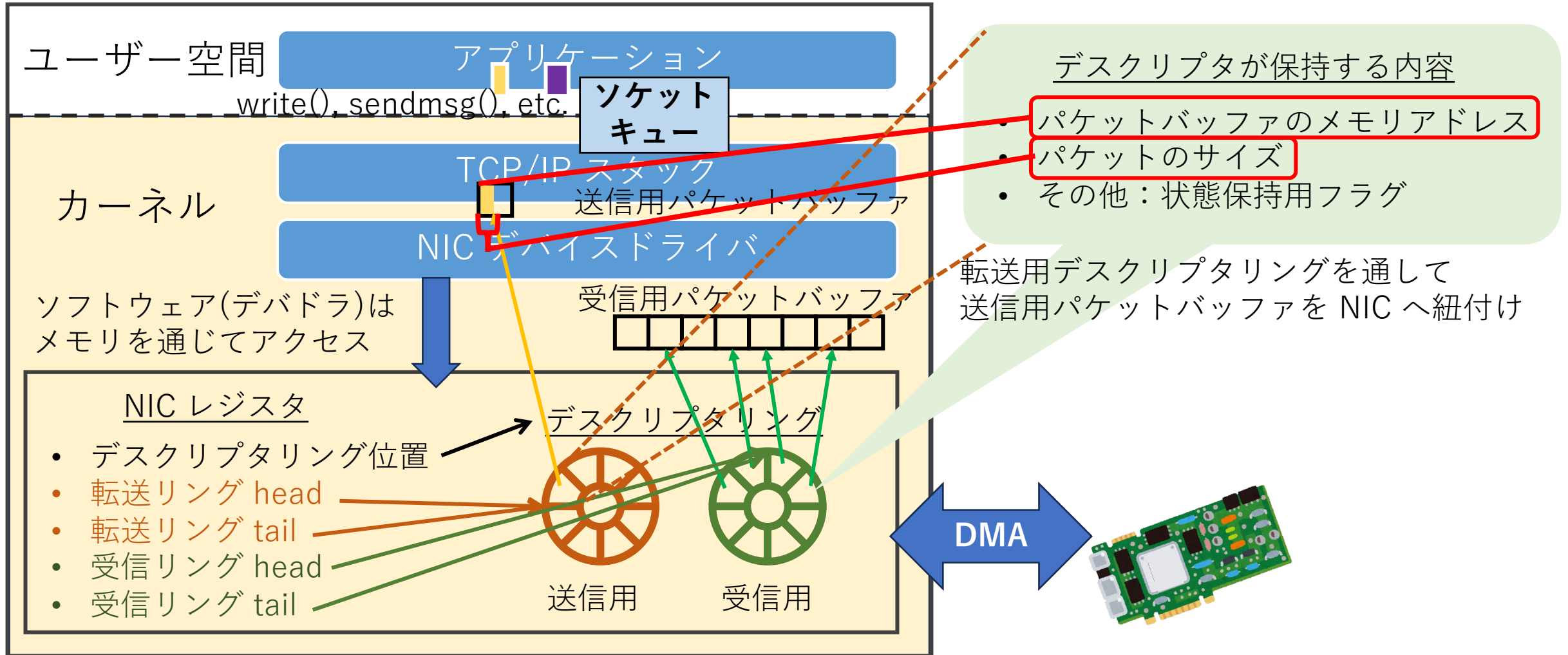
転送用デスクリプタリングを通して
送信用バッファを NIC へ紐付け



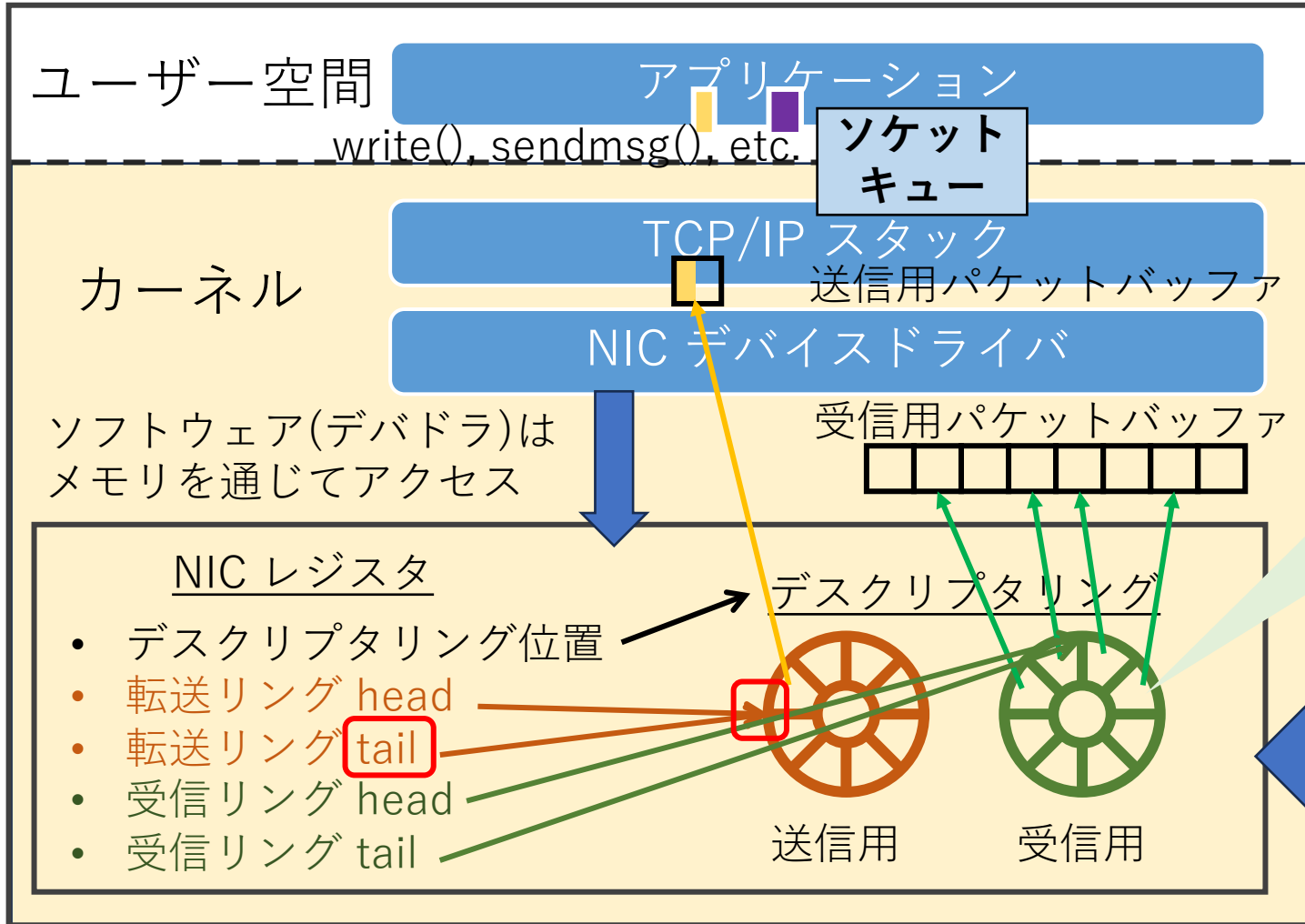
(ある程度) 一般的な送信処理の流れ



(ある程度) 一般的な送信処理の流れ



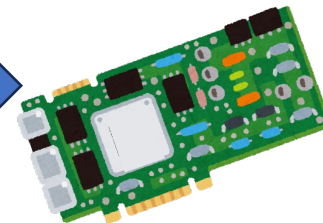
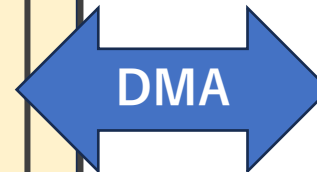
(ある程度) 一般的な送信処理の流れ



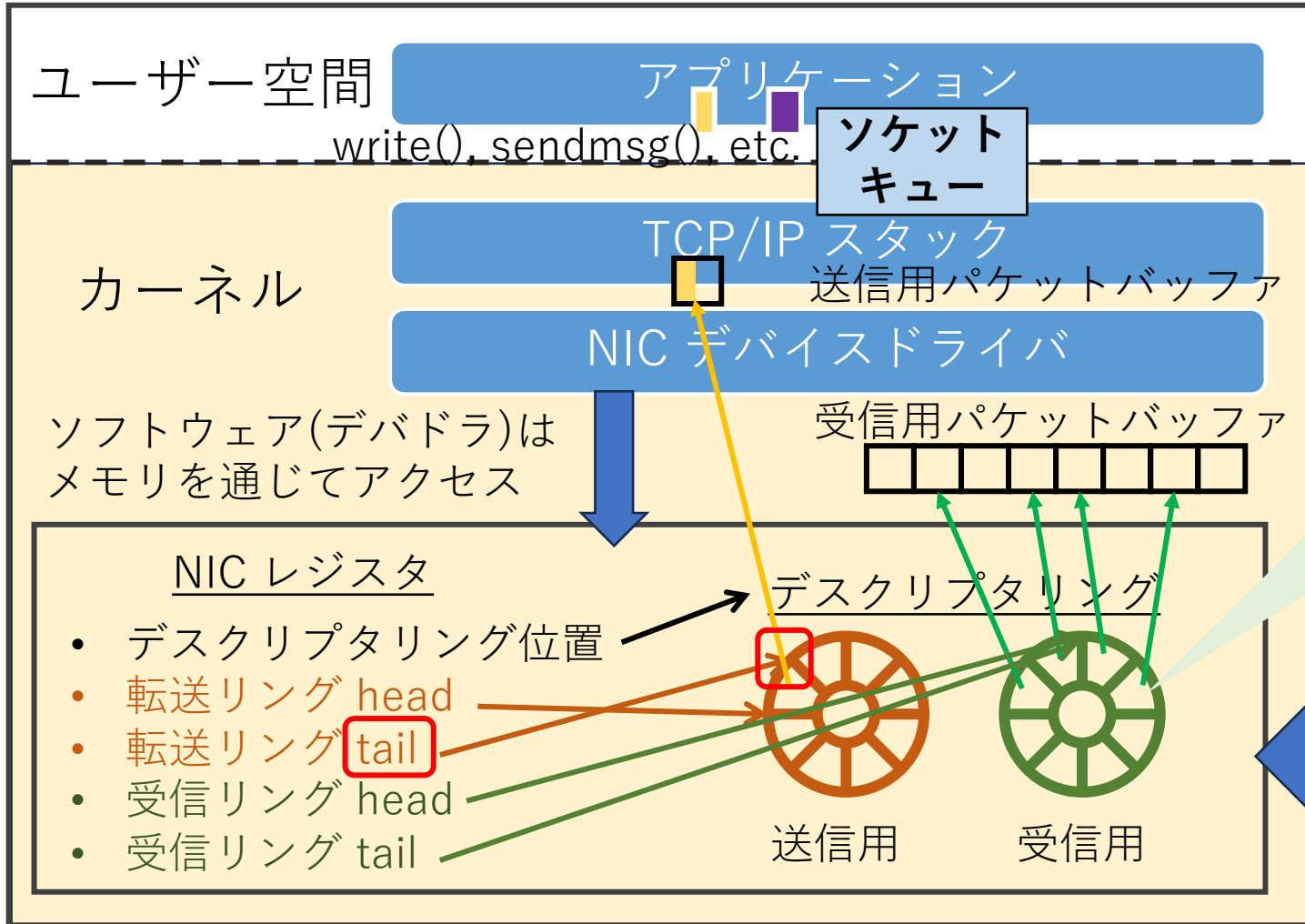
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC レジスタの転送リングの tail の値を更新



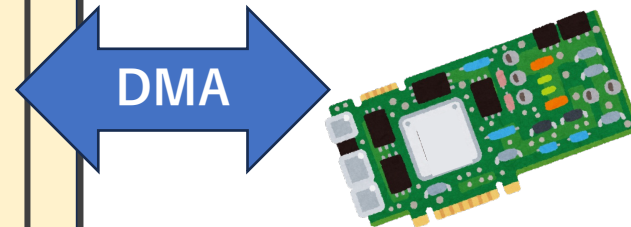
(ある程度) 一般的な送信処理の流れ



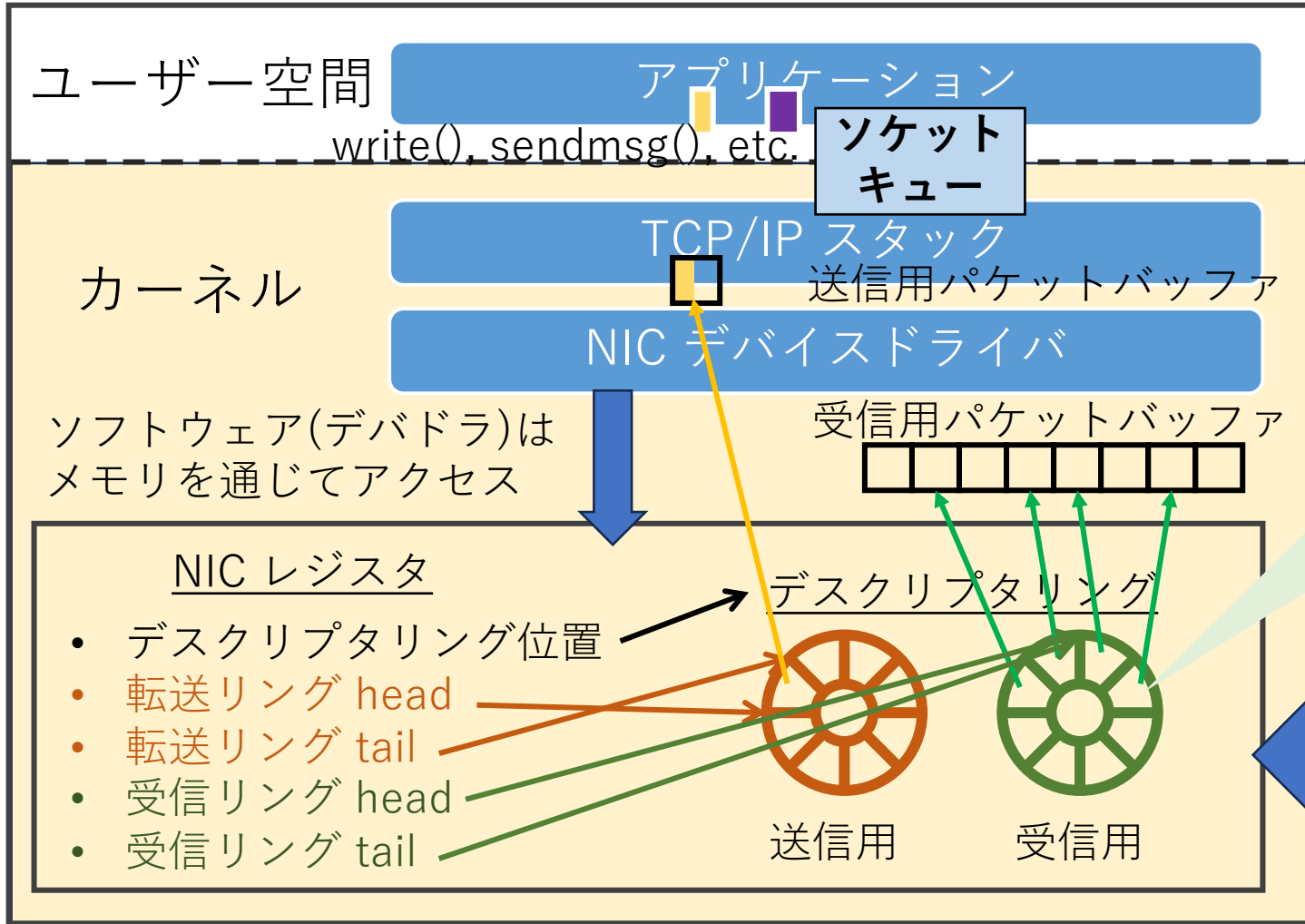
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC レジスタの転送リングの tail の値を更新



(ある程度) 一般的な送信処理の流れ

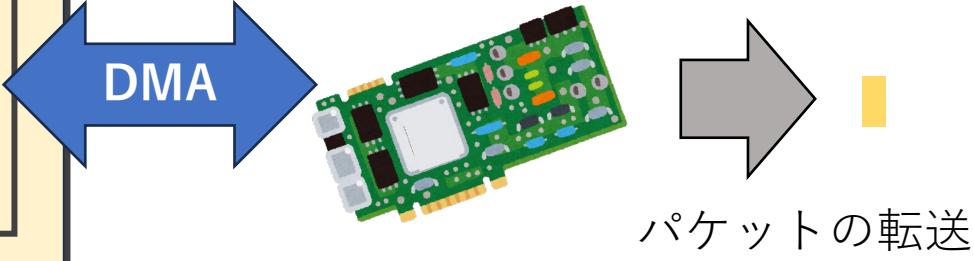


デスクリプタが保持する内容

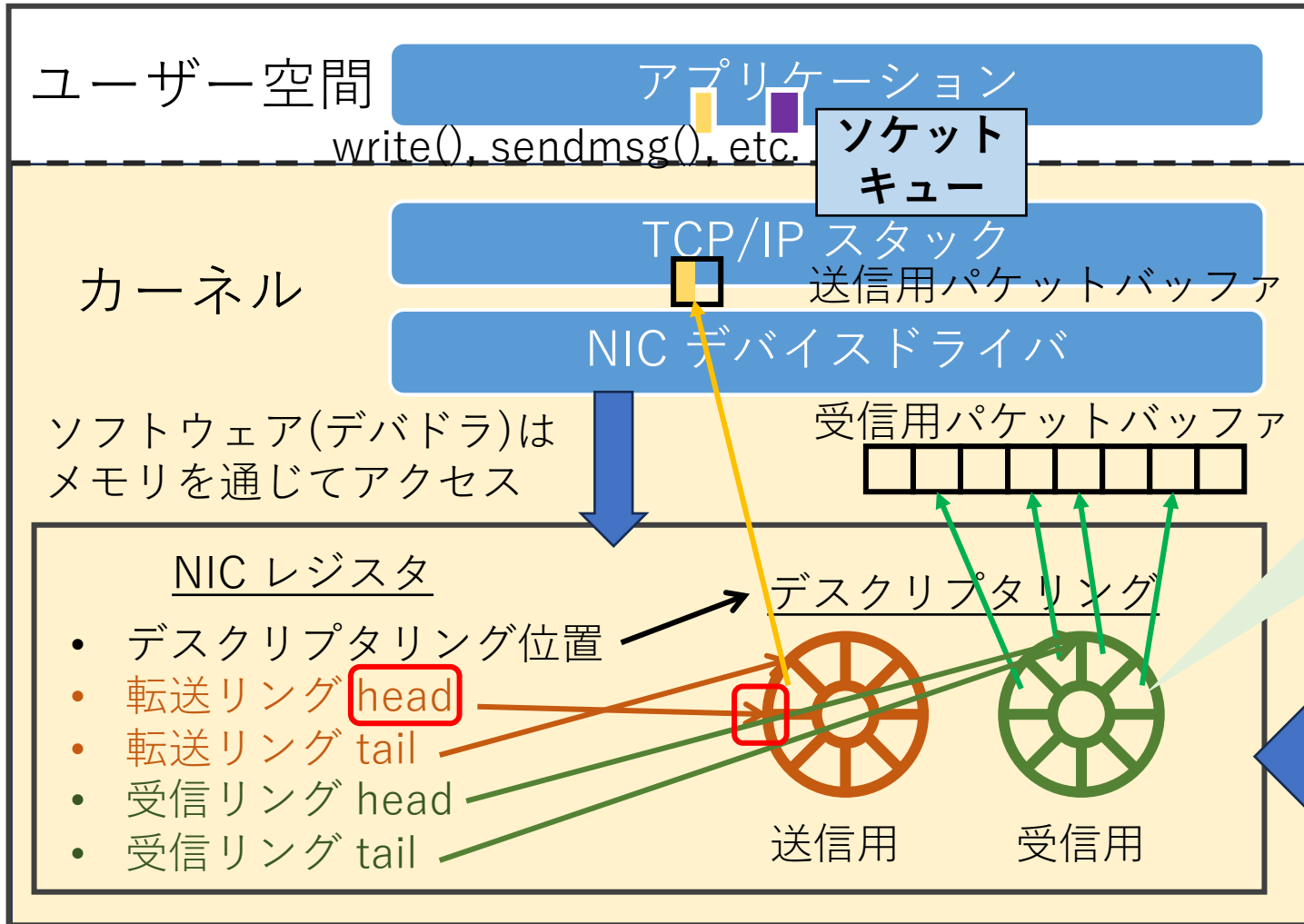
- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC レジスタの転送リングの tail の値を更新

これをきっかけに NIC からパケットが転送される



(ある程度) 一般的な送信処理の流れ



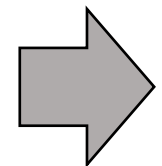
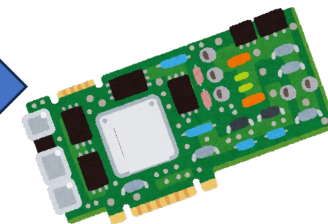
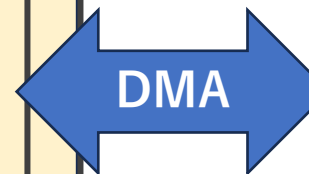
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC レジスタの転送リングの tail の値を更新

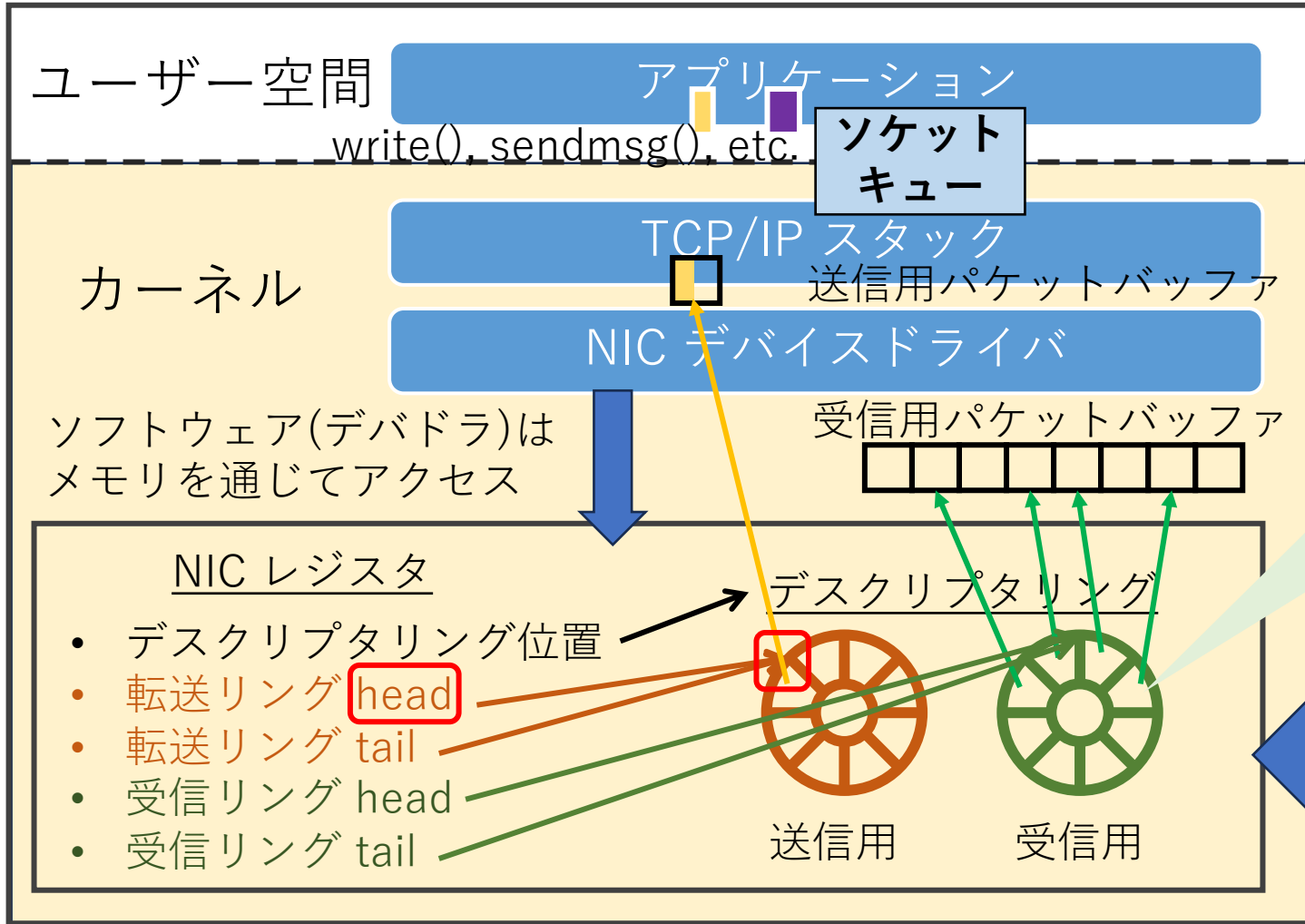
これをきっかけに NIC から
パケットが転送される

転送完了後、
NIC が転送リングの head を進める



パケットの転送

(ある程度) 一般的な送信処理の流れ



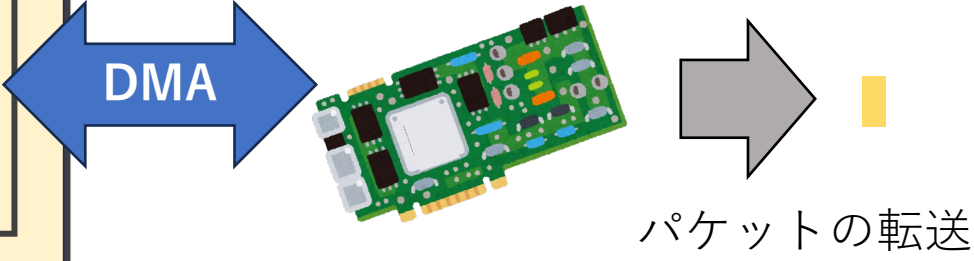
デスクリプタが保持する内容

- パケットバッファのメモリアドレス
- パケットのサイズ
- その他：状態保持用フラグ

NIC レジスタの転送リングの tail の値を更新

これをきっかけに NIC からパケットが転送される

転送完了後、NIC が転送リングの head を進める



研究紹介

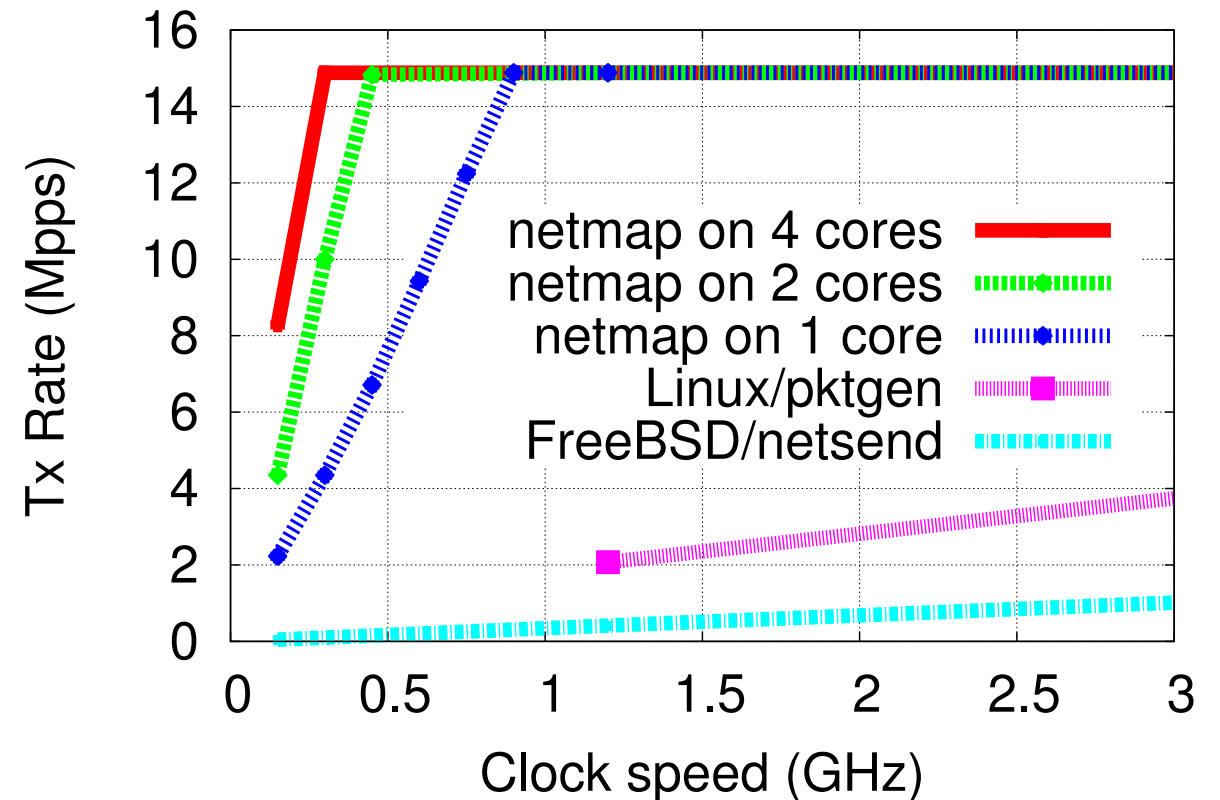
パケット I/O 性能について

カーネルをバイパスするパケット I/O フレームワーク

パケット I/O フレームワーク

- カーネル（の大部分）をバイパスしてユーザー空間から NIC の I/O を実行できるようにする
 - DPDK (2010)
 - netmap (USENIX ATC 2012)

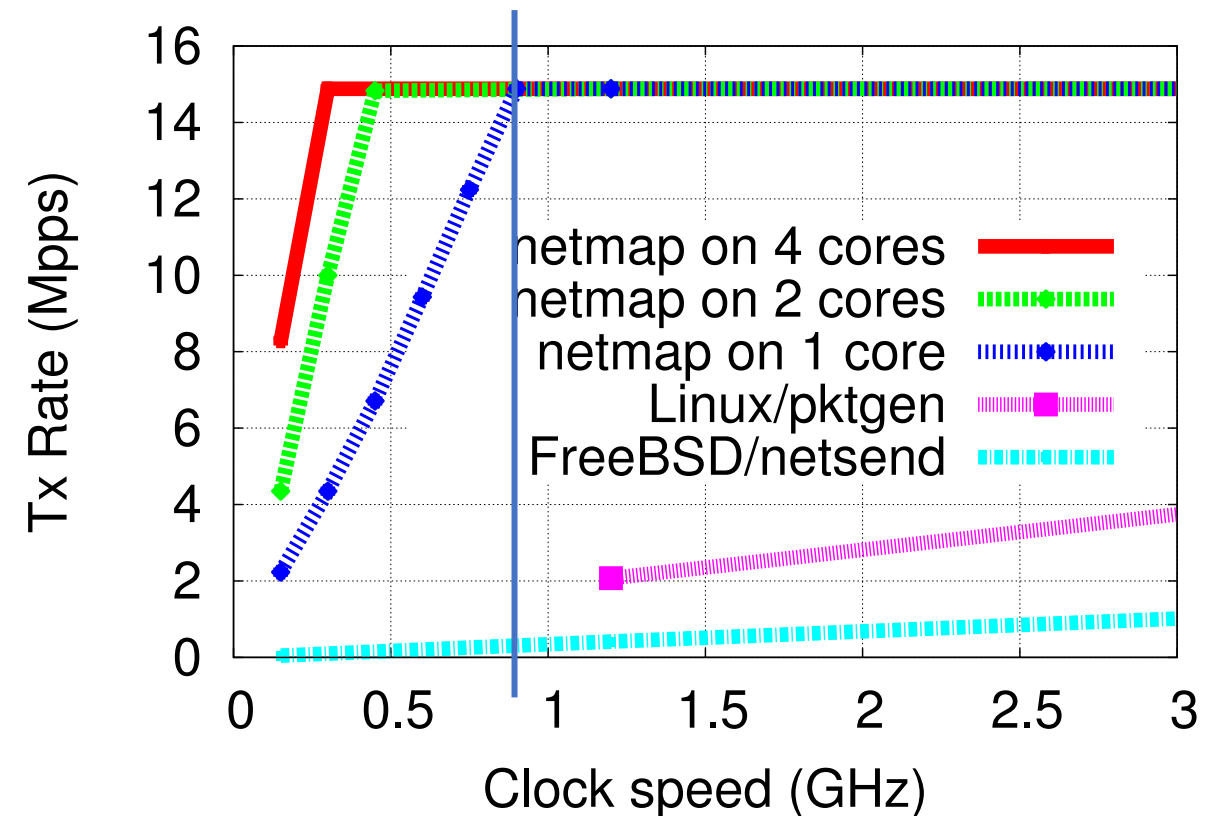
• 0.9 GHz で 10 Gbps NIC の
ラインレート (14.88 Mpps)
で送信できる



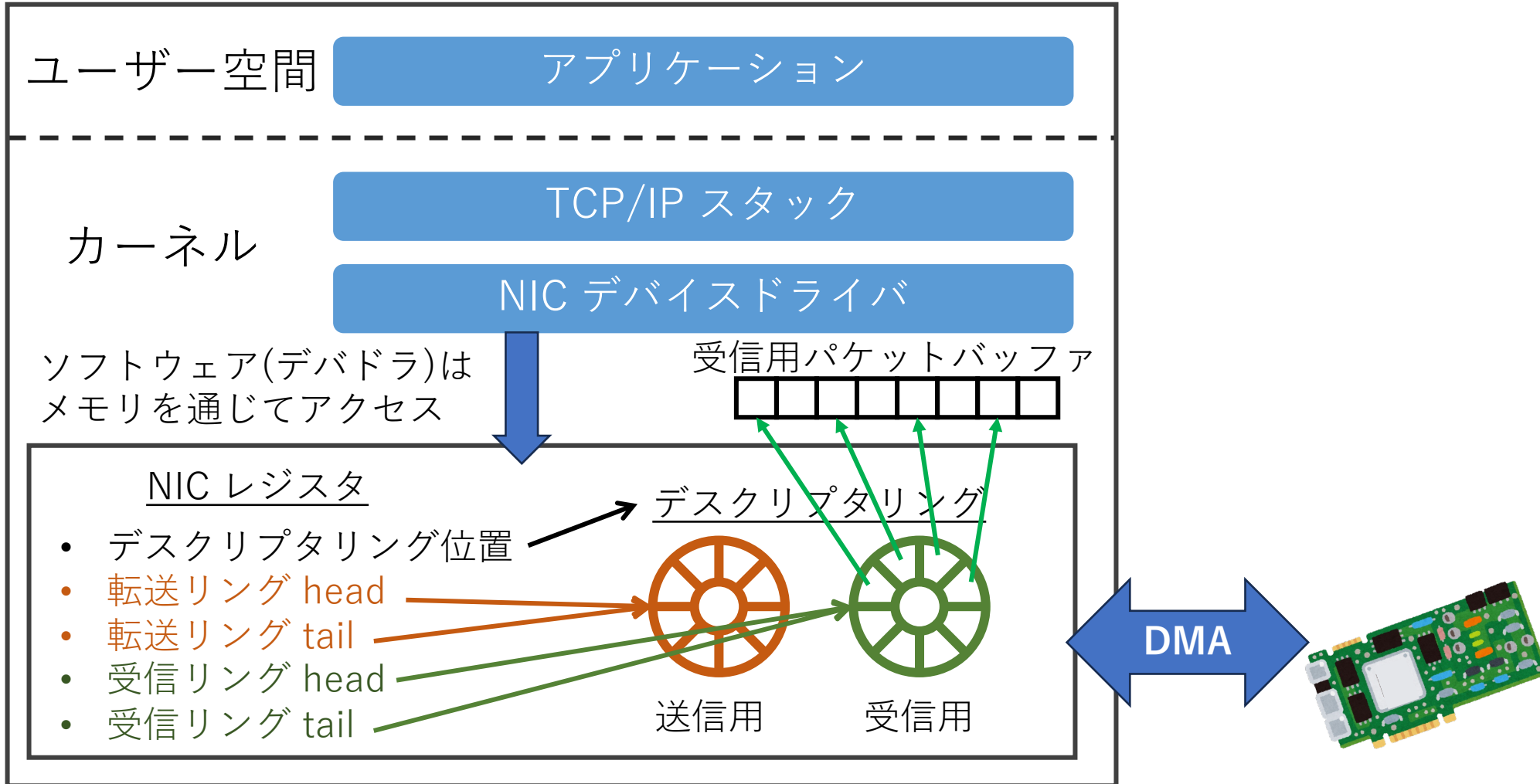
パケット I/O フレームワーク

- カーネル (の大部分) をバイパスしてユーザー空間から NIC の I/O を実行できるようにする
 - DPDK (2010)
 - netmap (USENIX ATC 2012)

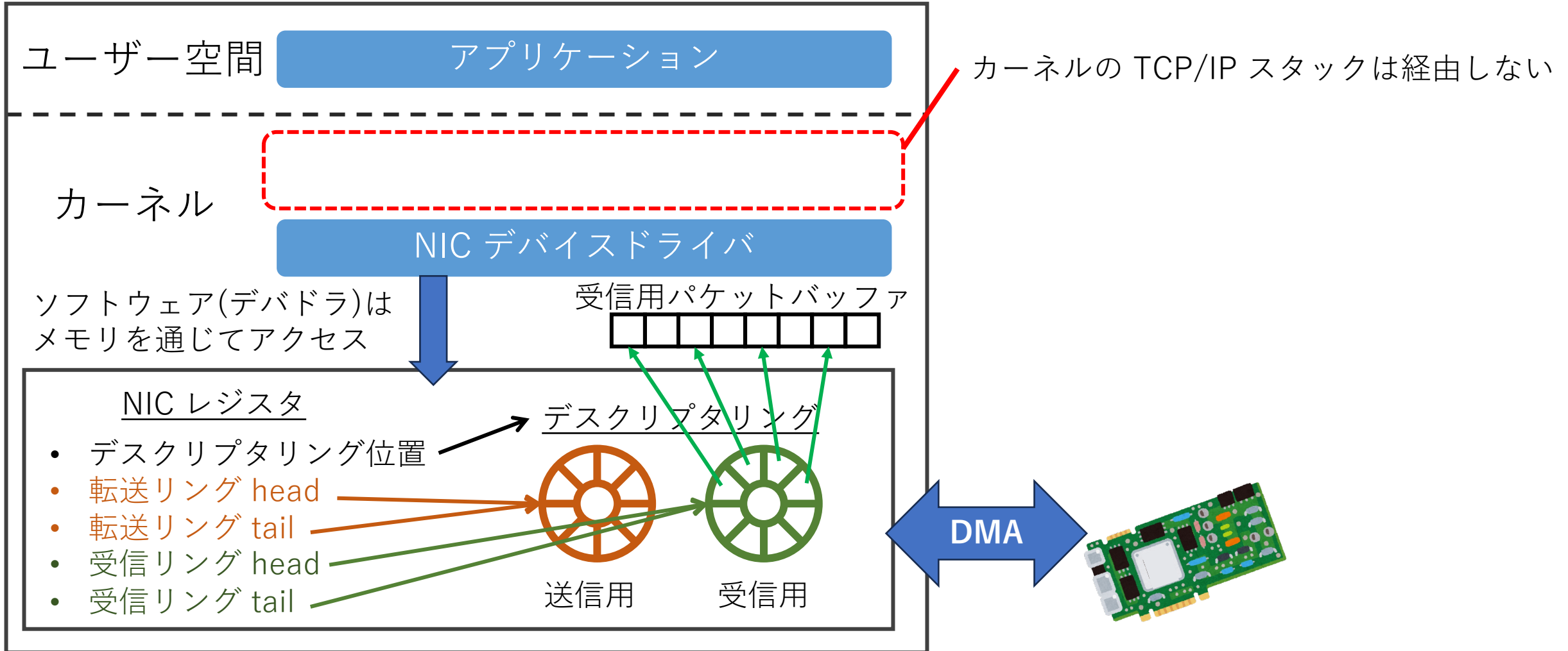
• 0.9 GHz で 10 Gbps NIC の
ラインレート (14.88 Mpps)
で送信できる



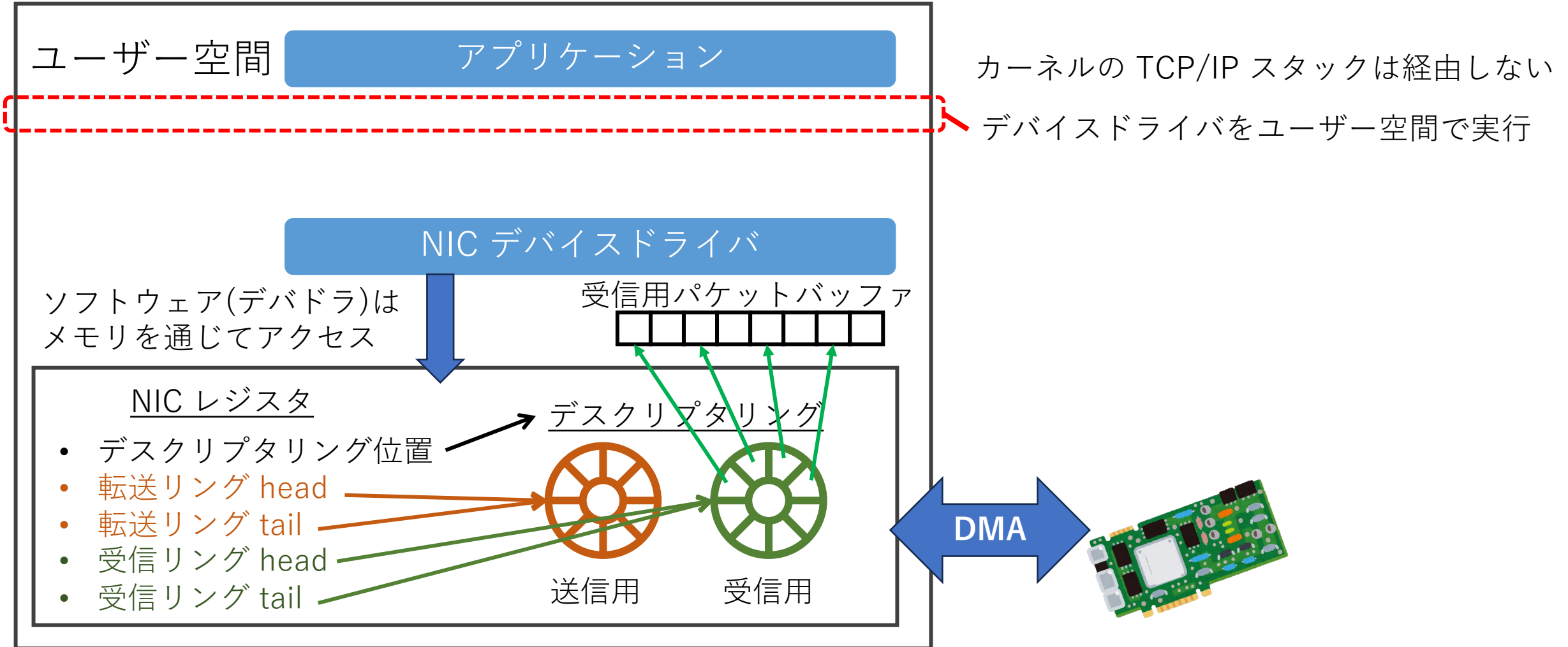
DPDK の場合



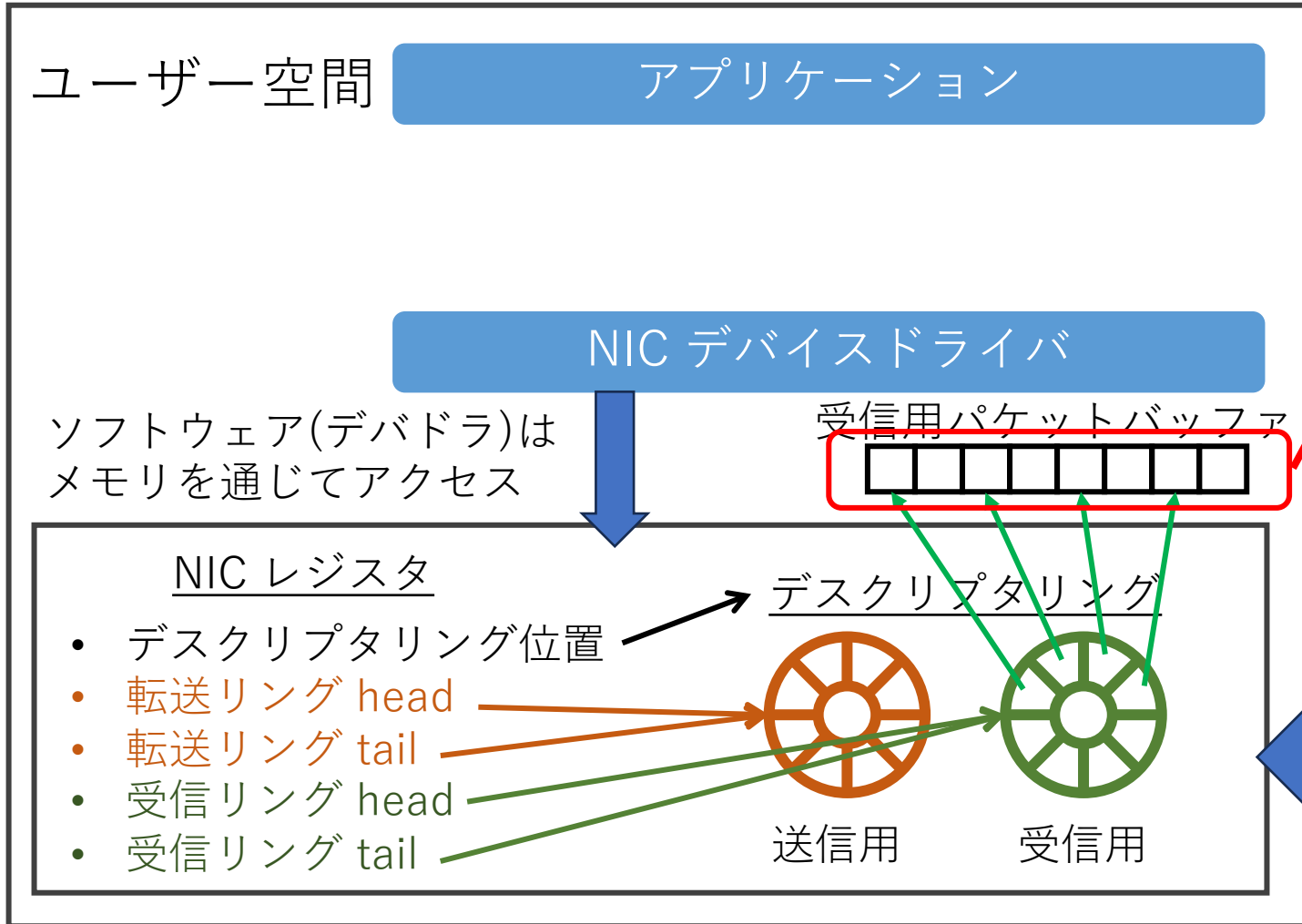
DPDK の場合



DPDK の場合



DPDK の場合

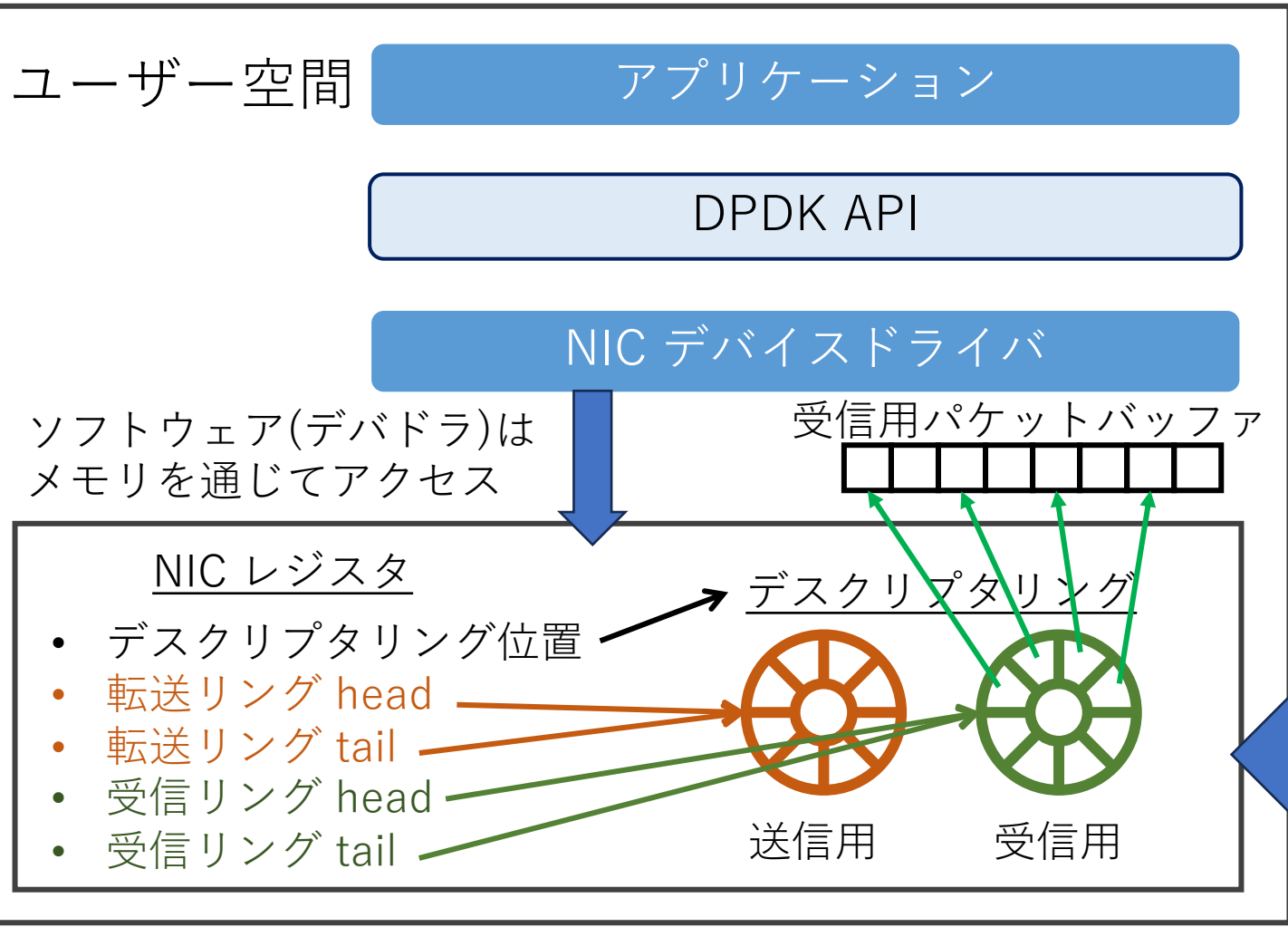


カーネルの TCP/IP スタックは経由しない
デバイスドライバをユーザー空間で実行

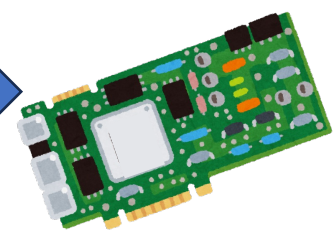
NIC に紐づいたパケットバッファも
ユーザー空間に配置

DMA

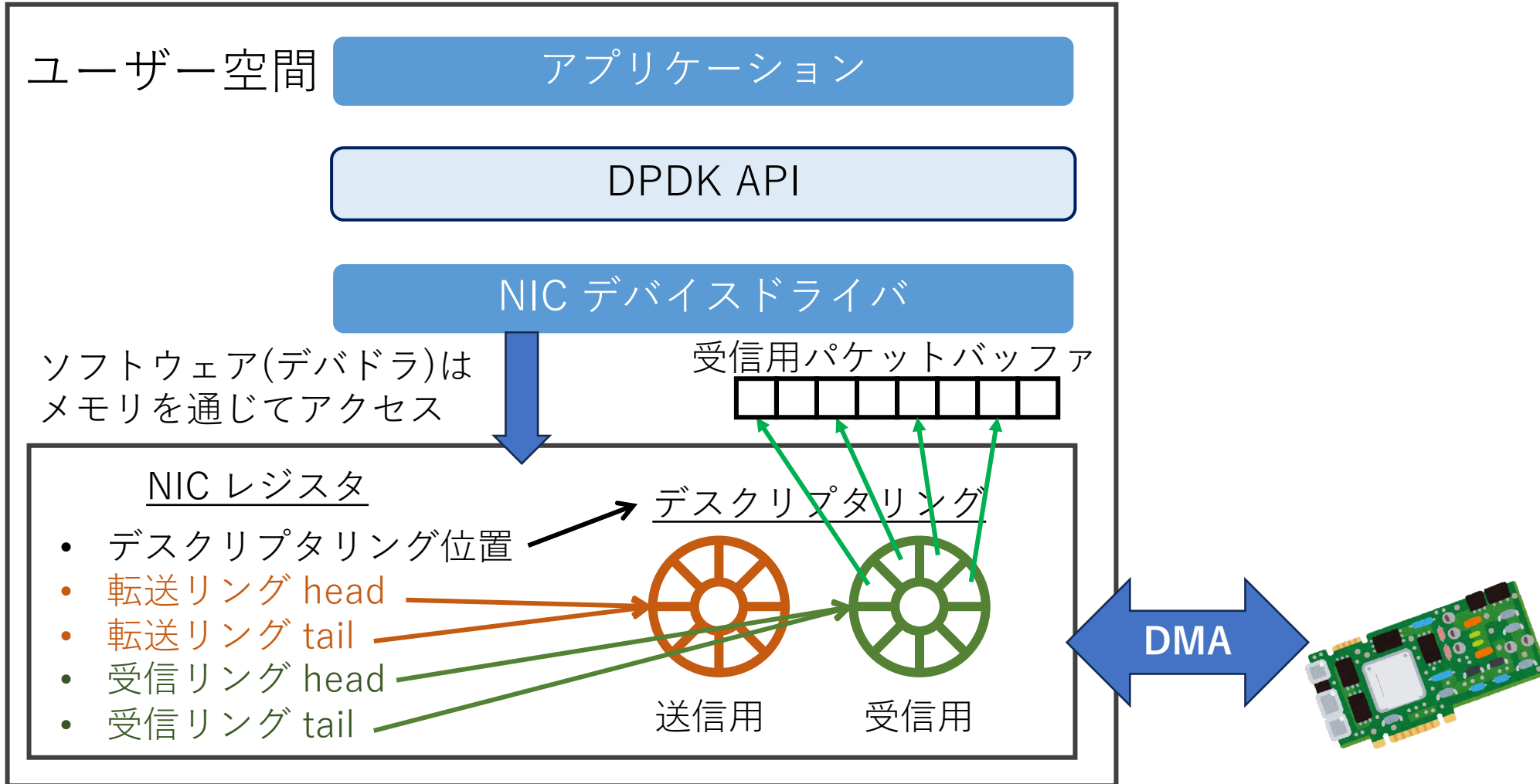
DPDK の場合



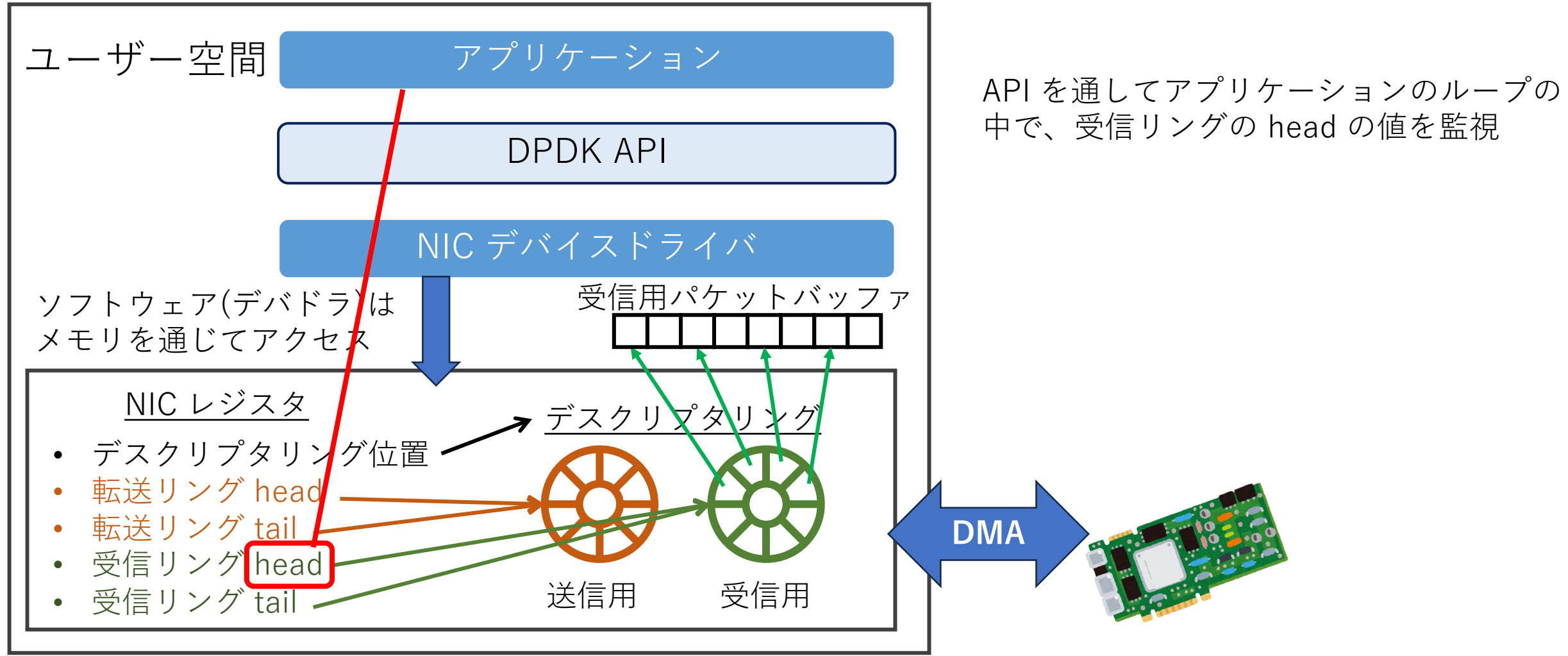
カーネルの TCP/IP スタックは経由しない
デバイスドライバをユーザー空間で実行
NIC に紐づいたパケットバッファもユーザー空間に配置



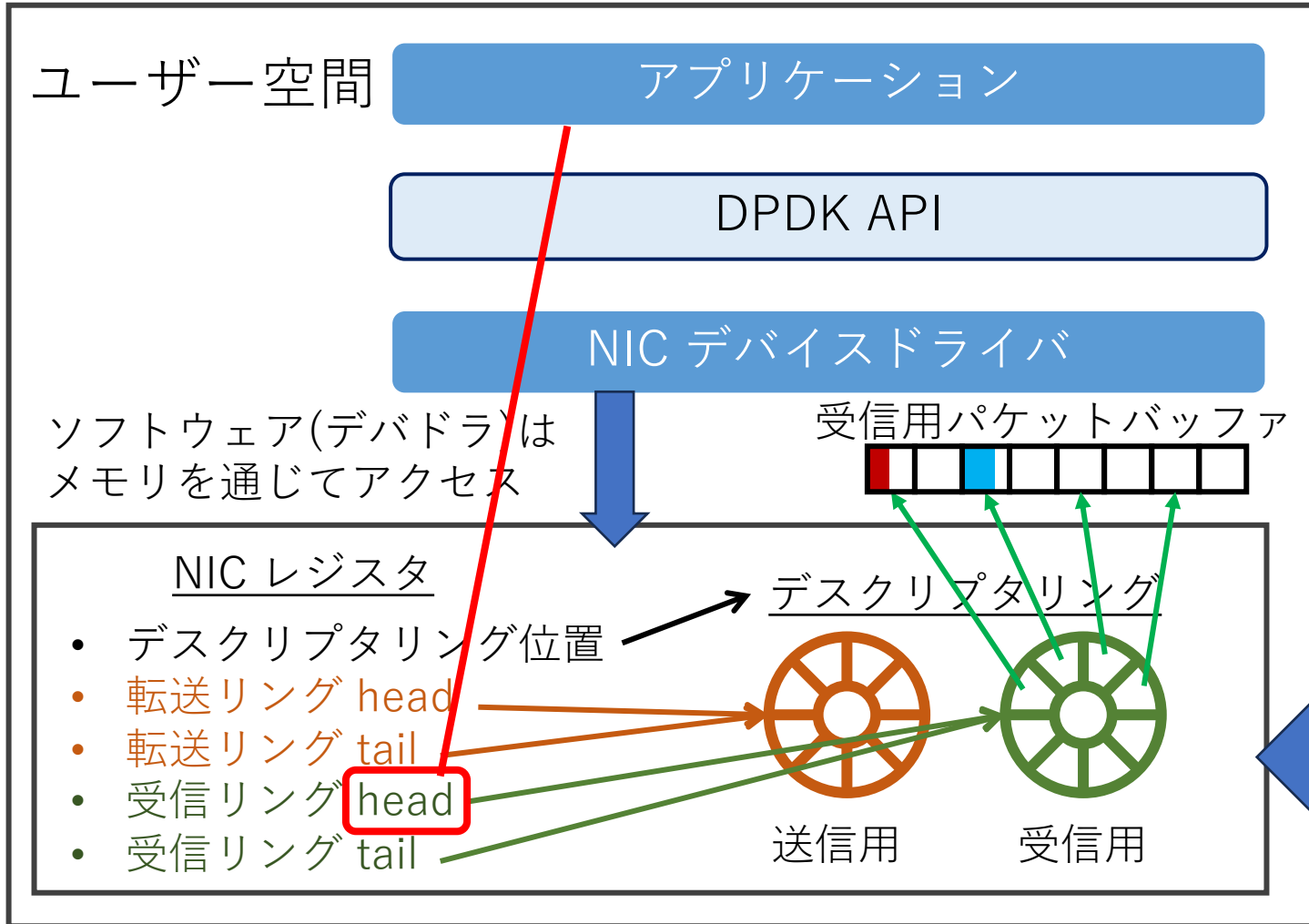
DPDK の場合：受信パケットの検知



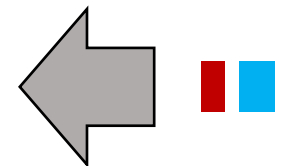
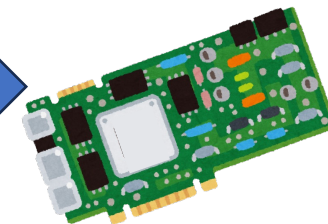
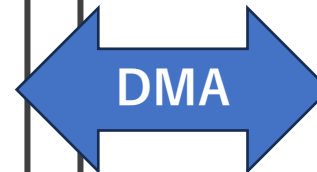
DPDK の場合：受信パケットの検知



DPDK の場合：受信パケットの検知

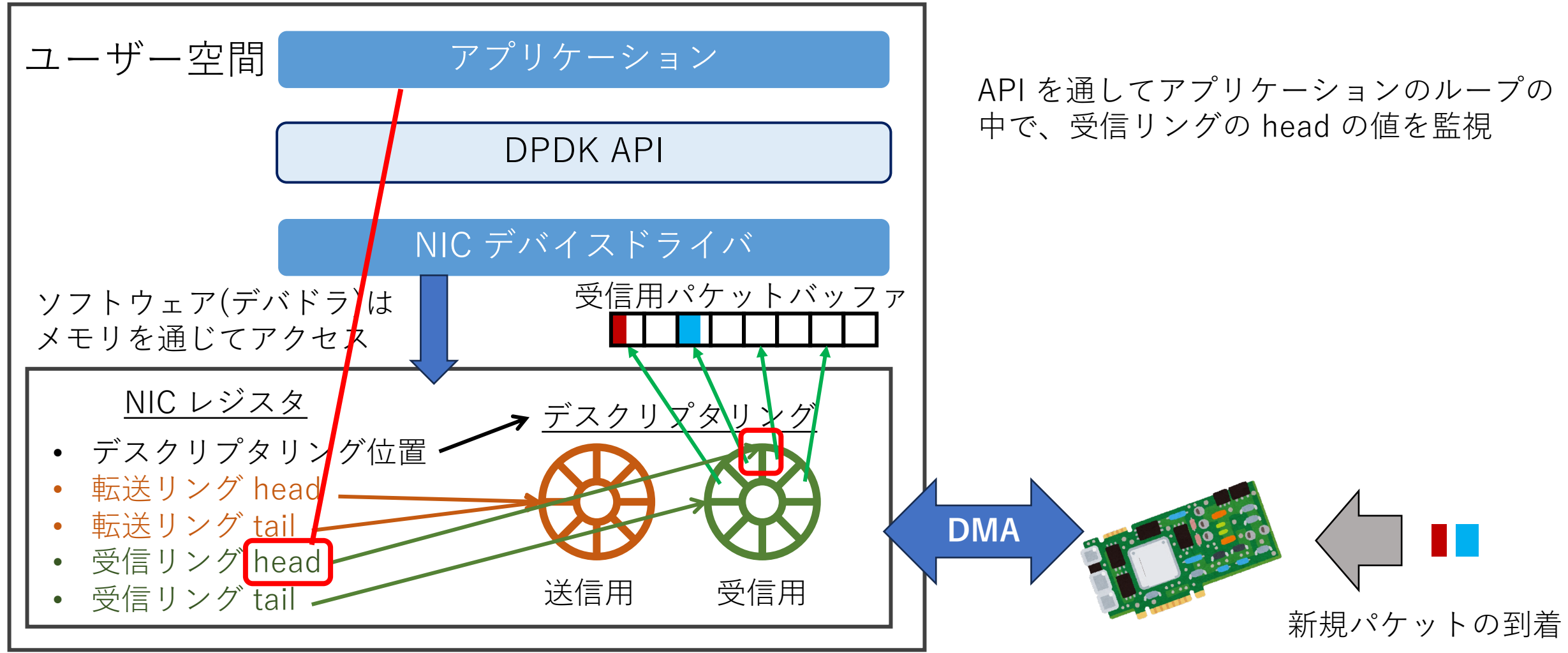


API を通してアプリケーションのループの中で、受信リングの head の値を監視

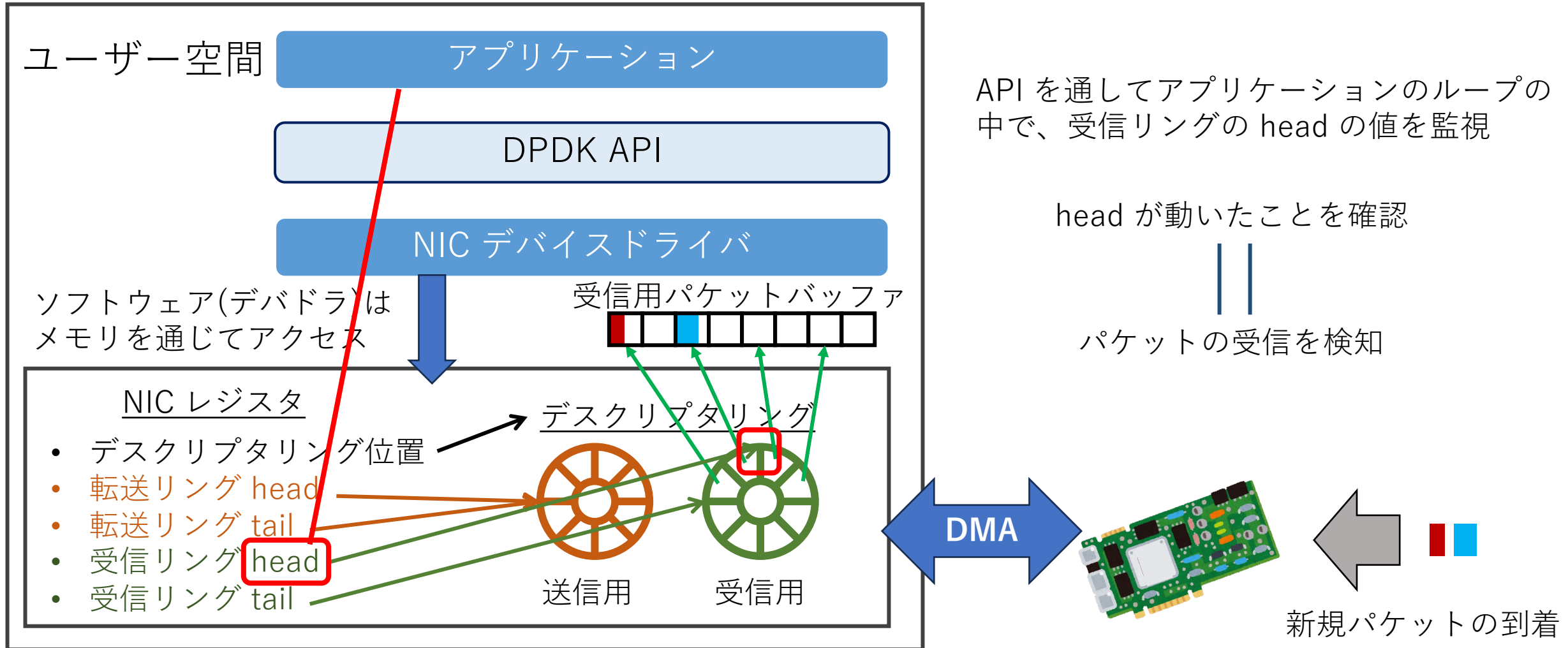


新規パケットの到着

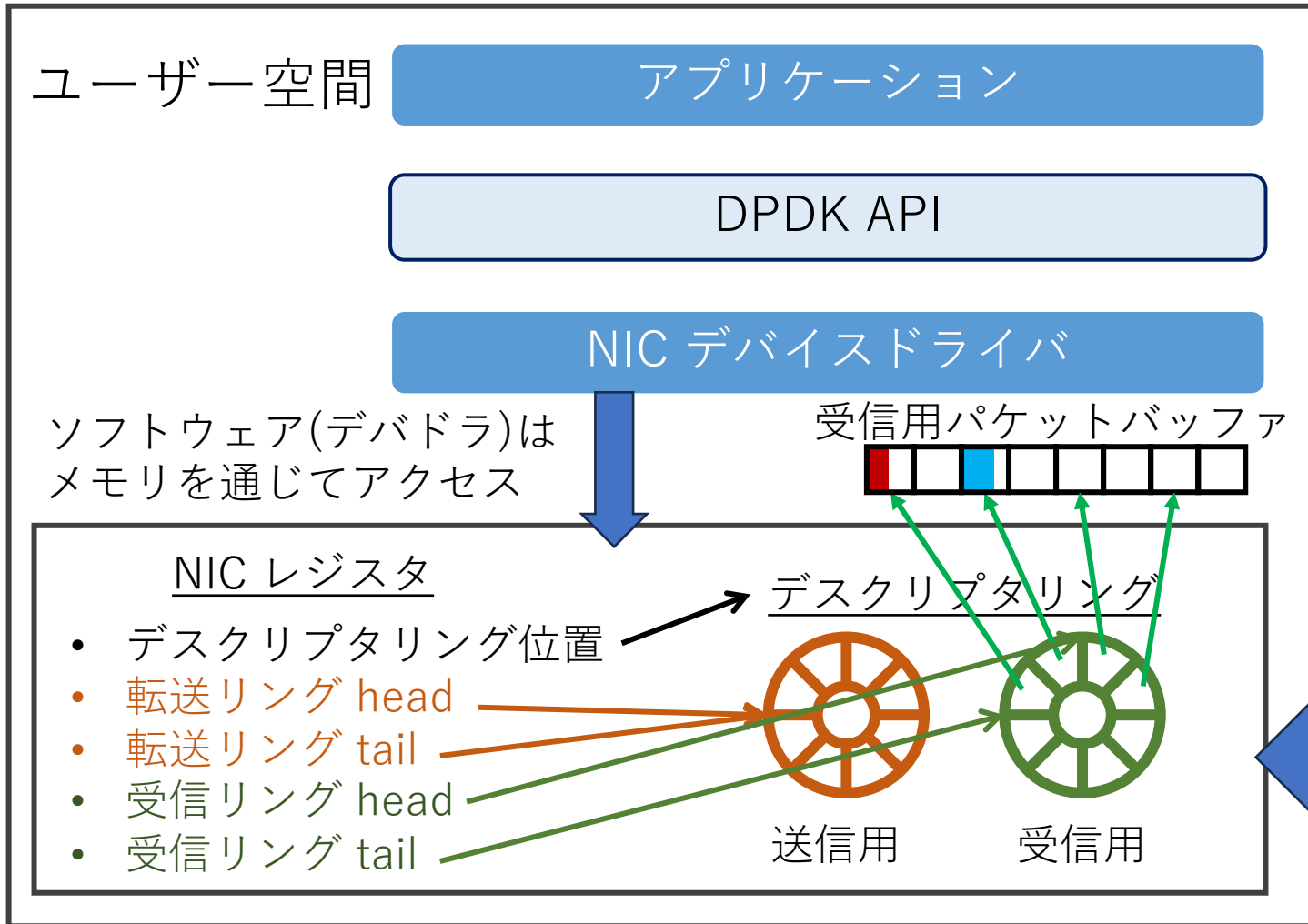
DPDK の場合：受信パケットの検知



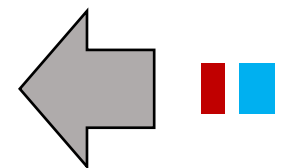
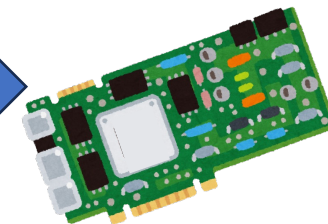
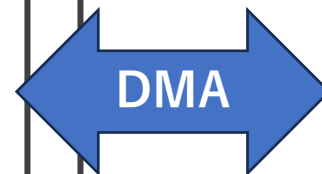
DPDK の場合：受信パケットの検知



DPDK の場合：受信パケットの読み込み

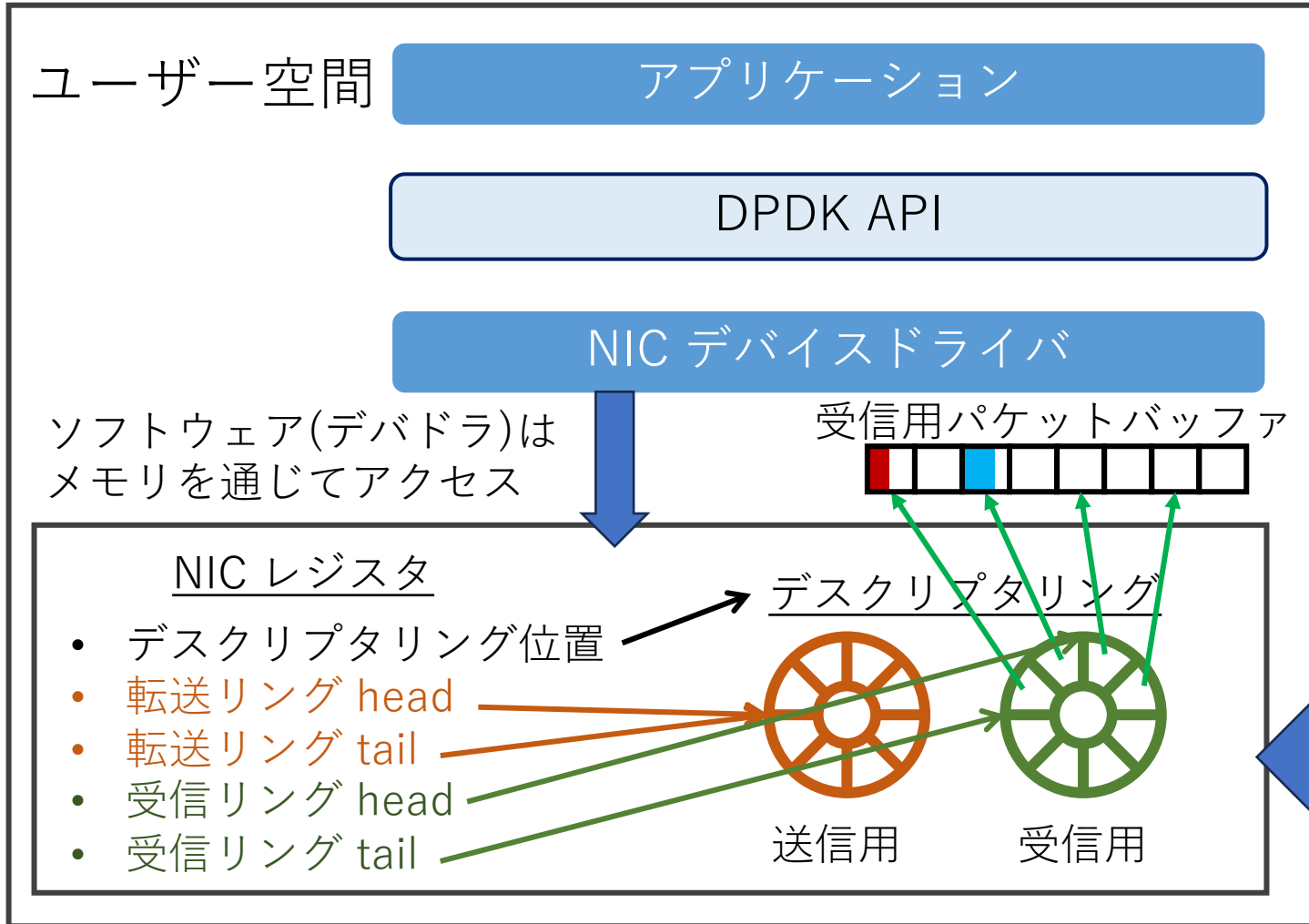


受信用パケットバッファはユーザー空間にあり、受信したデータは検知された段階で既にアプリケーションから見えている



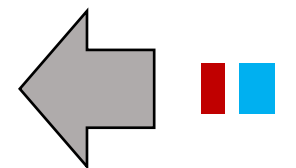
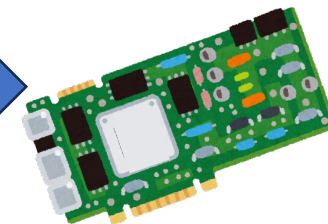
新規パケットの到着

DPDK の場合：受信パケットの読み込み



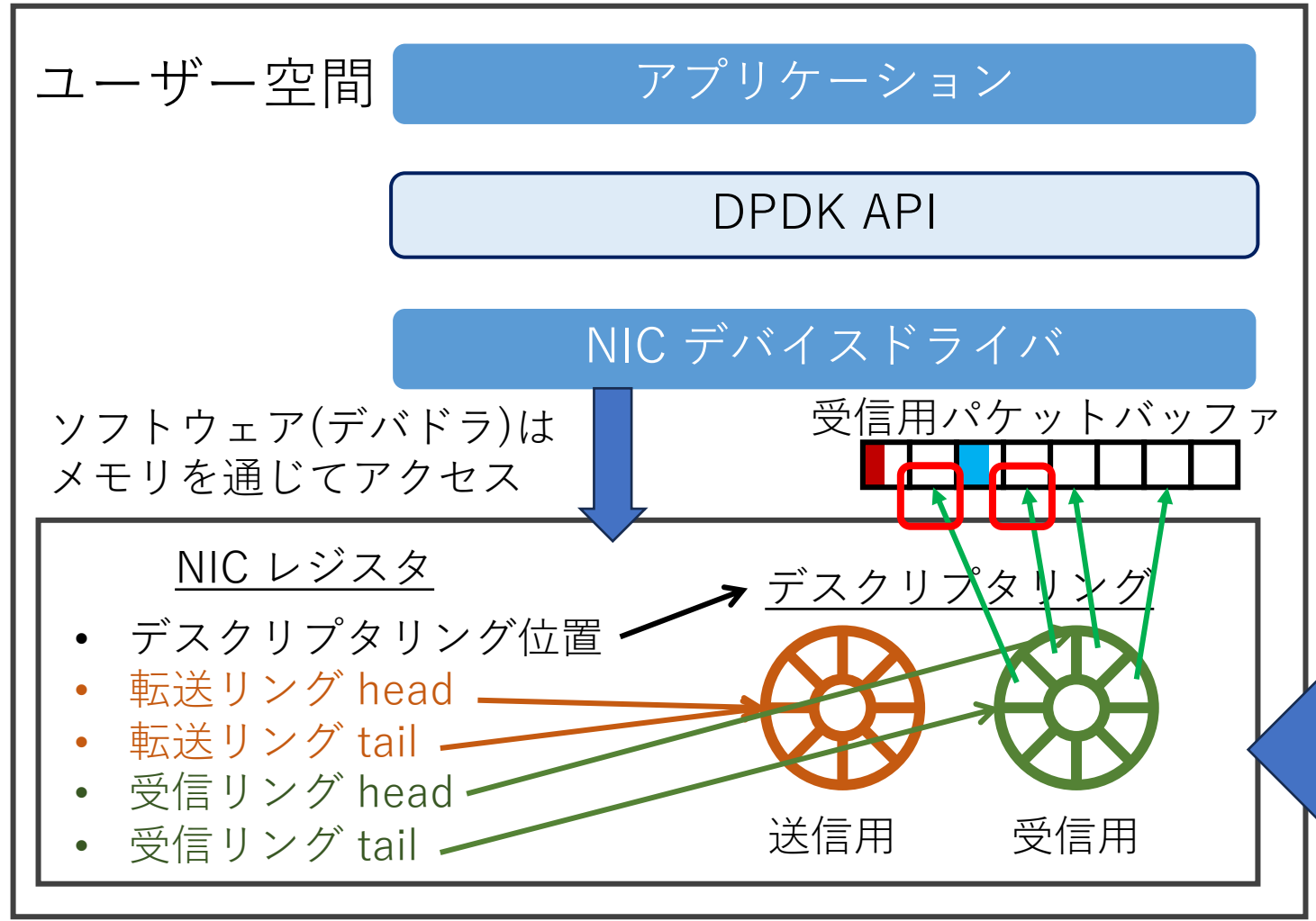
受信用パケットバッファはユーザー空間にあり、受信したデータは検知された段階で既にアプリケーションから見えている

なので、読み込みのために追加の作業はなし



新規パケットの到着

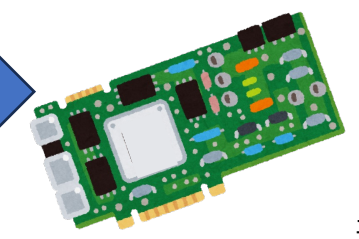
DPDK の場合：受信リング tail の更新



受信用バッファはユーザー空間にあり、受信したデータは検知された段階で既にアプリケーションから見えている

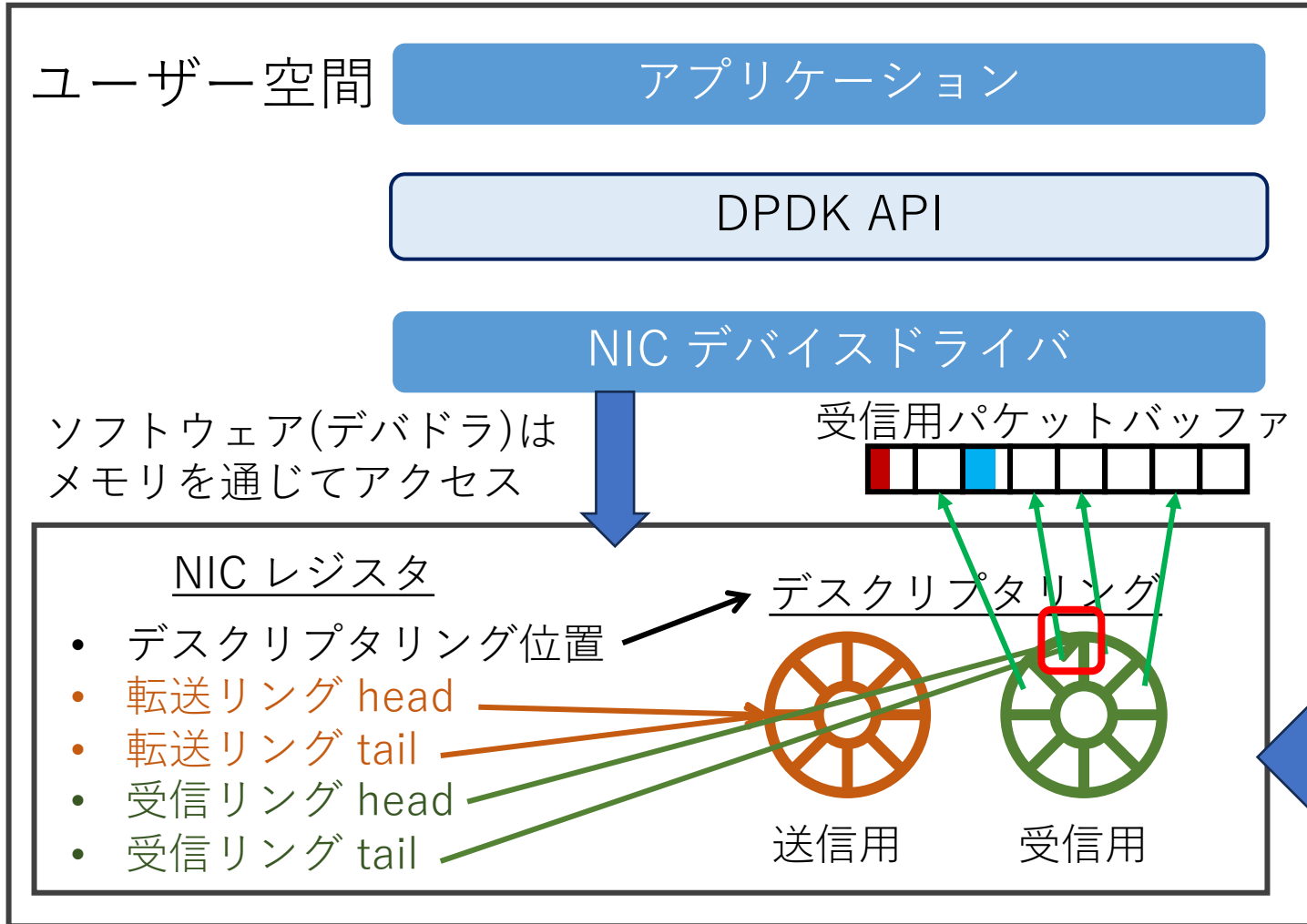
なので、読み込みのために追加の作業はなし

新しいパケットを受け取れるように別のバッファを紐づけて、tailを進める



新規パケットの到着

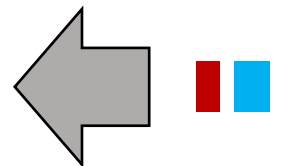
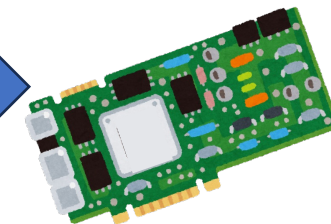
DPDK の場合：受信リング tail の更新



受信用バッファはユーザー空間にあり、受信したデータは検知された段階で既にアプリケーションから見えている

なので、読み込みのために追加の作業はなし

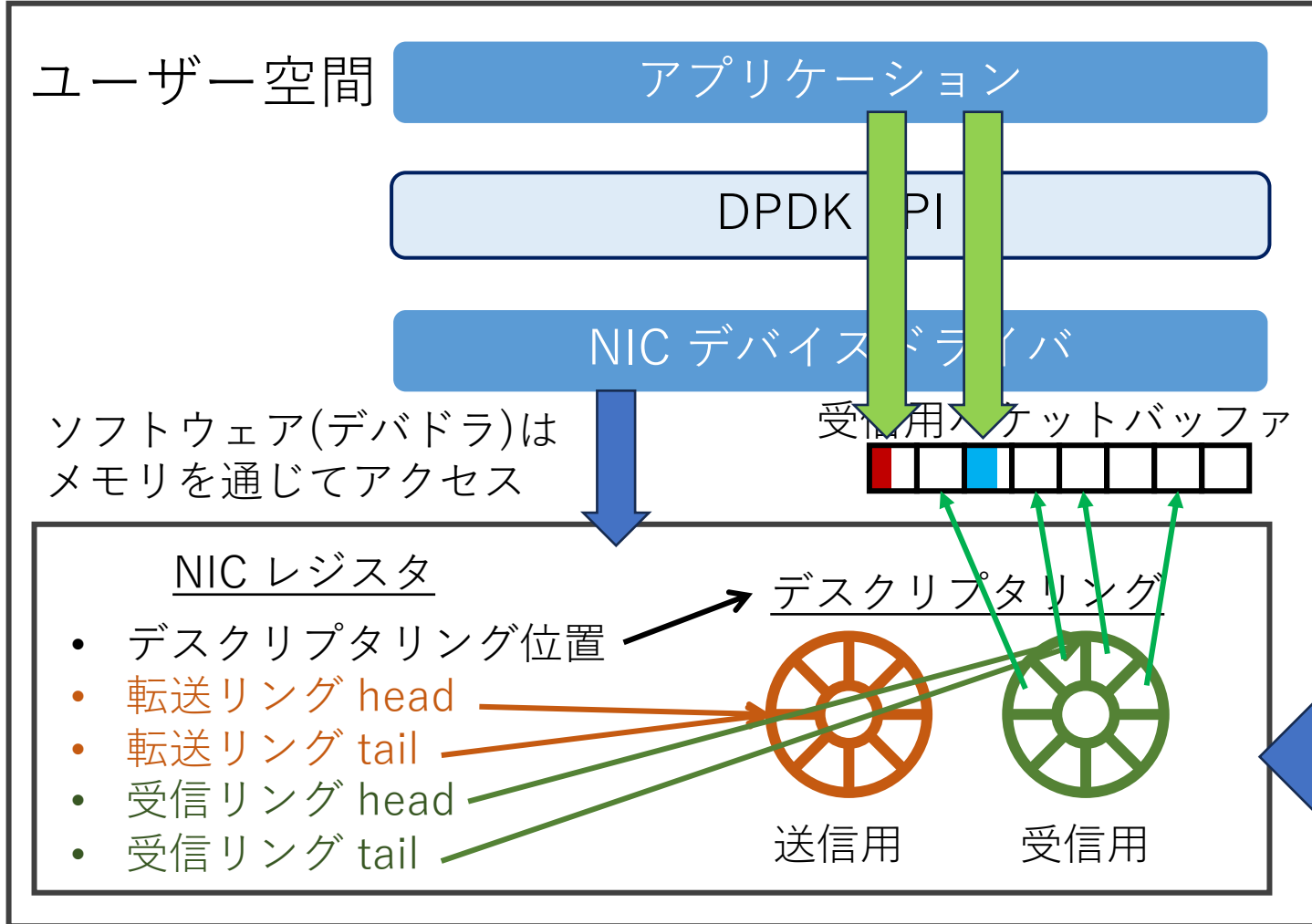
新しいパケットを受け取れるように別のバッファを紐づけて、tailを進める



新規パケットの到着

DPDK の場合

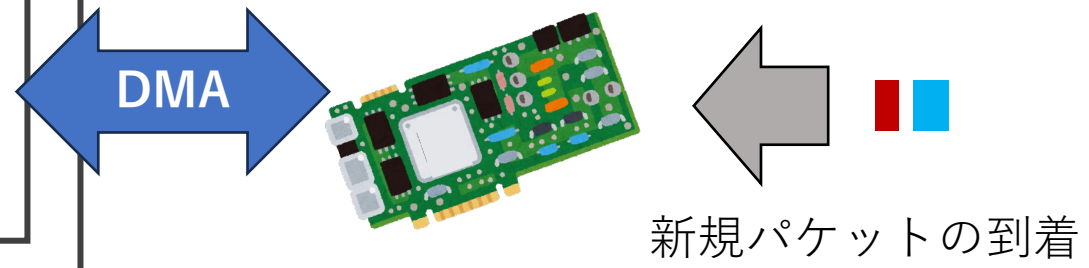
あとは、アプリが好きなように受信したデータを消費できる



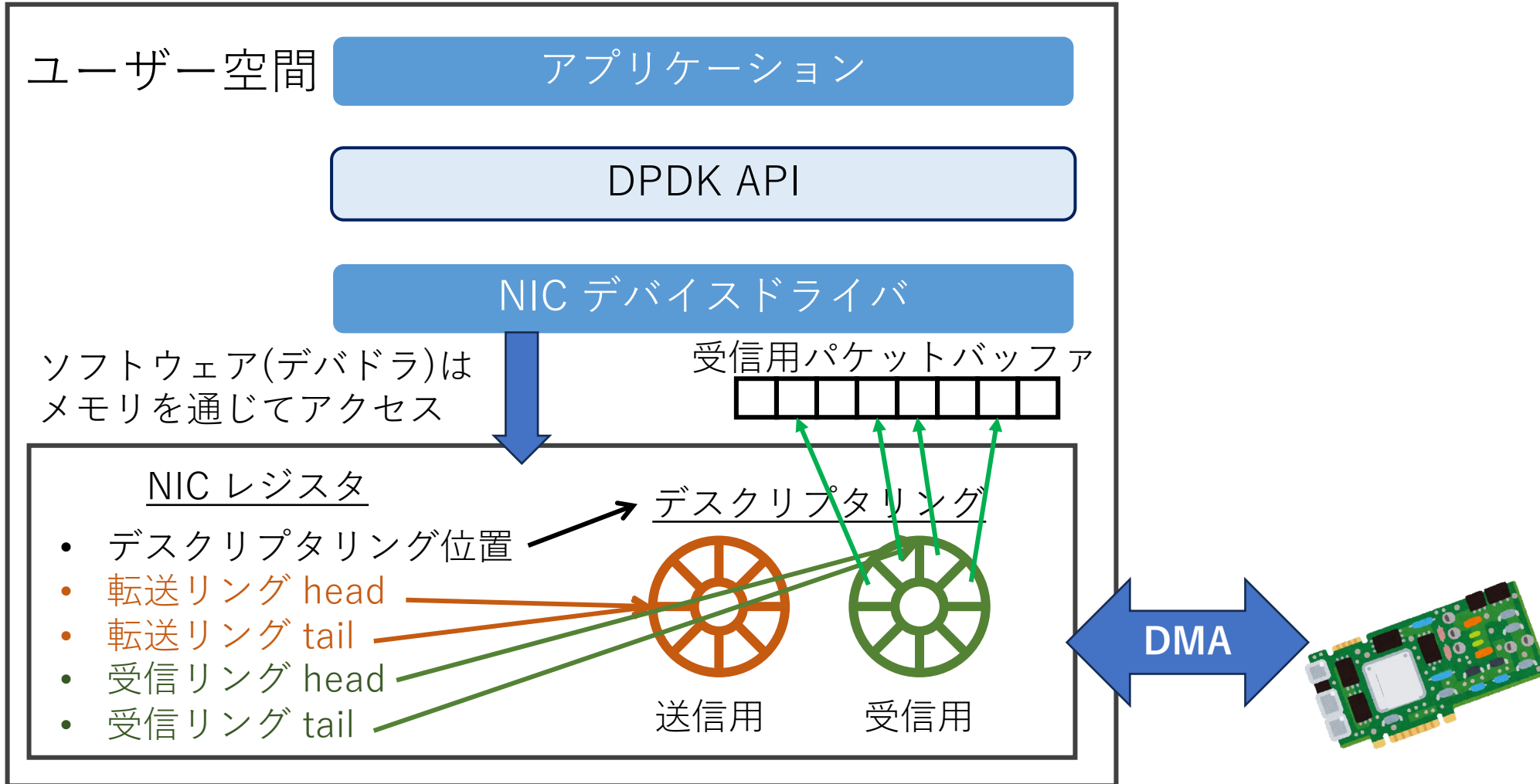
受信用パケットバッファはユーザー空間にあり、受信したデータは検知された段階で既にアプリケーションから見えている

なので、読み込みのために追加の作業はなし

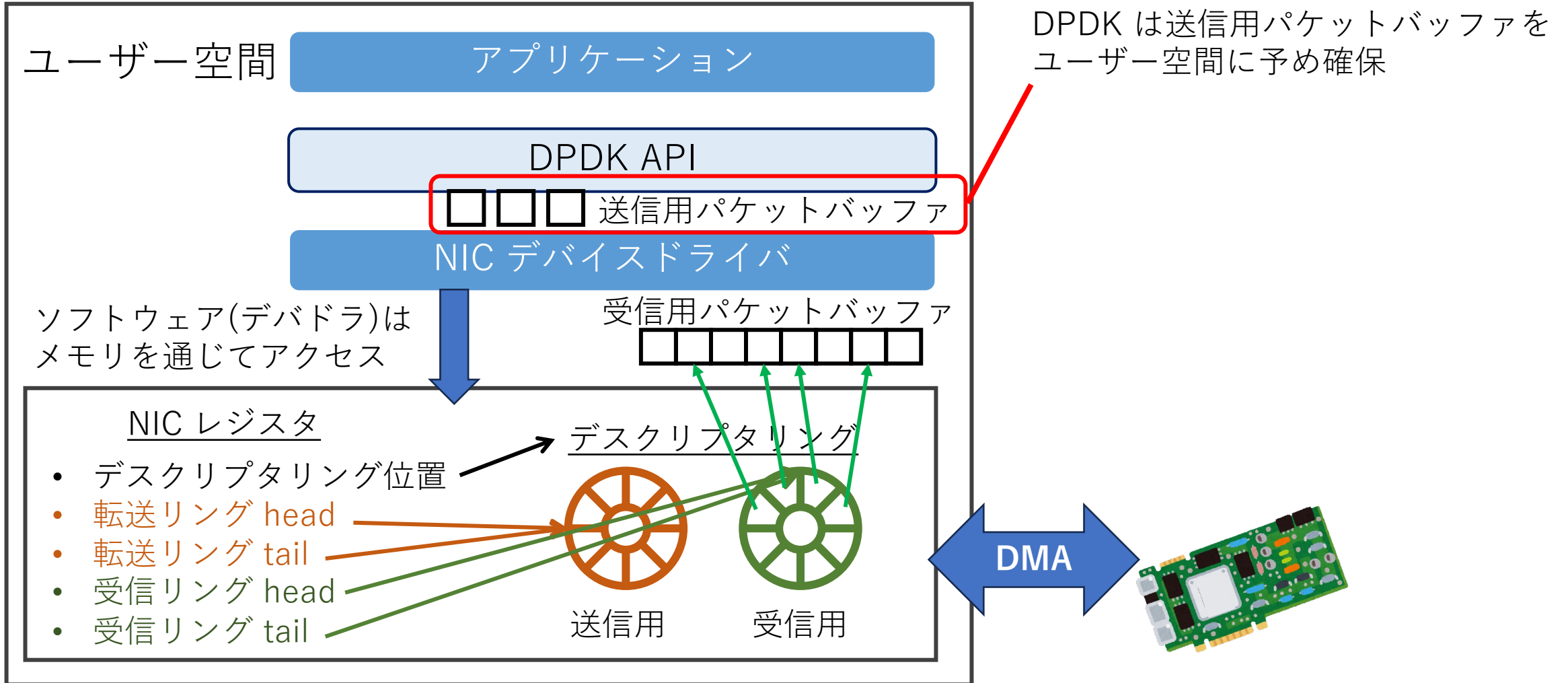
新しいパケットを受け取れるように別のバッファを紐づけて、tailを進める



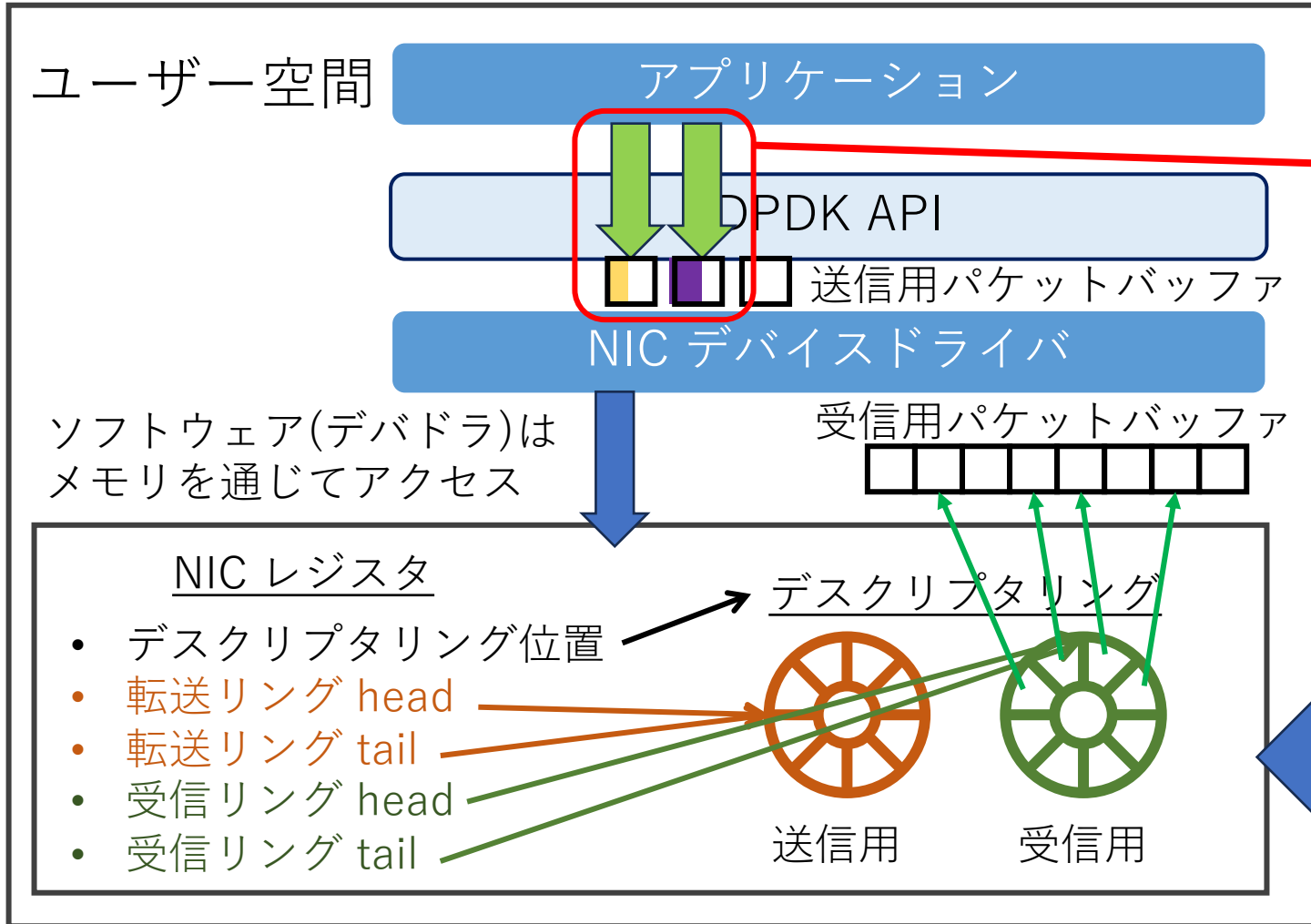
DPDK の場合：転送



DPDK の場合：転送

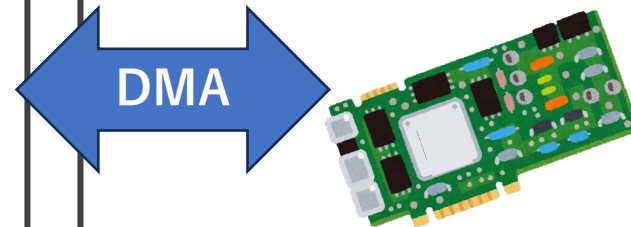


DPDK の場合：転送

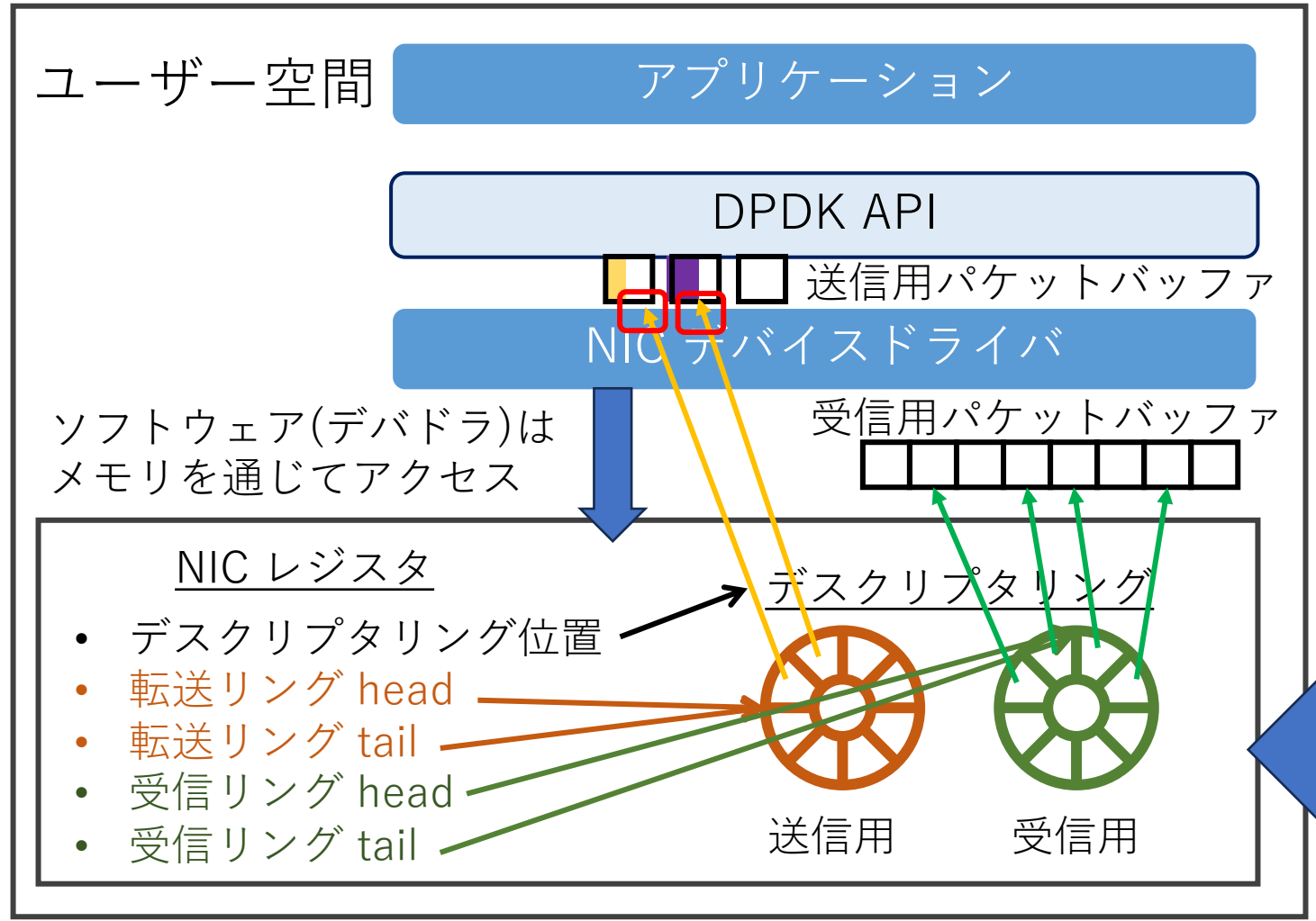


DPDK は送信用バッファをユーザー空間に予め確保

アプリケーションは確保された送信用バッファへ直接データを書き込む



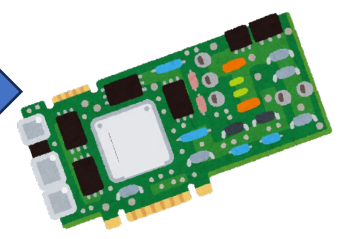
DPDK の場合：転送



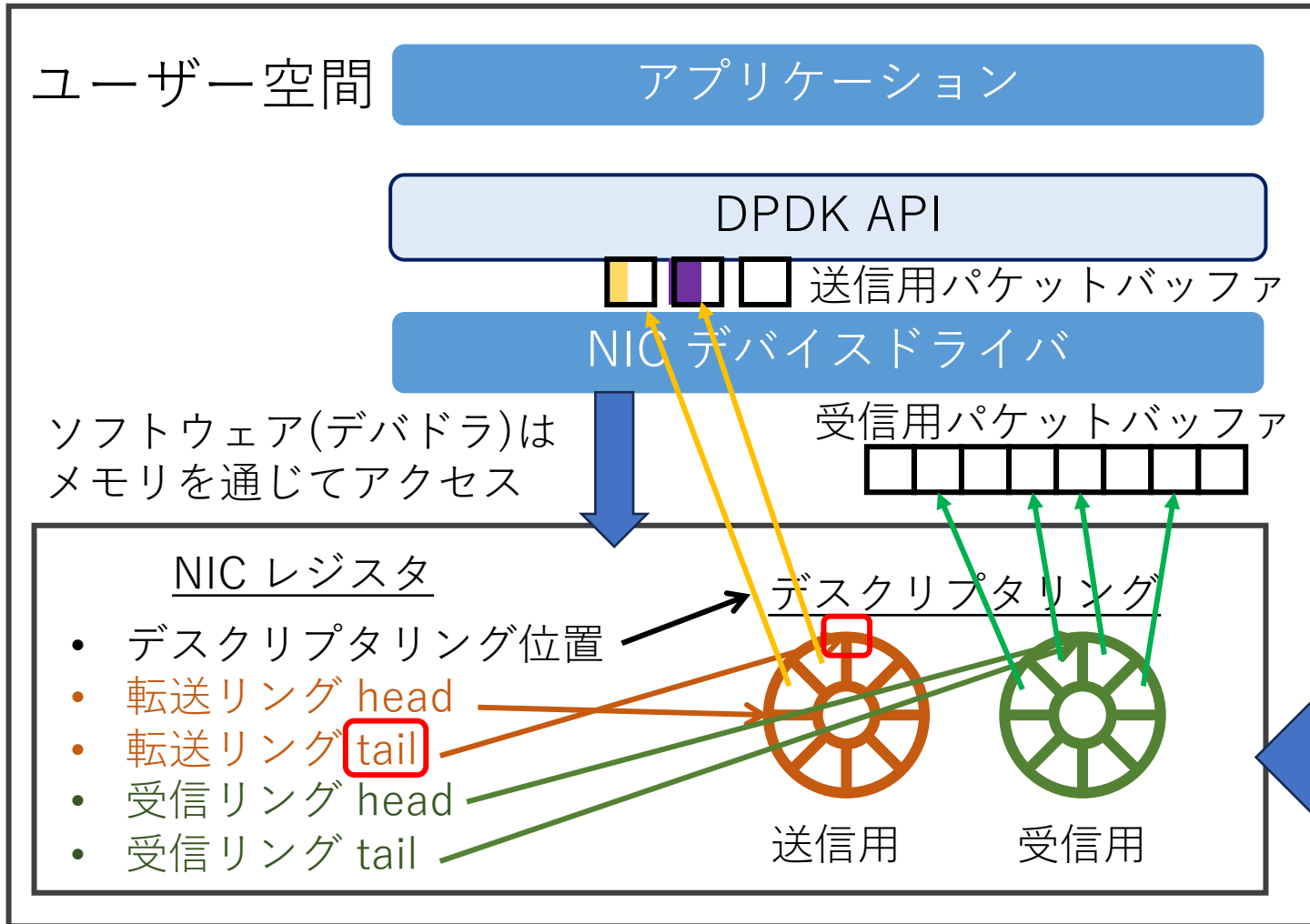
DPDK は送信用パケットバッファをユーザー空間に予め確保

アプリケーションは確保された送信用パケットバッファへ直接データを書き込む

DPDK は NIC のデスクリプタリングに送信用パケットバッファを紐付け



DPDK の場合：転送



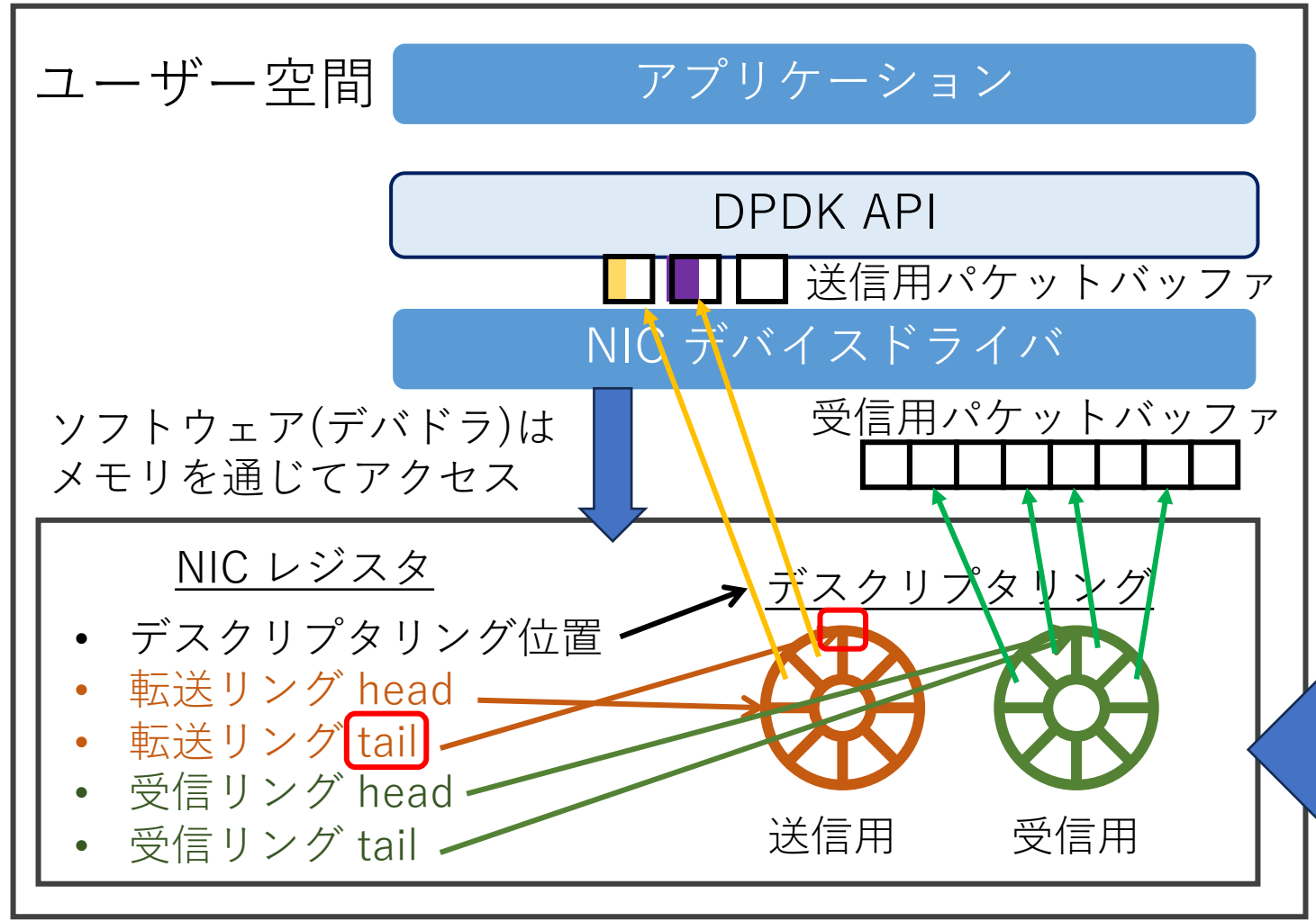
DPDK は送信用バッファをユーザー空間に予め確保

アプリケーションは確保された送信用バッファへ直接データを書き込む

DPDK は NIC のデスクリプタリングに送信用バッファを紐付け

その後、転送リングの tail を更新

DPDK の場合：転送



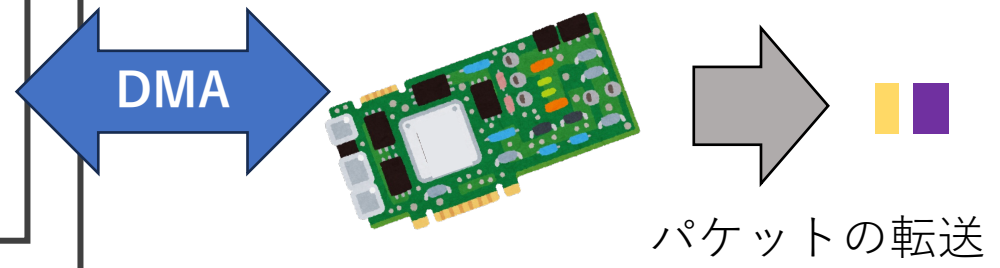
DPDK は送信用バッファをユーザー空間に予め確保

アプリケーションは確保された送信用バッファへ直接データを書き込む

DPDK は NIC のデスクリプタリングに送信用バッファを紐付け

その後、転送リングの tail を更新

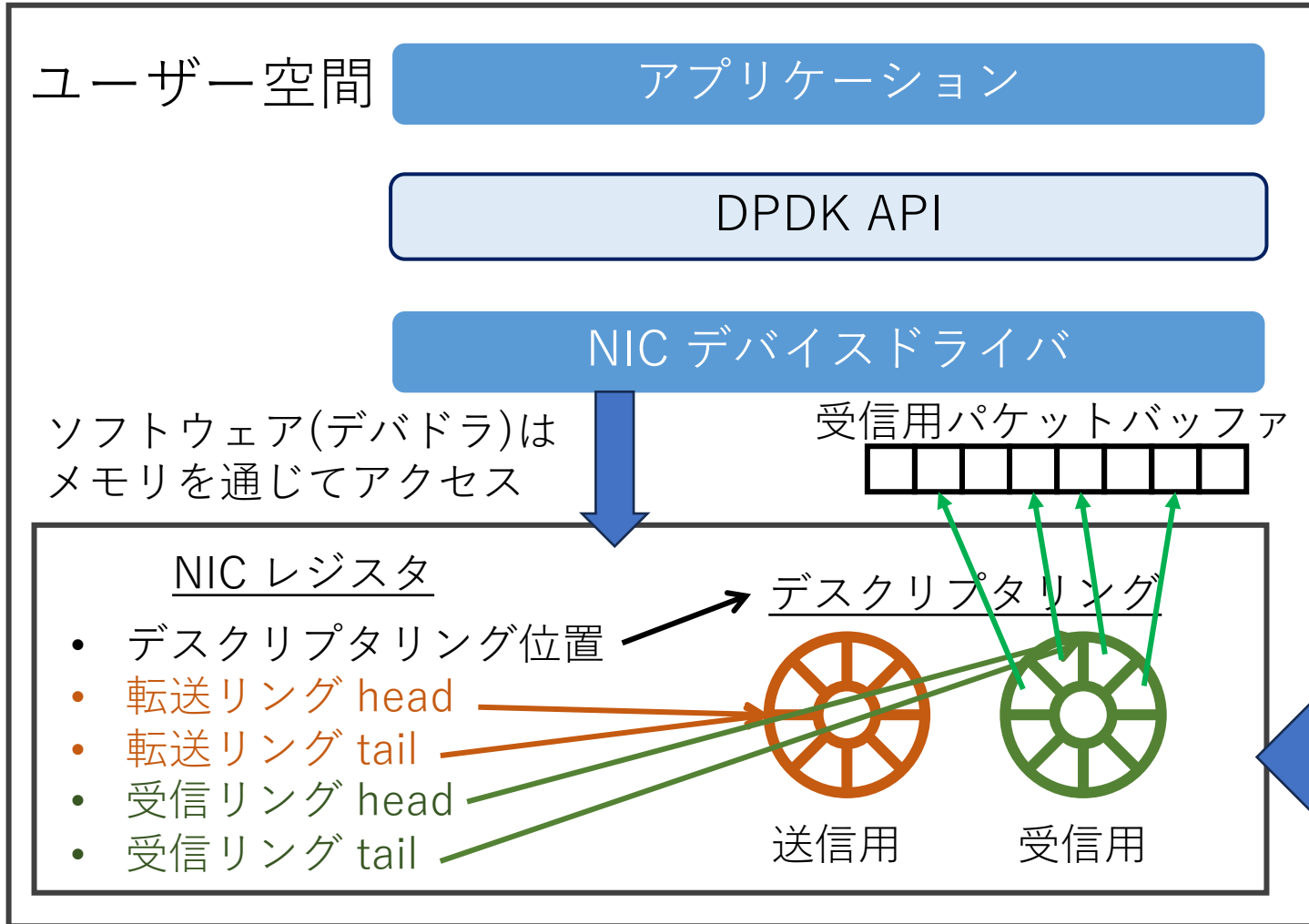
これをきっかけにパッケージが NIC から転送される



DPDK の場合：削減できるコスト

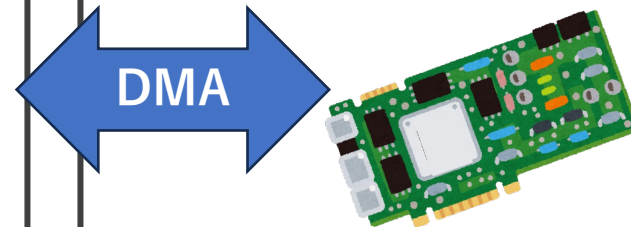
- 受信において、ハードウェア割り込みを起点としたカーネルスレッドの起動に伴うスケジューリング
- 受信において、ユーザー空間プロセスへの新規データの通知に伴うスケジューリング
- プロトコルスタック内の処理
- システムコール呼び出し
- ユーザー空間とカーネルの間での送受信に伴うメモリコピー

パケット I/O フレームワークの用途

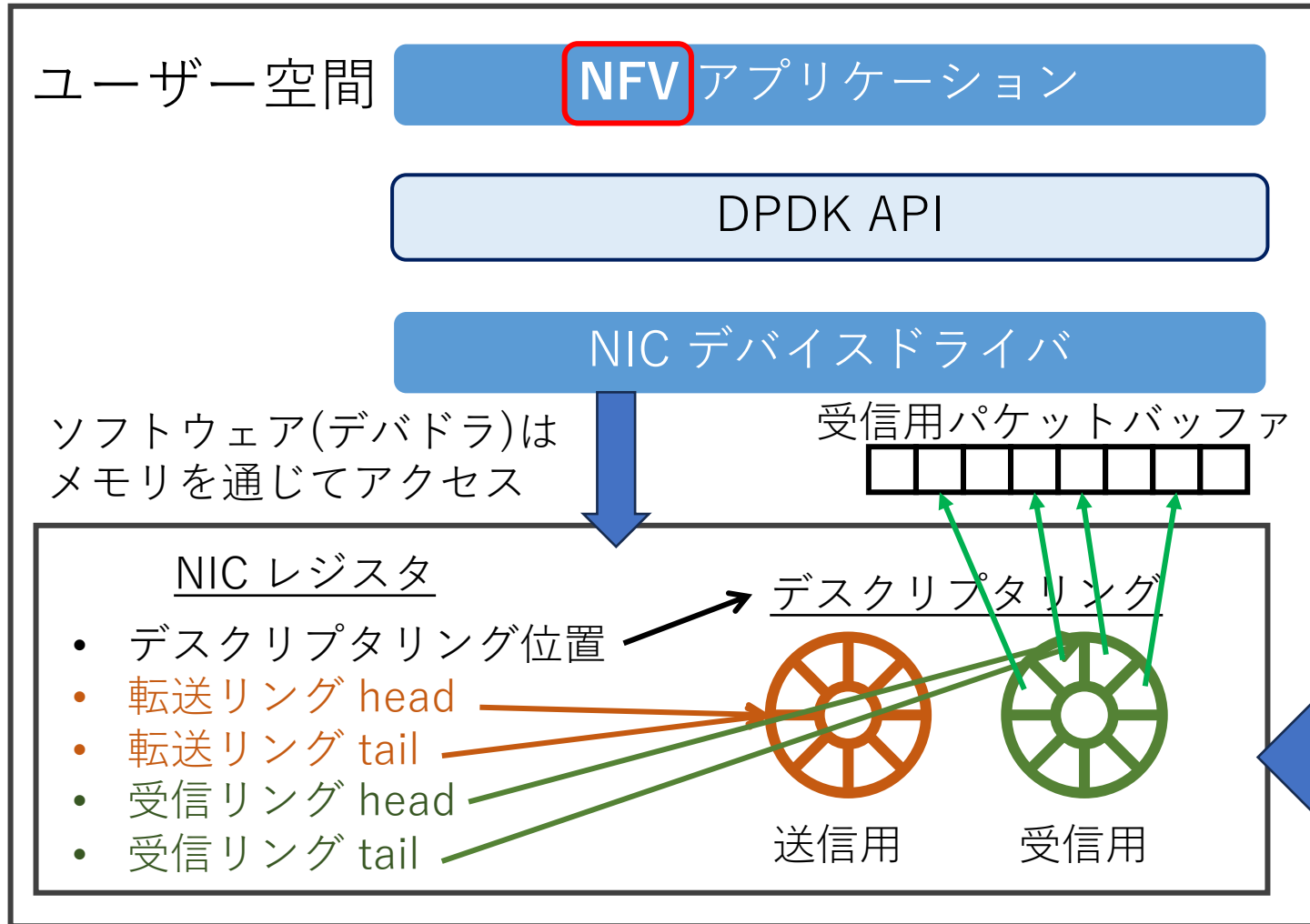


Network Function Virtualization (NFV)

汎用的なサーバーでネットワーク機能を動かす (e.g., Firewall, Router)

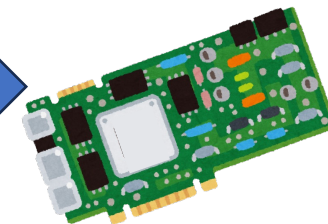


パケット I/O フレームワークの用途

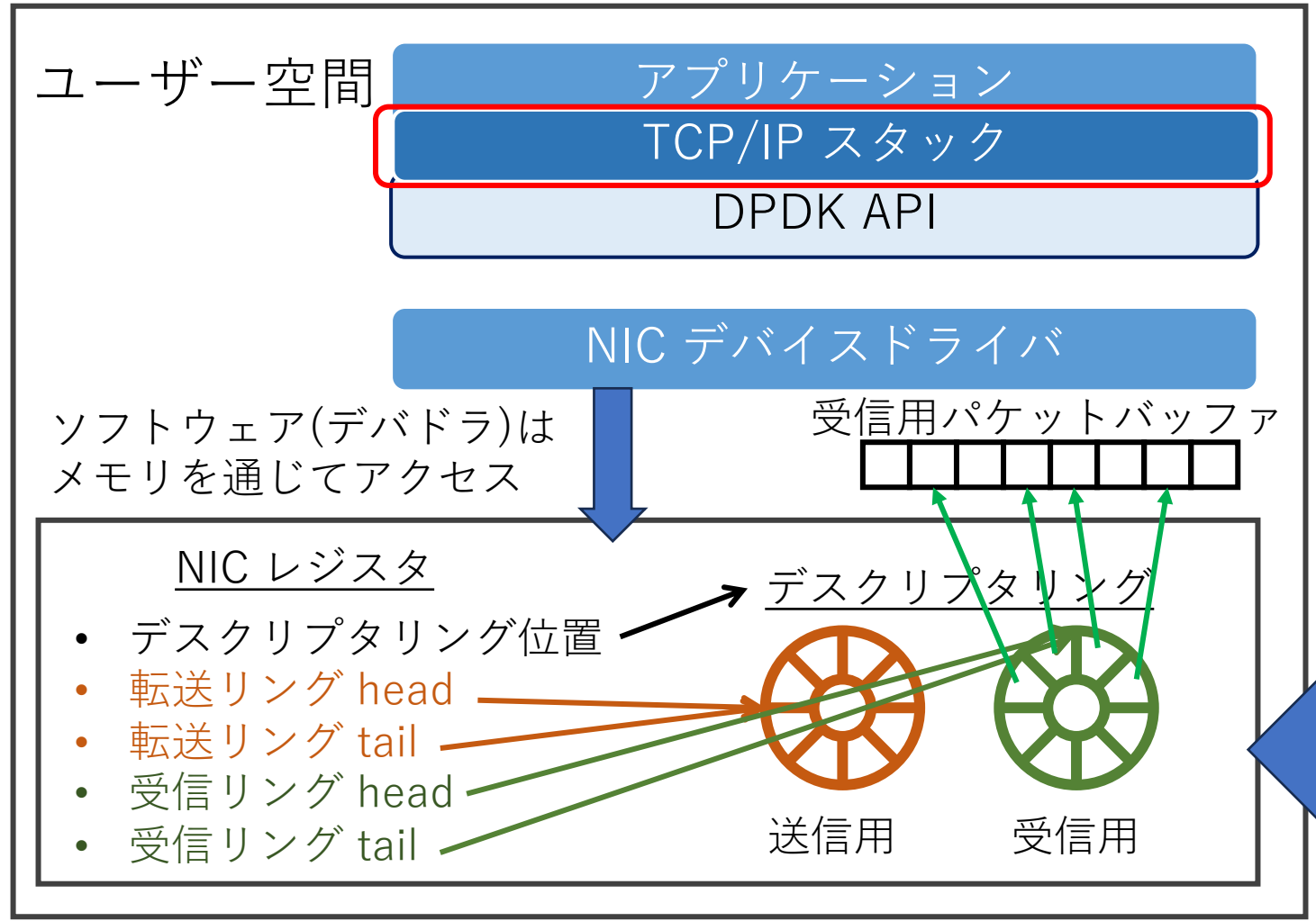


Network Function Virtualization (NFV)

汎用的なサーバーでネットワーク機能を動かす (e.g., Firewall, Router)



パケット I/O フレームワークの用途

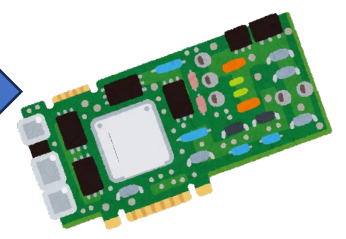


Network Function Virtualization (NFV)

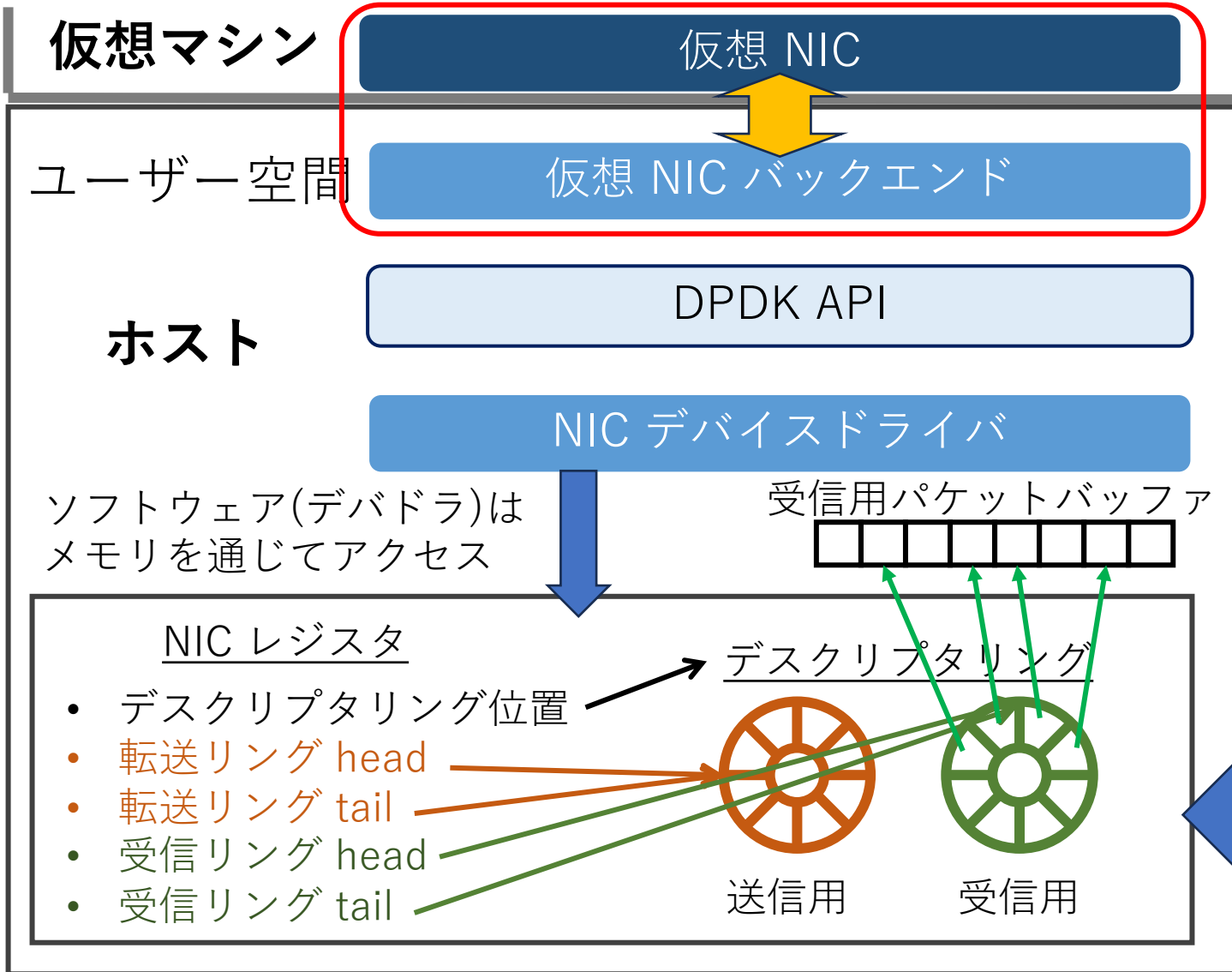
汎用的なサーバーでネットワーク機能を動かす (e.g., Firewall, Router)

サーバープログラムの高速化

ユーザー空間で動作する TCP/IP スタックと組み合わせる



パケット I/O フレームワークの用途



Network Function Virtualization (NFV)

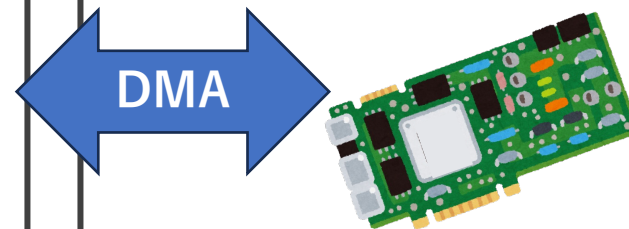
汎用的なサーバーでネットワーク機能を動かす (e.g., Firewall, Router)

サーバープログラムの高速化

ユーザー空間で動作する TCP/IP スタックと組み合わせる

仮想マシン通信の高速化

仮想 I/O バックエンドに組み込む



研究紹介

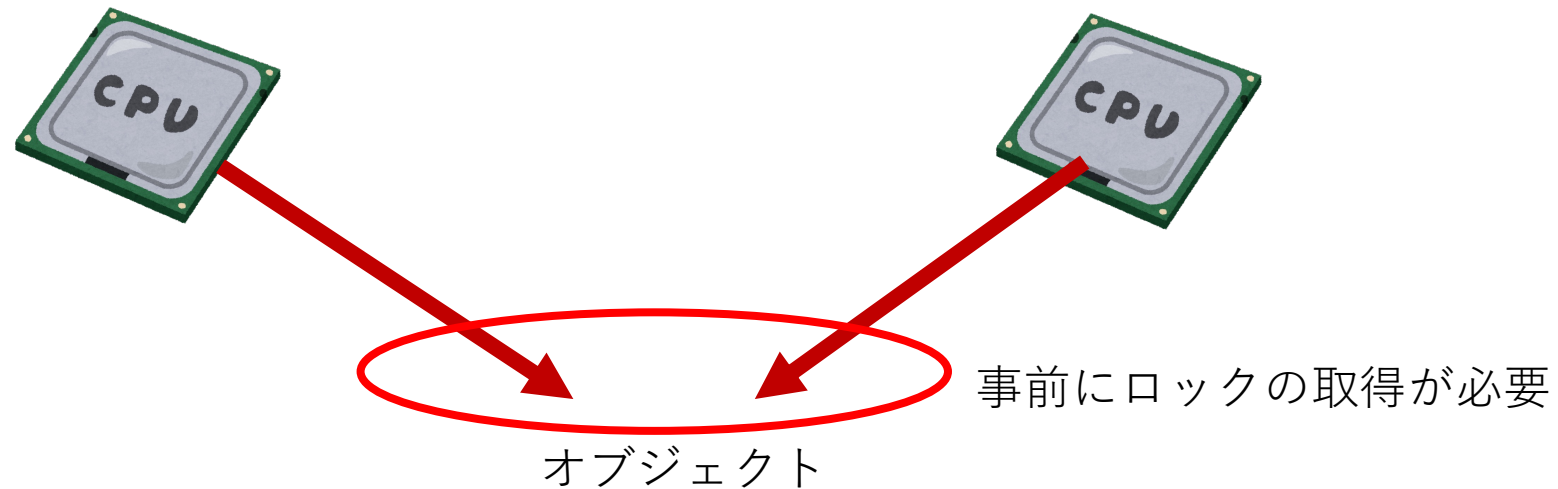
TCP/IP スタック設計

マルチコア環境でのスケーラビリティについて

基本的なハードウェア機能の説明

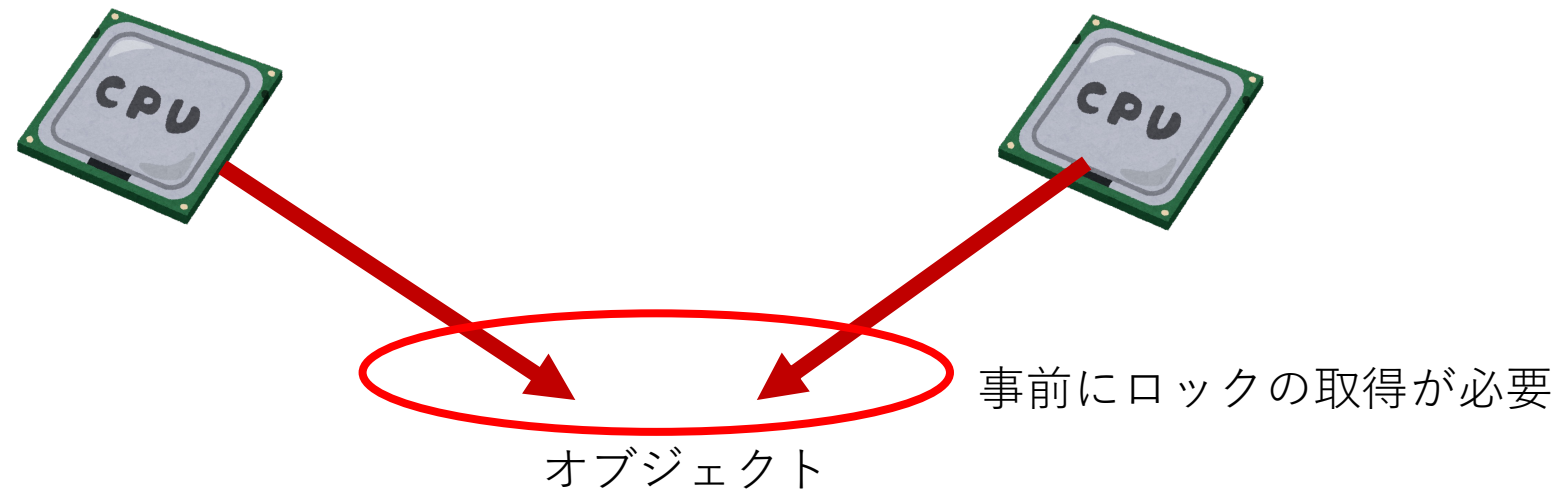
TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - CPU コア間で共有されるオブジェクトのアクセスにはロックの取得が必要 => ロック取得待機時間がボトルネックになる

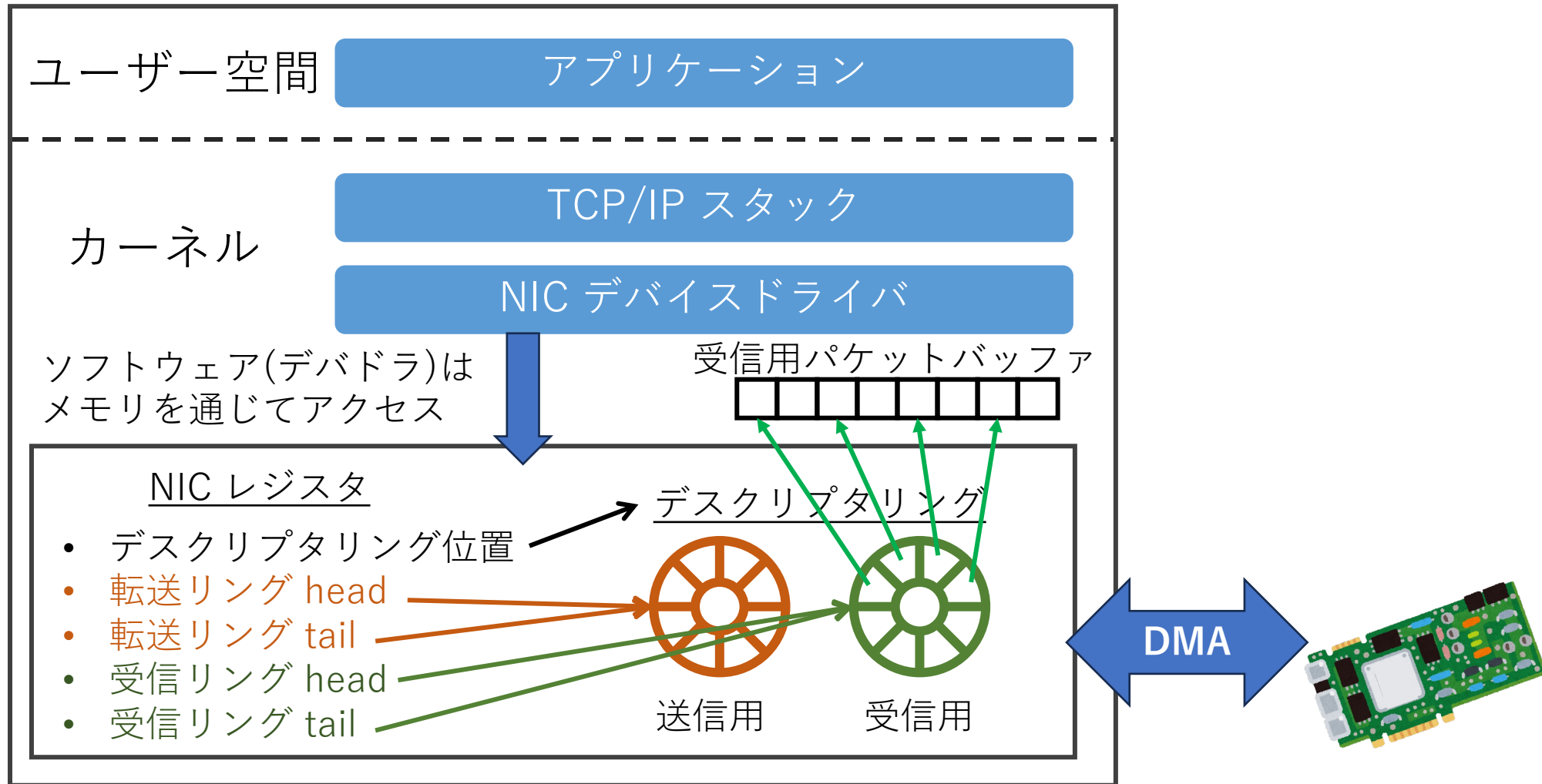


TCP/IP スタック設計の再考

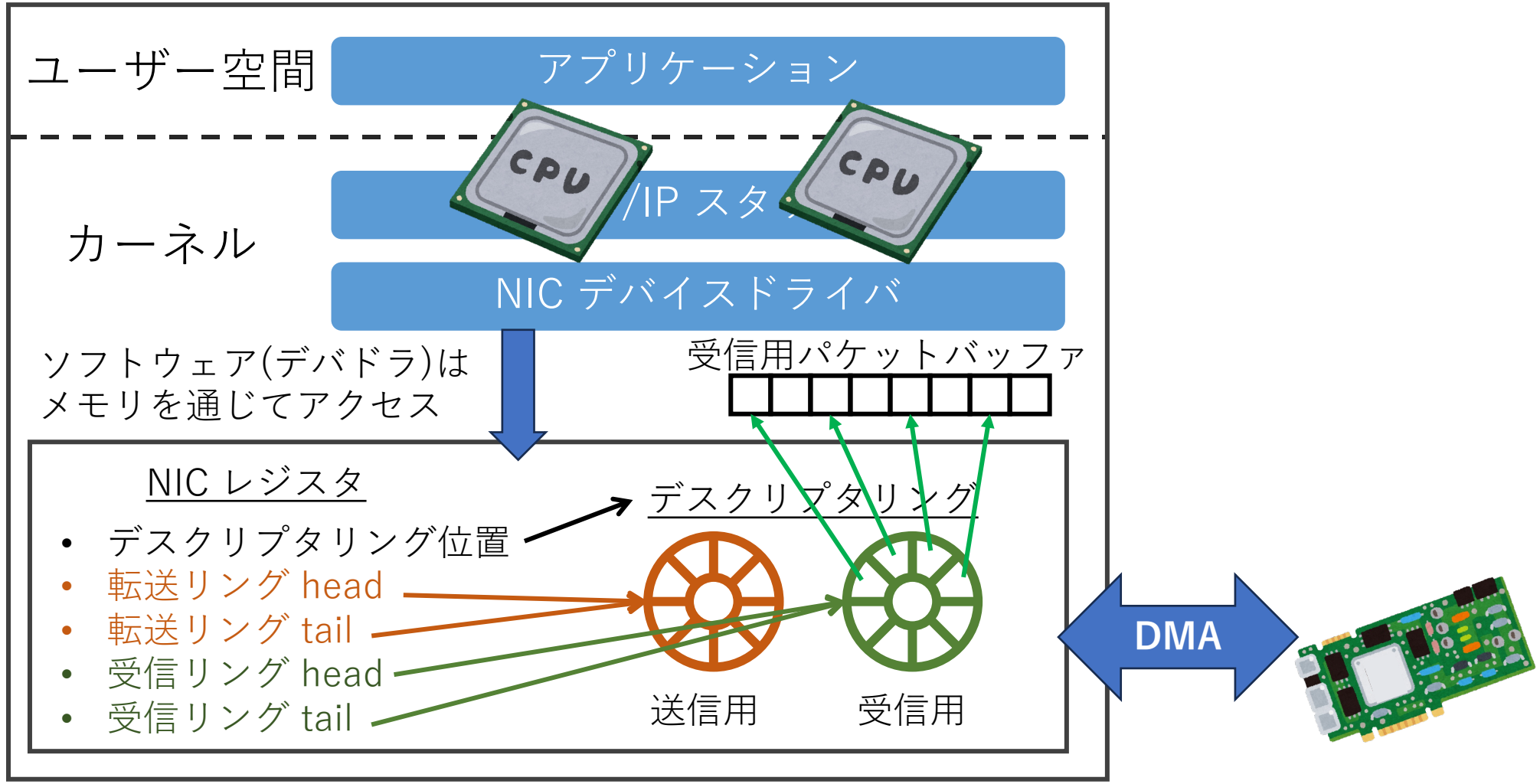
- マルチコア環境で性能をスケールさせる
 - CPU コア間で共有されるオブジェクトのアクセスにはロックの取得が必要 => ロック取得待機時間がボトルネックになる
 - 基本的なアイデア：CPU コア間で共有するオブジェクトを減らす



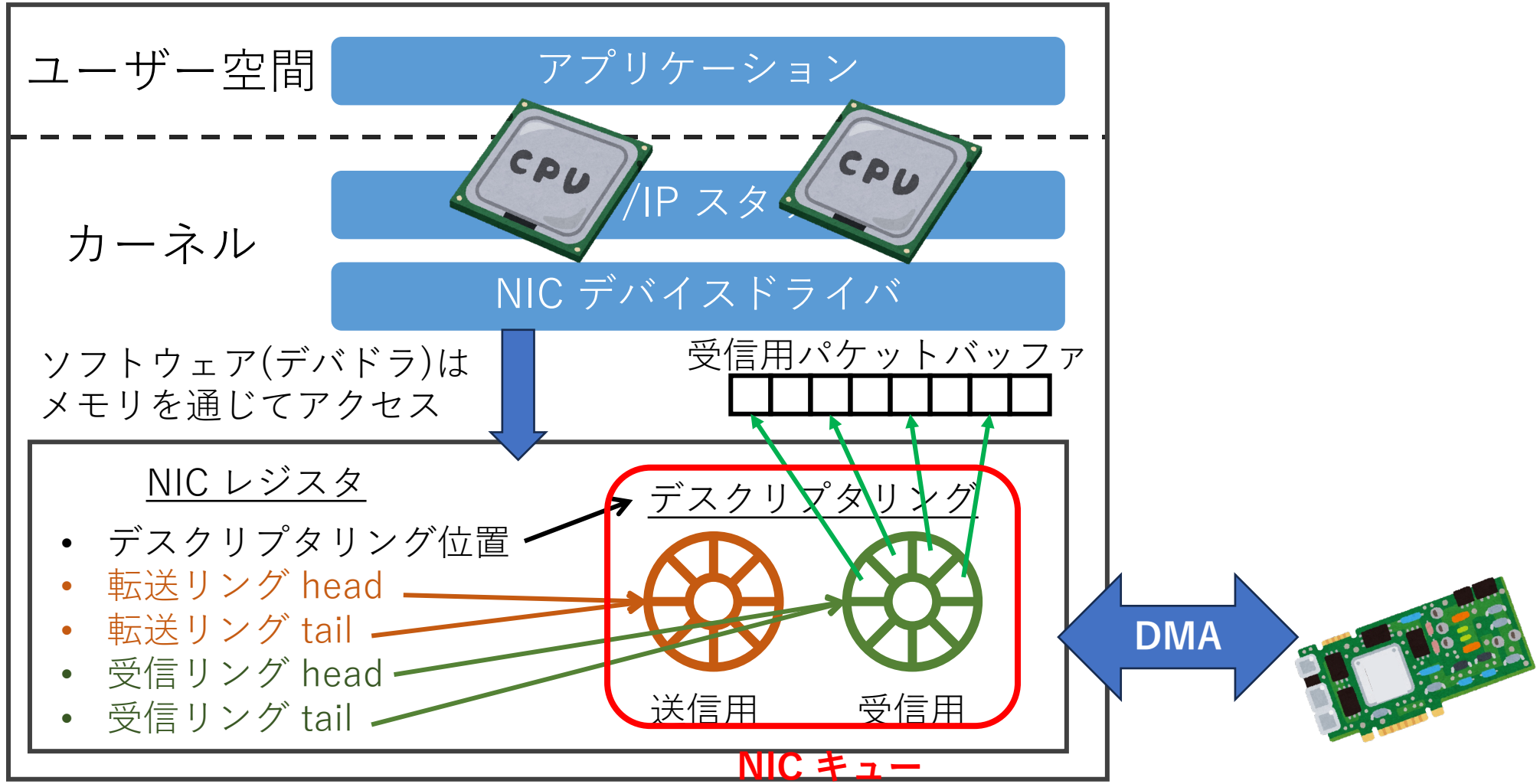
TCP/IP スタック設計の再考



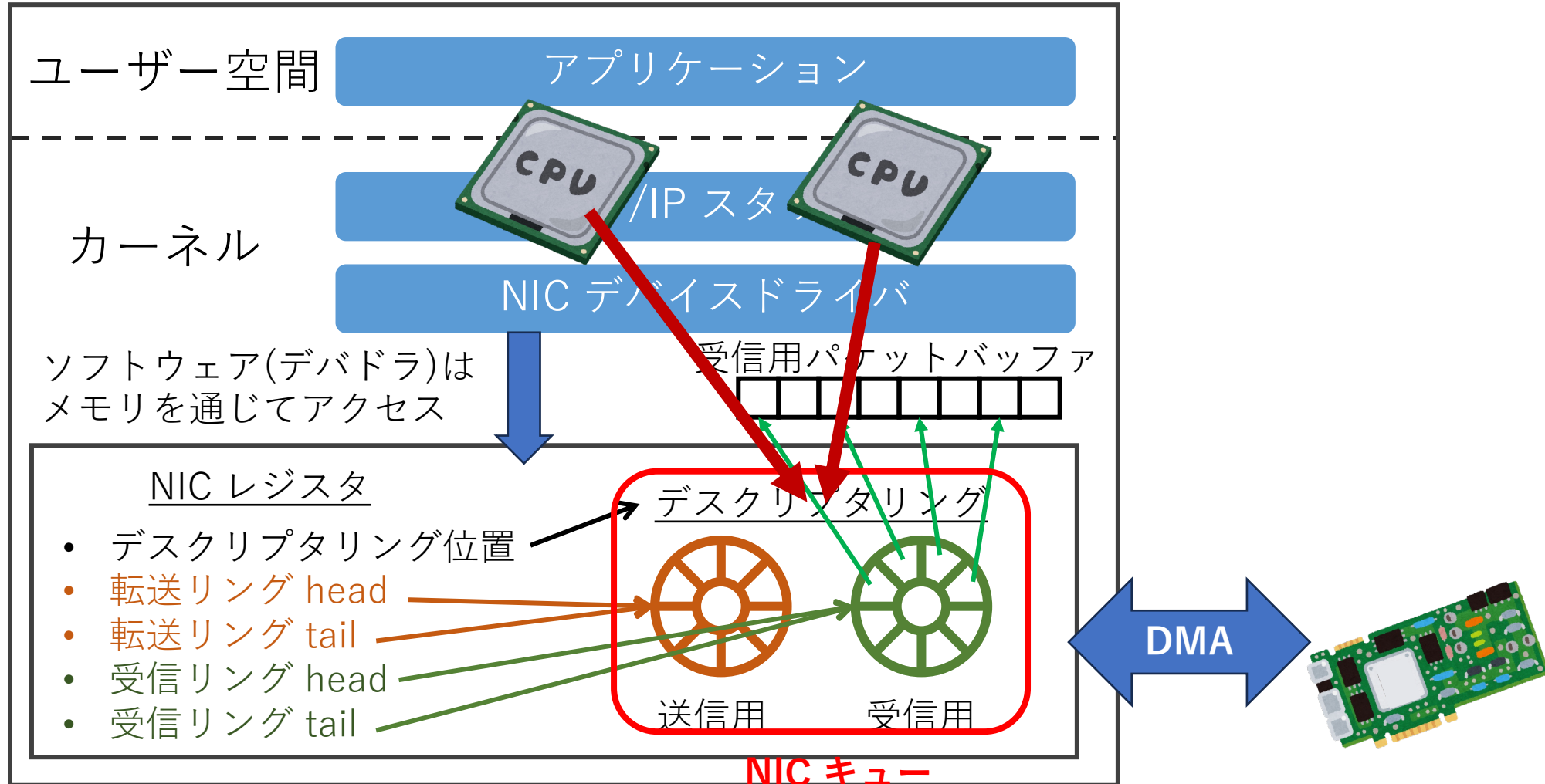
TCP/IP スタック設計の再考



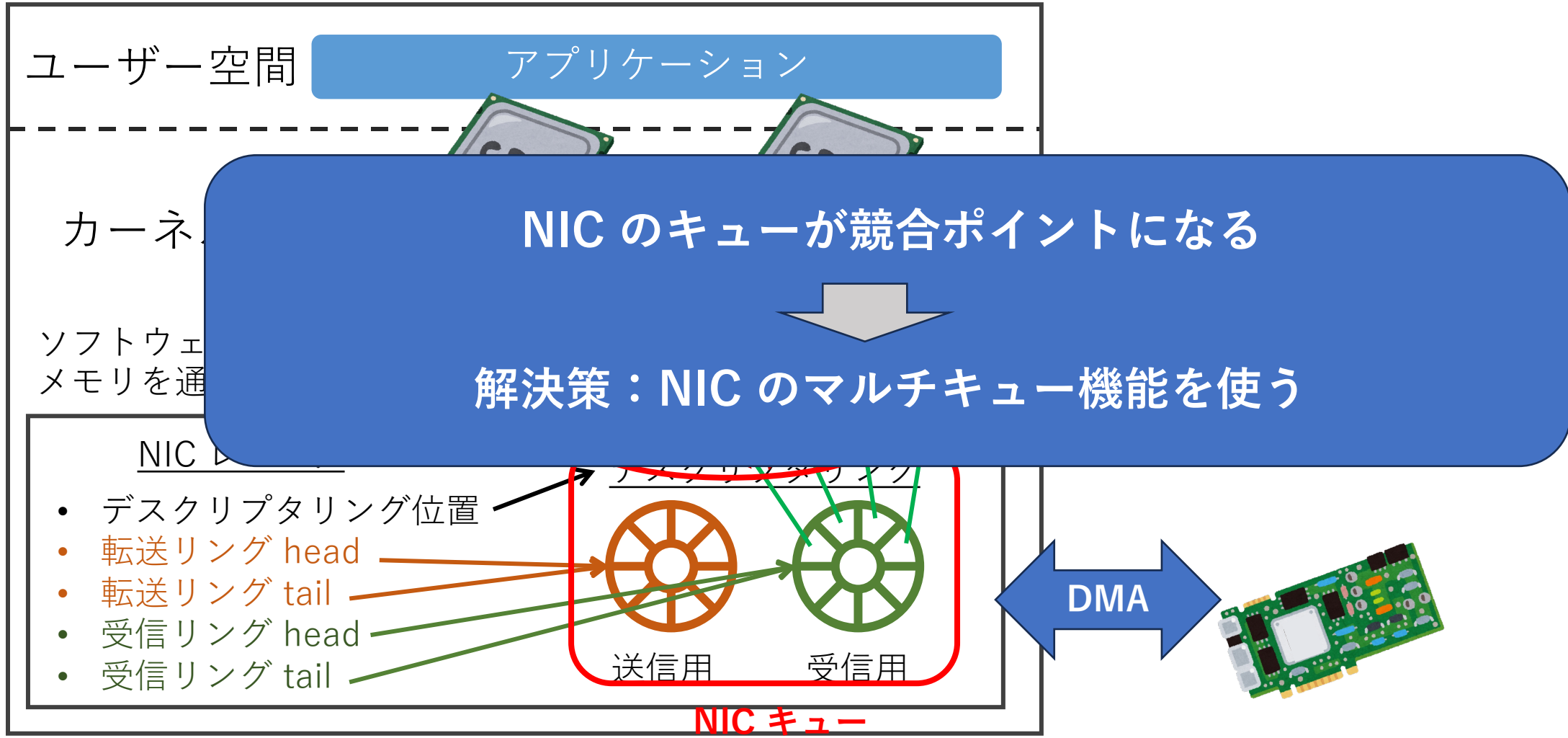
TCP/IP スタック設計の再考



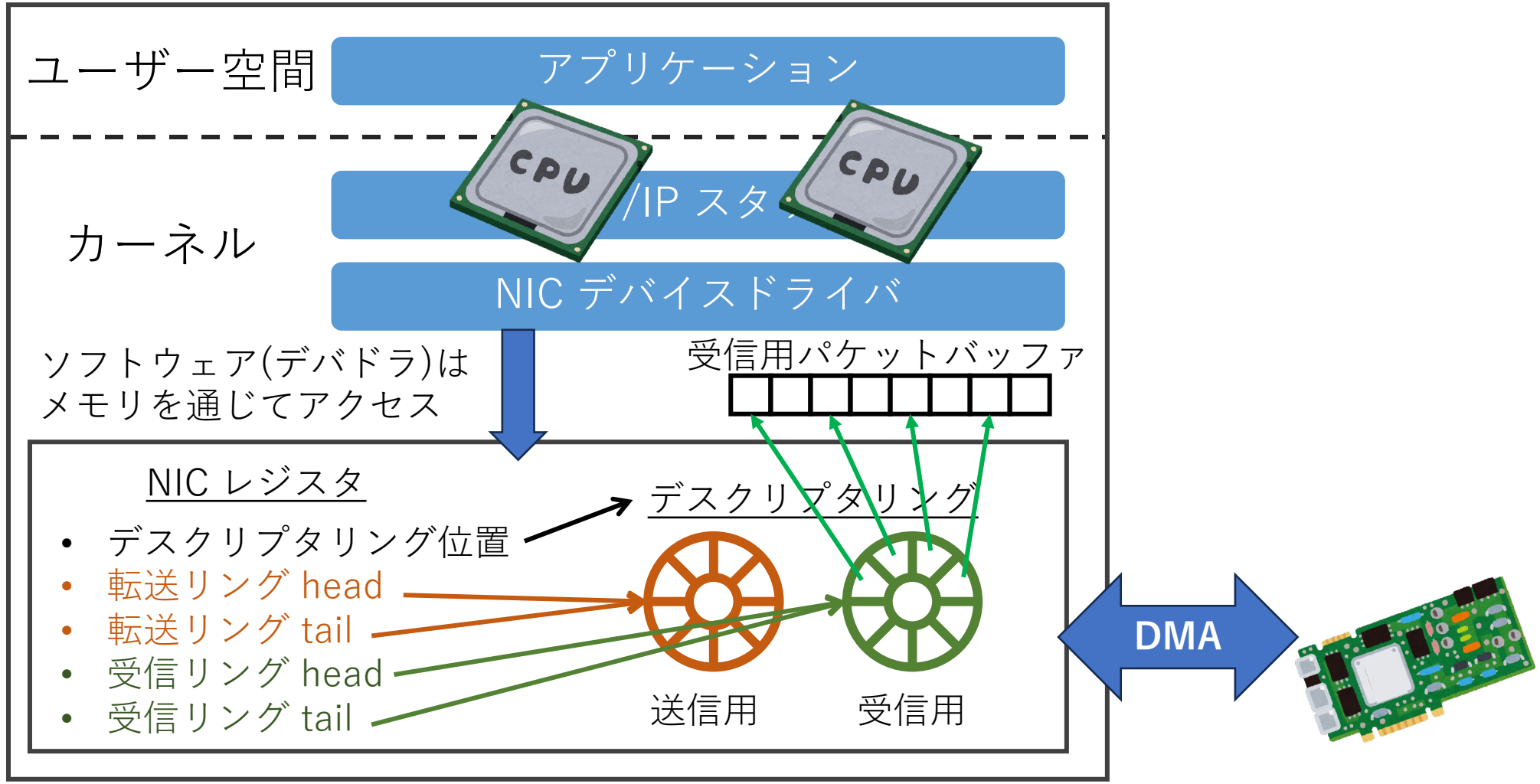
TCP/IP スタック設計の再考



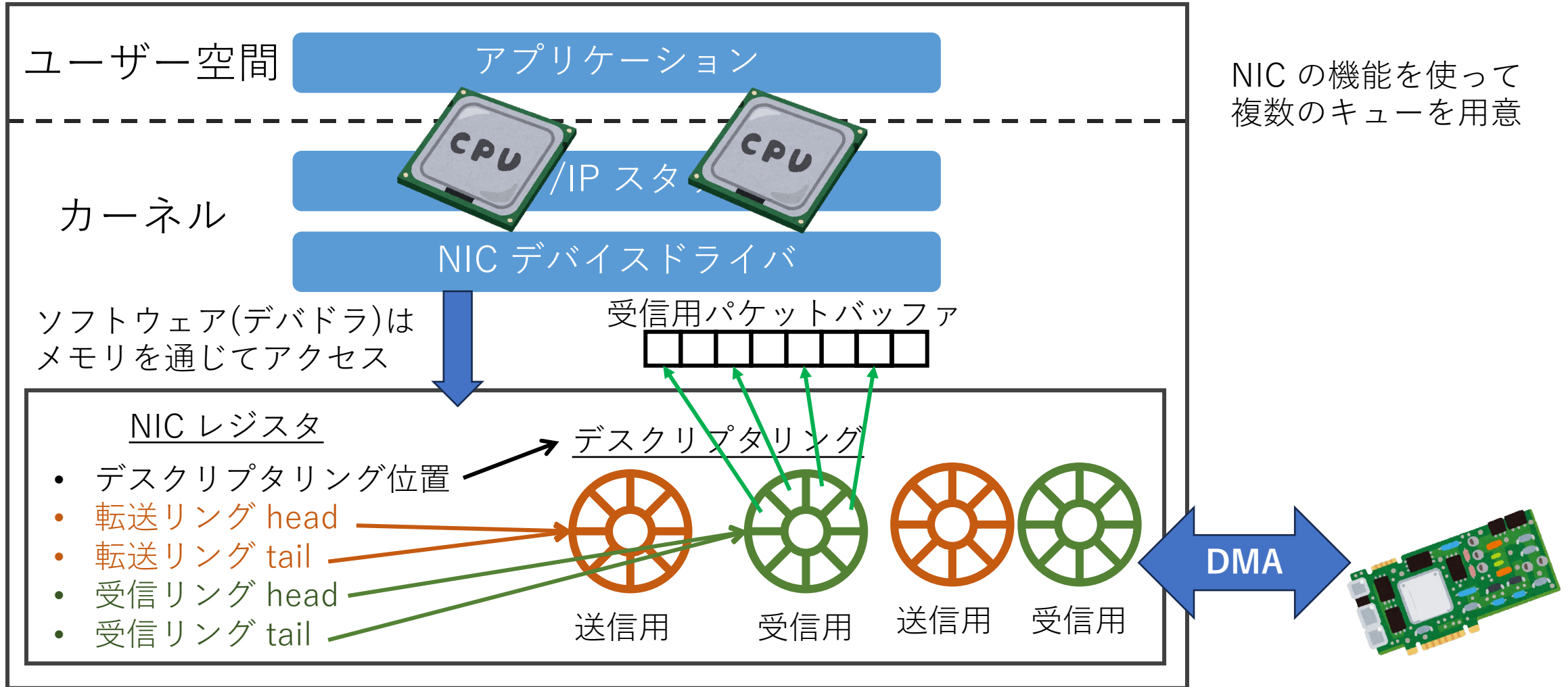
TCP/IP スタック設計の再考



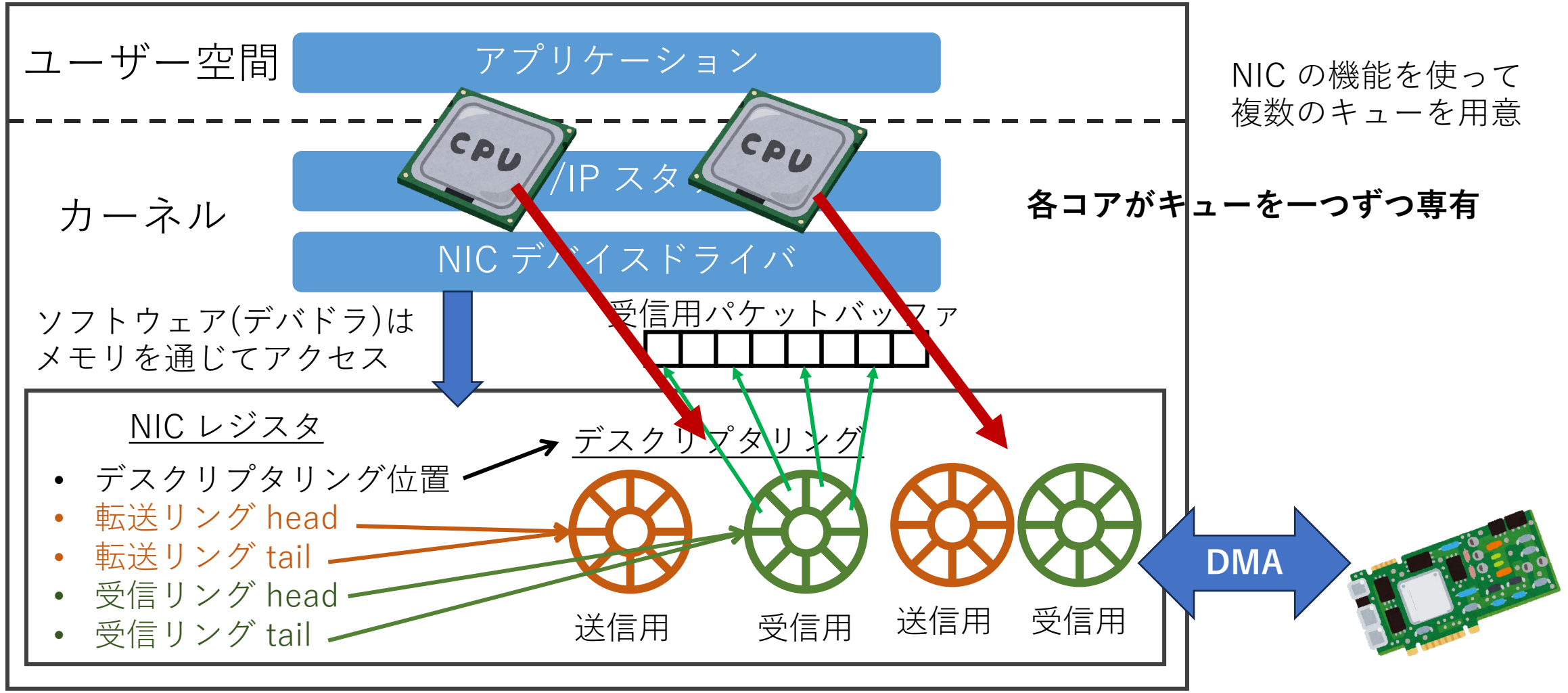
TCP/IP スタック設計の再考



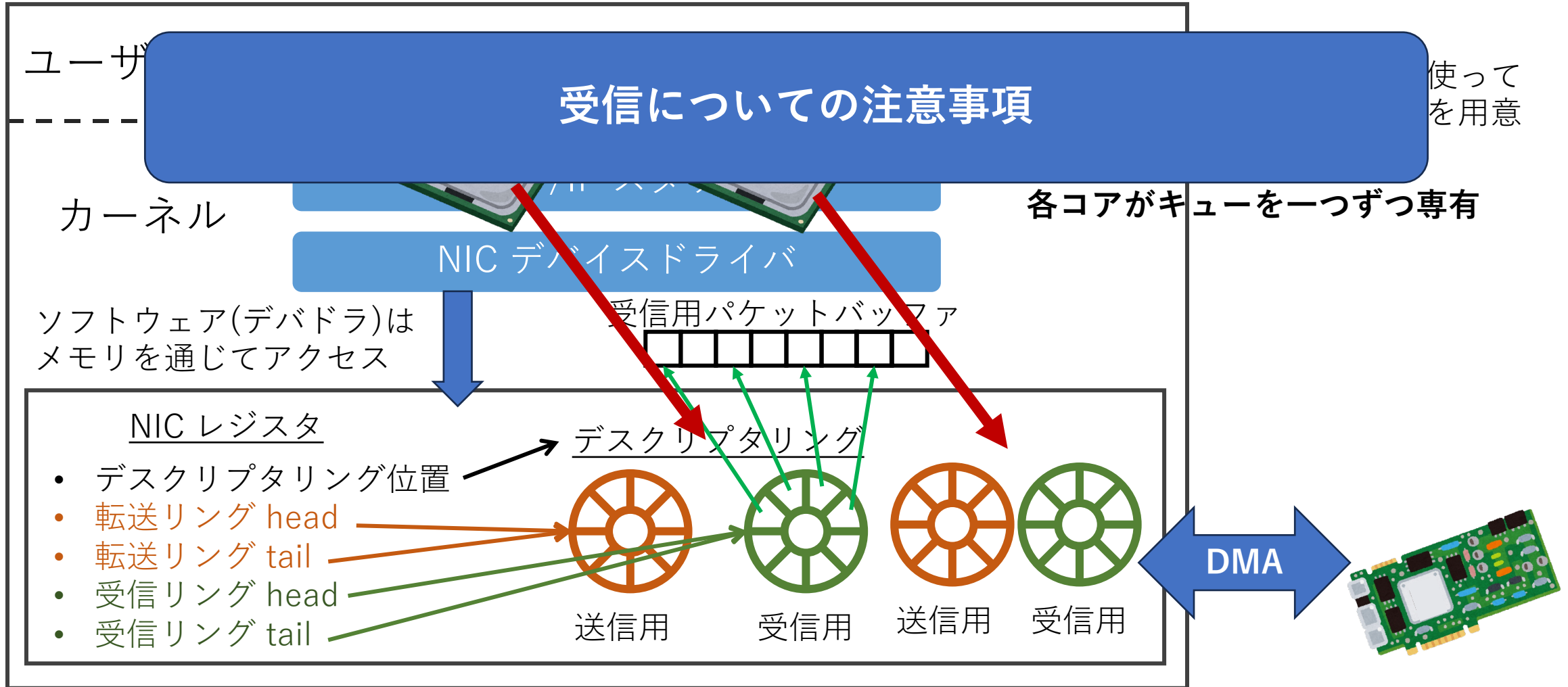
TCP/IP スタック設計の再考



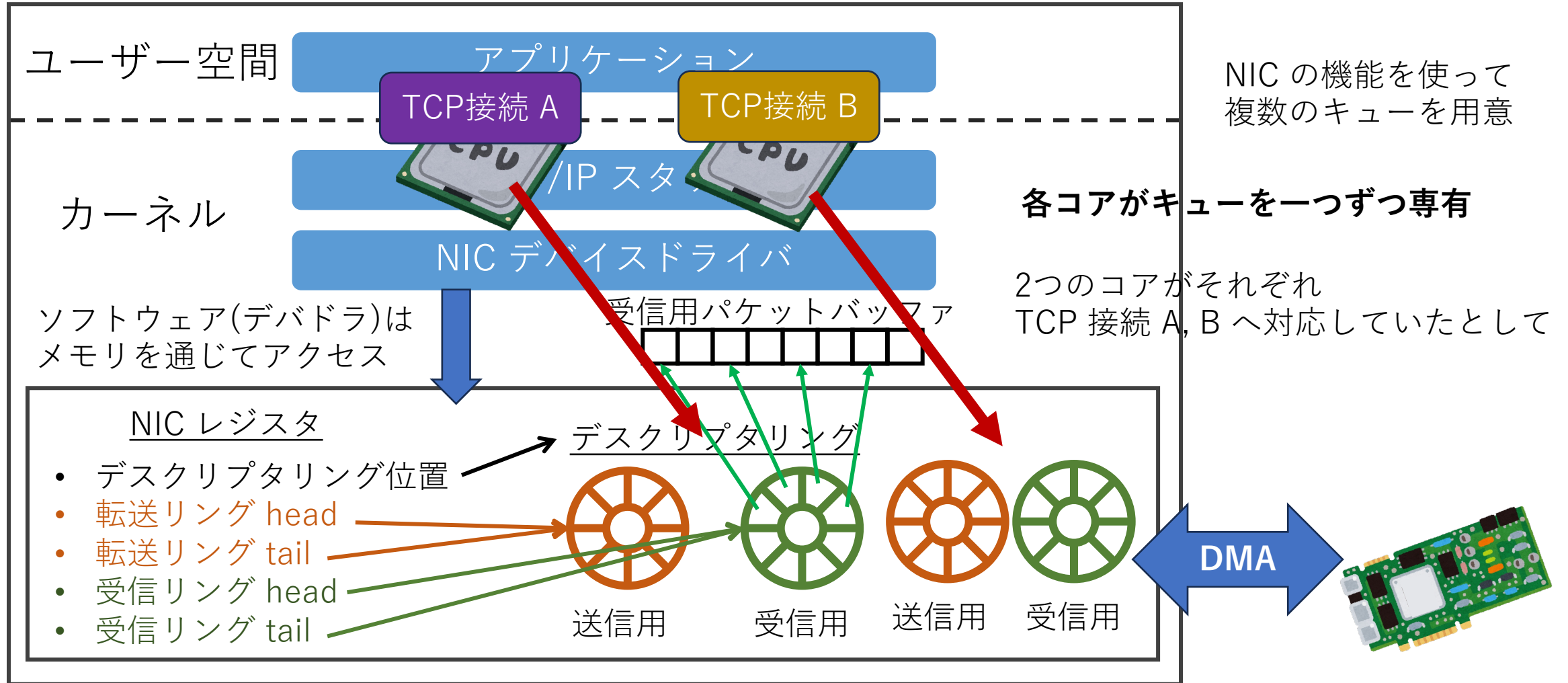
TCP/IP スタック設計の再考



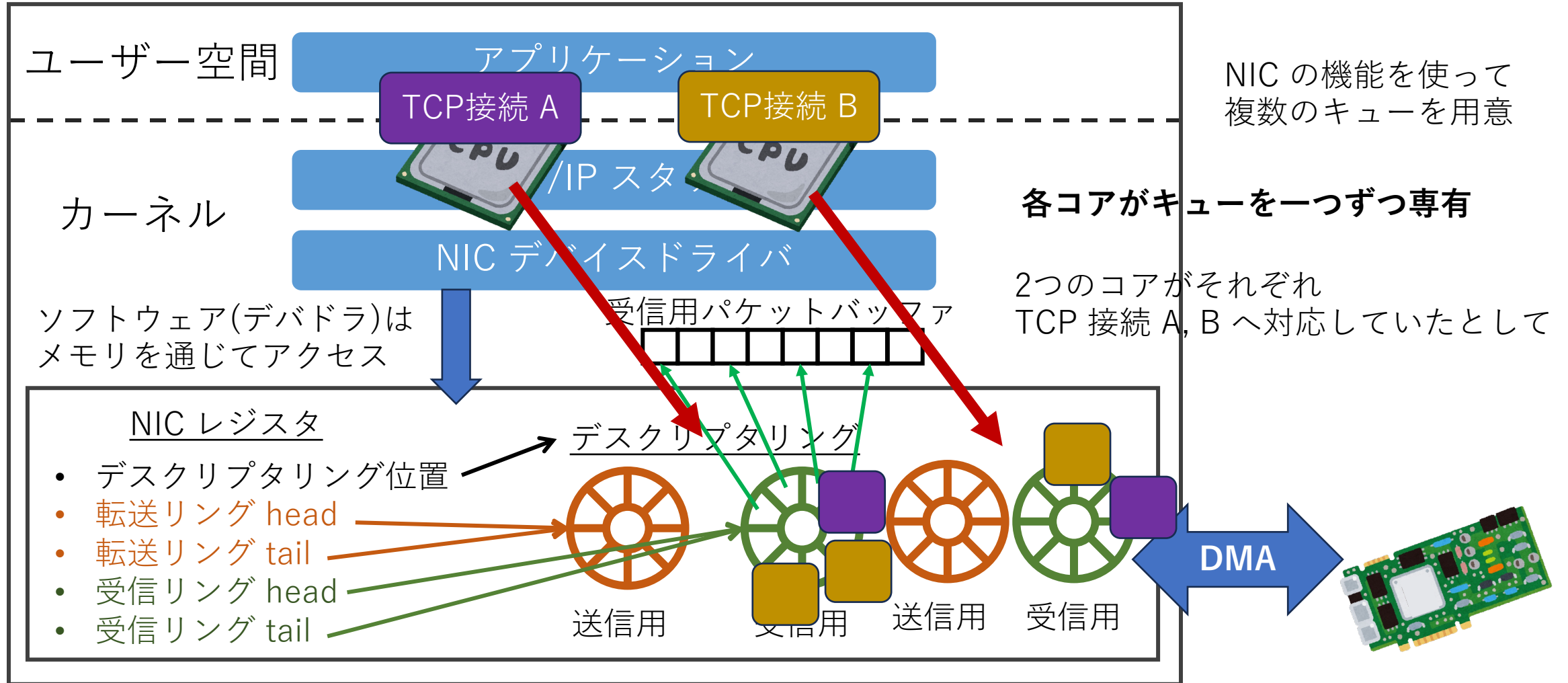
TCP/IP スタック設計の再考



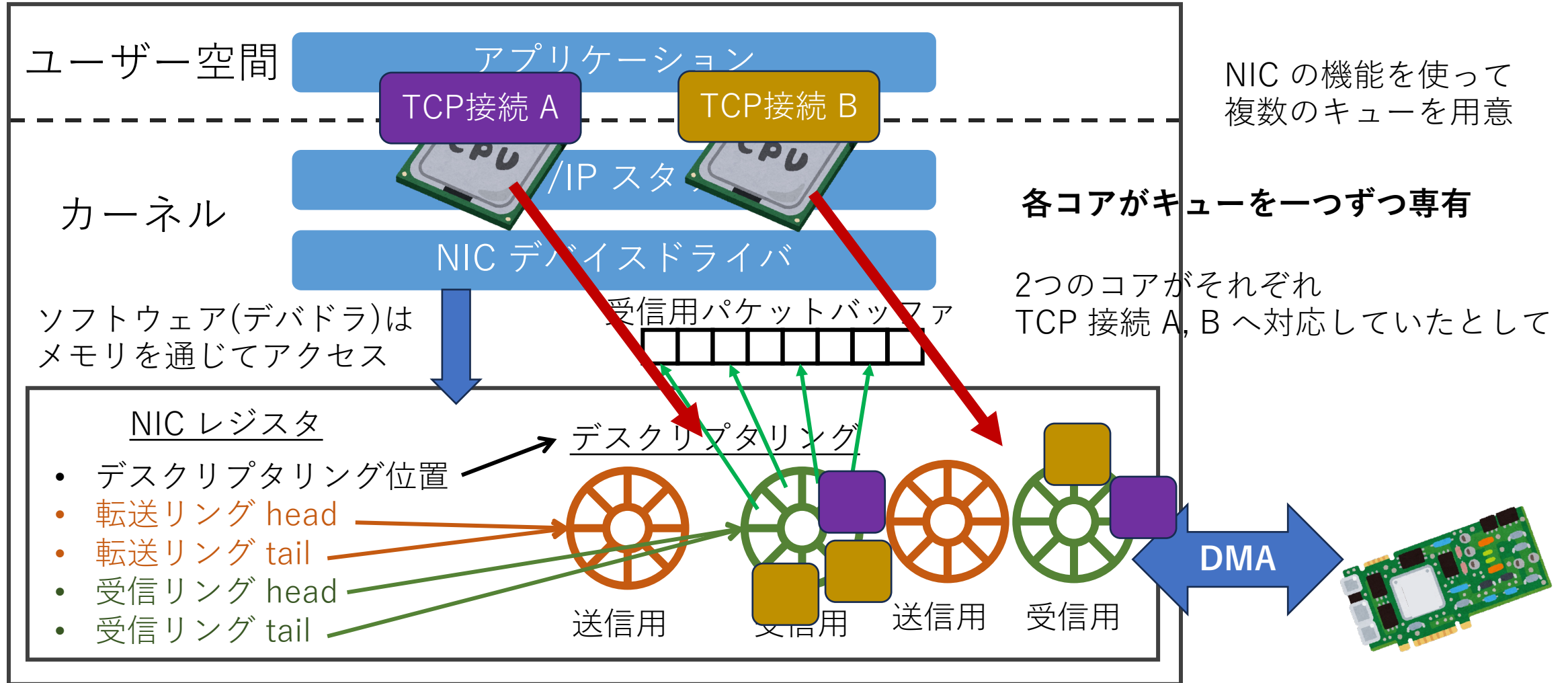
TCP/IP スタック設計の再考



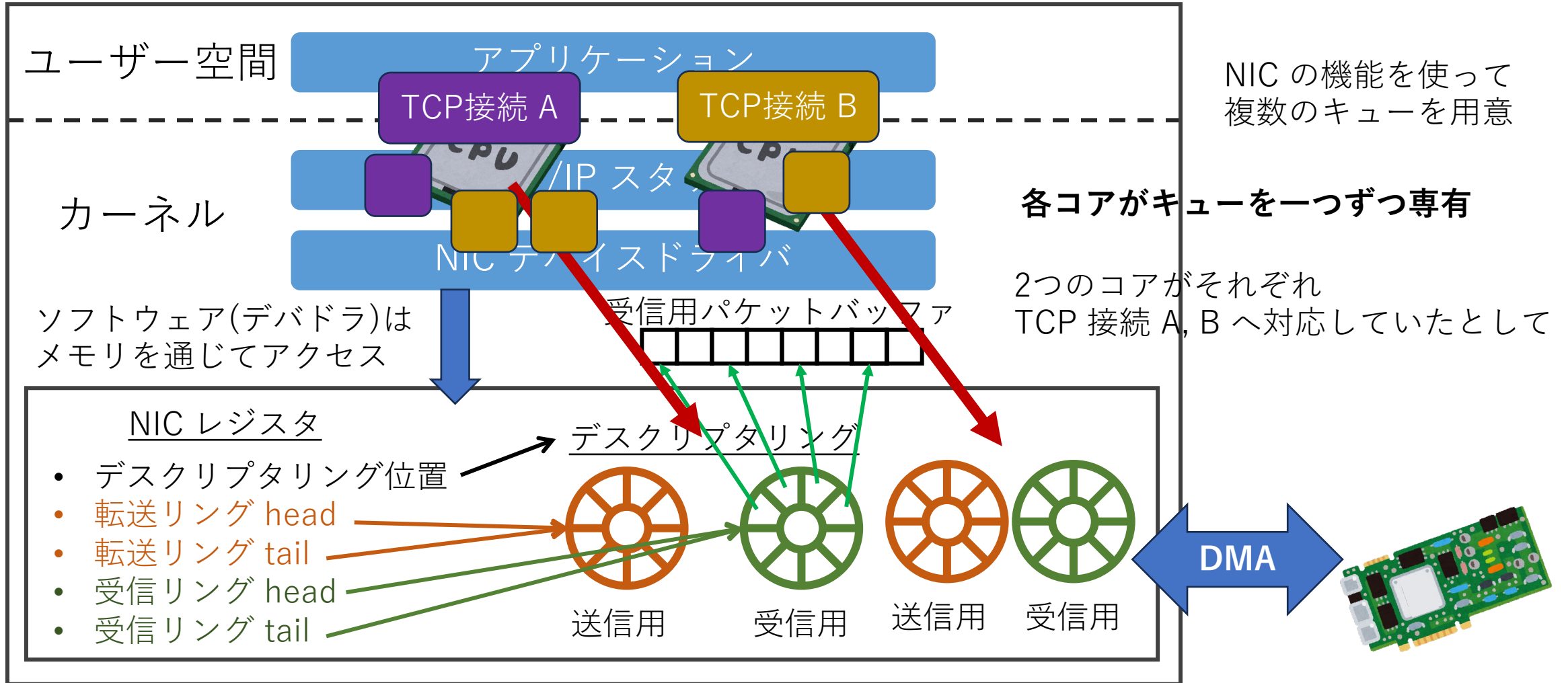
TCP/IP スタック設計の再考



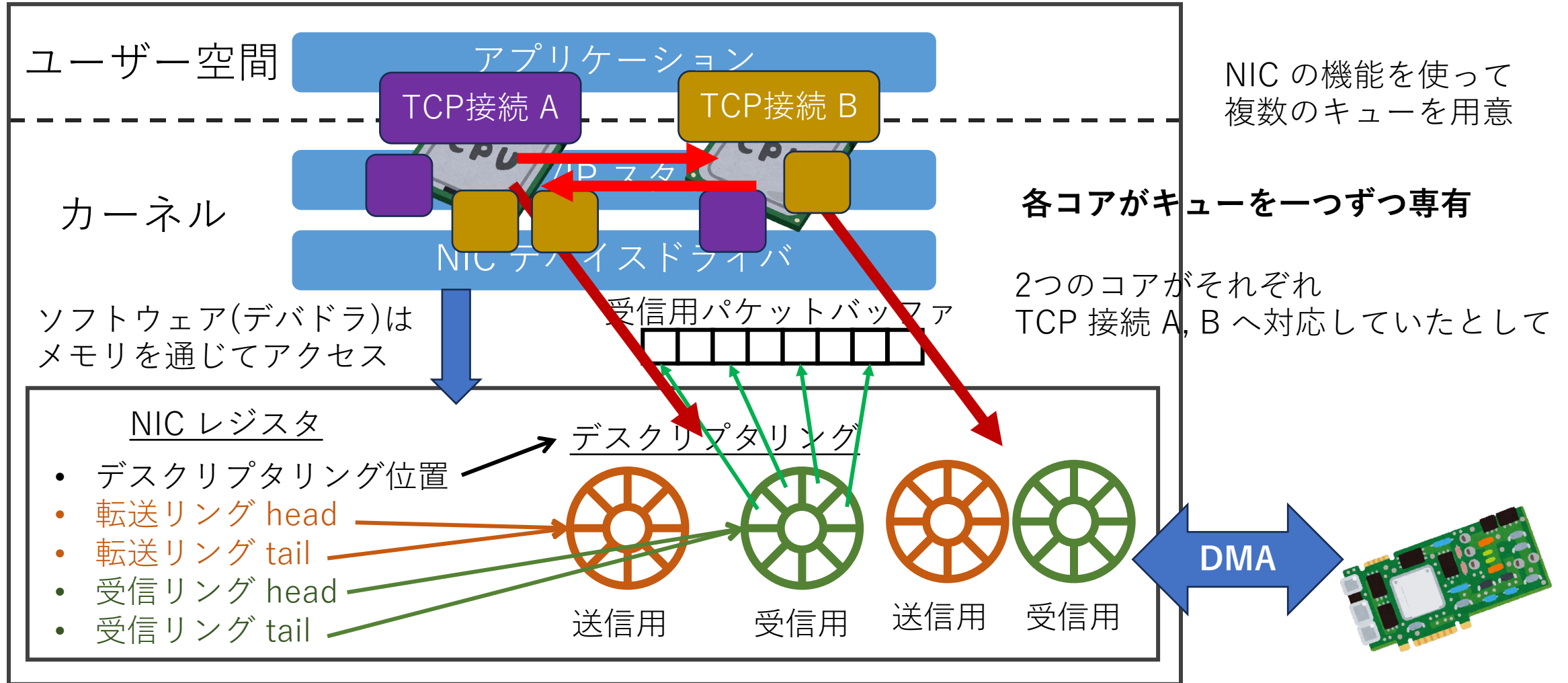
TCP/IP スタック設計の再考



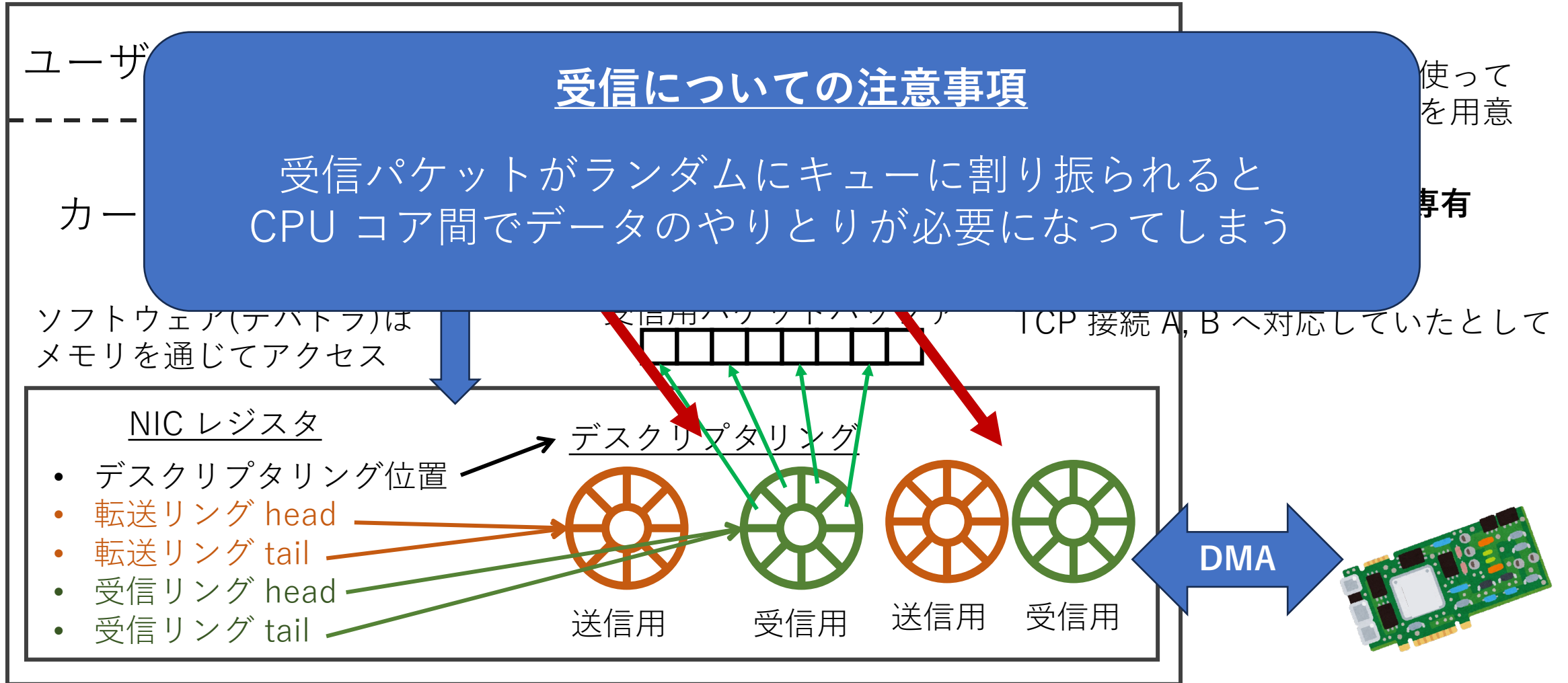
TCP/IP スタック設計の再考



TCP/IP スタック設計の再考



TCP/IP スタック設計の再考



TCP/IP スタック設計の再考

ユーザ

受信についての注意事項

受信パッケージがランダムにキューに割り振られると CPU コア間でデータのやりとりが必要になってしまう



解決策：NIC の Receive Side Scaling (RSS) 機能を使う

使って
を用意

専有

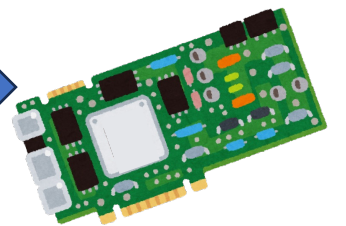
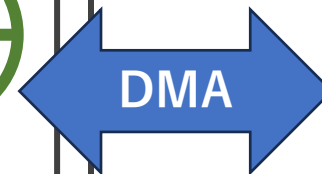
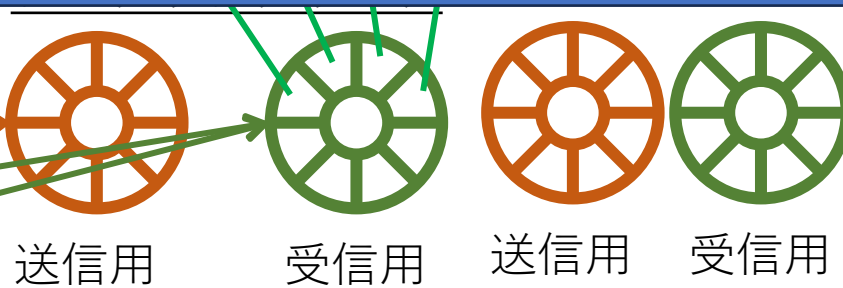
たとして

カー

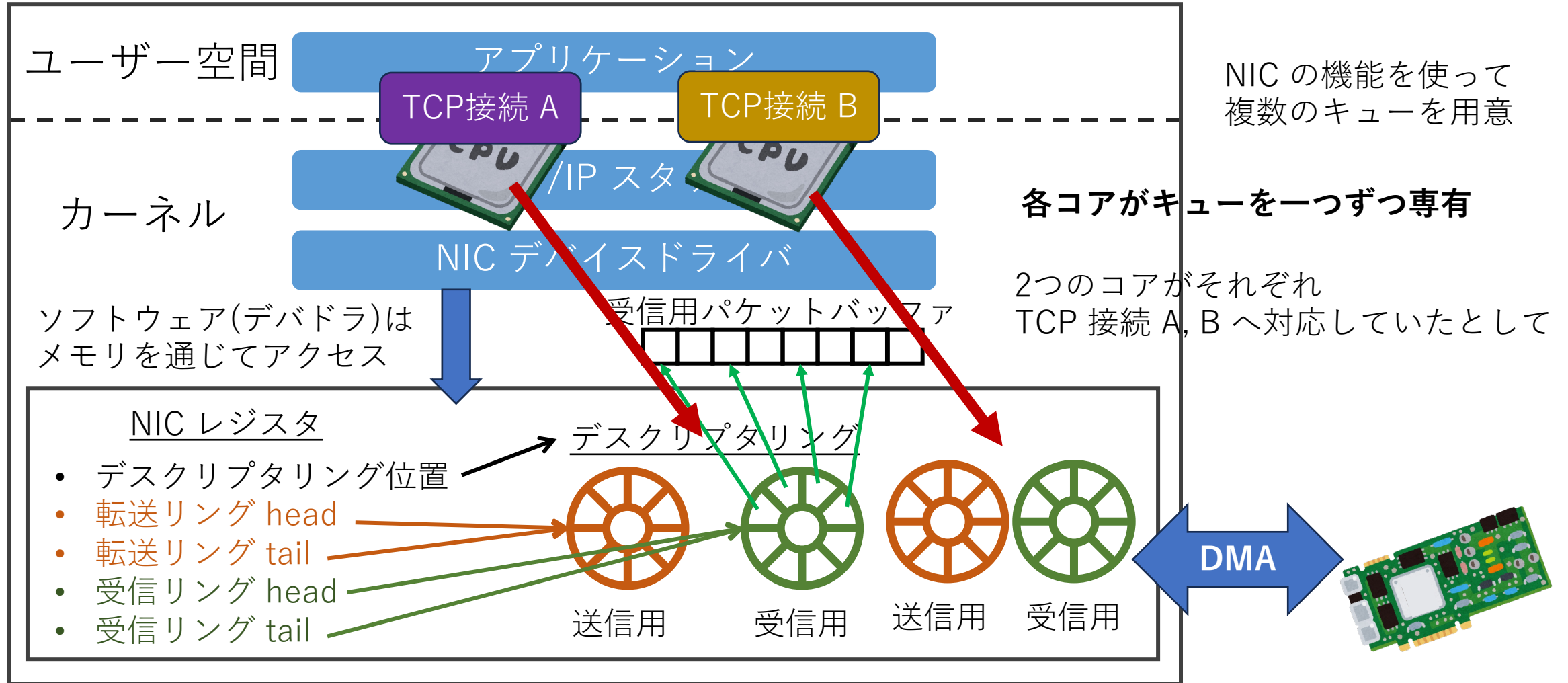
ソフト
メモリ

NIC

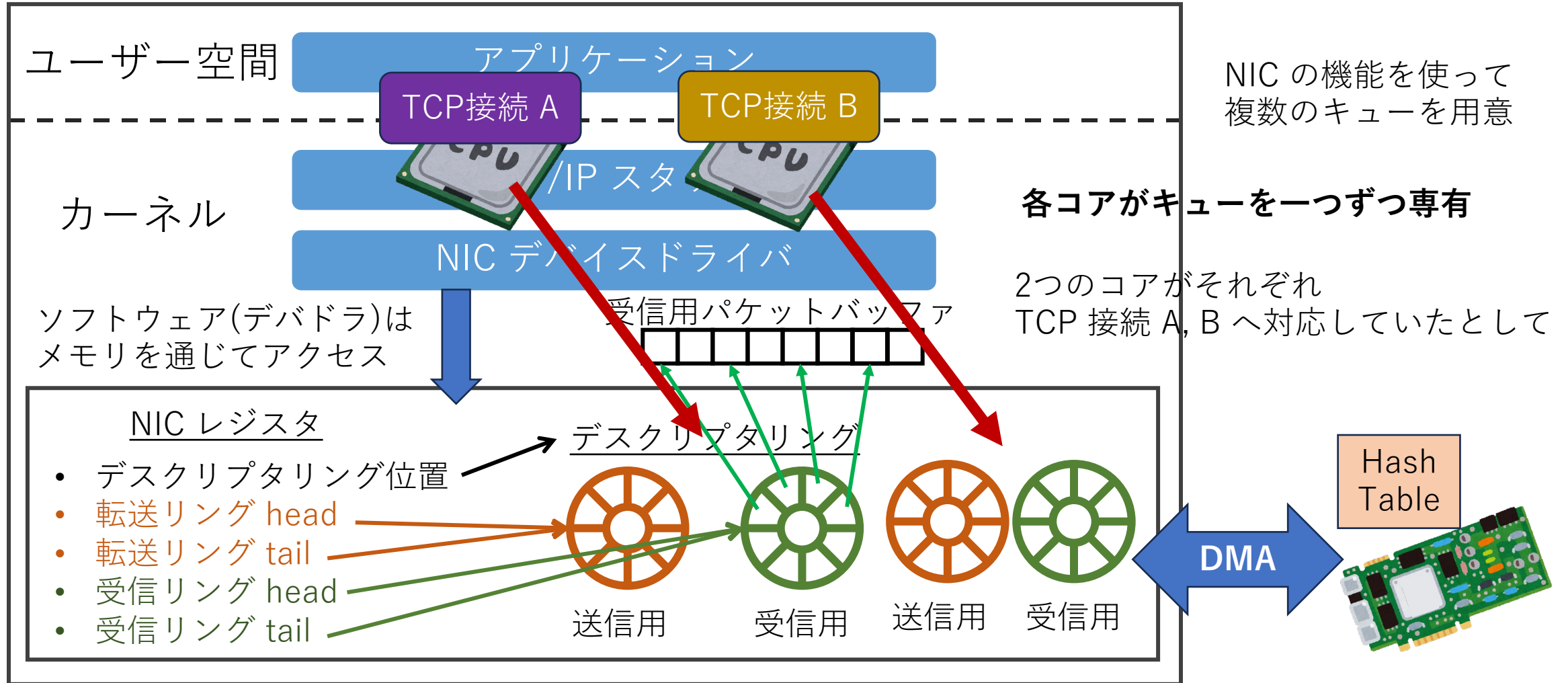
- デスクリプタリング位置
- 転送リング head
- 転送リング tail
- 受信リング head
- 受信リング tail



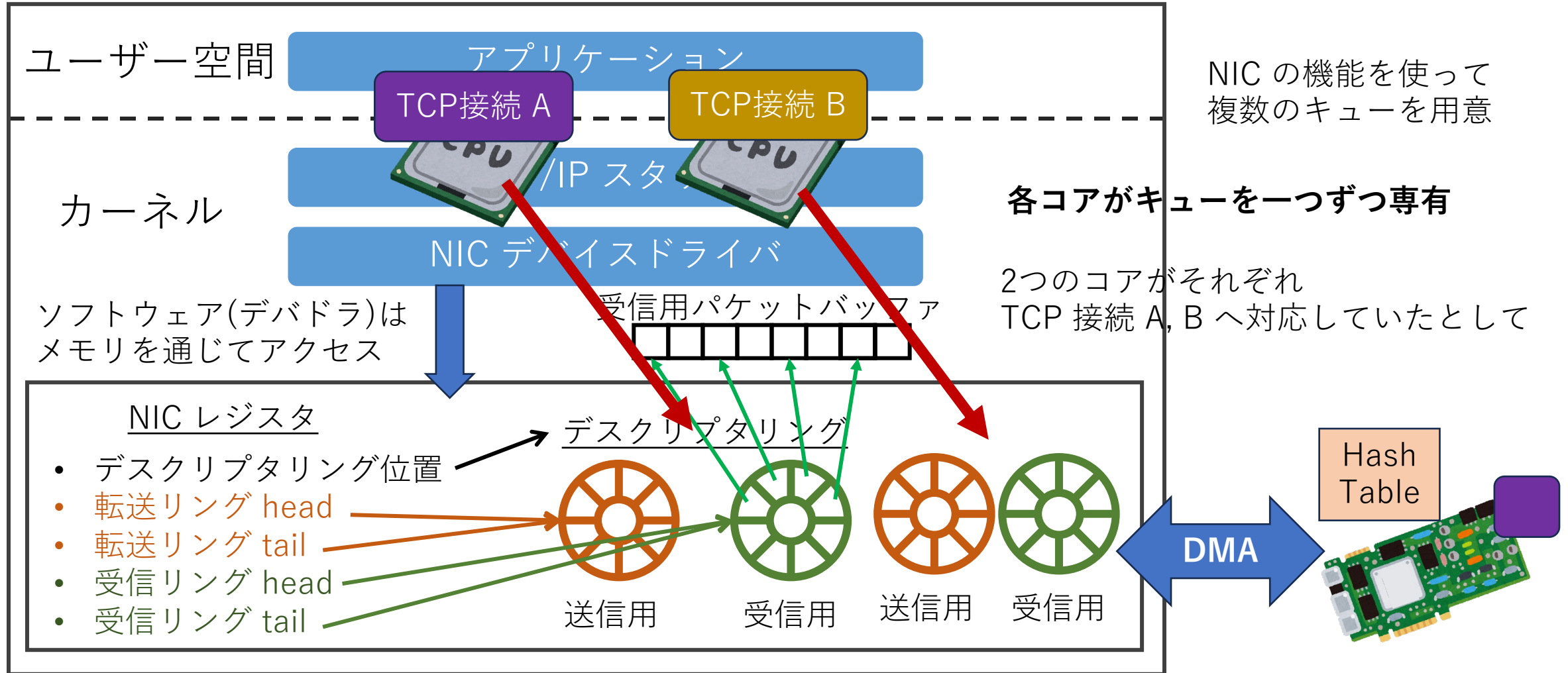
TCP/IP スタック設計の再考



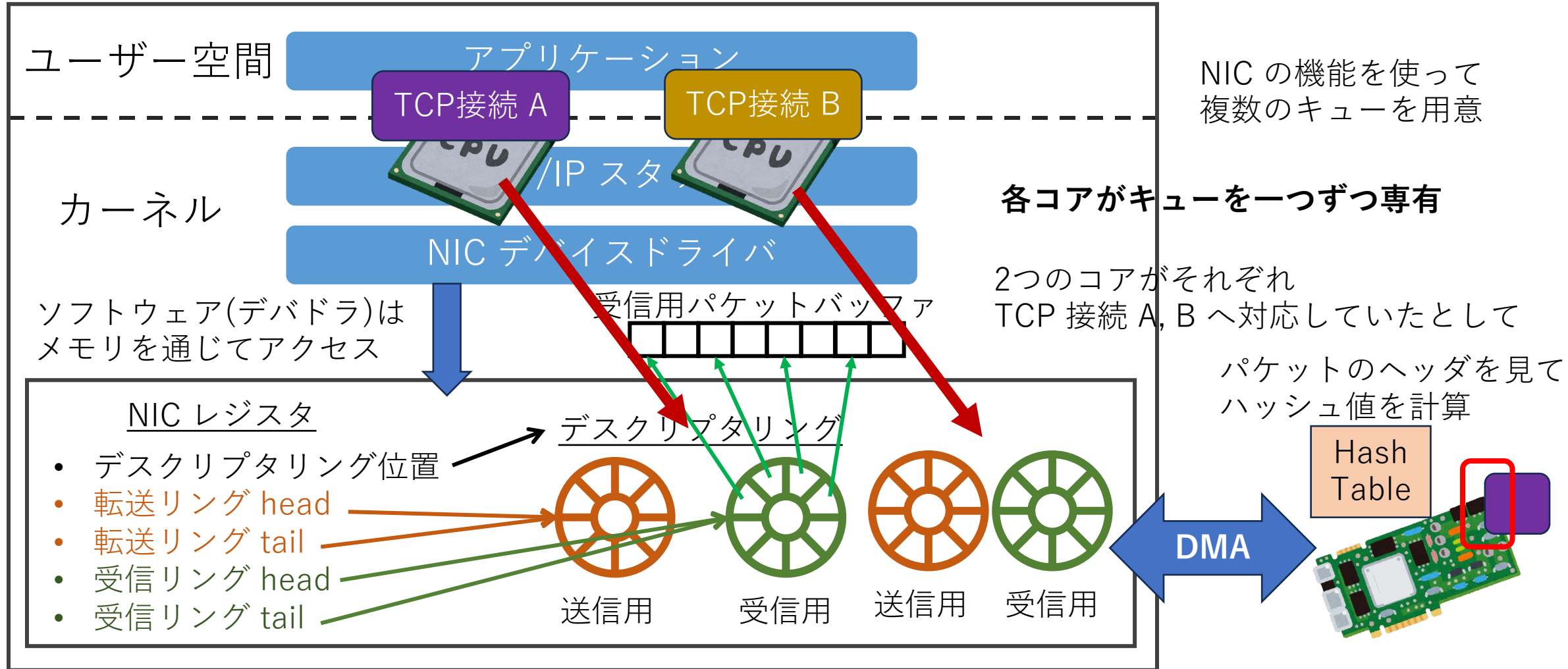
TCP/IP スタック設計の再考



TCP/IP スタック設計の再考

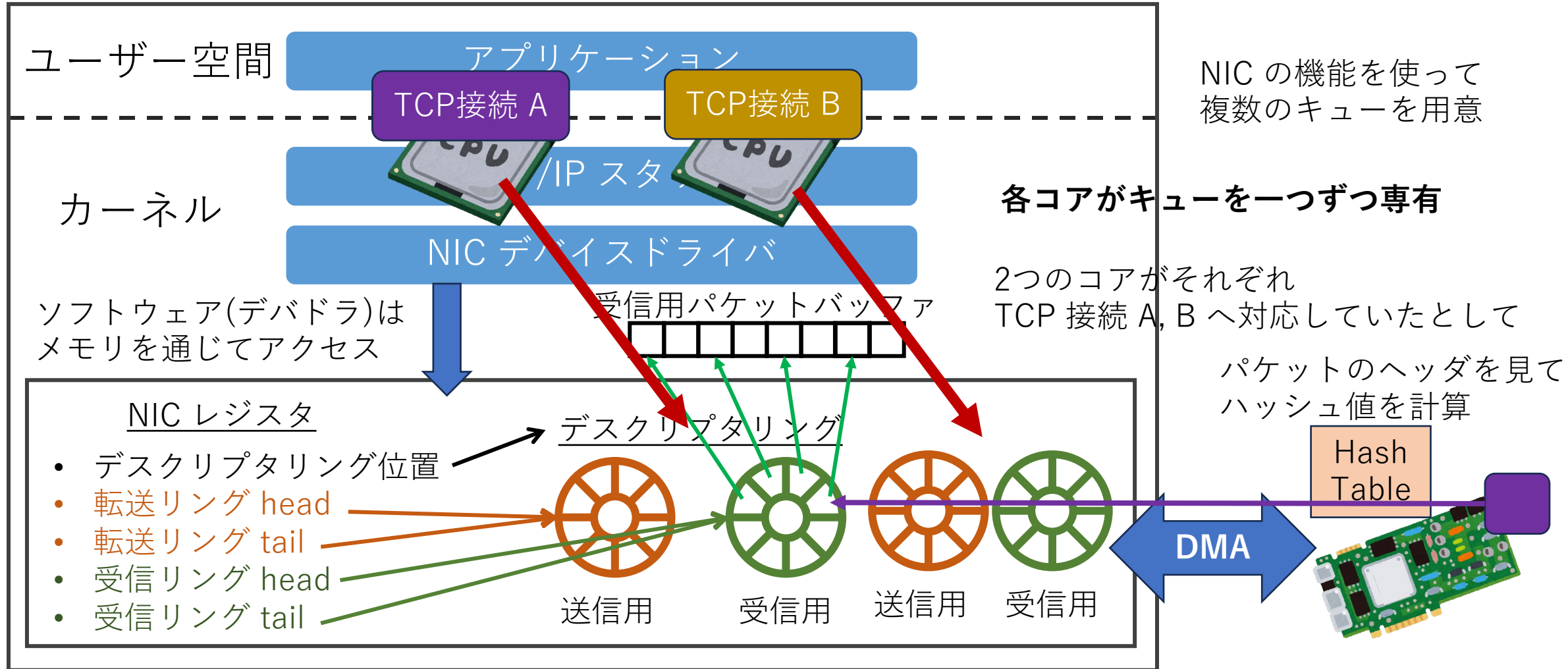


TCP/IP スタック設計の再考



TCP/IP スタック設計の再考

*RSS: Receive Side Scaling



NIC の機能を使って
複数のキューを用意

各コアがキューを一つずつ専有

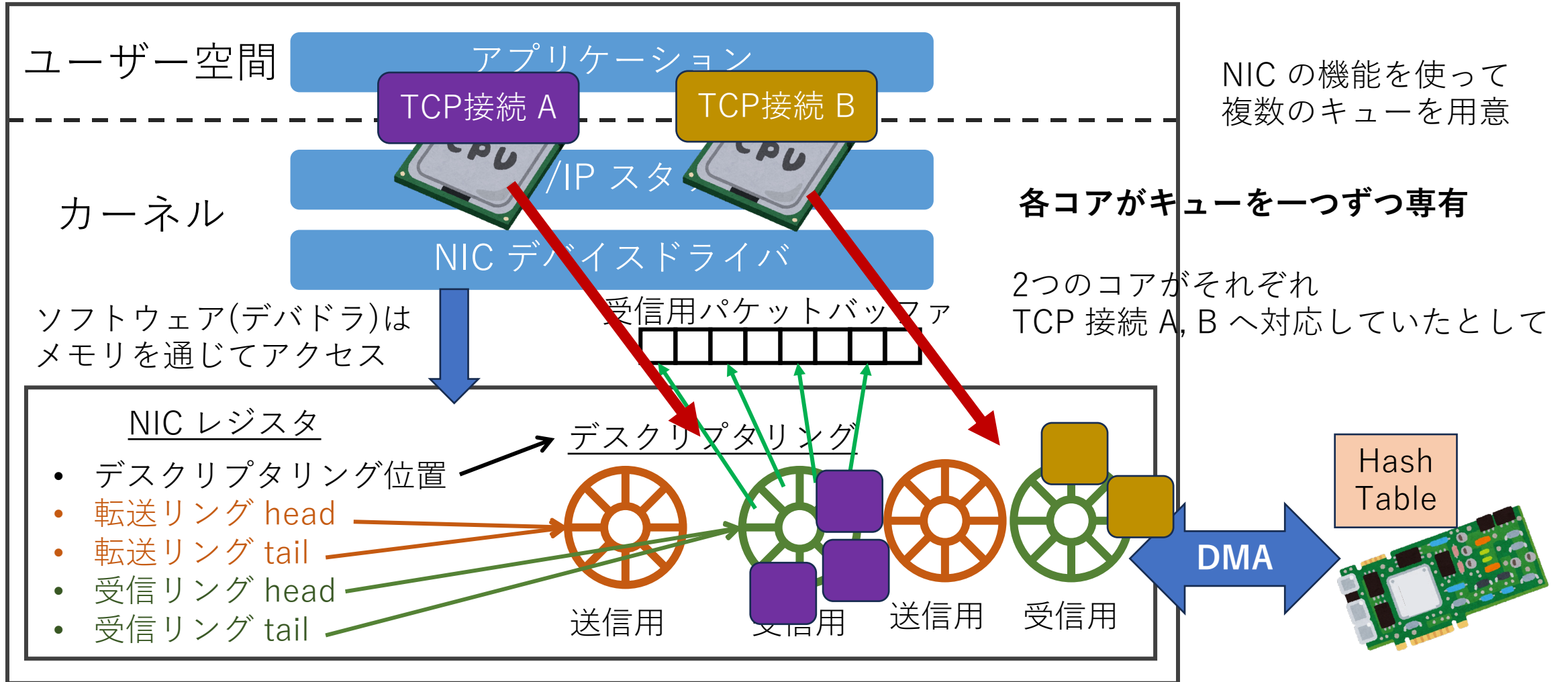
2つのコアがそれぞれ
TCP 接続 A, B へ対応していたとして

パケットのヘッダを見て
ハッシュ値を計算

ハッシュ値を元に hash table を参照して宛先キューを決める

TCP/IP スタック設計の再考

*RSS: Receive Side Scaling



TCP/IP スタック設計の再考

*RSS: Receive Side Scaling

ユーザー

受信についての注意事項

受信パッケージがランダムにキューに割り振られると CPU コア間でデータのやりとりが必要になってしまう



解決策：NIC の Receive Side Scaling (RSS) 機能を使う

RSS のおかげで、特定の TCP 接続のパッケージは 特定のキューで受信されるようにできる

ソフトウェア
メモリを

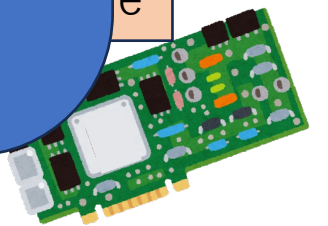
- デス
- 転送
- 転送
- 受信リ
- 受信リング tail

送信用

受信用

送信用

受信用



を使って
を用意

専有

たとして

h
e

TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用

TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用

ここまでは NIC のハードウェア機能の話

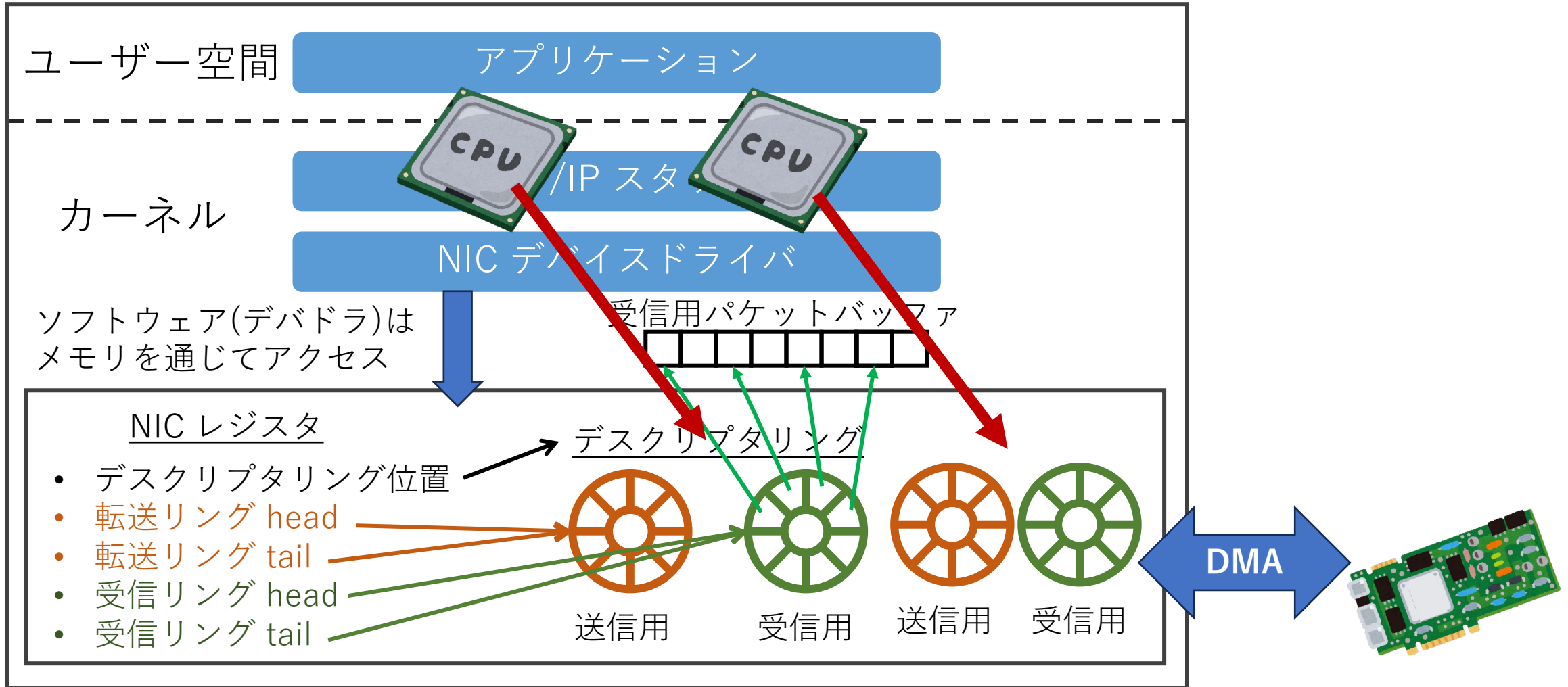
研究紹介

TCP/IP スタック設計

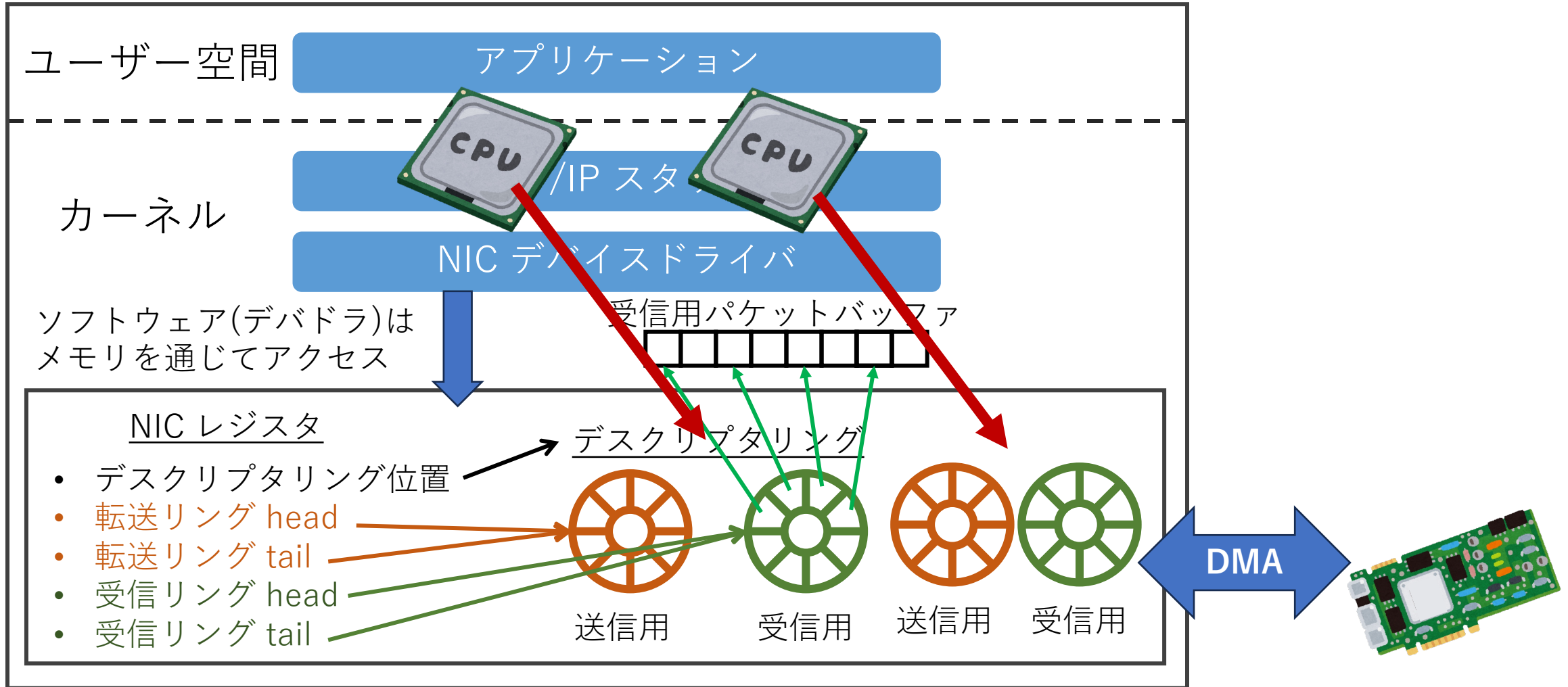
マルチコア環境でのスケーラビリティについて

ソフトウェアでもコア間の共有オブジェクトを減らす

TCP/IP スタック設計の再考

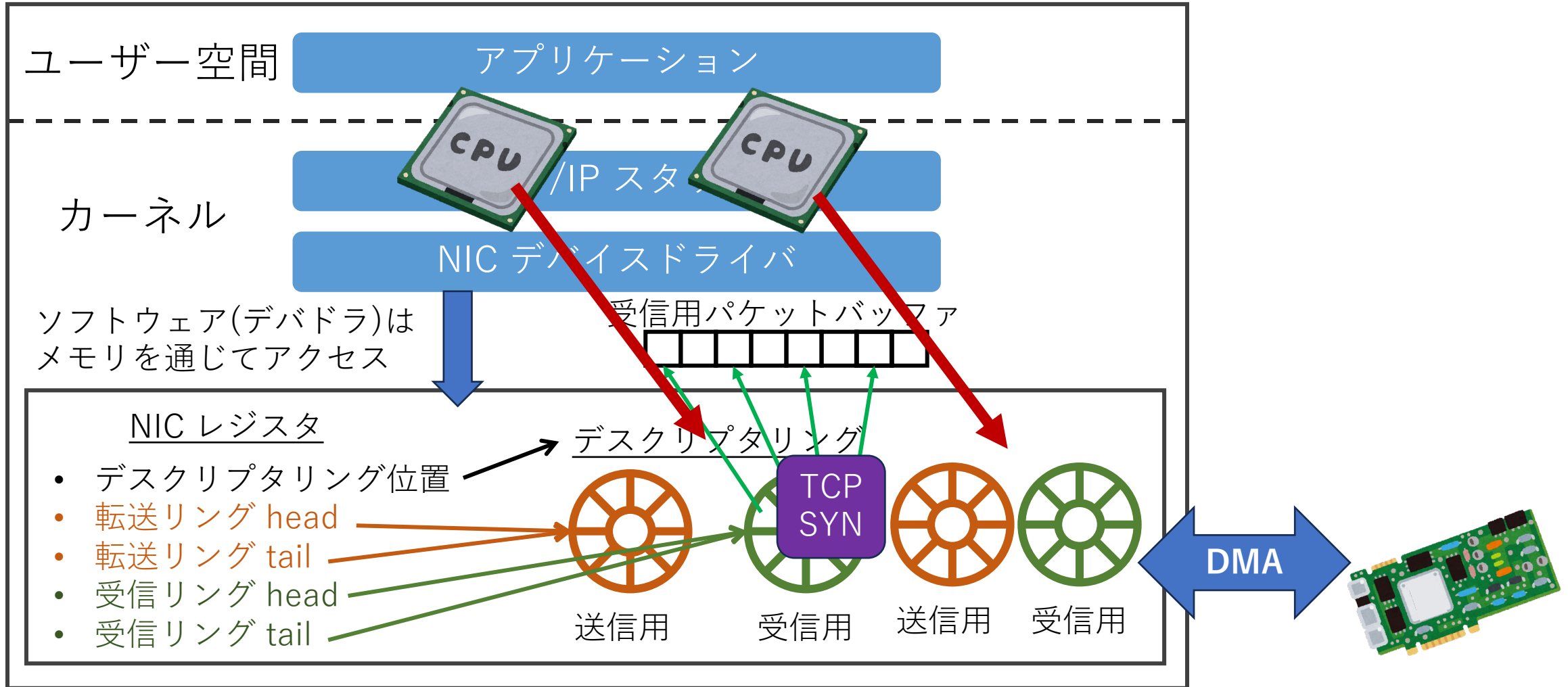


TCP/IP スタック設計の再考



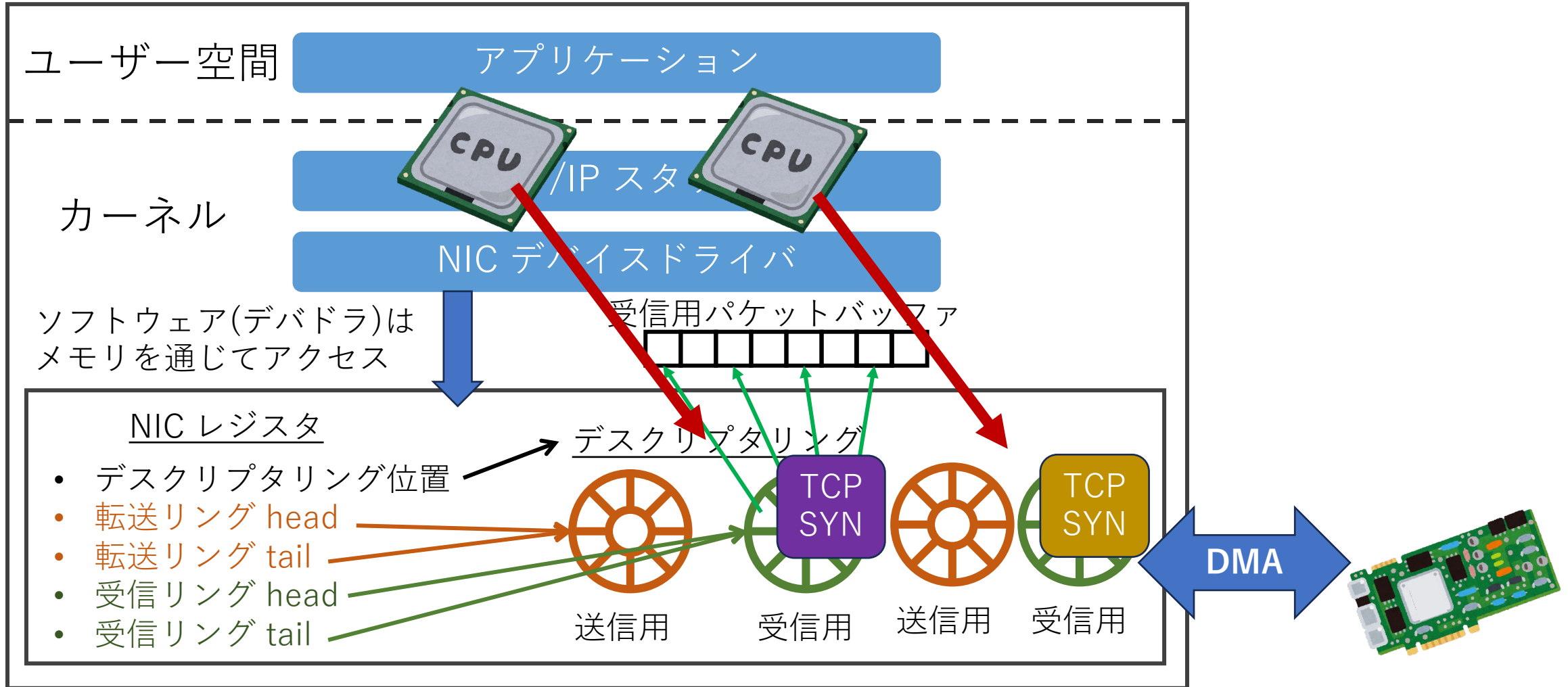
TCP の接続確立時

TCP/IP スタック設計の再考



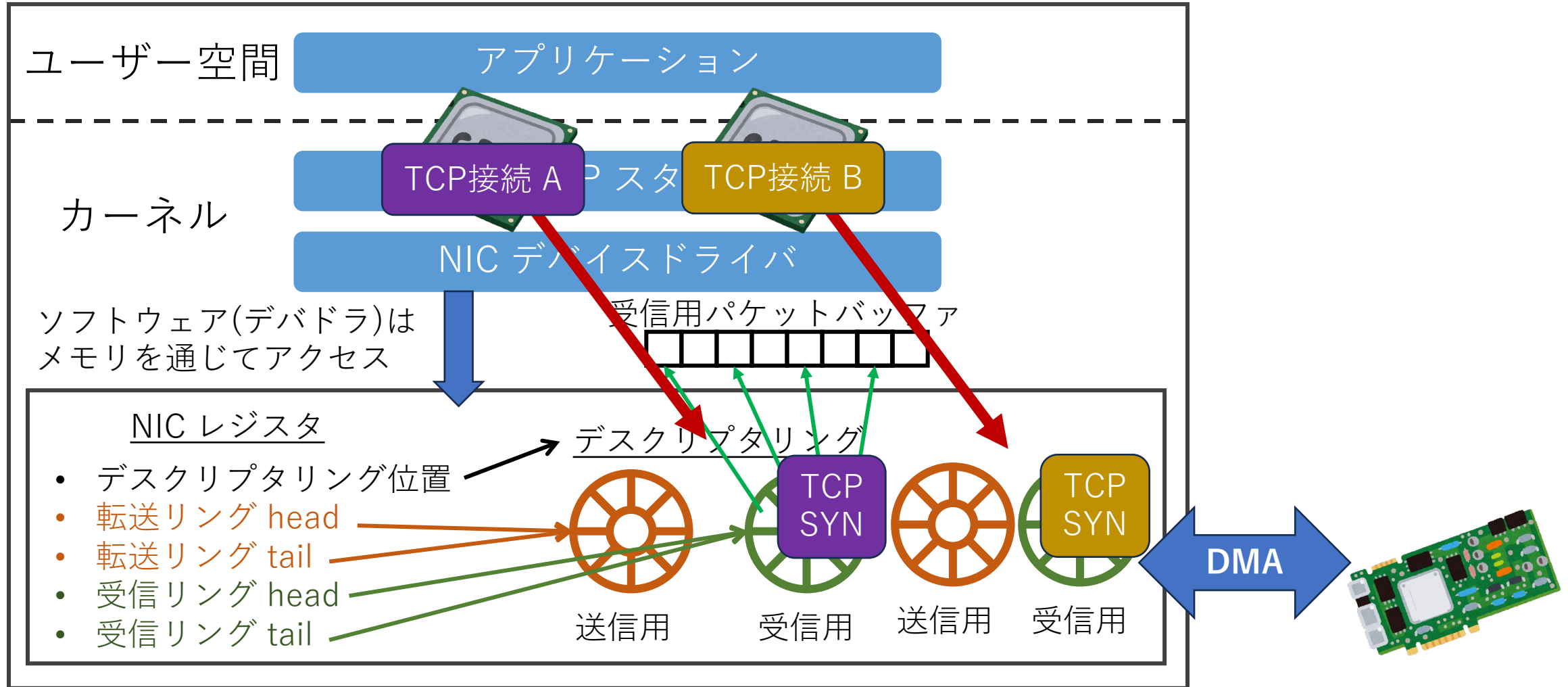
TCP の接続確立時

TCP/IP スタック設計の再考

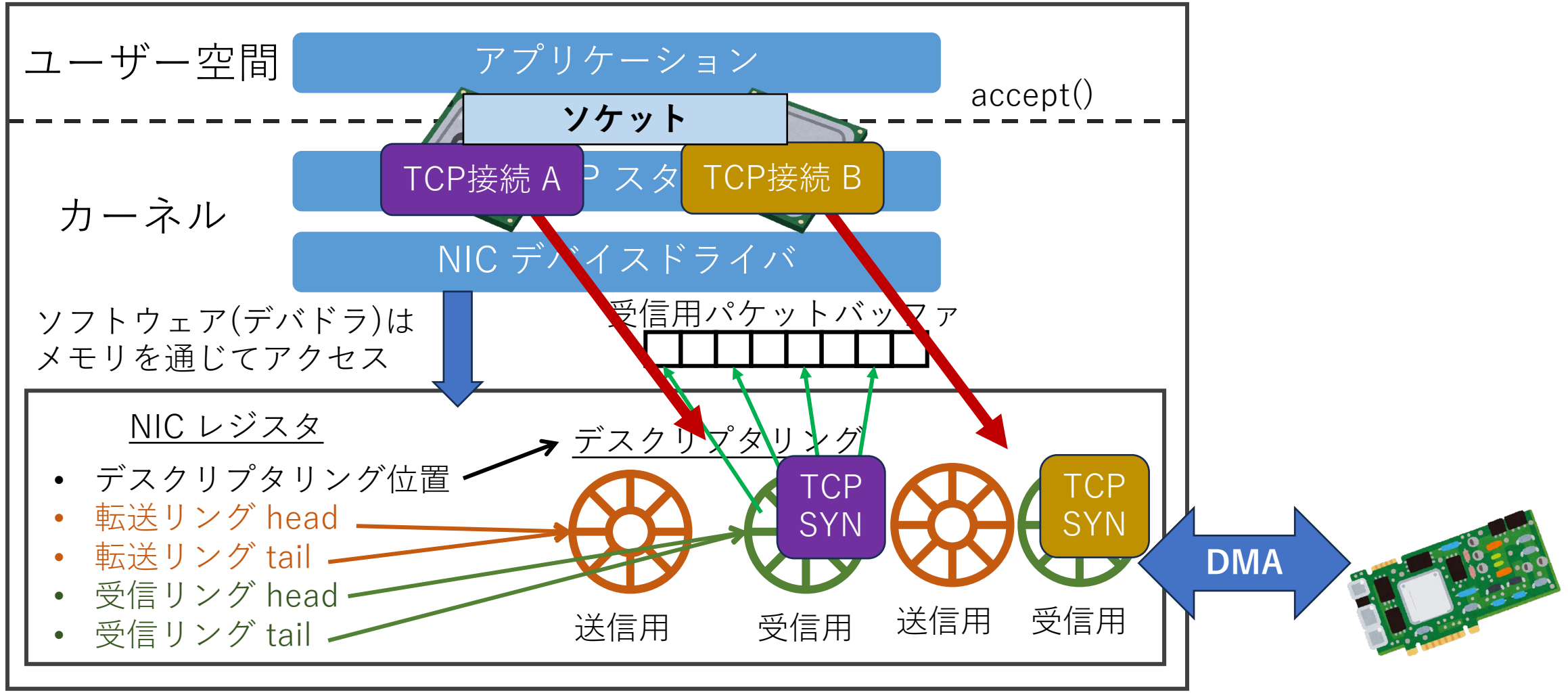


TCP の接続確立時

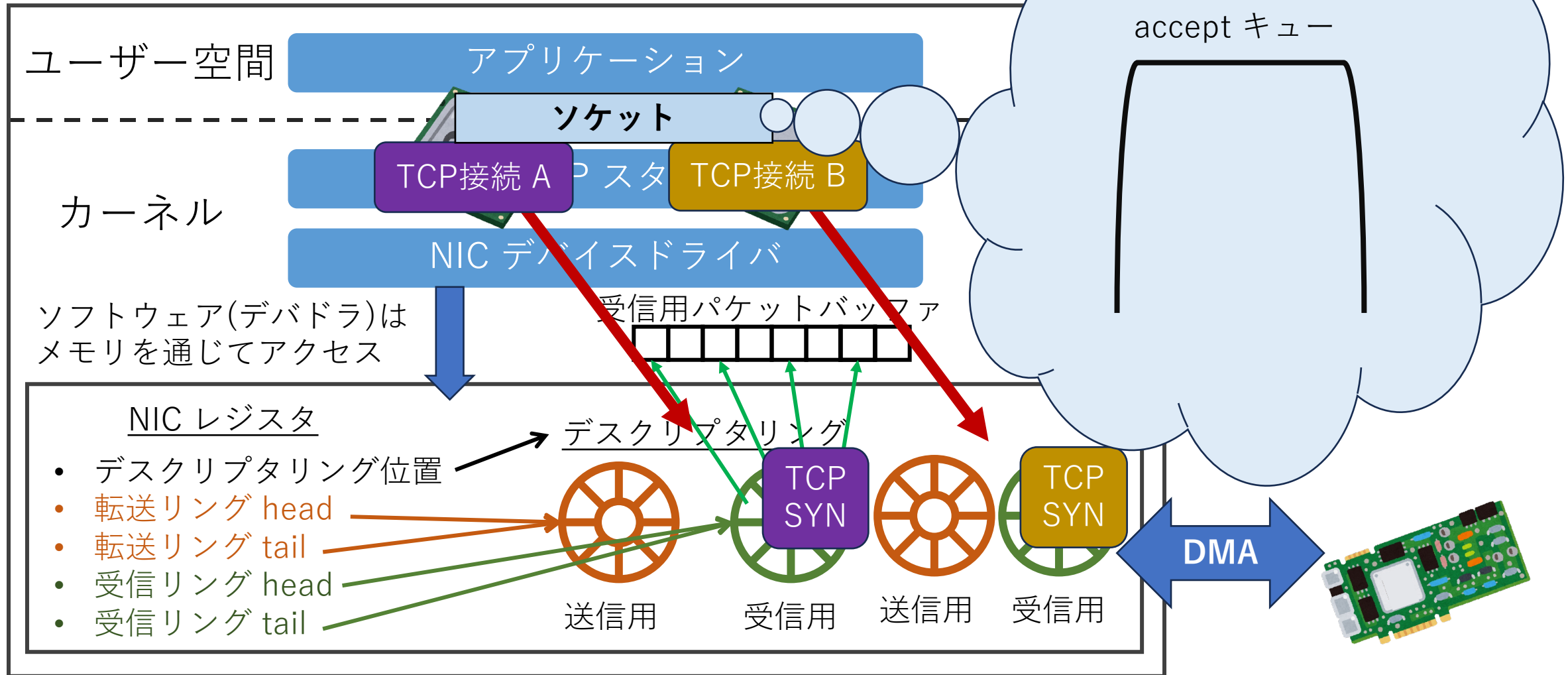
TCP/IP スタック設計の再考



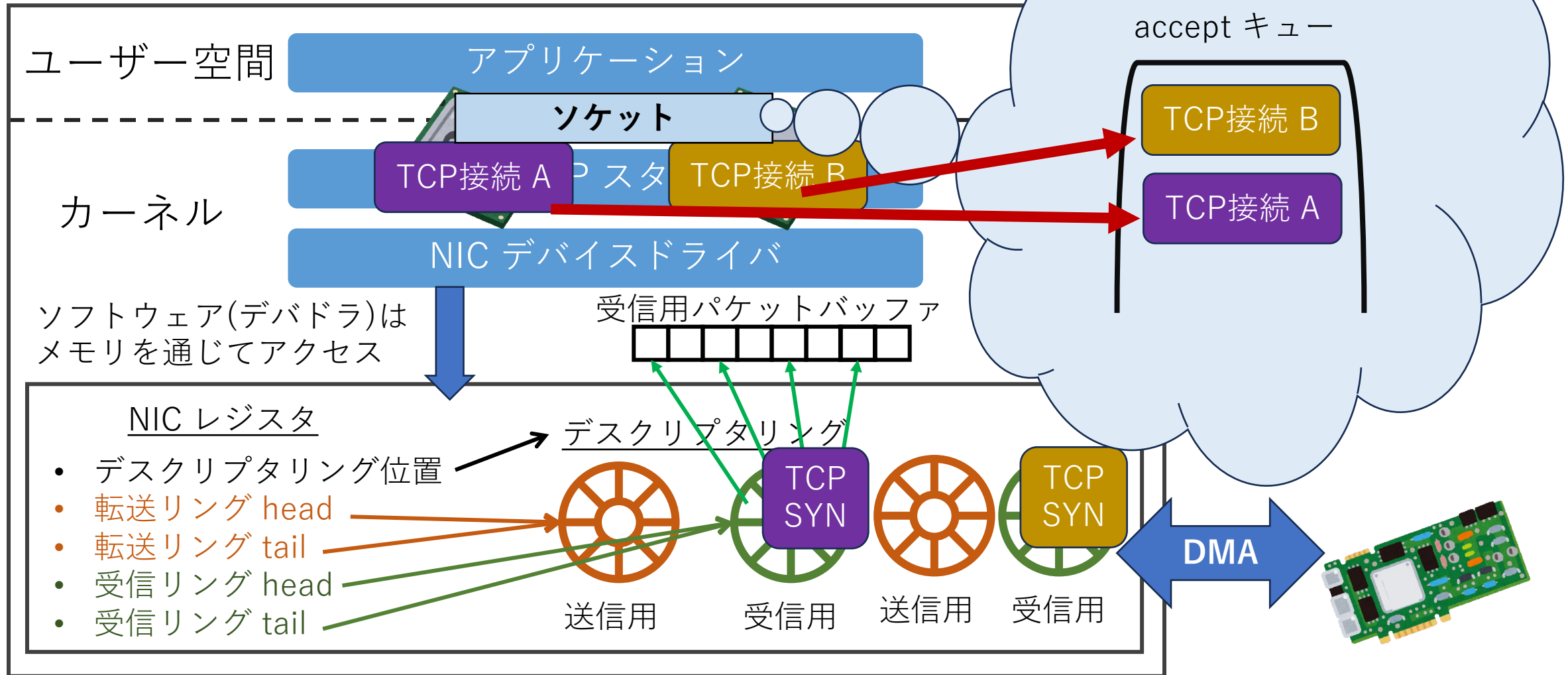
TCP/IP スタック設計の再考



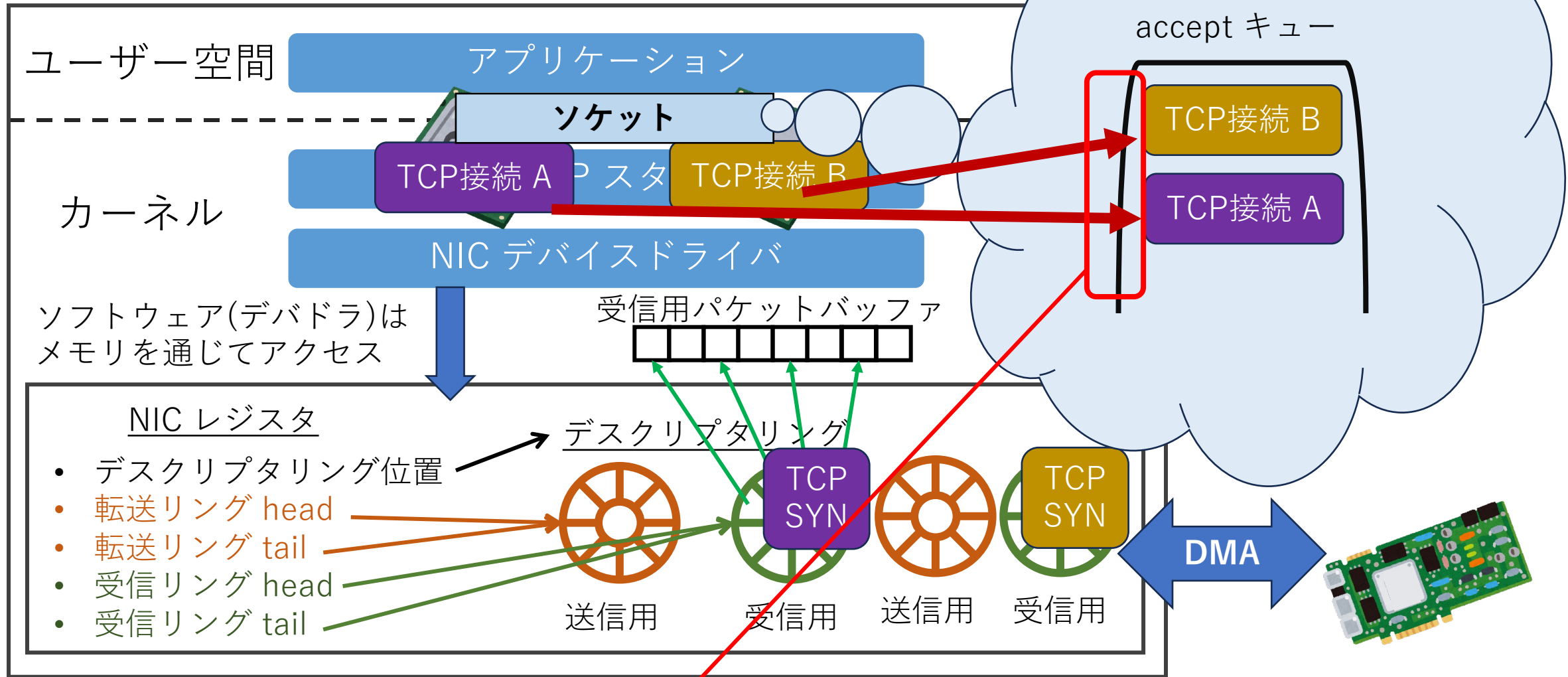
TCP/IP スタック設計の再考



TCP/IP スタック設計の再考



TCP/IP スタック設計の再考

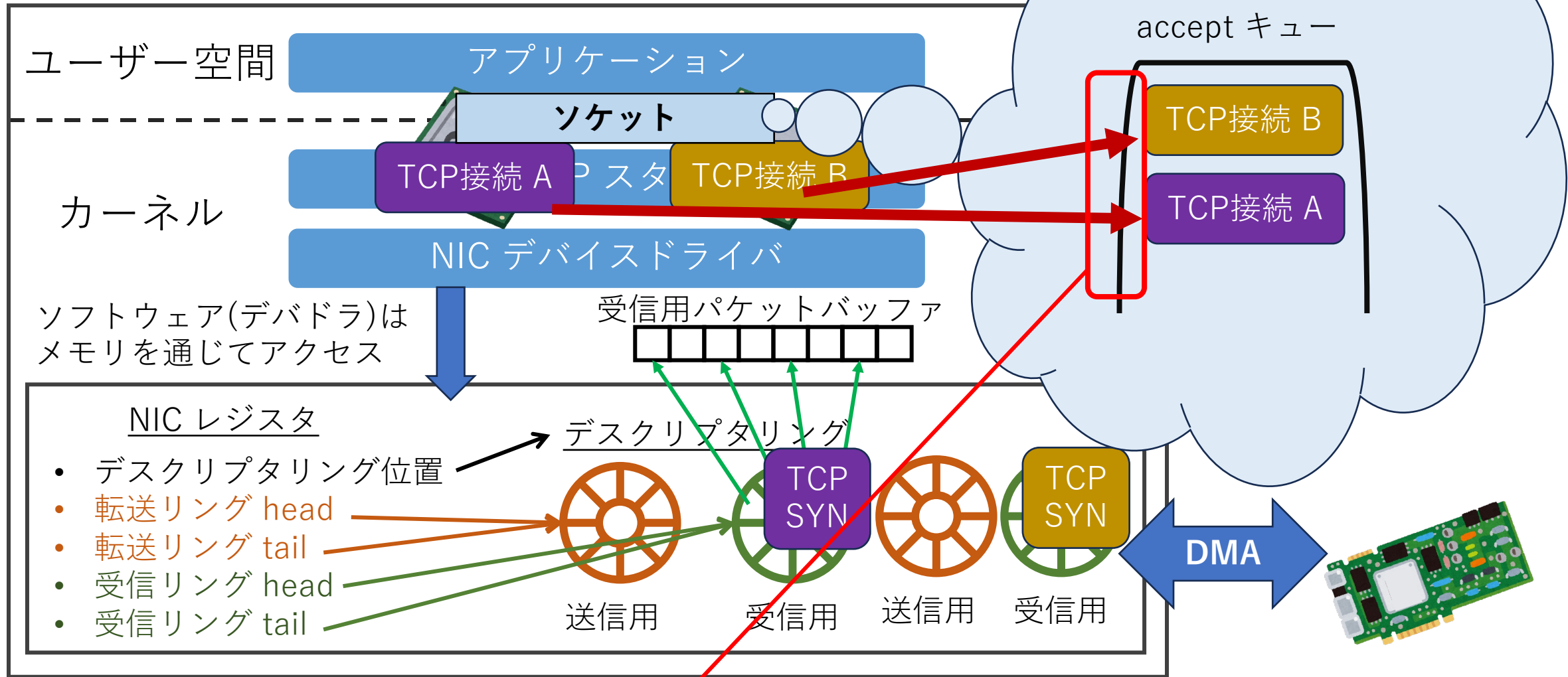


accept キューはソケットごとに1つずつしかないため、コア間での競合のポイントになる

TCP/IP スタック設計の再考

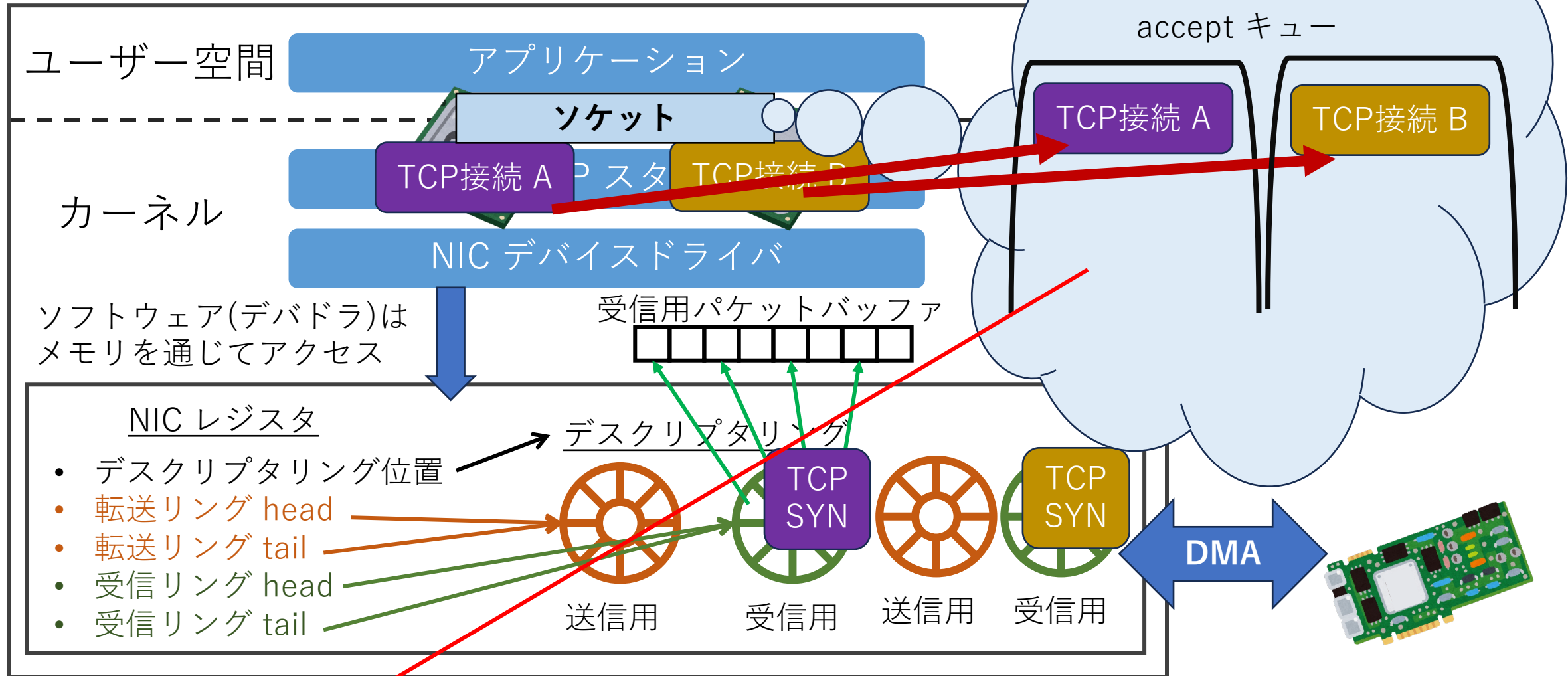
- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - Affinity-Accept (EuroSys 2012)
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

TCP/IP スタック設計の再考



accept キューはソケットごとに1つずつしかないため、コア間での競合のポイントになる

TCP/IP スタック設計の再考



解決策：コアごとに accept キューを用意する

TCP/IP スタック設計の再考

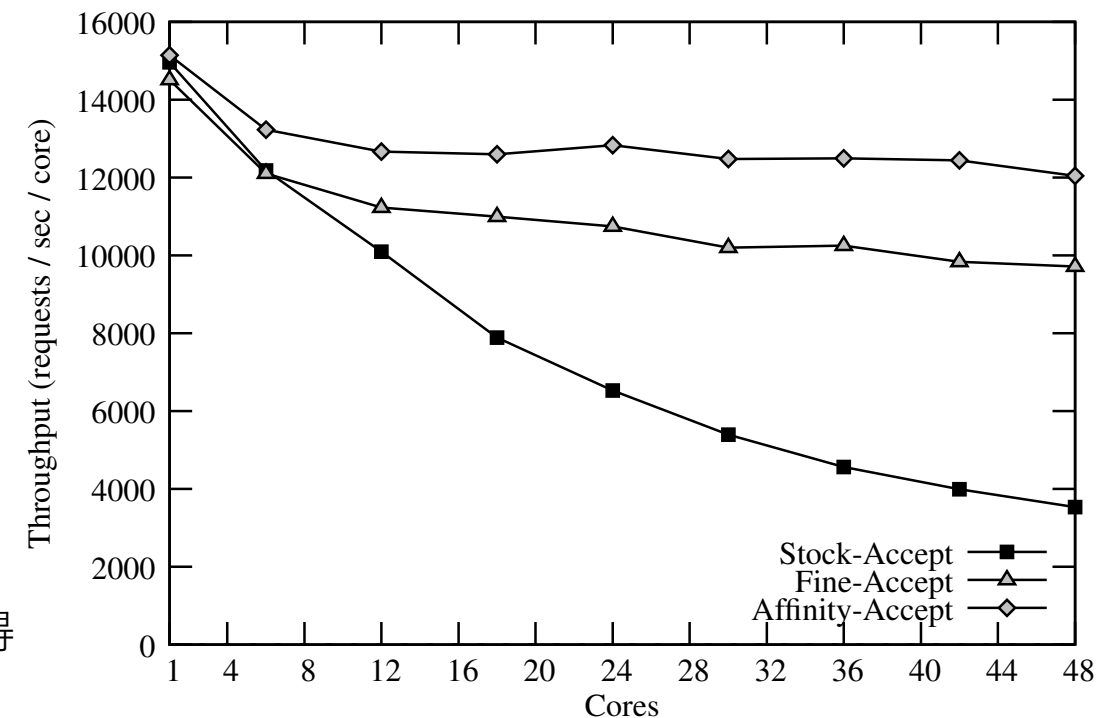
- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - Affinity-Accept (EuroSys 2012)
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

TCP/IP スタック設計の再考

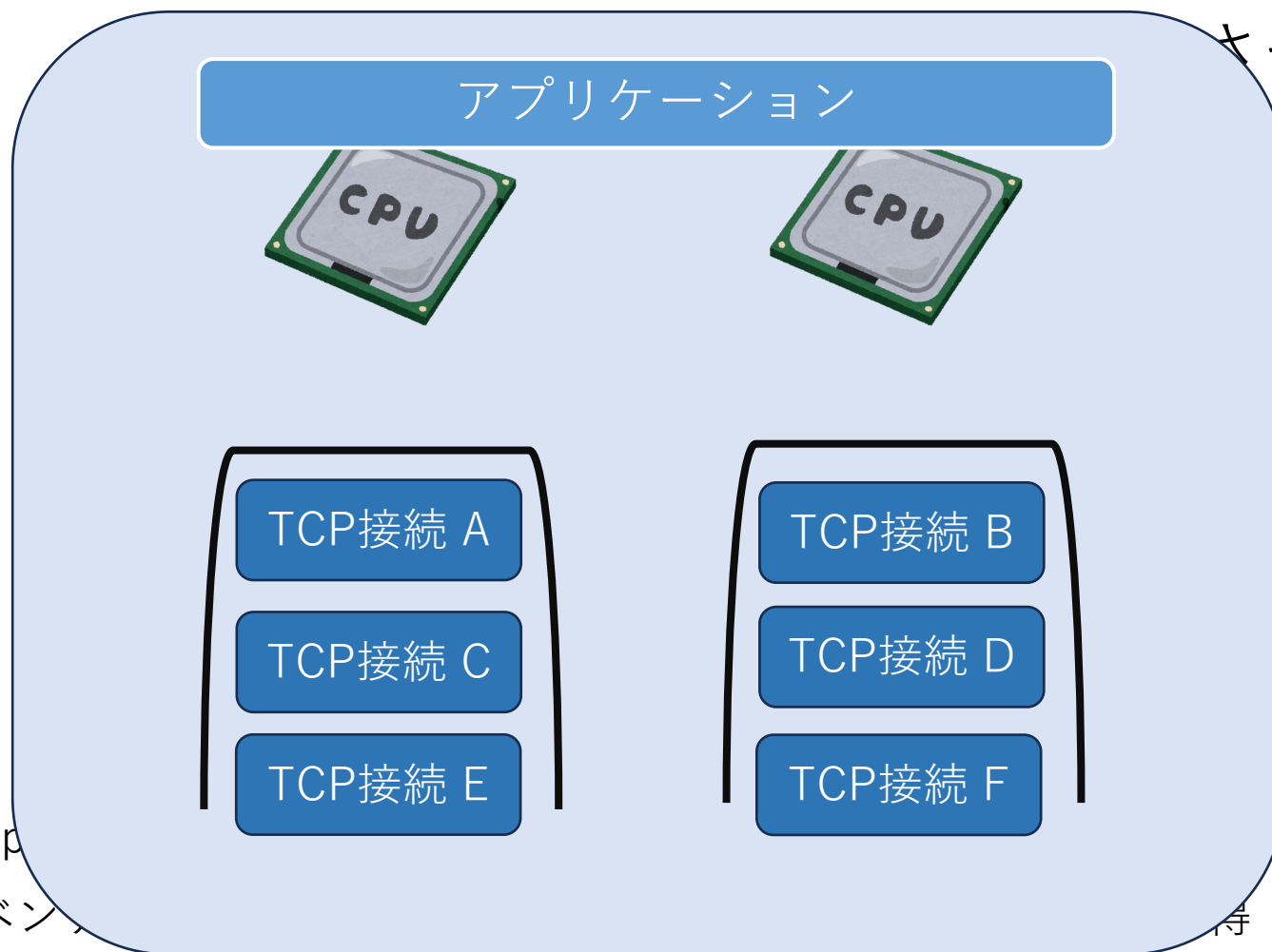
- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - **Affinity-Accept (EuroSys 2012)**
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

Apache HTTP server パフォーマンス

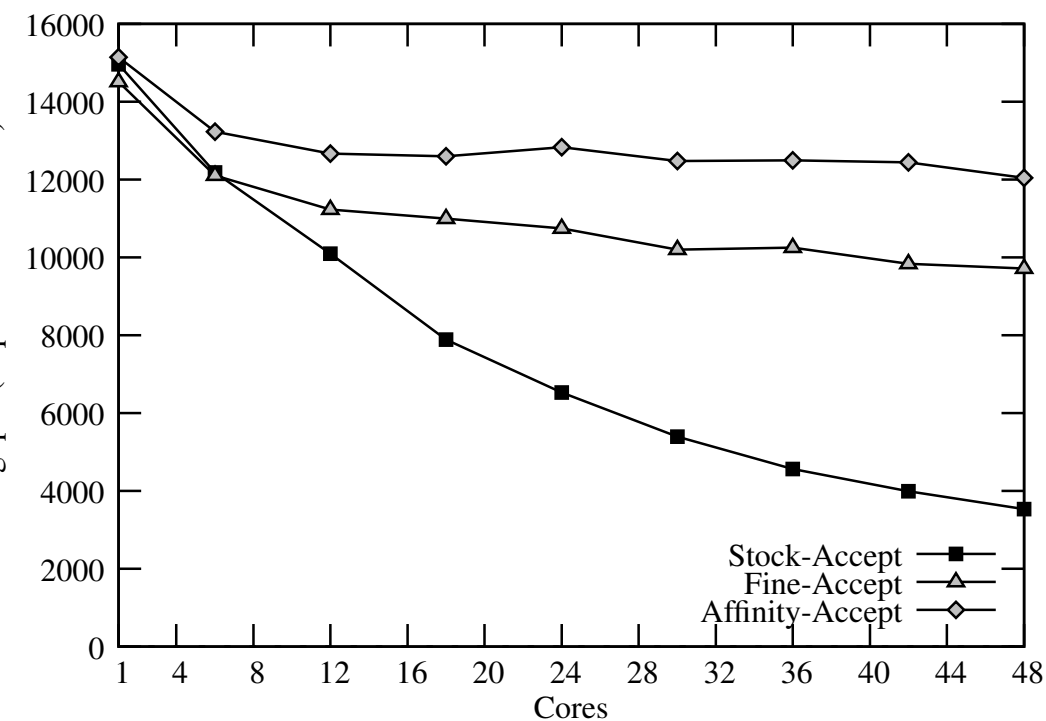
ベンチマーククライアントは1回の TCP 接続で6ファイル取得



TCP/IP スタック設計の再考

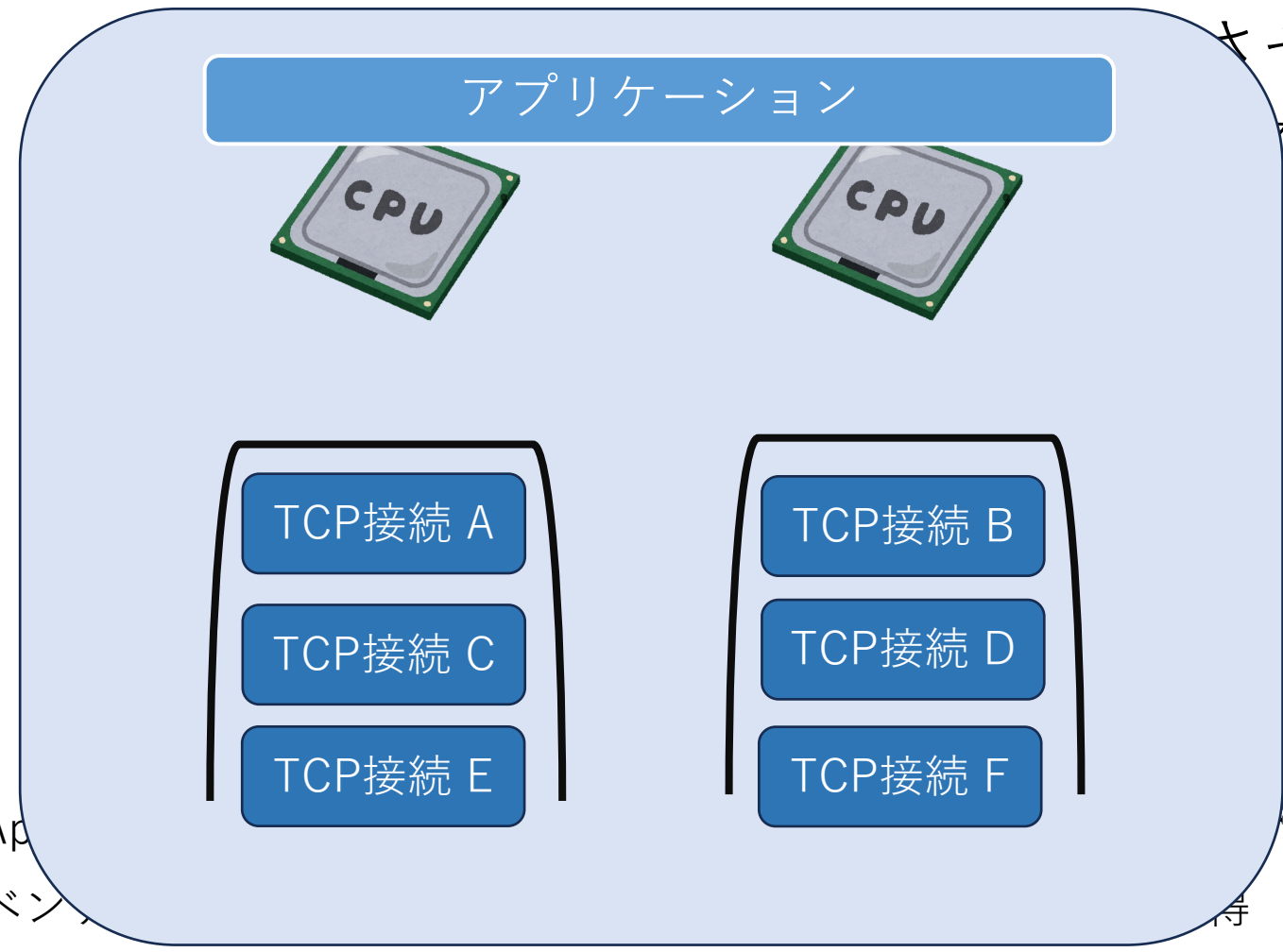


減らせる
オブジェクトを減らす
キュー機能を利用

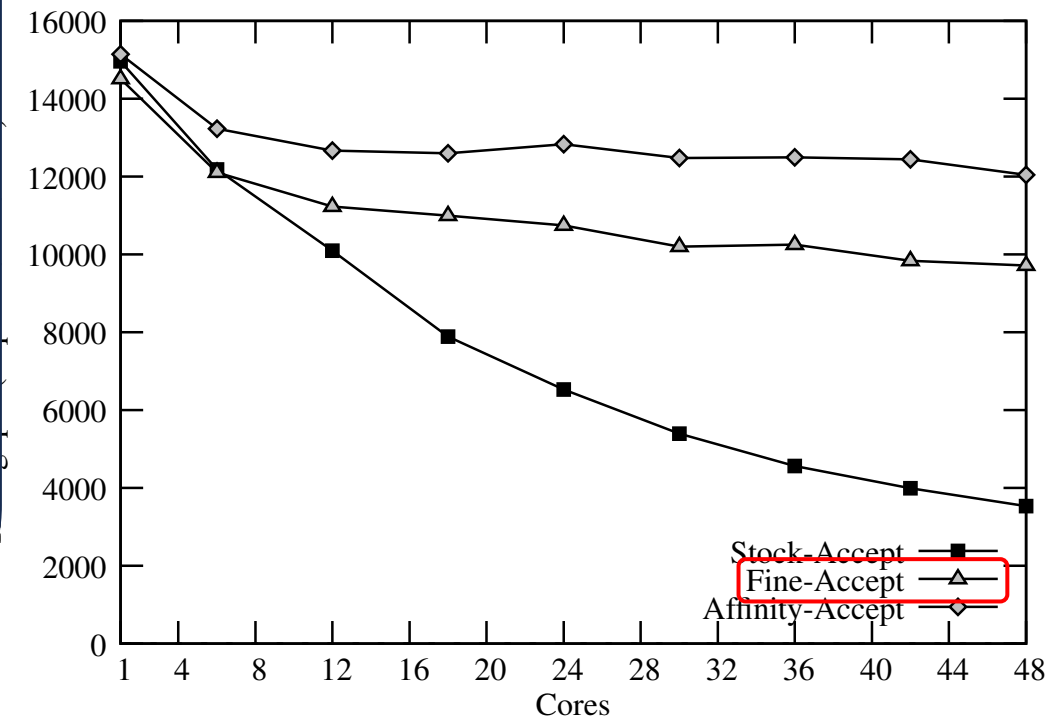


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept



減らせる
オブジェクトを減らす
キュー機能を利用

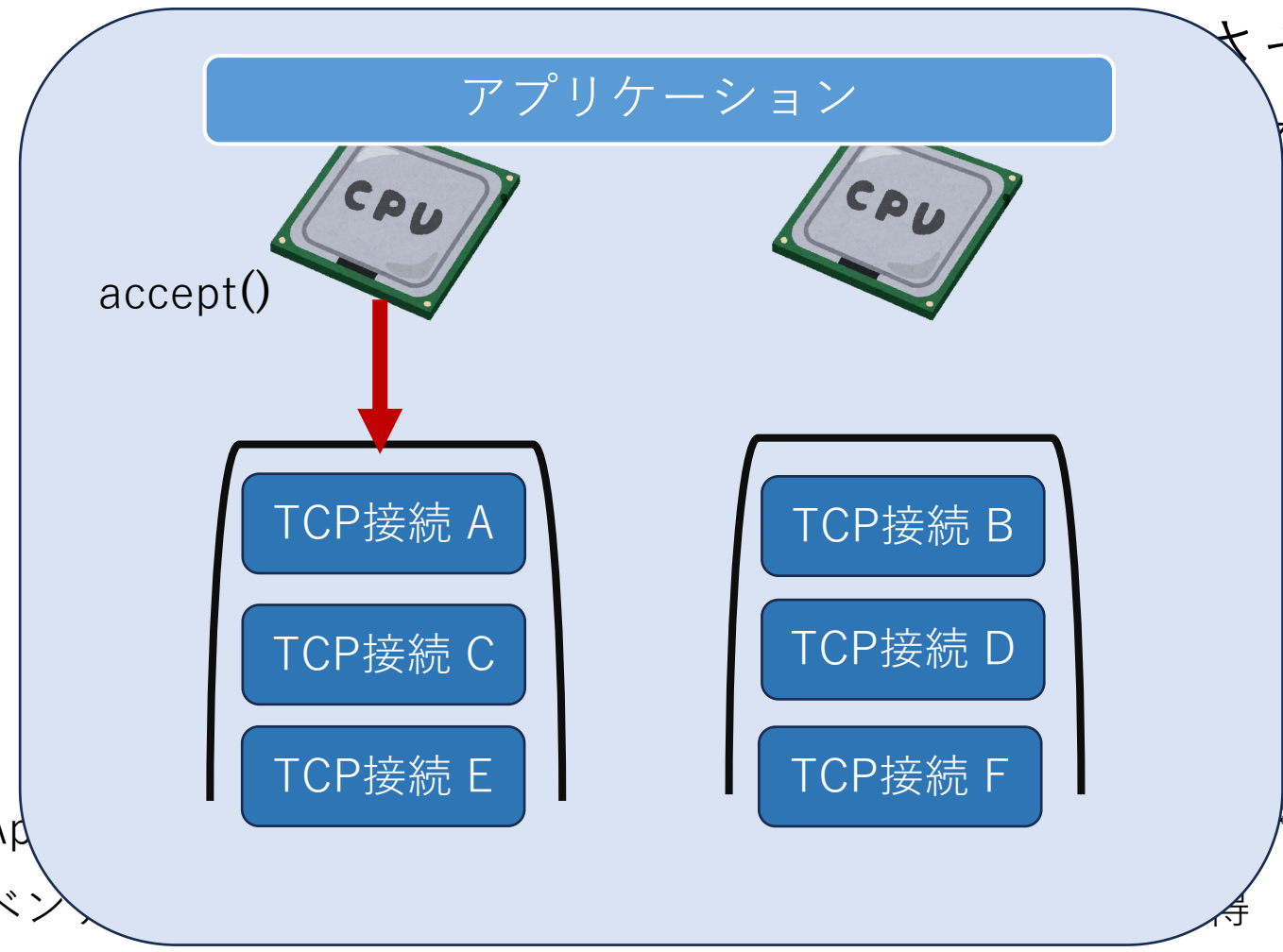


Ap
ベン

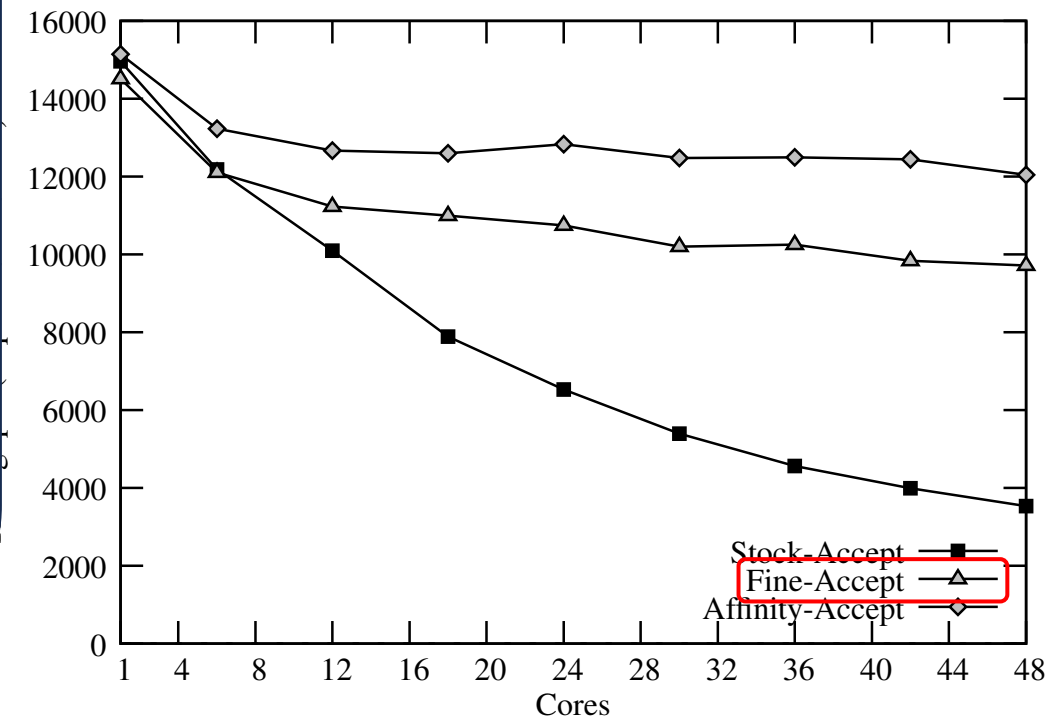
待

TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept



減らせる
オブジェクトを減らす
キュー機能を利用

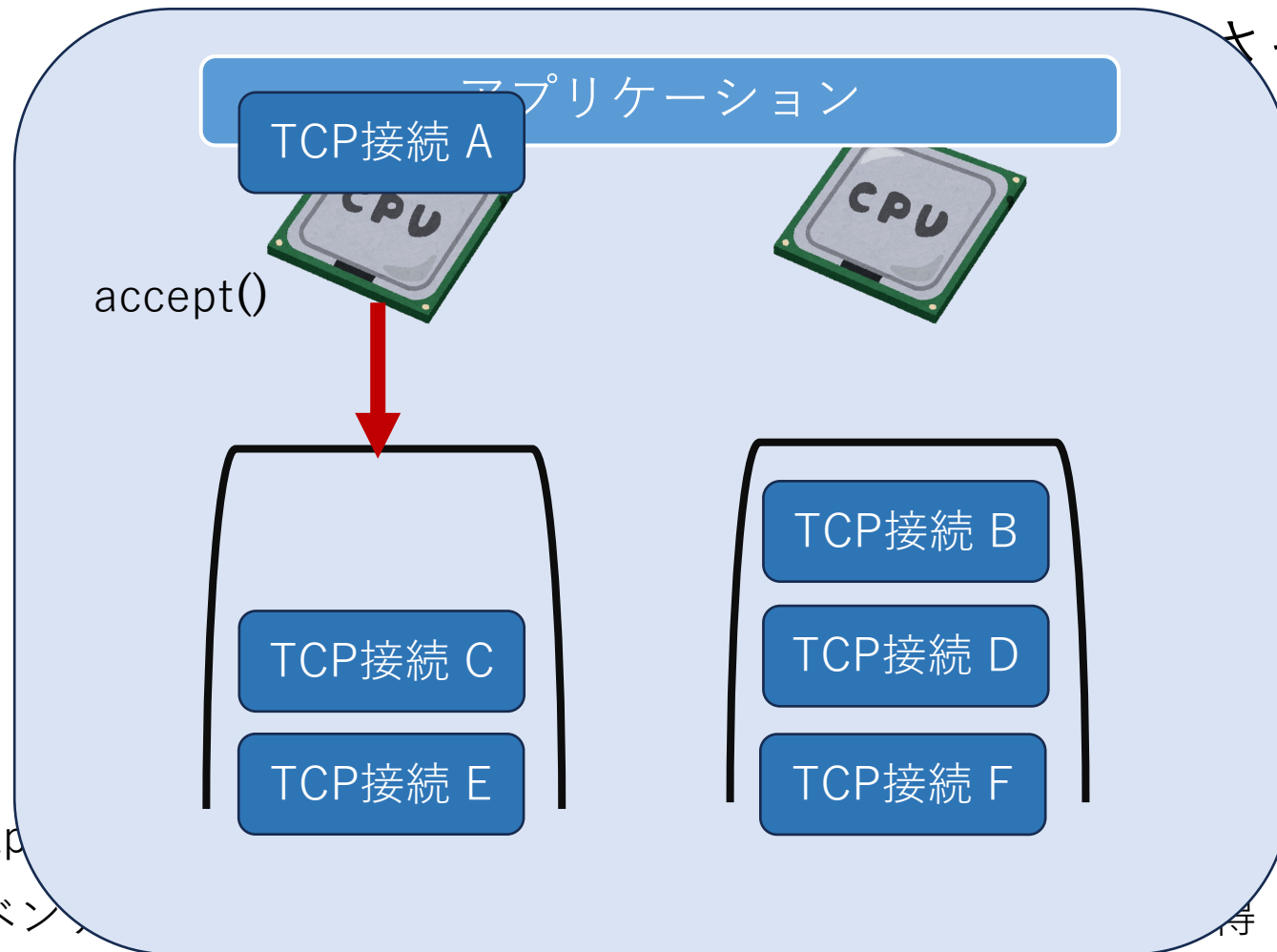


Ap
ベン

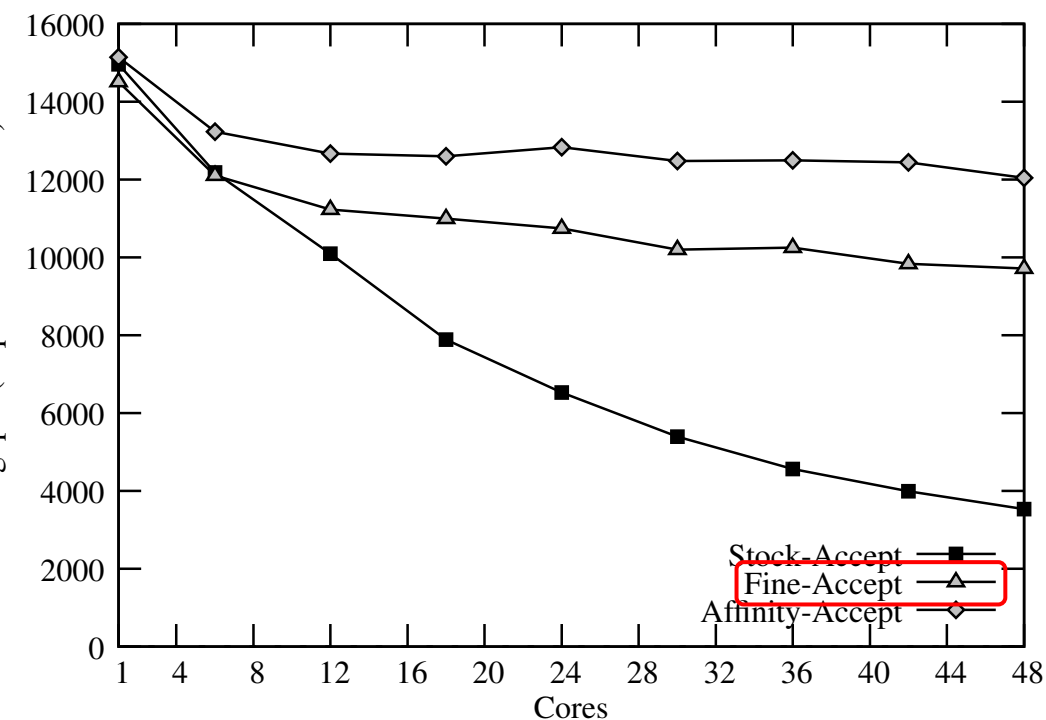
待

TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

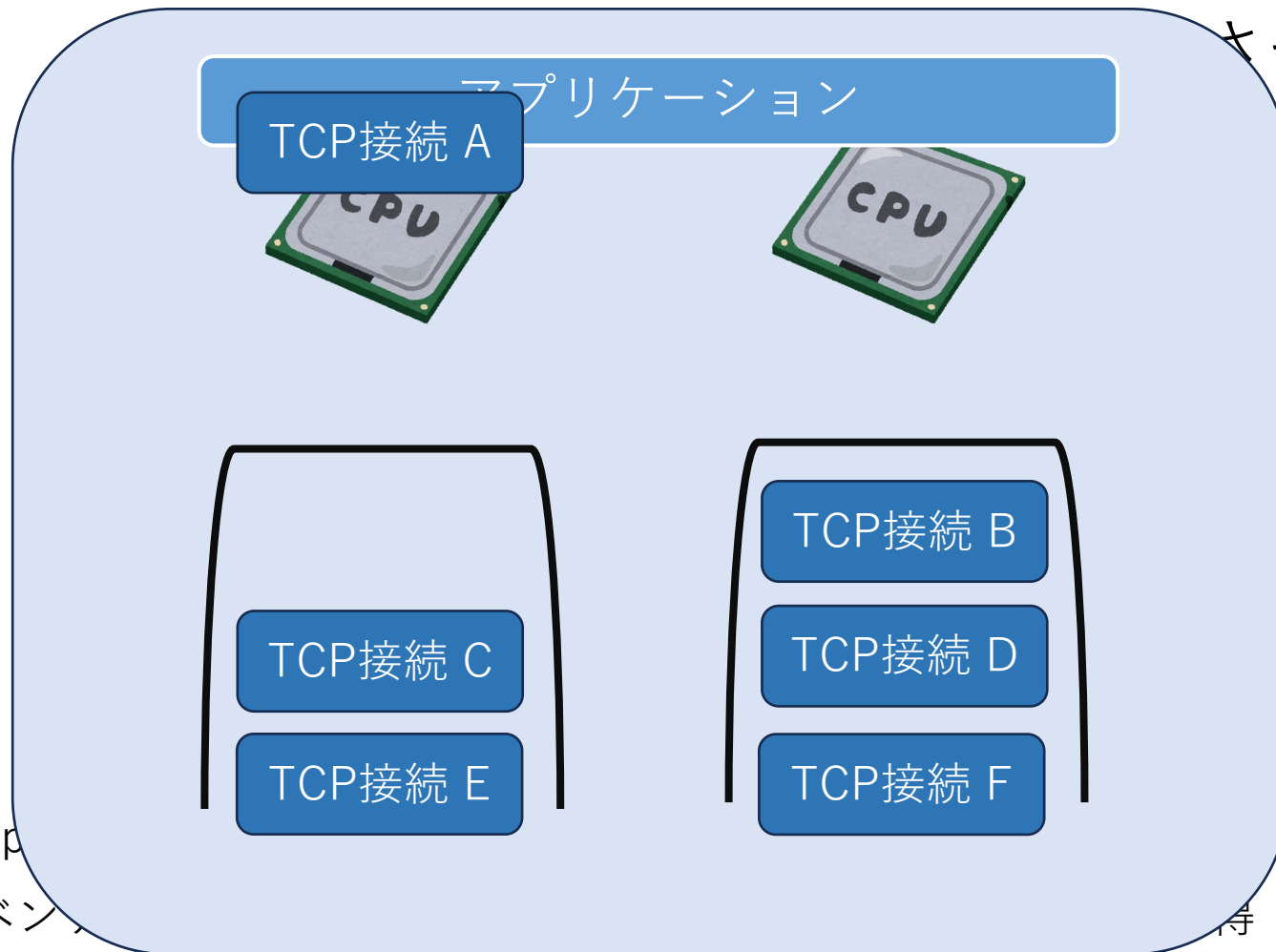


受けさせる
オブジェクトを減らす
キュー機能を利用

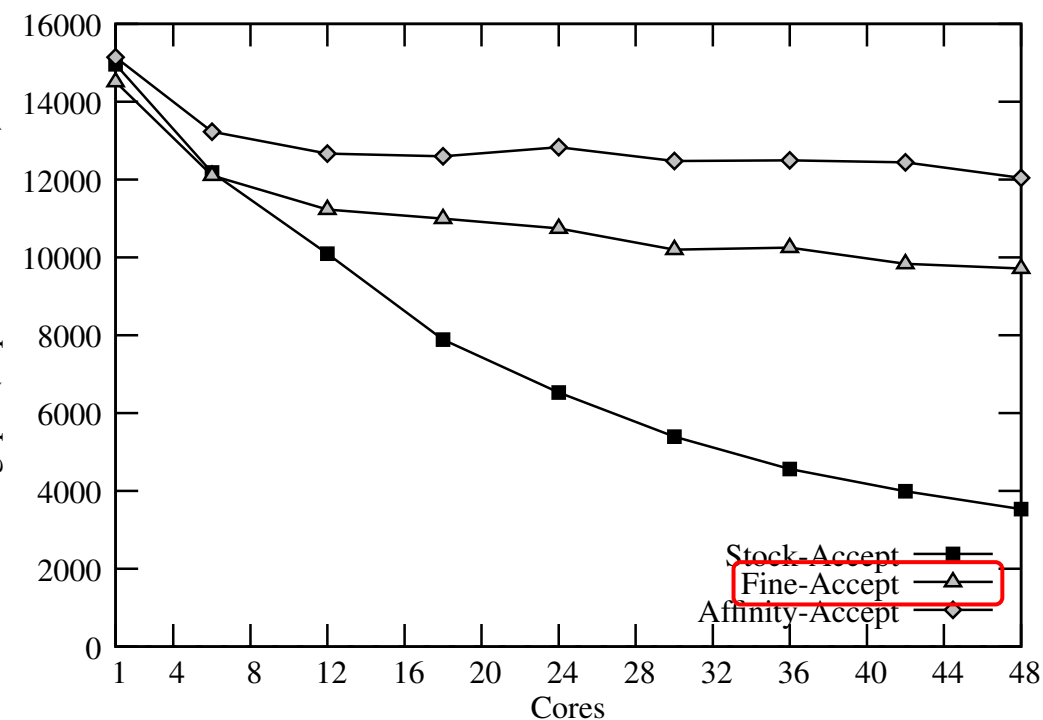


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

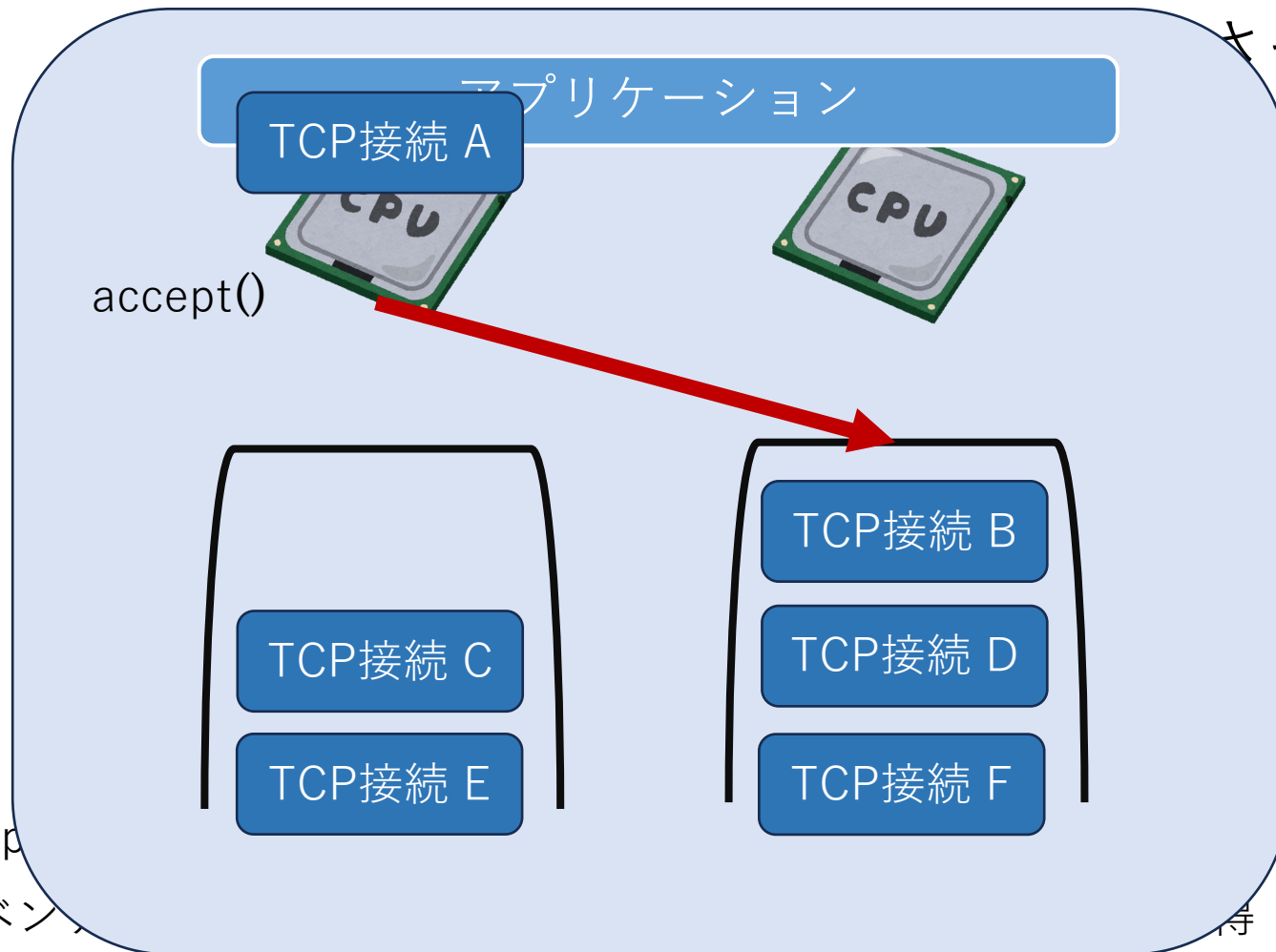


減らせる
オブジェクトを減らす
キュー機能を利用

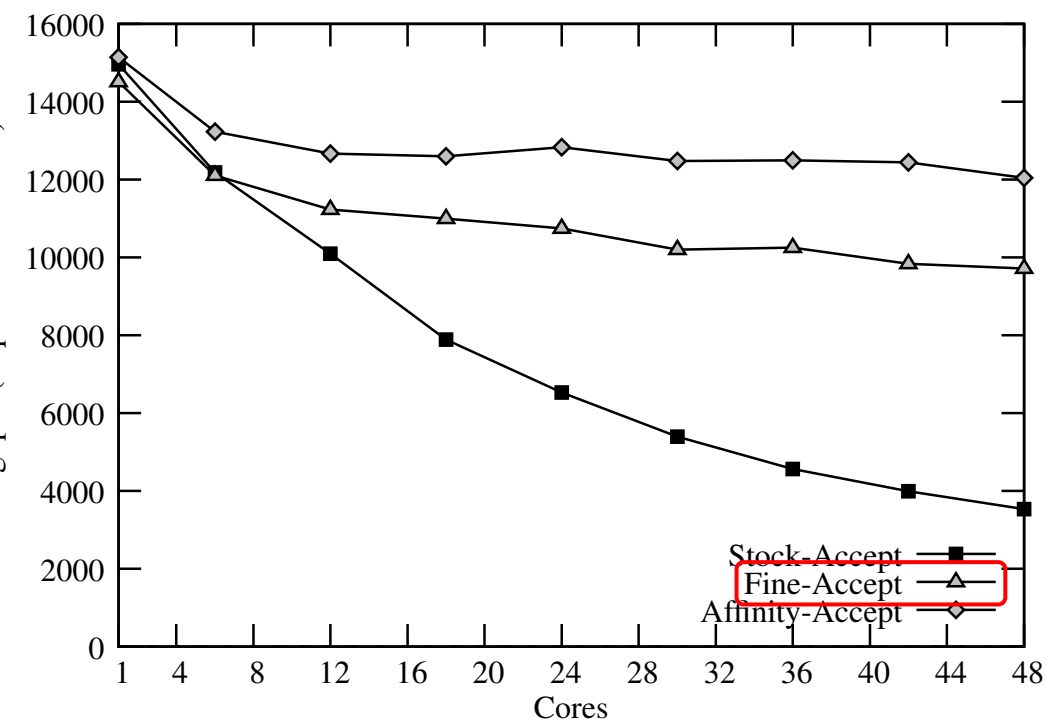


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

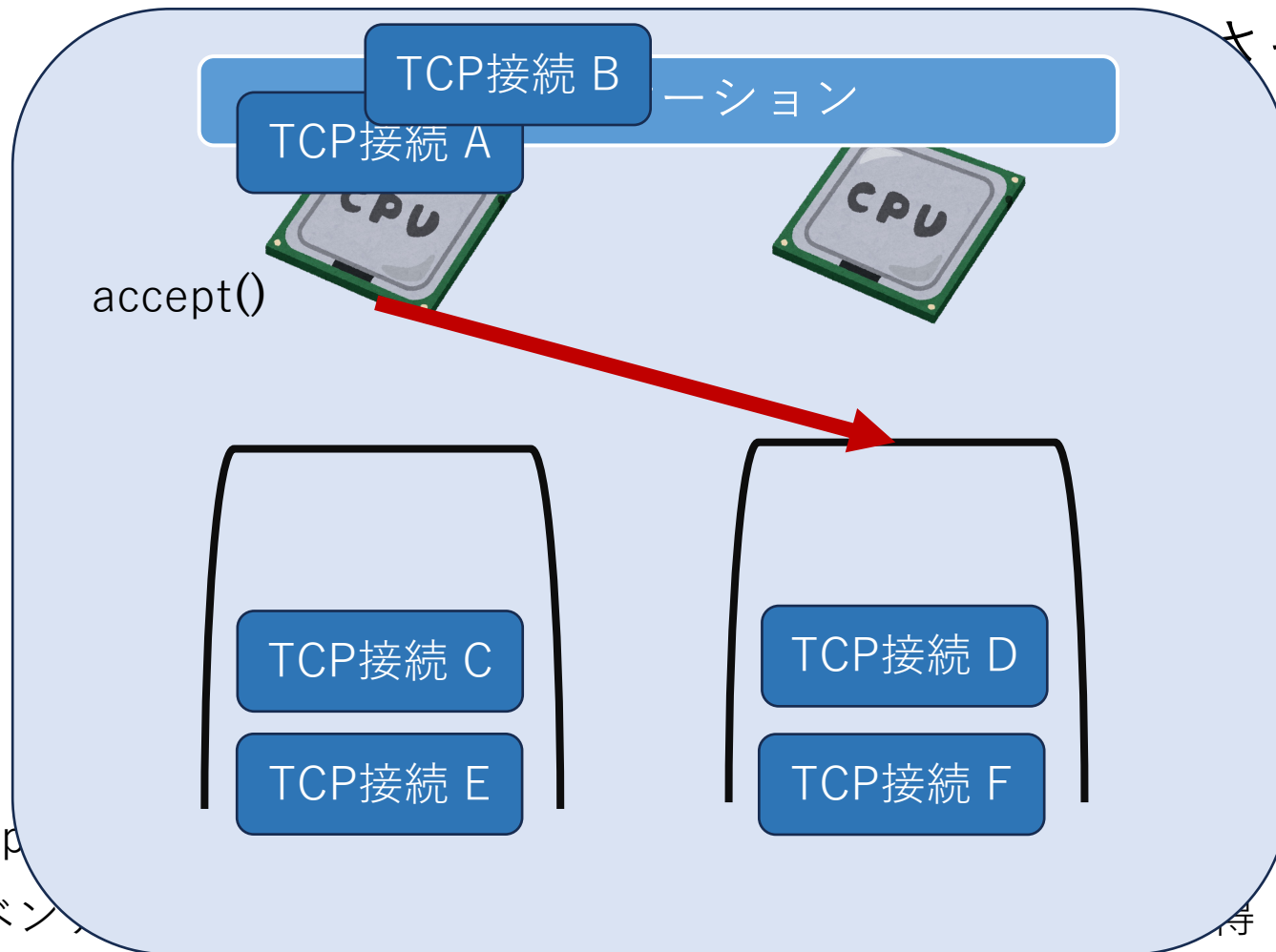


減らせる
オブジェクトを減らす
キュー機能を利用

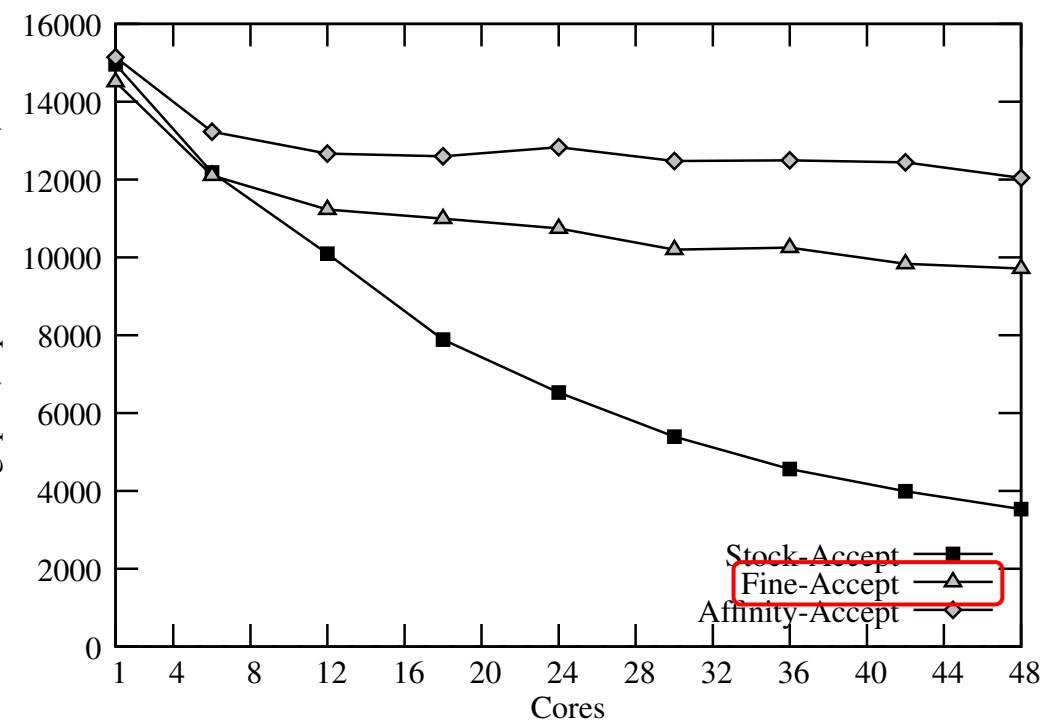


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

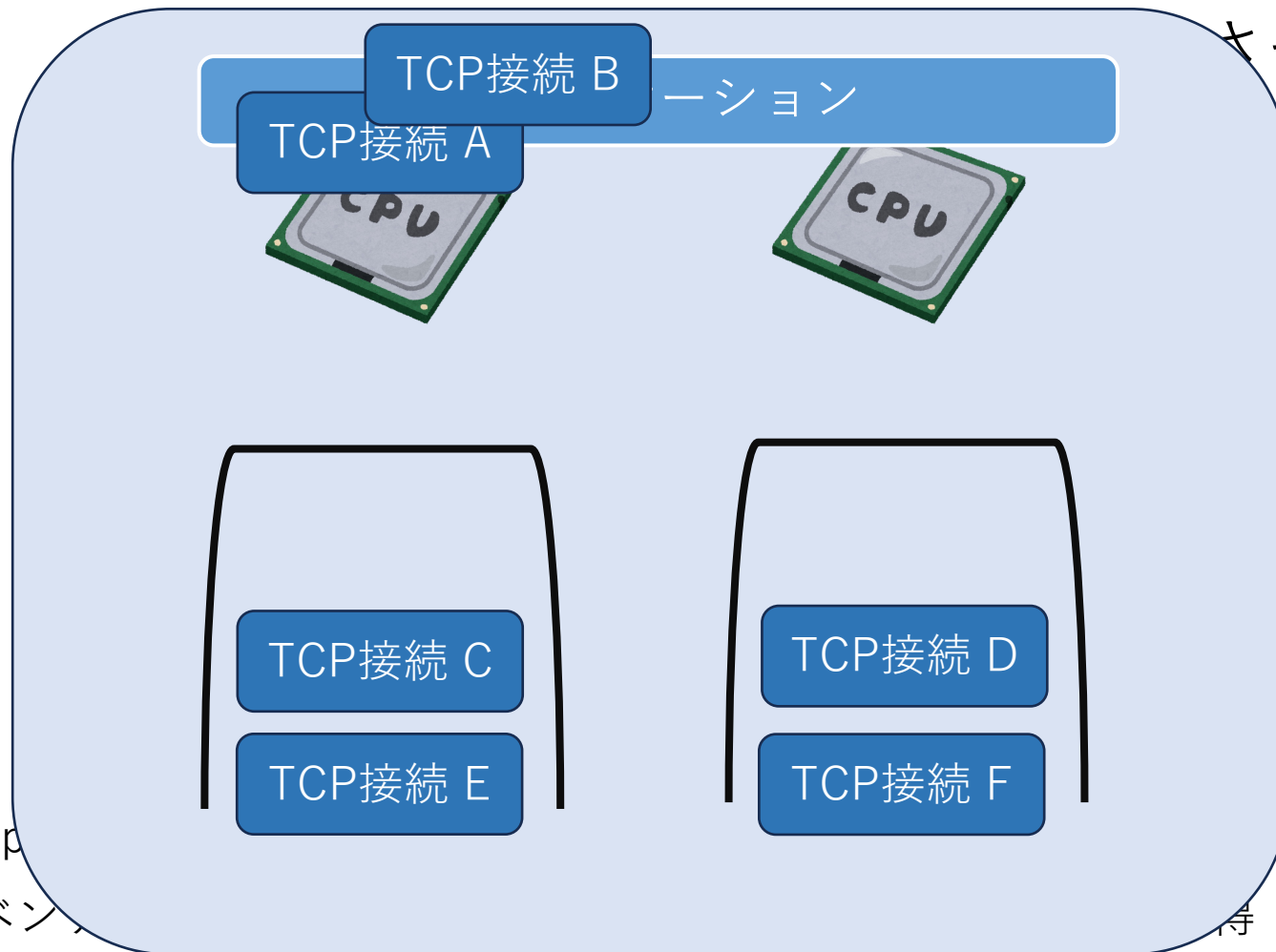


させる
オブジェクトを減らす
キュー機能を利用

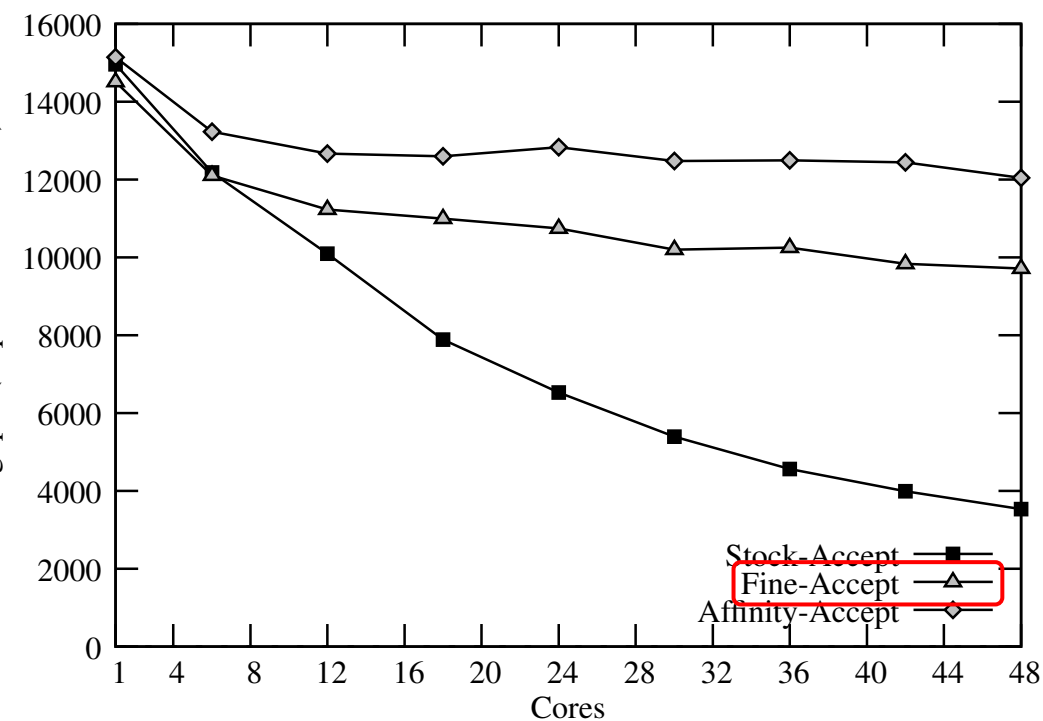


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

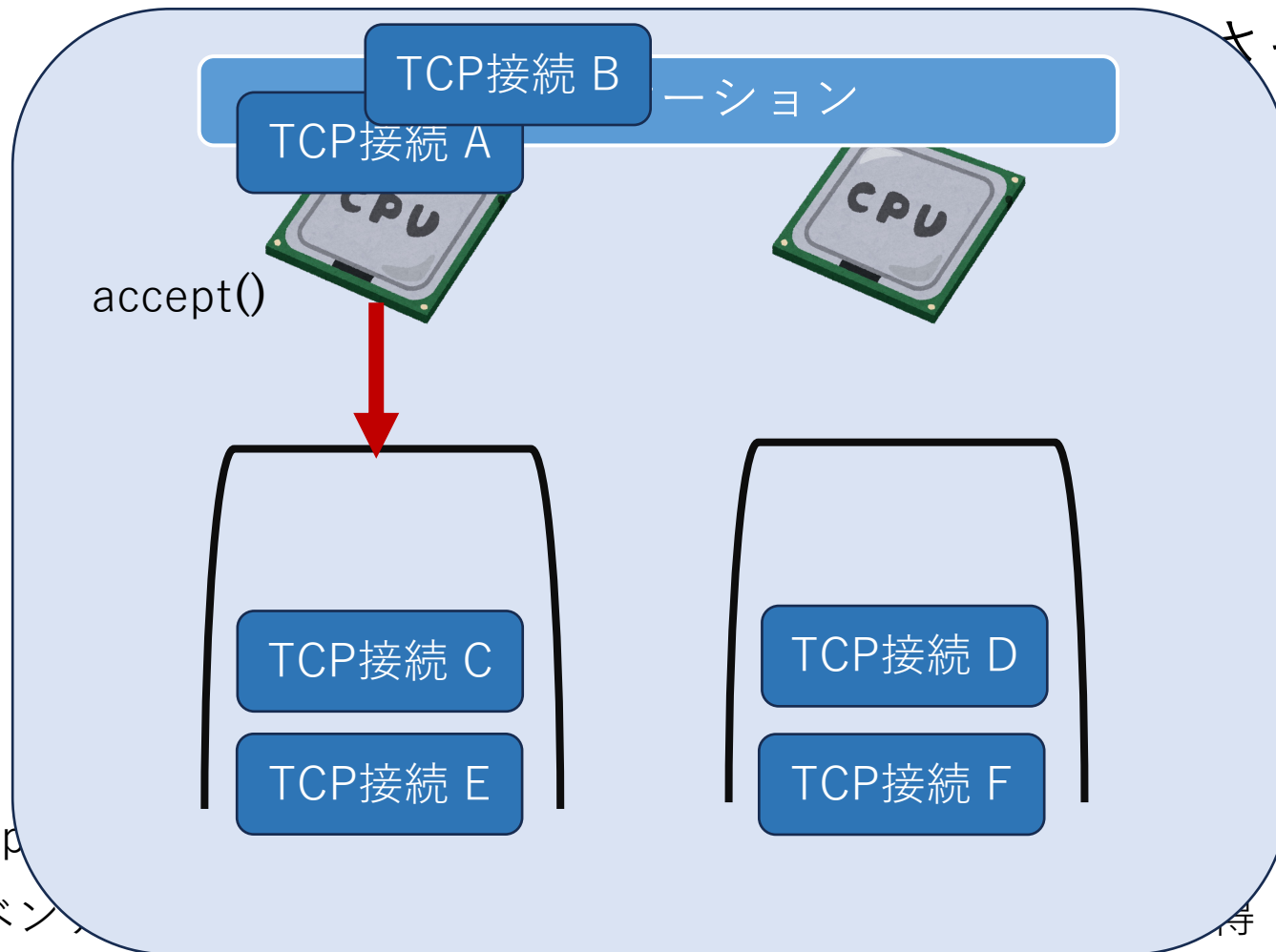


減らせる
オブジェクトを減らす
キュー機能を利用

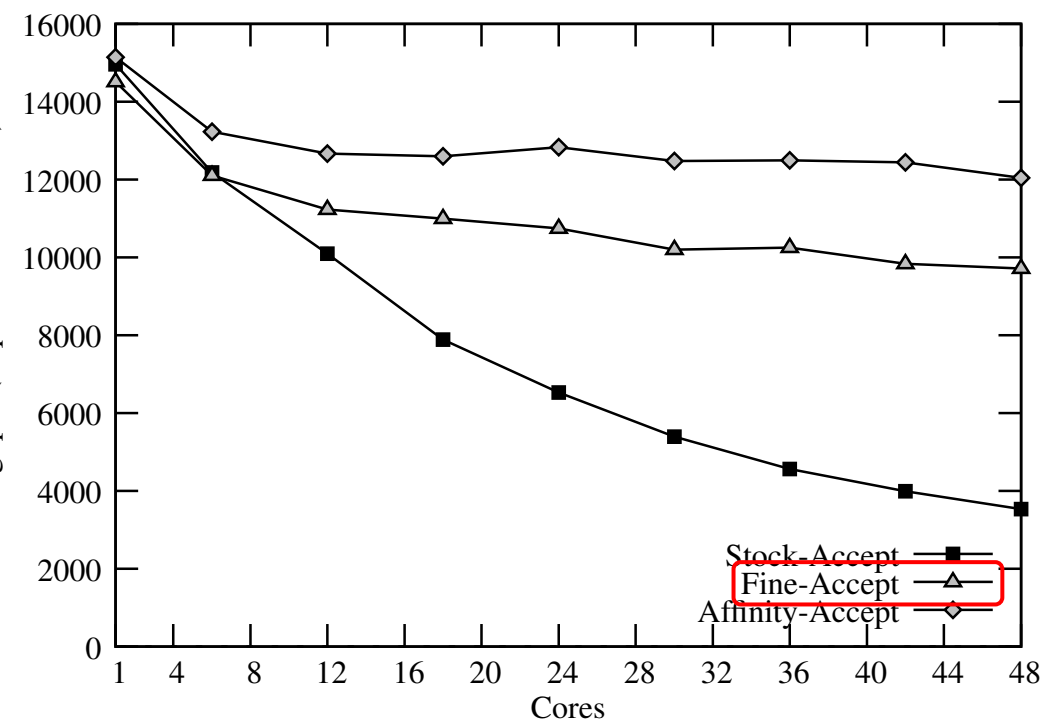


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

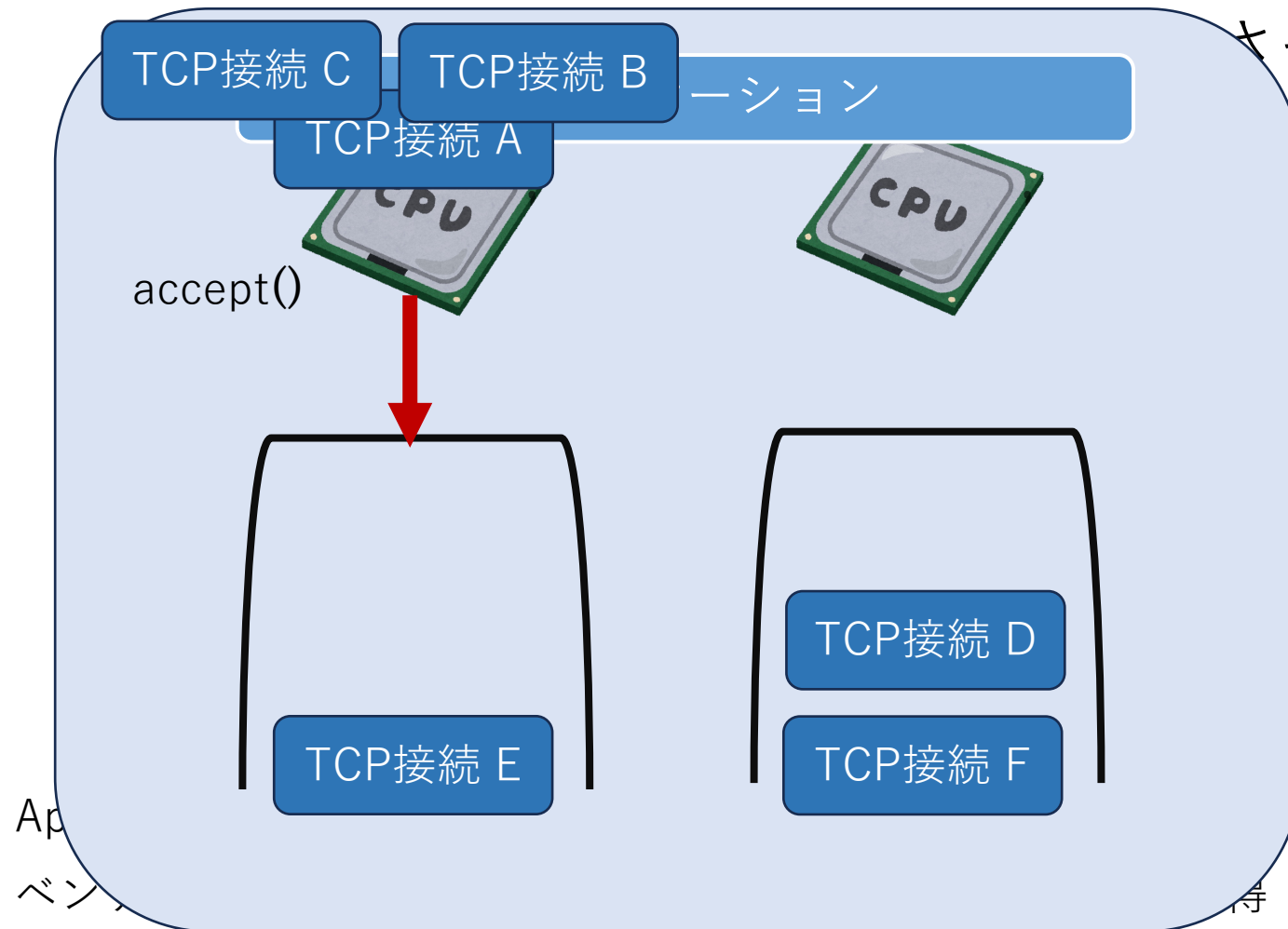


減らせる
オブジェクトを減らす
キュー機能を利用

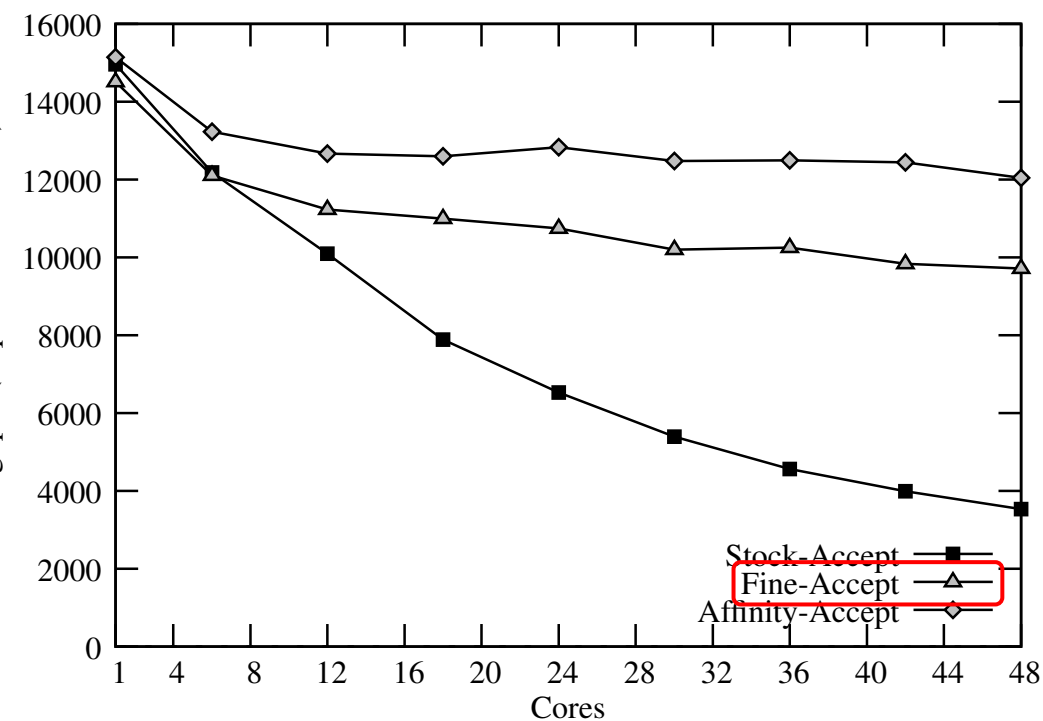


TCP/IP スタック設計の再考

Fine-Accept: 全ての accept キューからラウンドロビンで accept

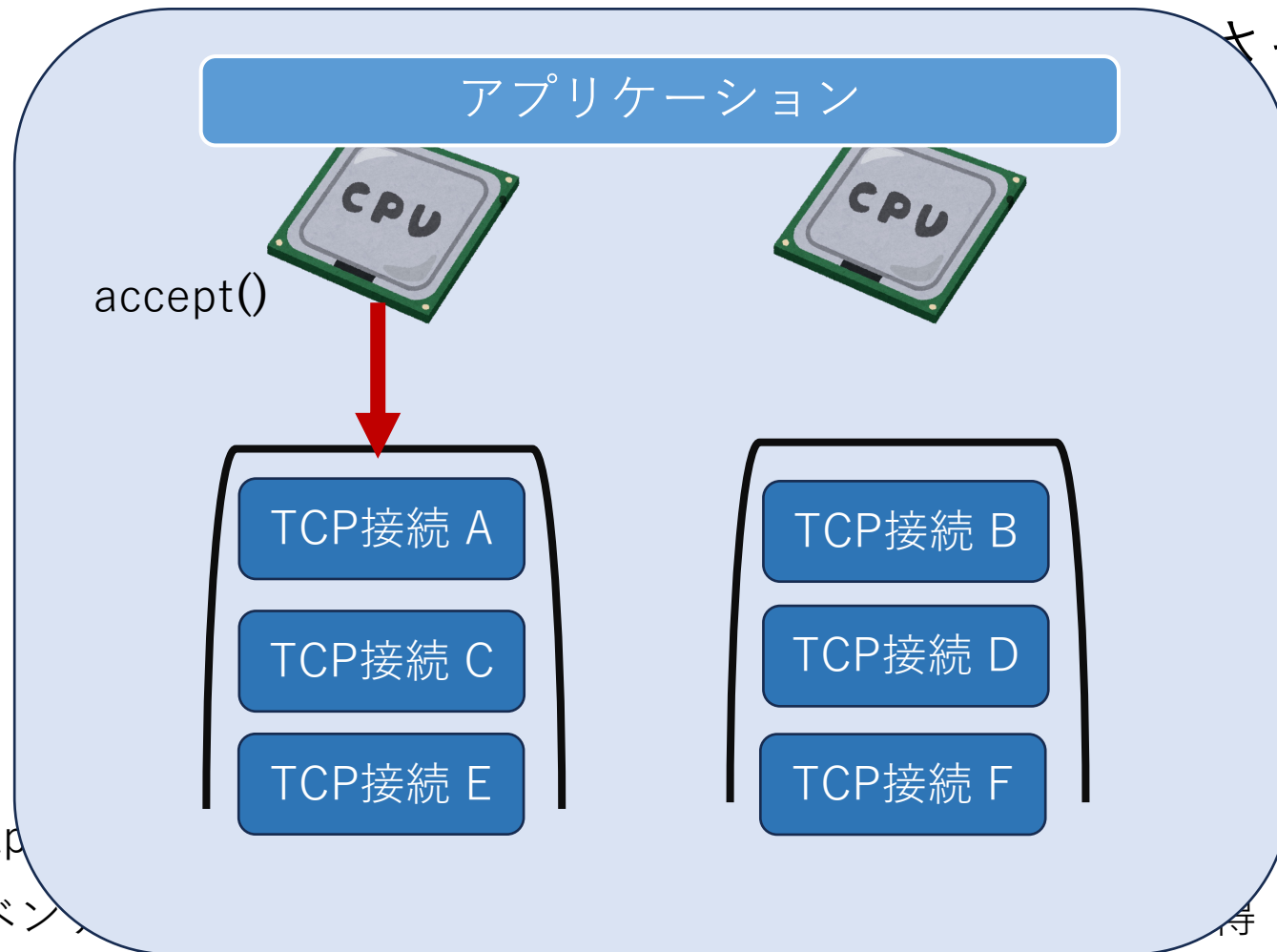


させる
オブジェクトを減らす
キュー機能を利用

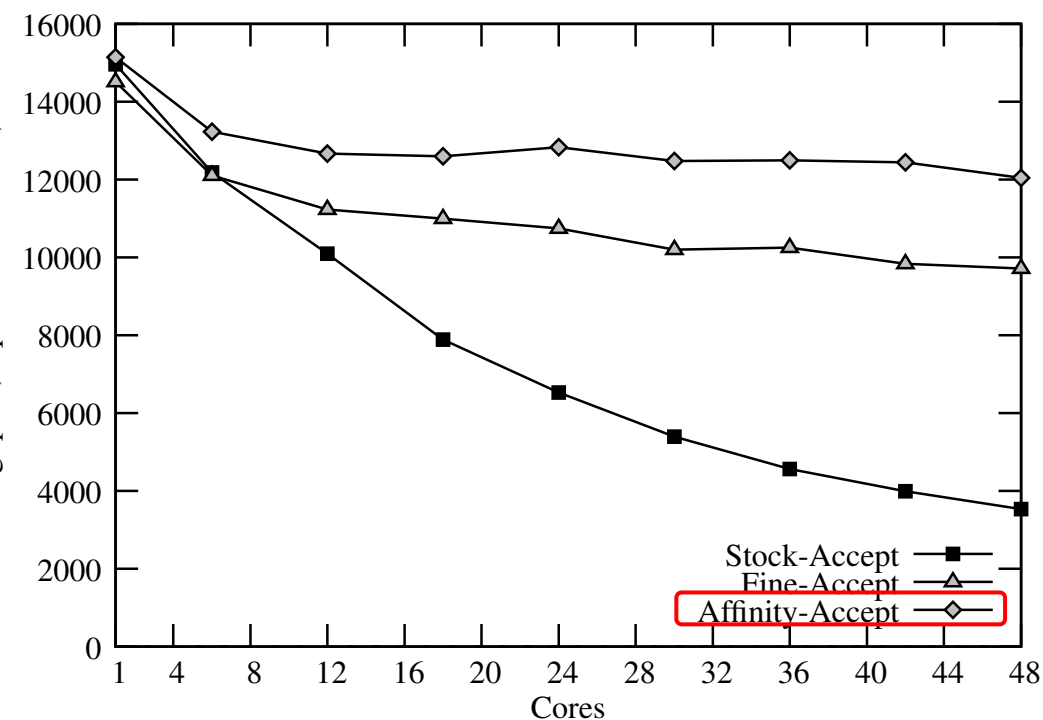


TCP/IP スタック設計の再考

Affinity-Accept: 基本的に特定の accept キューからのみ accept

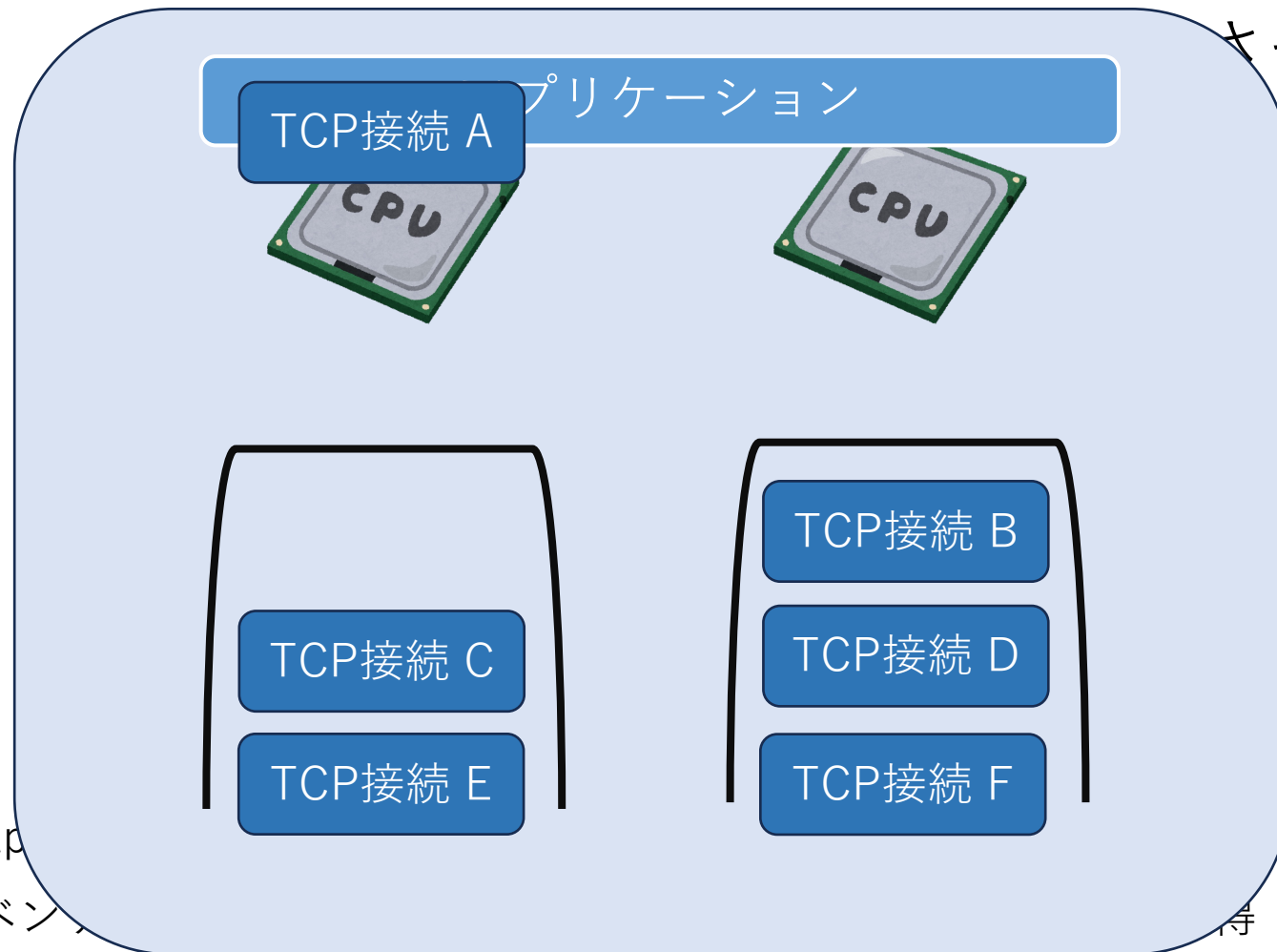


減らせる
オブジェクトを減らす
キュー機能を利用

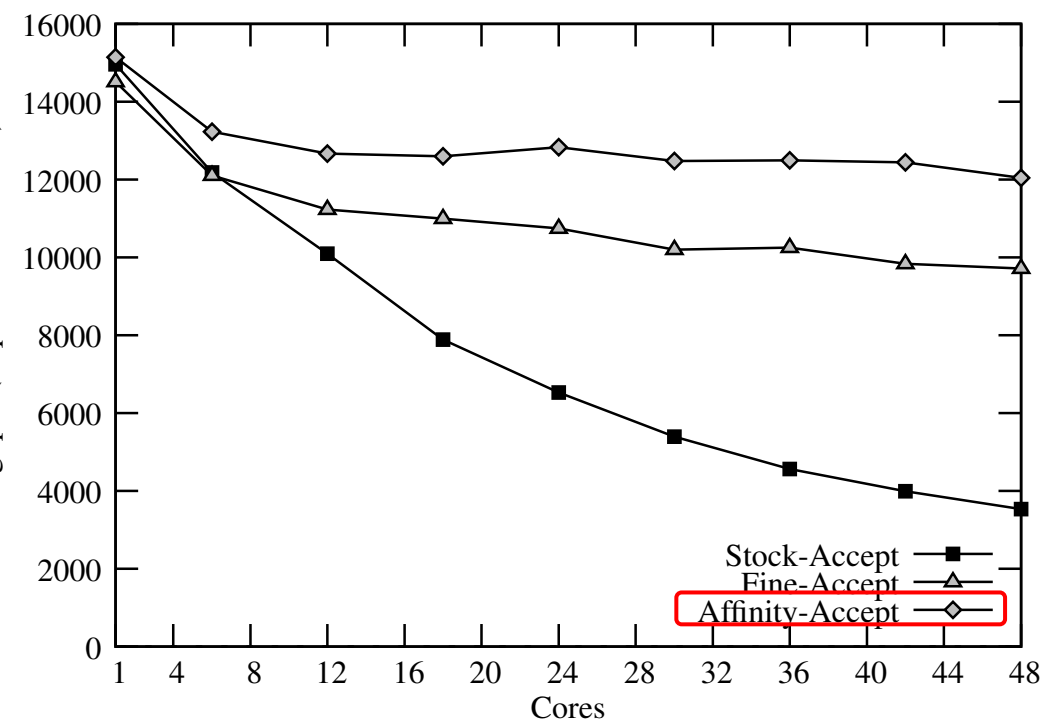


TCP/IP スタック設計の再考

Affinity-Accept: 基本的に特定の accept キューからのみ accept

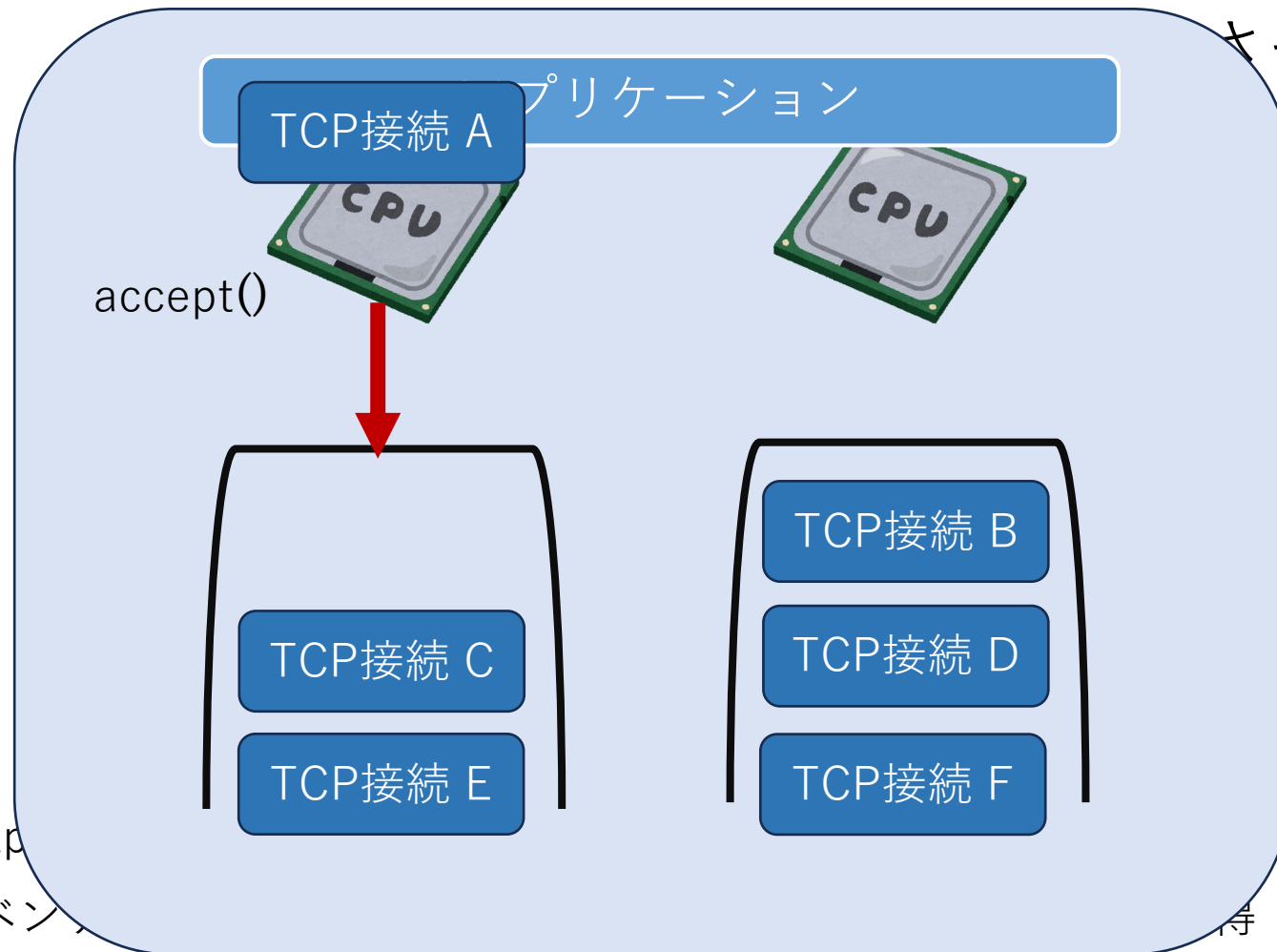


減らせる
オブジェクトを減らす
キュー機能を利用

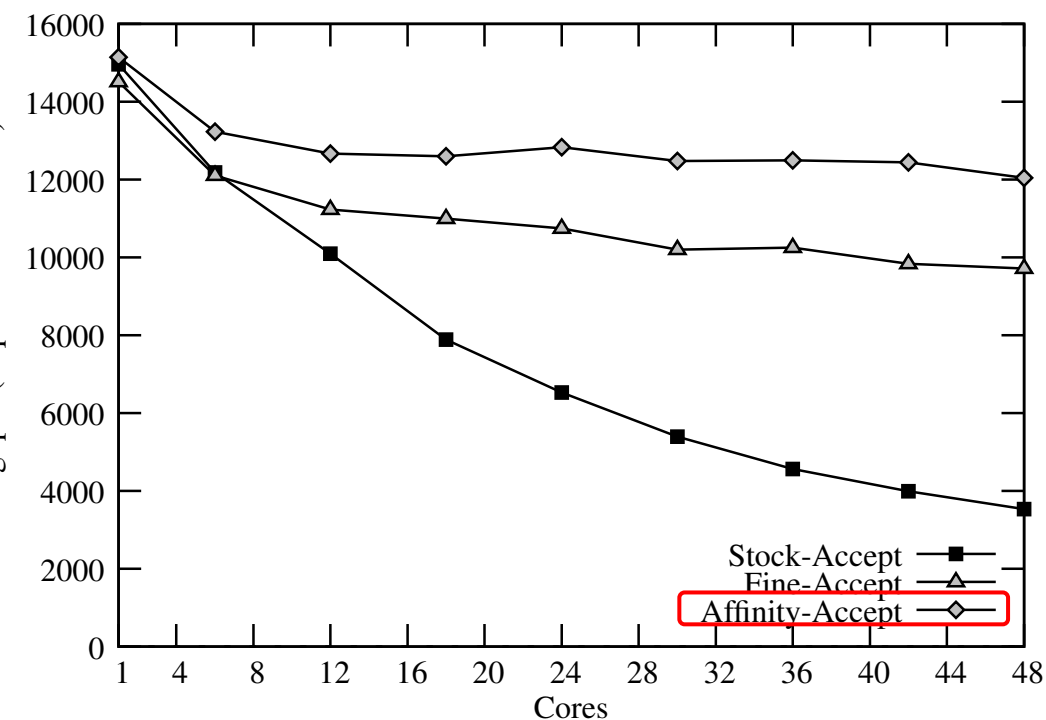


TCP/IP スタック設計の再考

Affinity-Accept: 基本的に特定の accept キューからのみ accept

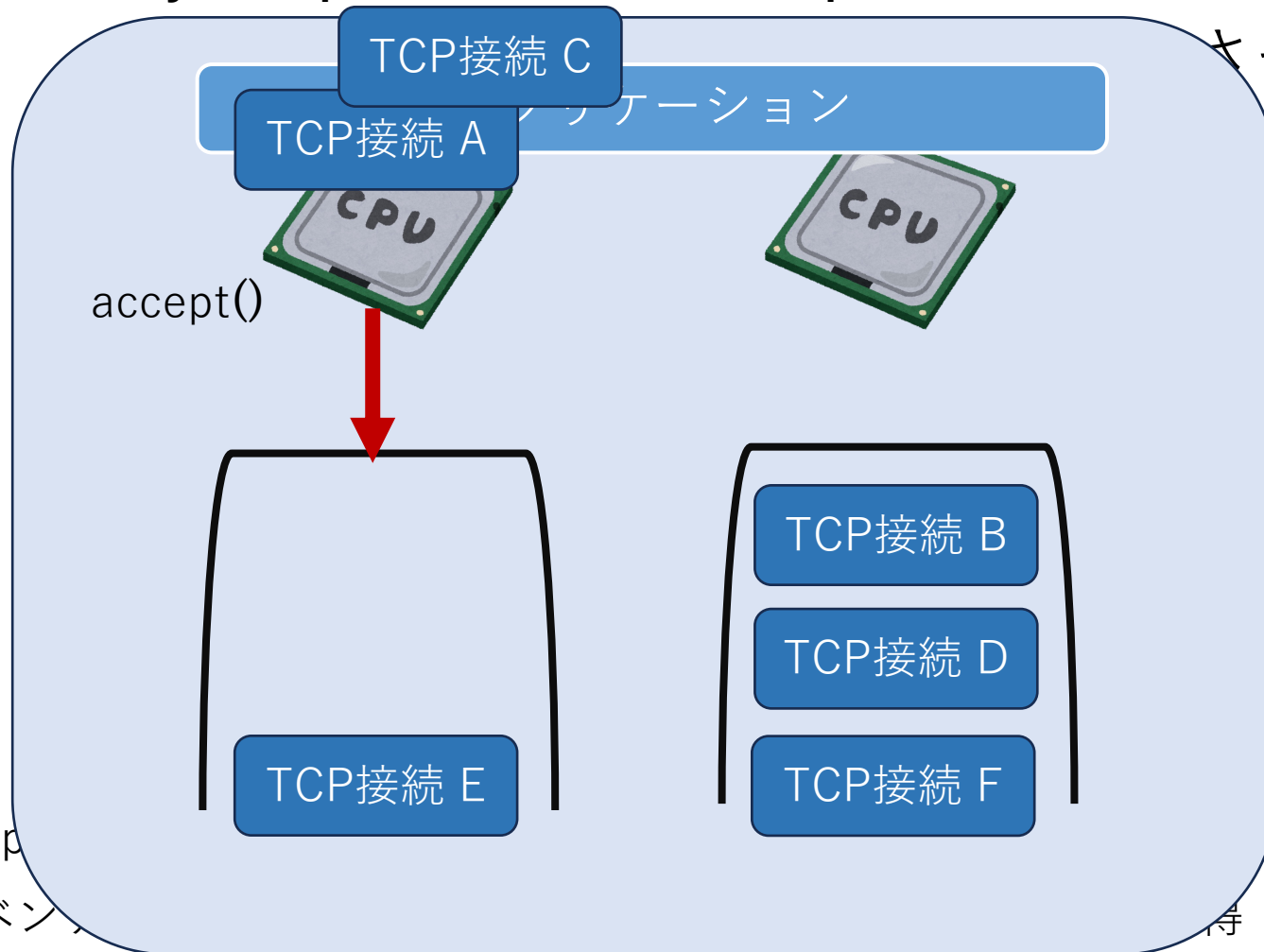


させる
オブジェクトを減らす
キュー機能を利用

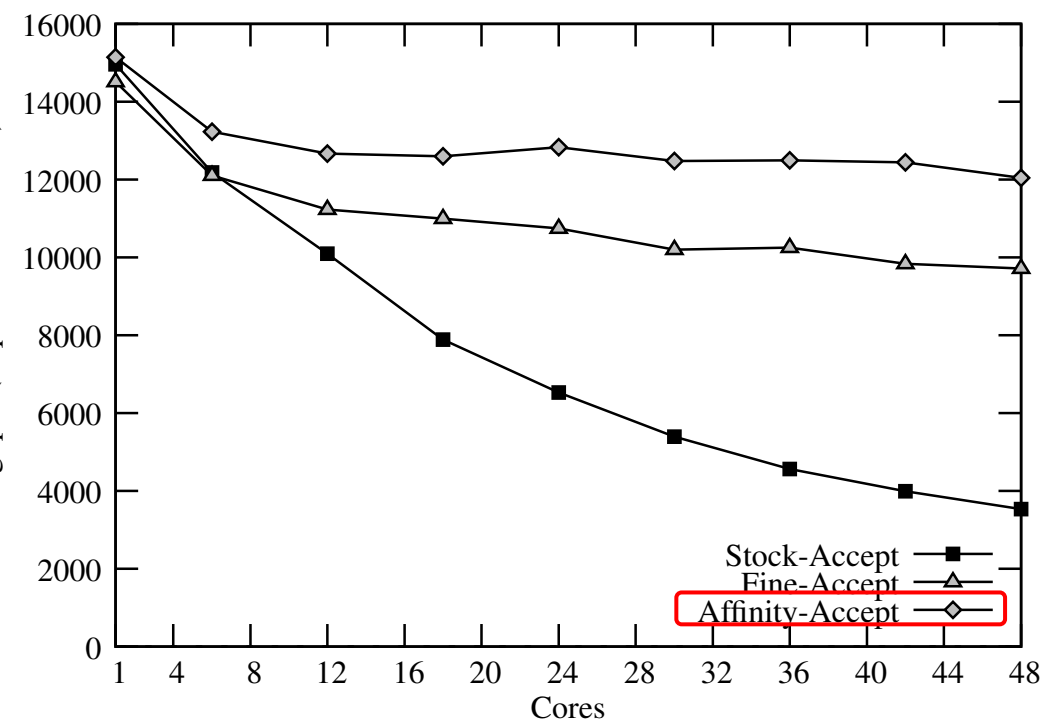


TCP/IP スタック設計の再考

Affinity-Accept: 基本的に特定の accept キューからのみ accept



受け取る
オブジェクトを減らす
キュー機能を利用

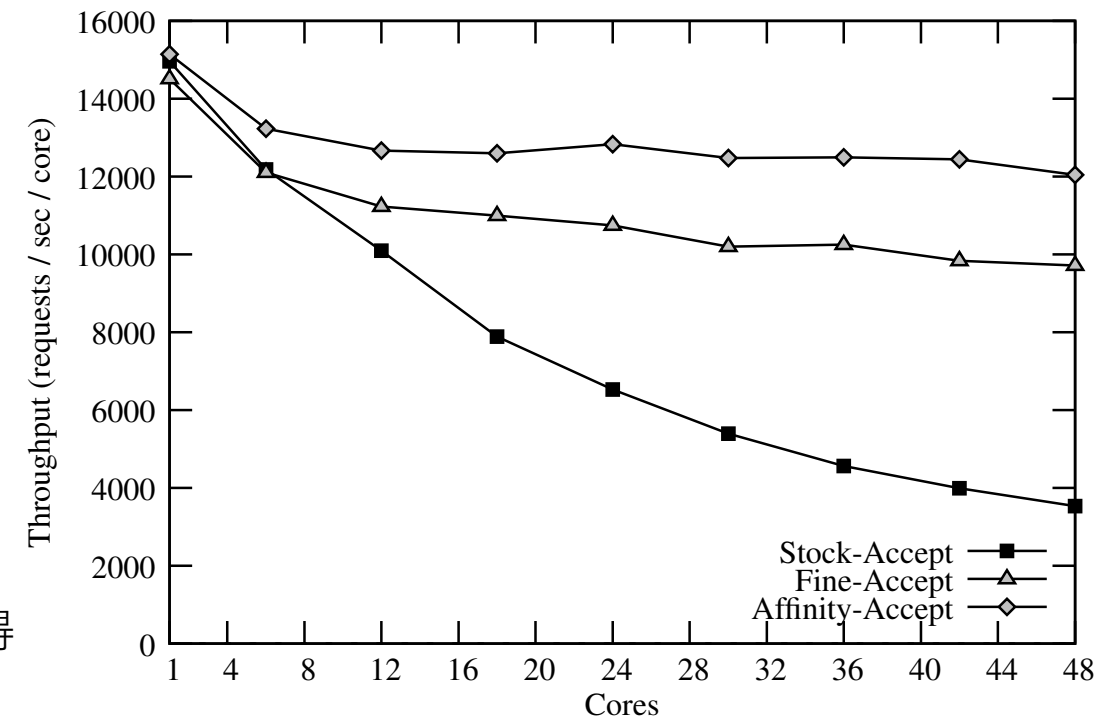


TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - **Affinity-Accept (EuroSys 2012)**
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

Apache HTTP server パフォーマンス

ベンチマーククライアントは1回の TCP 接続で6ファイル取得

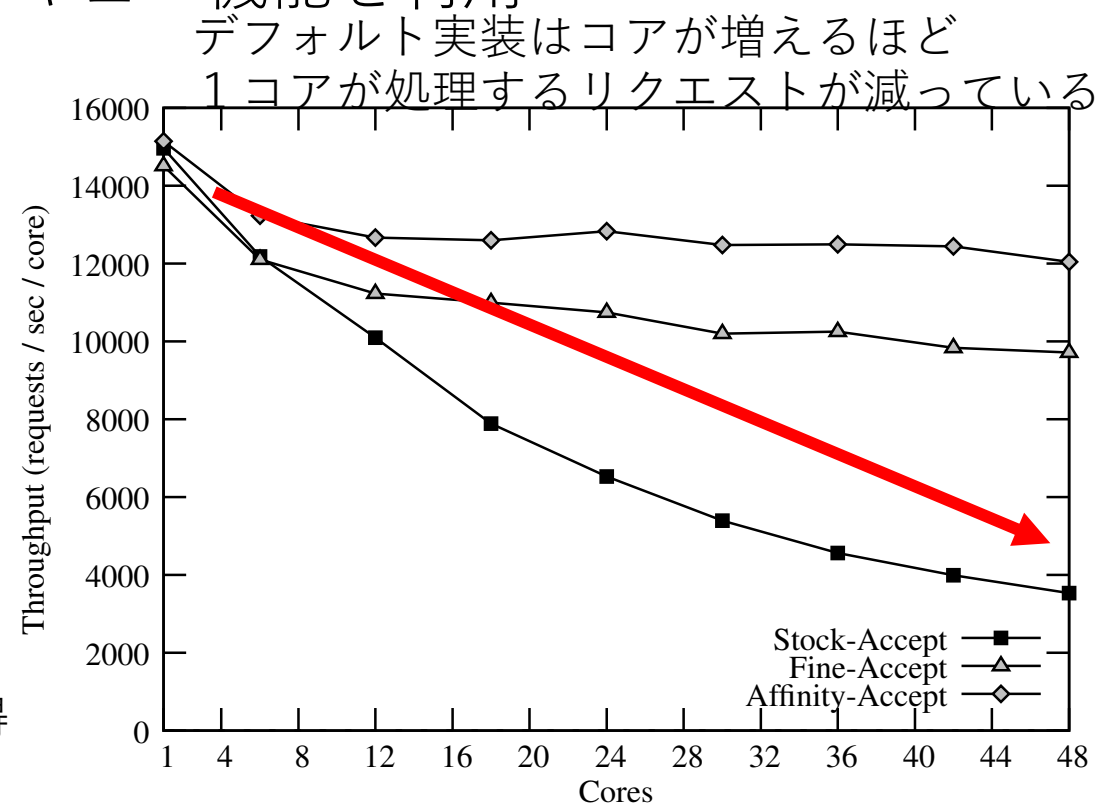


TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - **Affinity-Accept (EuroSys 2012)**
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

Apache HTTP server パフォーマンス

ベンチマーククライアントは1回の TCP 接続で6ファイル取得

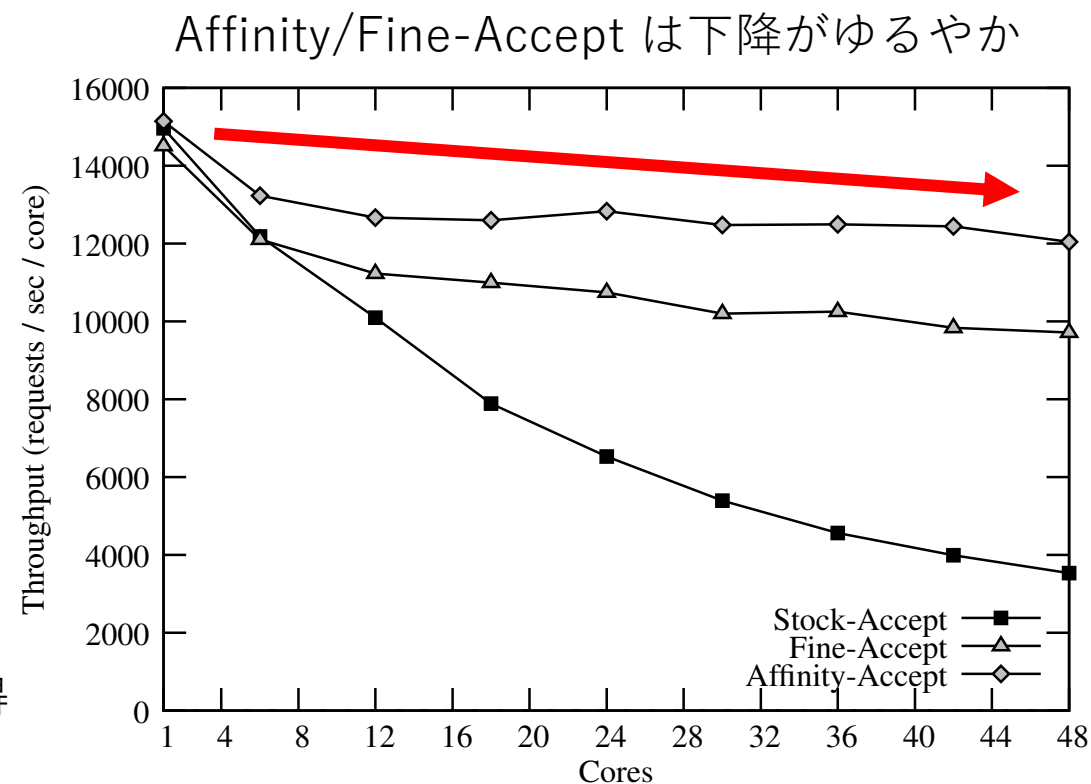


TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - **Affinity-Accept (EuroSys 2012)**
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

Apache HTTP server パフォーマンス

ベンチマーククライアントは1回の TCP 接続で6ファイル取得

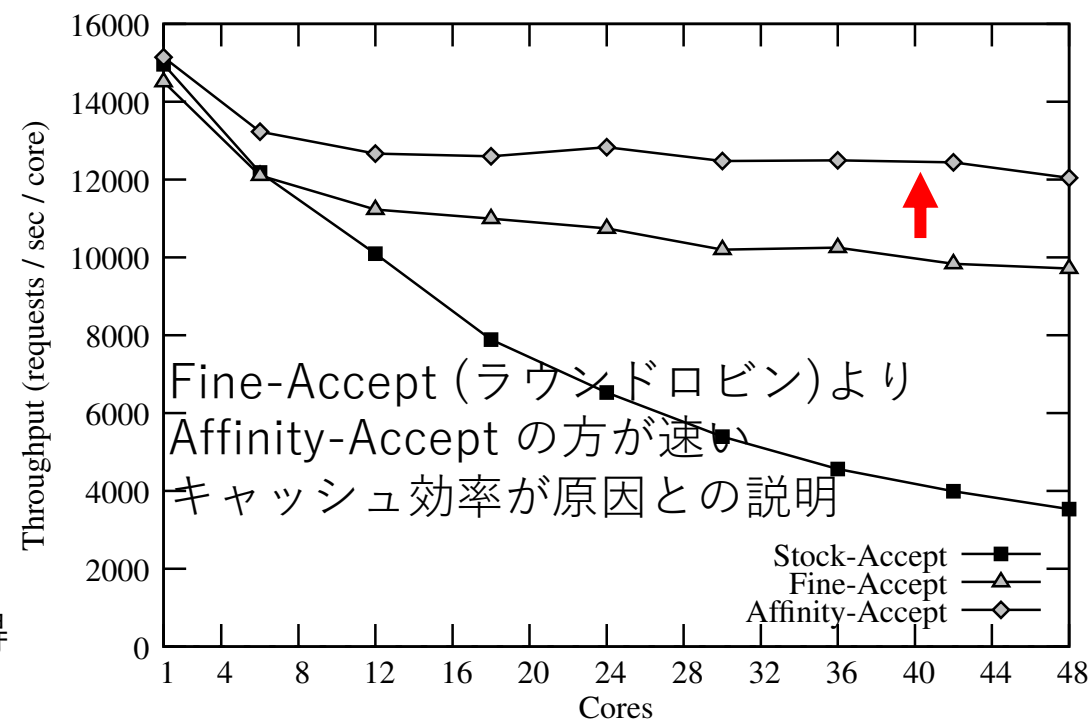


TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - **Affinity-Accept (EuroSys 2012)**
 - MegaPipe (OSDI 2012)
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

Apache HTTP server パフォーマンス

ベンチマーククライアントは1回の TCP 接続で6ファイル取得



TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
 - accept のスケーラビリティに関して
 - Affinity-Accept (EuroSys 2012)
 - **MegaPipe (OSDI 2012)**
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

POSIX socket に変わる API の提案
- accept のキューをコアごとに分ける

TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
 - accept のスケーラビリティに関して
 - Affinity-Accept (EuroSys 2012)
 - **MegaPipe (OSDI 2012)**
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLoS 2016)

POSIX socket に変わる API の提案

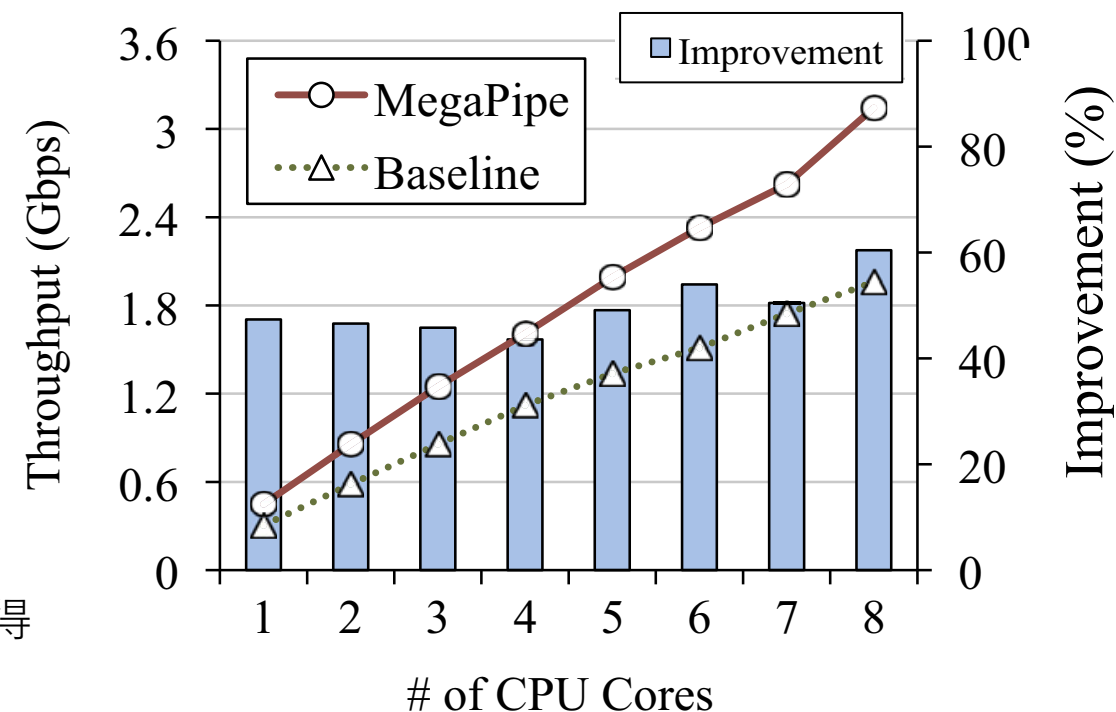
- accept のキューをコアごとに分ける
- ファイルデスクリプタのテーブルも分ける
- 複数のリクエストをバッチ可能 (FlexSC と同様の効果を期待)

TCP/IP スタック設計の再考

- マルチコア環境で性能をスケールさせる
 - 基本的なアイデア：コア間で共有するオブジェクトを減らす
 - NIC のキューについて：NIC のマルチキュー機能を利用
 - Receive Side Scaling (RSS) も利用
- accept のスケーラビリティに関して
 - Affinity-Accept (EuroSys 2012)
 - **MegaPipe (OSDI 2012)**
 - mTCP (NSDI 2014)
 - Fastsocket (ASPLOS 2016)

nginx HTTP server パフォーマンス

ベンチマーククライアントは1回の TCP 接続で6ファイル取得



研究紹介

TCP/IP スタック設計

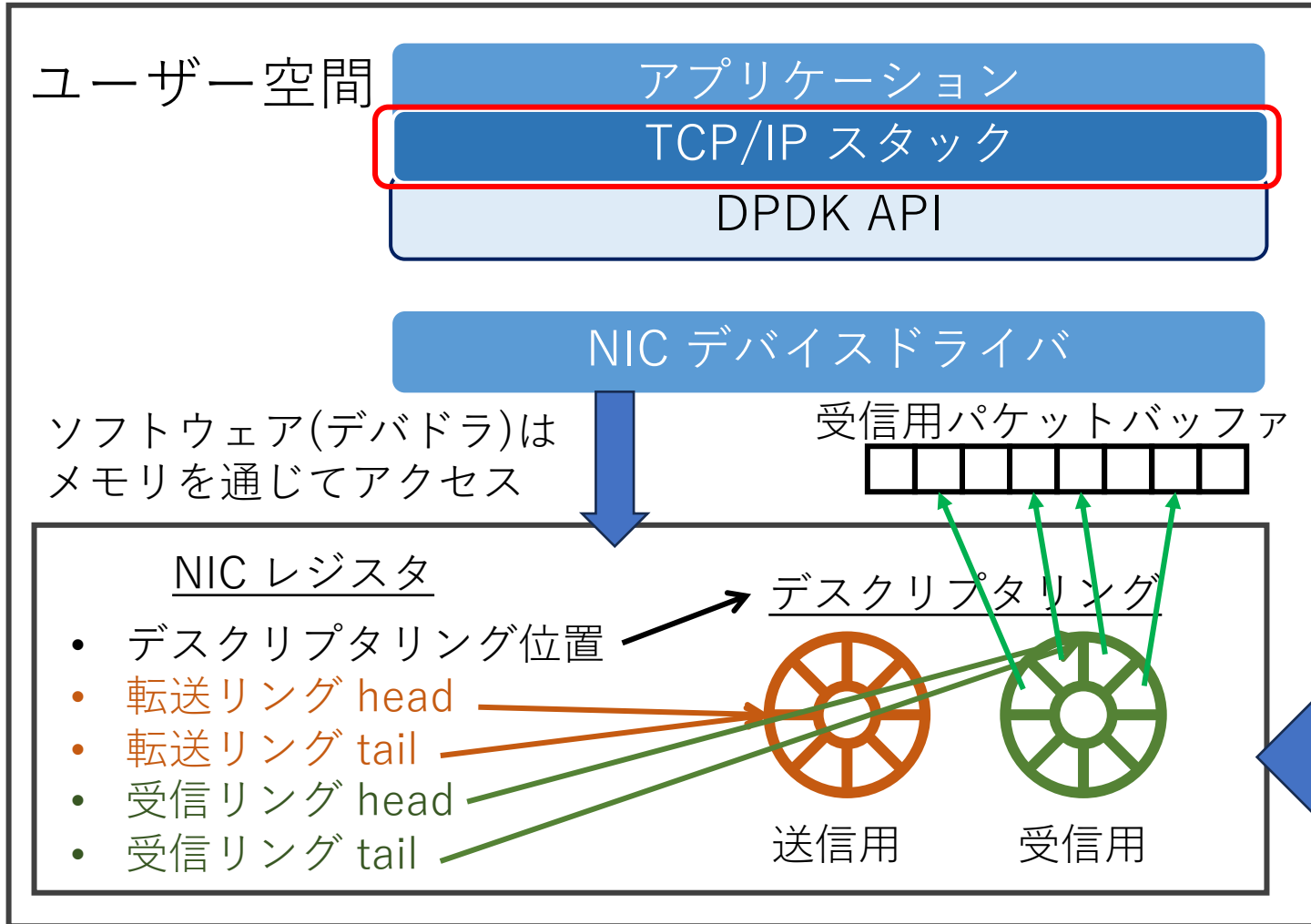
パケット I/O フレームワークを適用する

ユーザー空間 TCP/IP スタック実装

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - zIO (OSDI 2021)
 - Demikernel (SOSP 2021)

パケット I/O フレームワークの用途

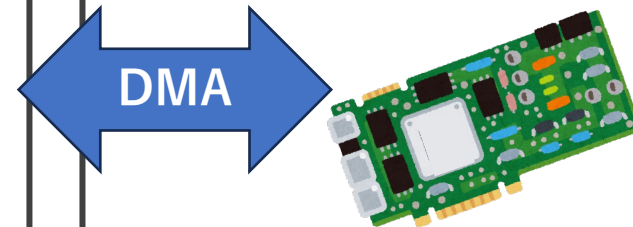


Network Function Virtualization (NFV)

汎用的なサーバーでネットワーク機能を動かす (e.g., Firewall, Router)

サーバープログラムの高速化

ユーザー空間で動作する
TCP/IP スタックと組み合わせる



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - **Sandstorm (SIGCOMM 2014)** Web サーバー
netmap + ユーザー空間 TCP/IP スタック
(コンテンツをパケットバッファ上に事前配置)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- **Sandstorm (SIGCOMM 2014)**

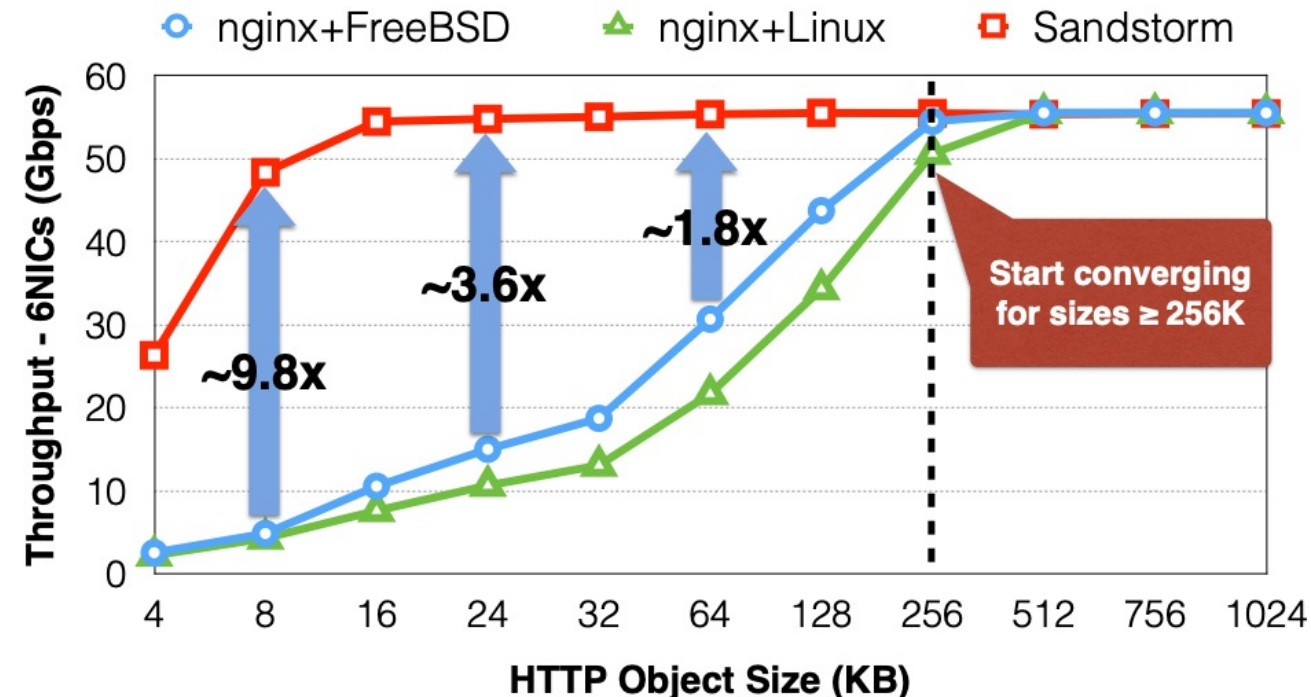
- mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

Web サーバー

netmap + ユーザー空間 TCP/IP スタック

(コンテンツをパケットバッファ上に事前配置)

コンテンツ配送速度 (6つの 10Gbps NIC 合計)



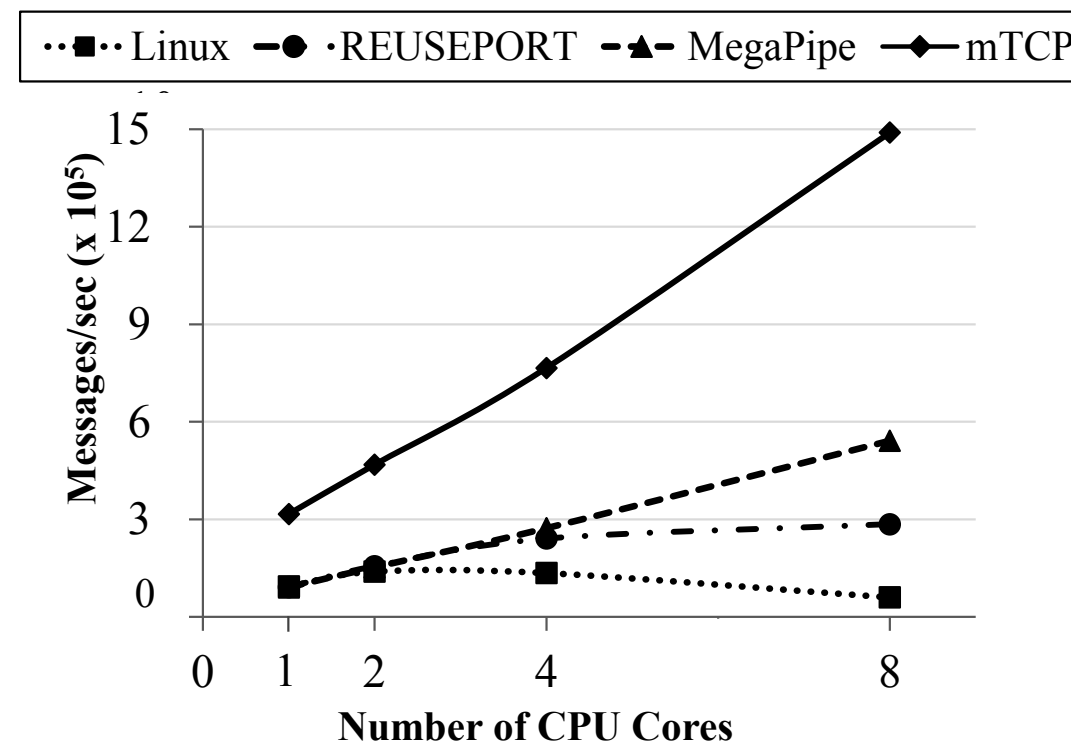
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - **mTCP (NSDI 2014)**
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)
- ユーザー空間 TCP/IP スタック
- accept キュー等をコアごとに用意
 - リクエストをバッチ

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - **mTCP (NSDI 2014)**
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

ユーザー空間 TCP/IP スタック
- accept キュー等をコアごとに用意
- リクエストをバッチ



TCP/IP スタック設計の再考

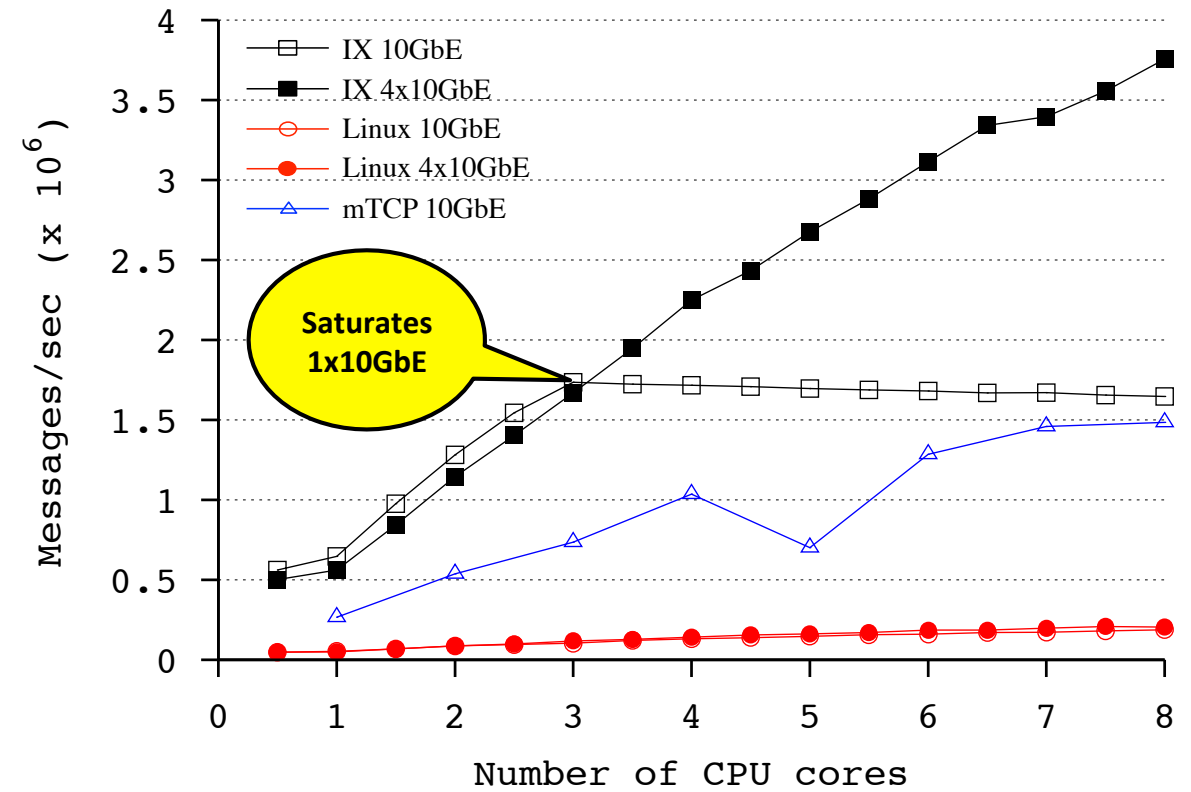
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - **Arrakis (OSDI 2014)**
 - **IX (OSDI 2014)**
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)
- 新しい OS
- アプリ + lwIP が直接 NIC へアクセス
 - NIC の SR-IOV 機能で多重化

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - **IX (OSDI 2014)**
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

新しい OS

- アプリ + lwIP が直接 NIC へアクセス
- NIC の SR-IOV 機能で多重化



研究紹介

TCP/IP スタック設計

パケット I/O フレームワークを適用する

既存の OS の TCP/IP スタックを使えるようにする

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

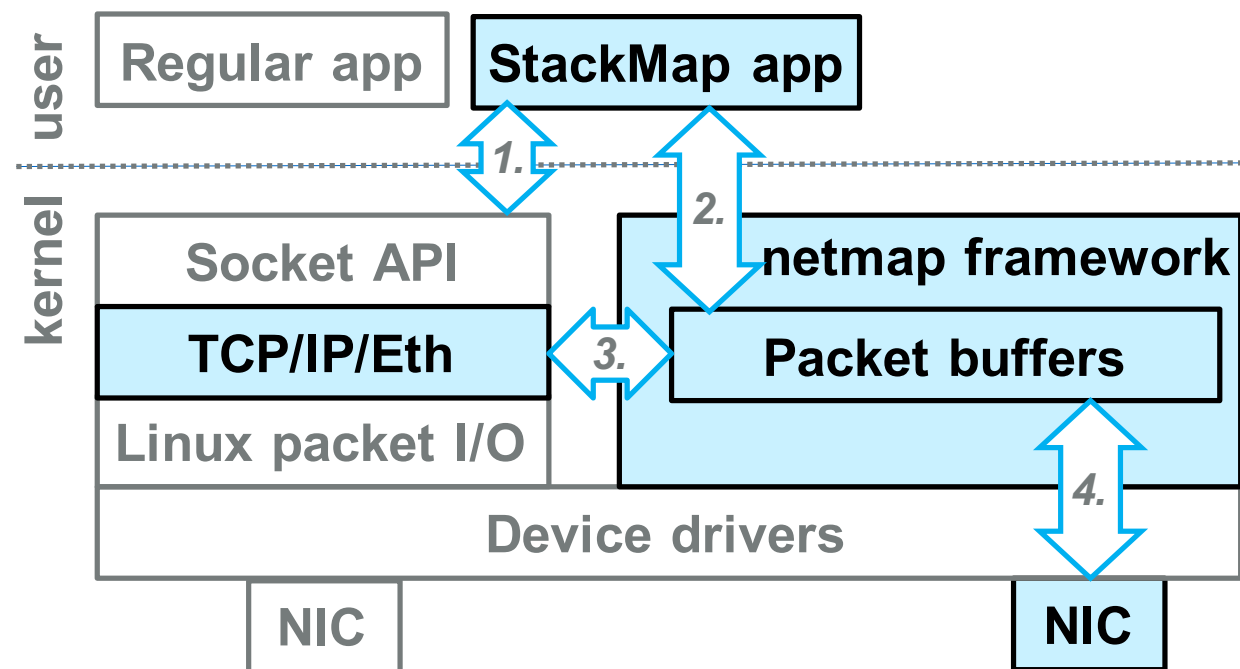
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)

- **StackMap (USENIX ATC 2016)**

- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

netmap + カーネル TCP/IP スタック

- アプリの API とデータパスは (ほぼ) netmap
- ヘッダの処理にカーネル TCP/IP スタックを利用



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

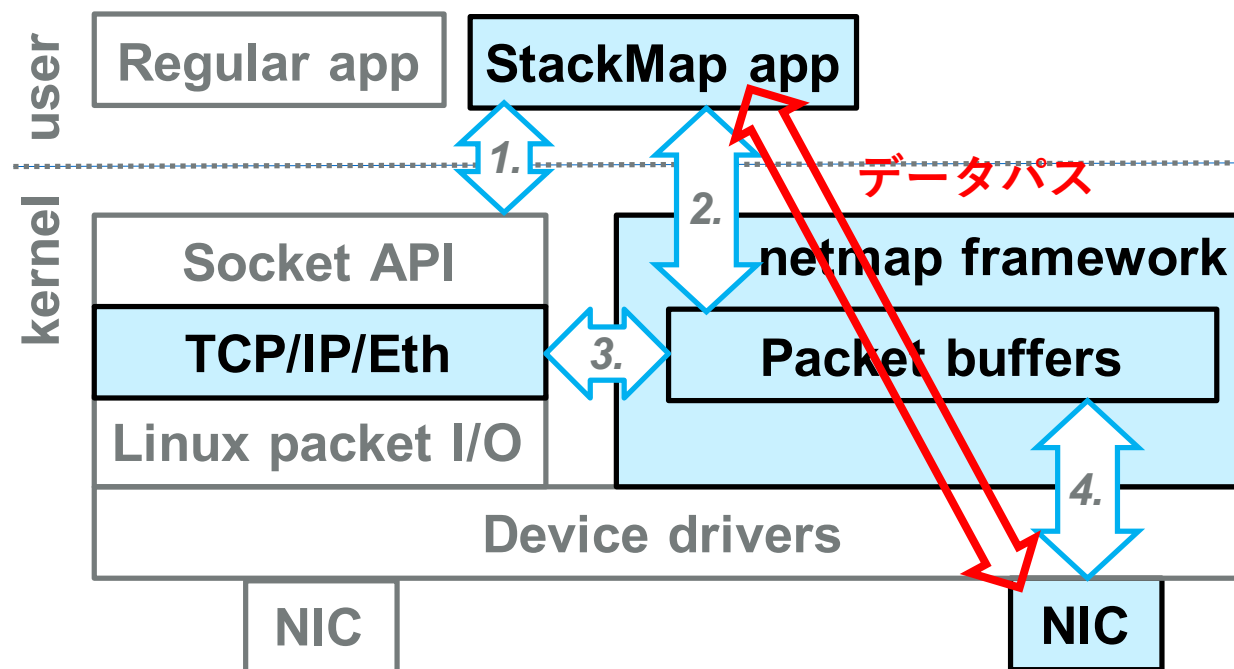
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)

- **StackMap (USENIX ATC 2016)**

- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

netmap + カーネル TCP/IP スタック

- アプリの API とデータパスは (ほぼ) netmap
- ヘッダの処理にカーネル TCP/IP スタックを利用



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)

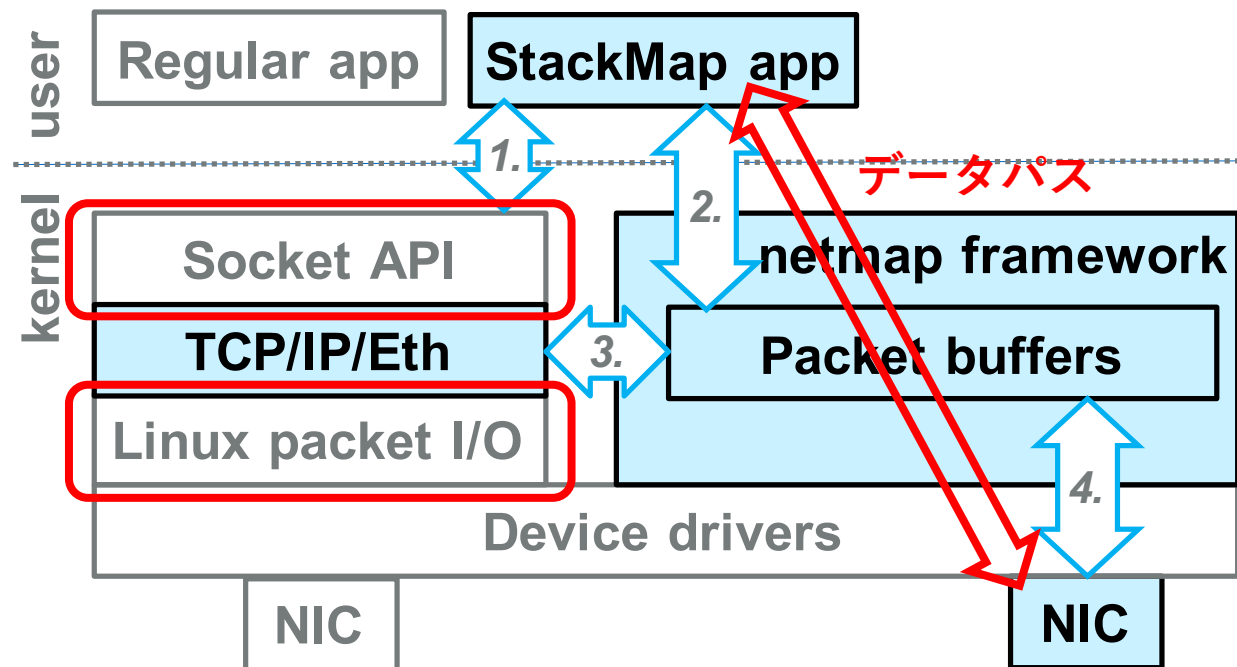
netmap + カーネル TCP/IP スタック

- アプリの API とデータパスは (ほぼ) netmap
- ヘッダの処理にカーネル TCP/IP スタックを利用

パケットは **Socket API と Linux の通常の I/O サブシステムをバイパス**

- **StackMap (USENIX ATC 2016)**

- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

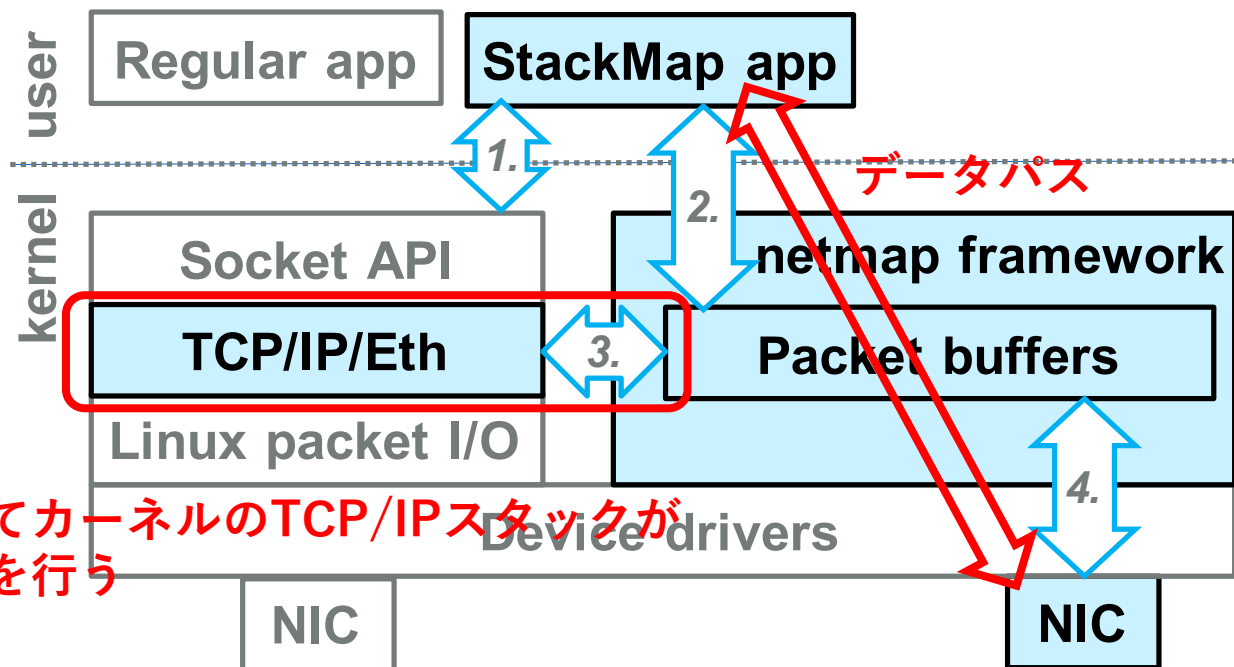
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)

netmap + カーネル TCP/IP スタック
- アプリの API とデータパスは (ほぼ) netmap
- ヘッダの処理にカーネル TCP/IP スタックを利用

パケットは Socket API と Linux の通常の I/O サブシステムをバイパス

- **StackMap (USENIX ATC 2016)**

- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



送受信に際してカーネルのTCP/IPスタックが
ヘッダの処理を行う

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

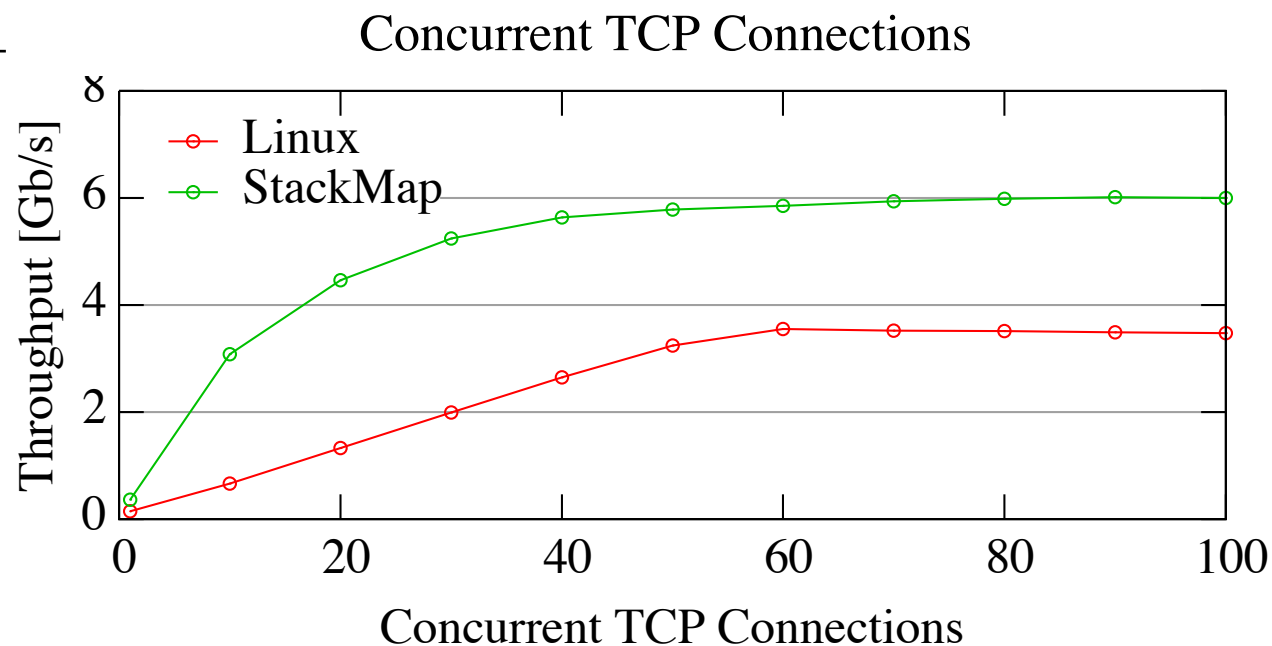
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)

- **StackMap (USENIX ATC 2016)**

- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

netmap + カーネル TCP/IP スタック
- アプリの API とデータパスは (ほぼ) netmap
- ヘッダの処理にカーネル TCP/IP スタックを利用

1 コアで 1 KB データを送信



研究紹介

TCP/IP スタック設計

パケット I/O フレームワークを適用する

ディスクとの親和性を高める

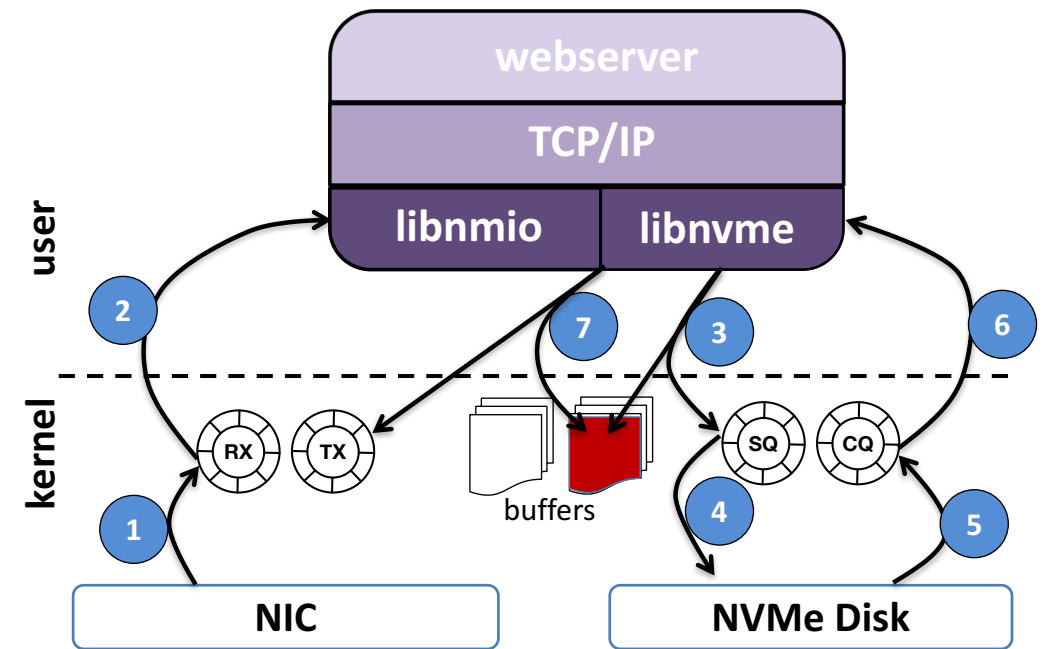
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化



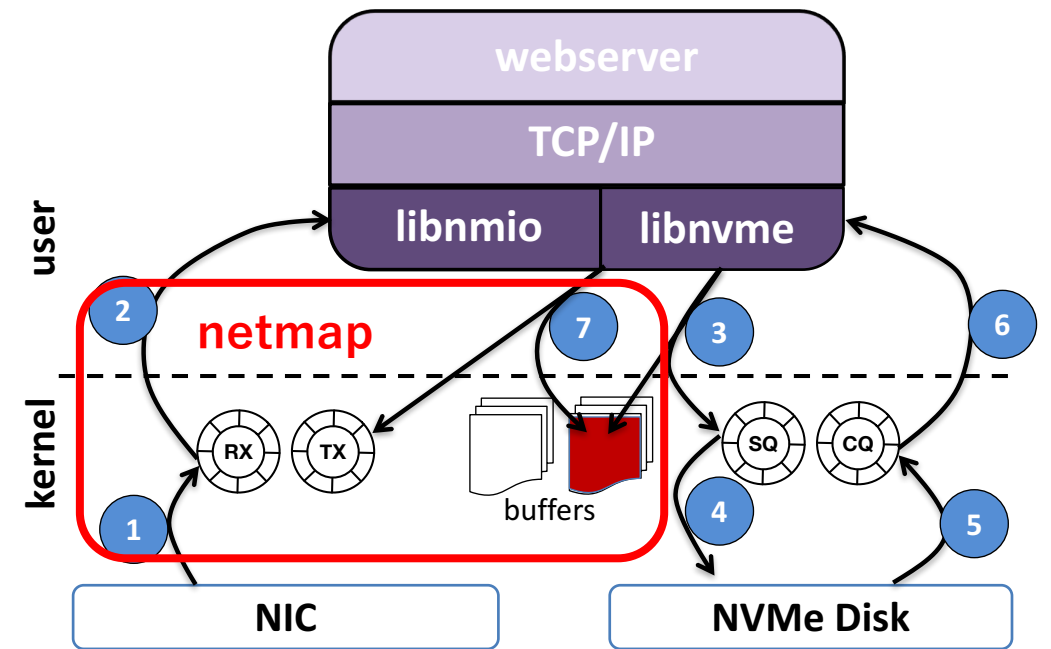
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化



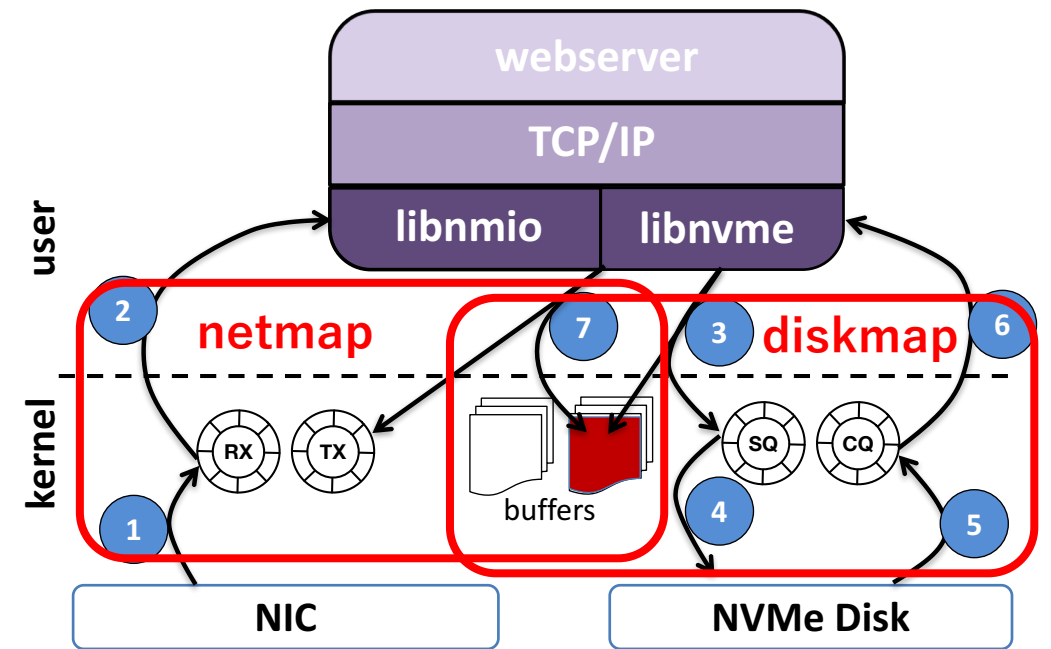
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化



TCP/IP スタック設計の再考

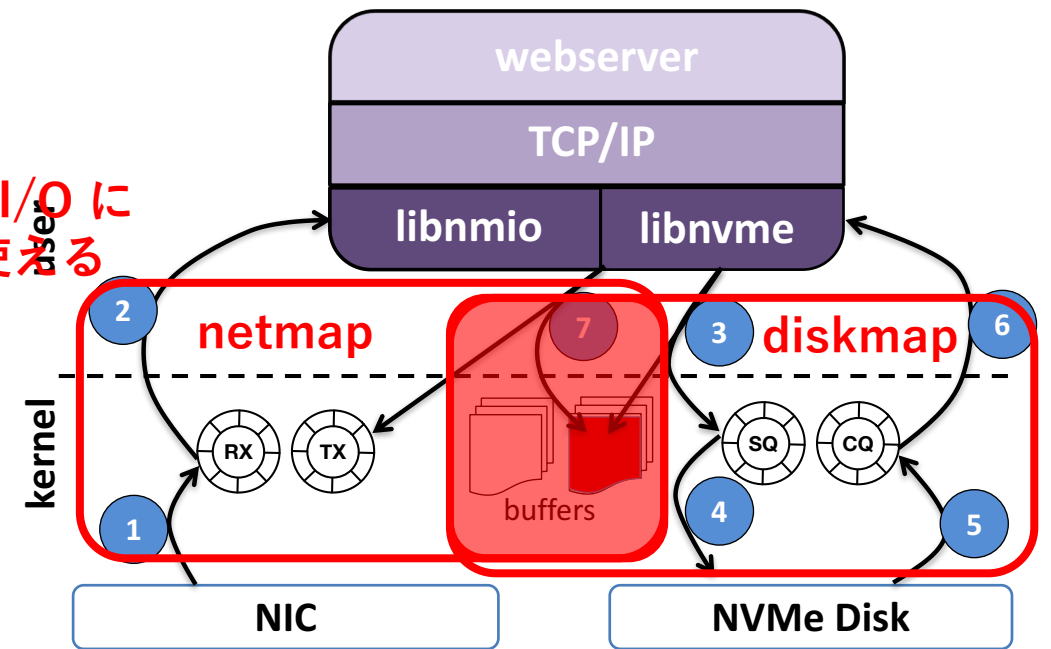
• パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化

NIC I/O と Disk I/O に
同じバッファが使える



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

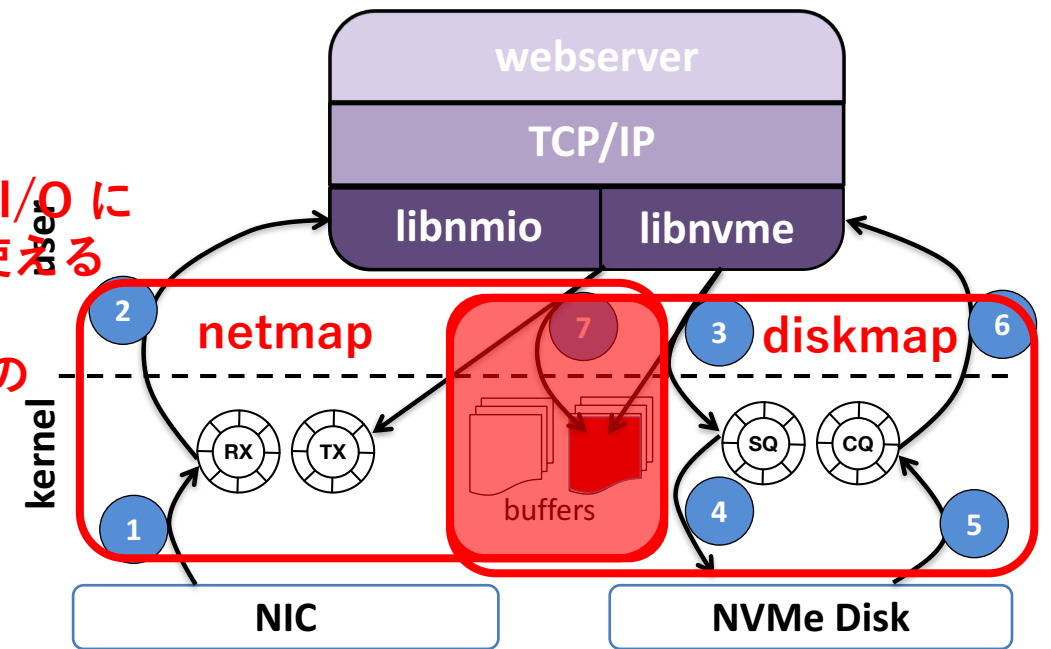
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化

NIC I/O と Disk I/O に
同じバッファが使える

Disk から NIC への
データの移動に
コピーが不要



TCP/IP スタック設計の再考

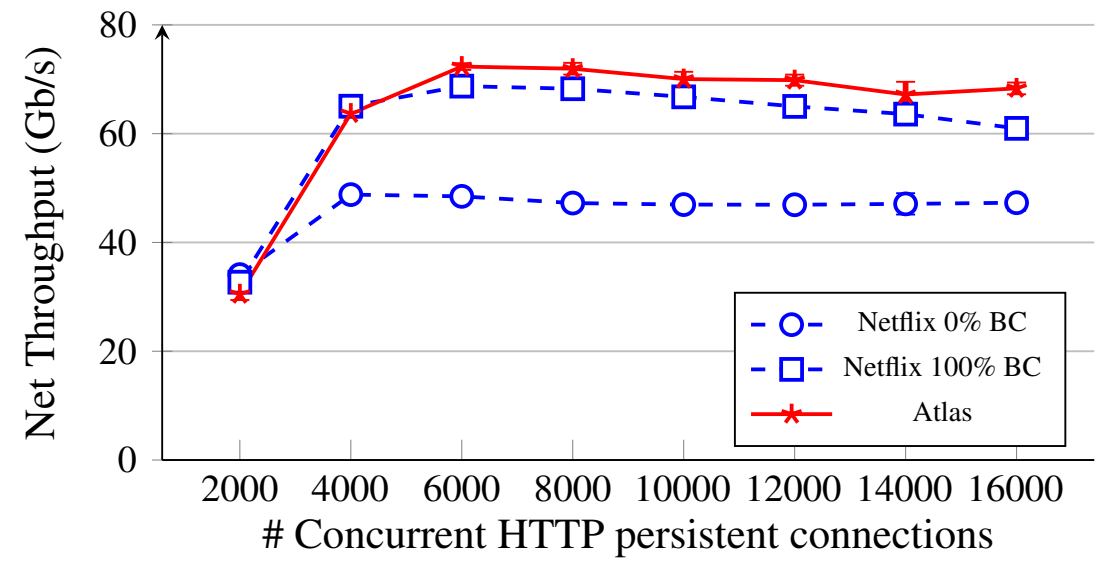
- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- **Atlas (SIGCOMM 2017)**
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

- ディスクアクセス時にカーネルをバイパスする *diskmap* 機構を追加
- *diskmap* を netmap と統合して、ディスクからの読み取ったデータの配送を効率化

コンテンツ配送速度 (2つの 40Gbps NIC 合計)



(a) Network throughput (Error bars indicate the 95% CI)

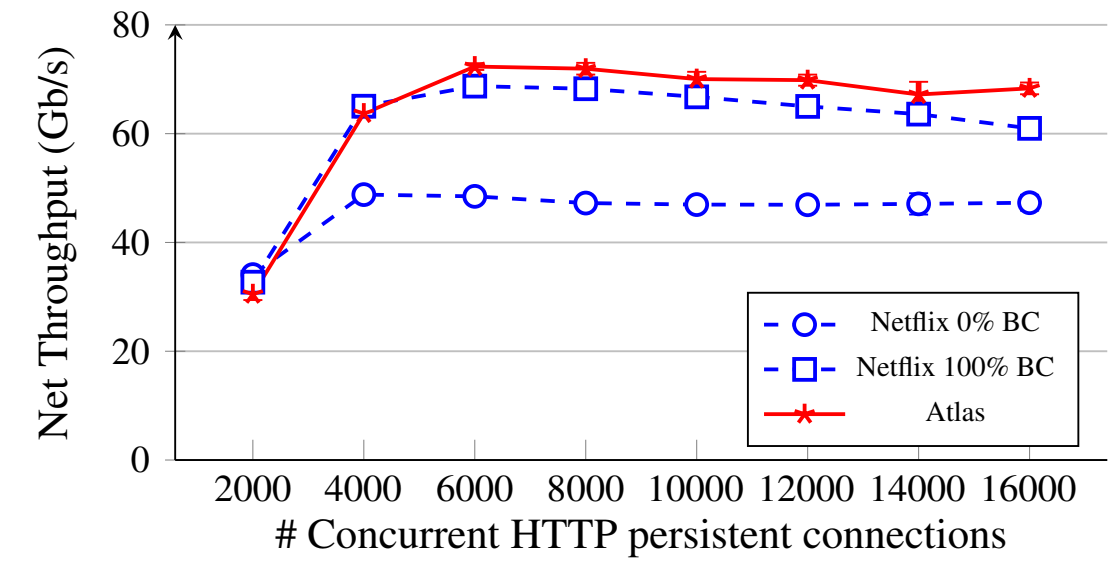
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - **Atlas (SIGCOMM 2017)**
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

ビデオストリーミング用サーバー (Sandstorm 拡張)

Netflix の最適化を含む FreeBSD との比較
BC: 配送コンテンツのバッファキャッシュヒット率

研究の取ったアプローチの配信を効率化
コンテンツ配送速度 (2つの 40Gbps NIC 合計)

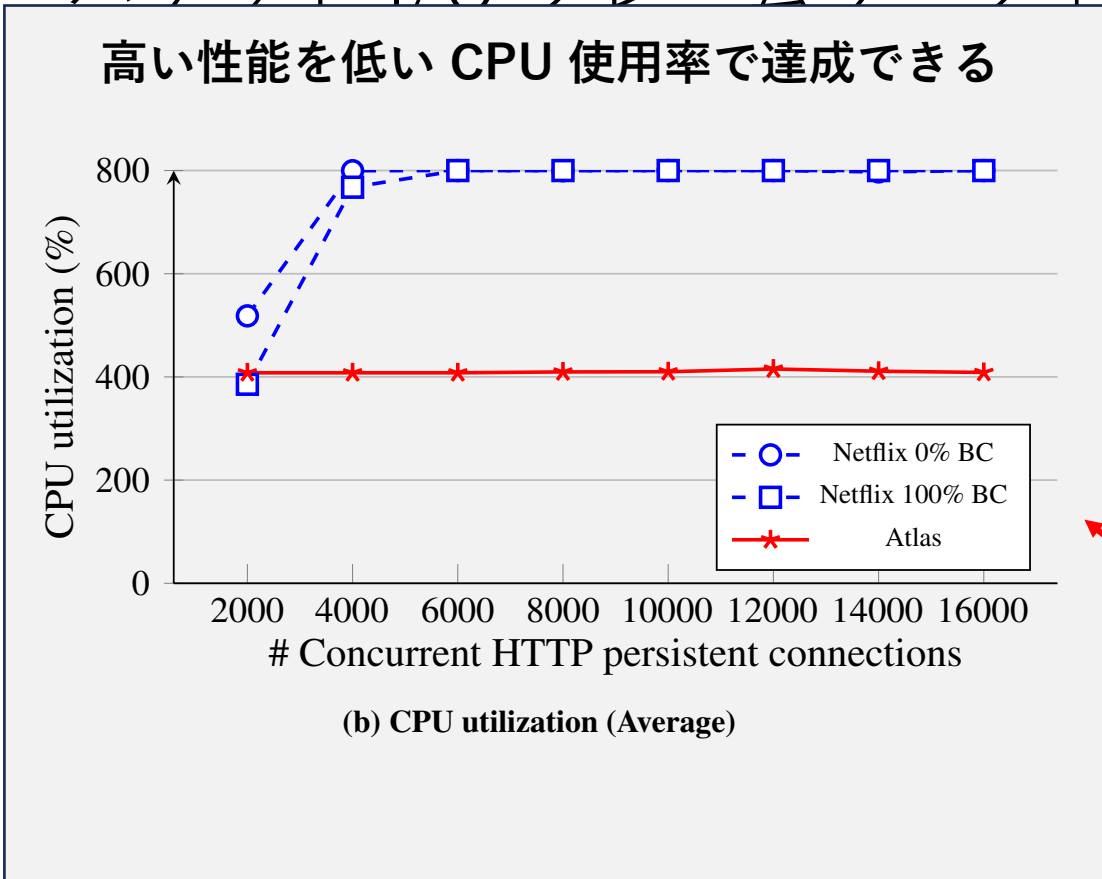


(a) Network throughput (Error bars indicate the 95% CI)

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

高い性能を低い CPU 使用率で達成できる



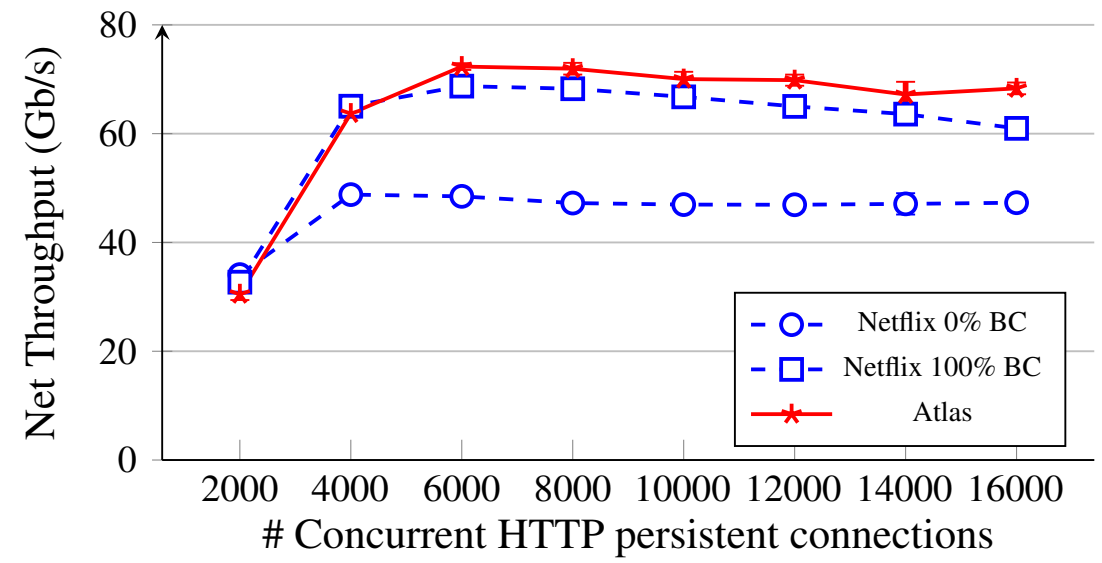
(b) CPU utilization (Average)

ビデオストリーミング用サーバー (Sandstorm 拡張)

Netflix の最適化を含む FreeBSD との比較
BC: 配送コンテンツのバッファキャッシュヒット率

研究の取ったアプローチの配送を効率化

コンテンツ配送速度 (2つの 40Gbps NIC 合計)



(a) Network throughput (Error bars indicate the 95% CI)

- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

研究紹介

TCP/IP スタック設計

パケット I/O フレームワークを適用する

各リクエストの処理時間の分散に配慮する

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

着眼点

既存のシステムは
アプリへ届くリクエストにかかる
処理時間のばらつきへの考慮が不十分



Head-of-Line Blocking 問題を引き起こす

拡張

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

アプリスレッド
on CPU Core0

アプリスレッド
on CPU Core1

拡張

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

アプリスレッド
on CPU Core0

アプリスレッド
on CPU Core1

CPU Core0 用

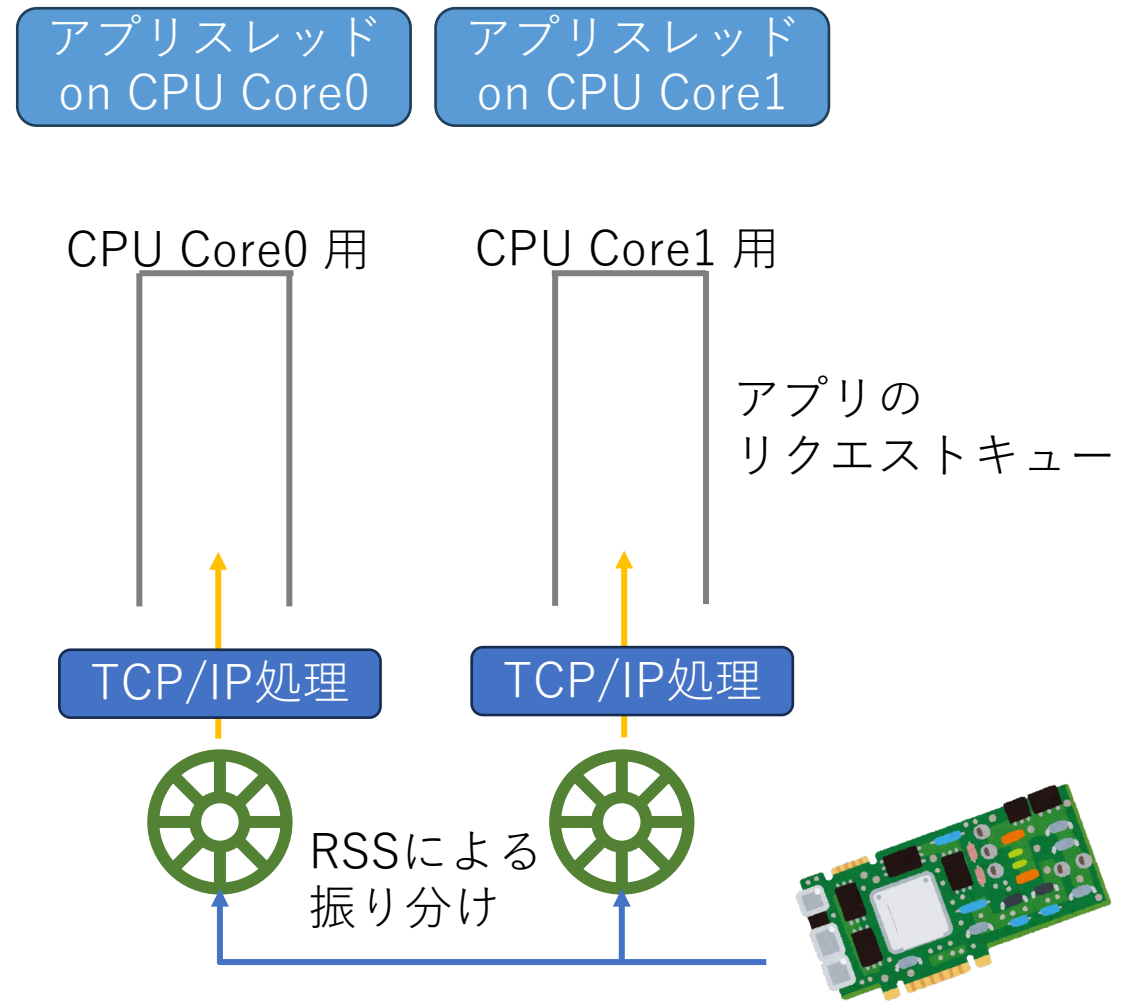
CPU Core1 用

アプリの
リクエストキュー



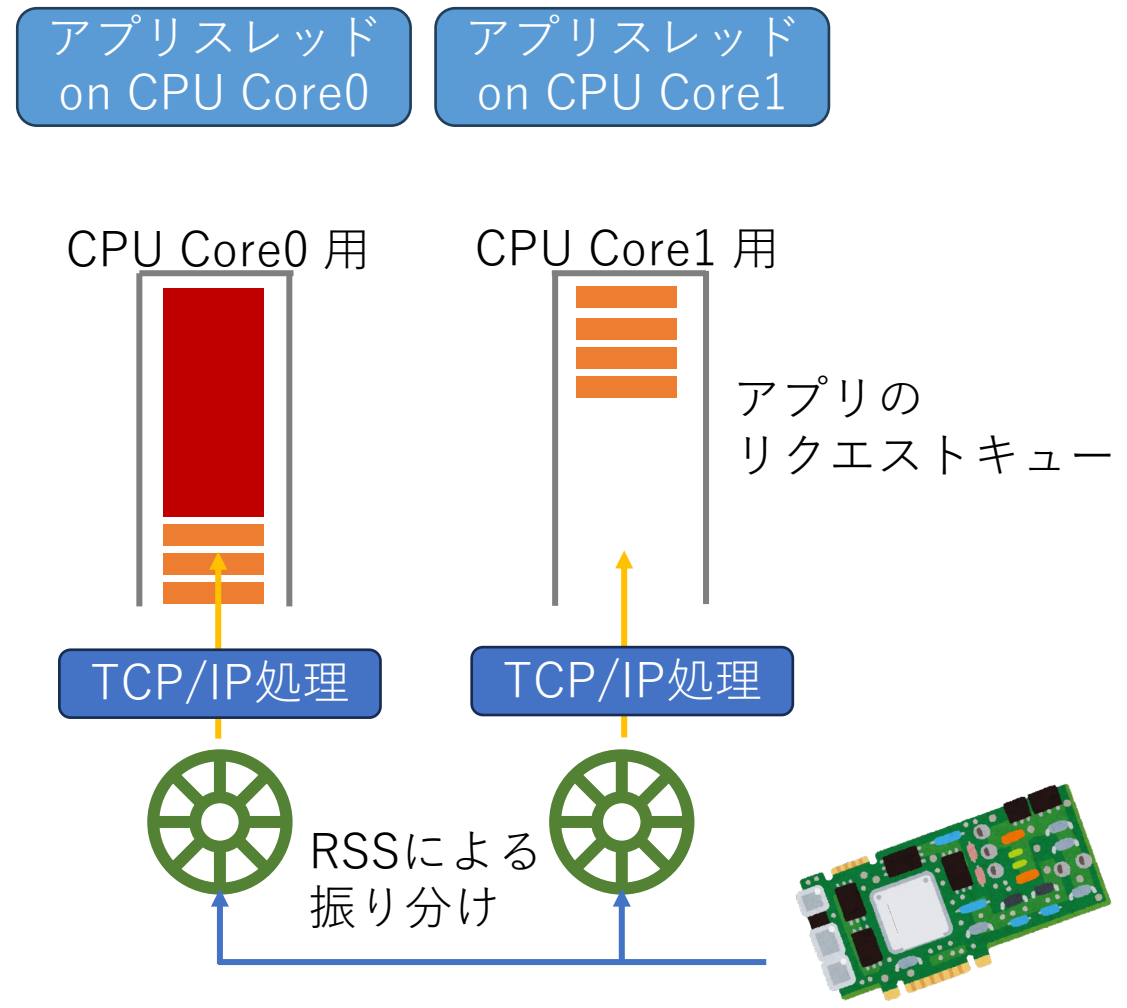
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

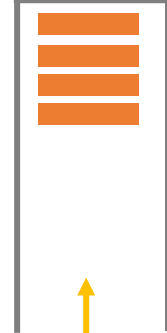
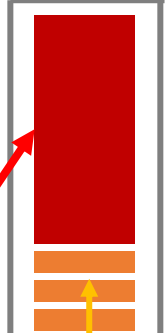
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

アプリスレッド
on CPU Core0

アプリスレッド
on CPU Core1

CPU Core0 用

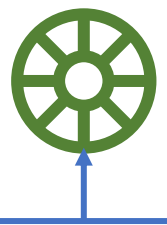
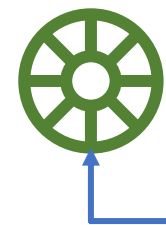
CPU Core1 用



アプリの
リクエストキュー

TCP/IP処理

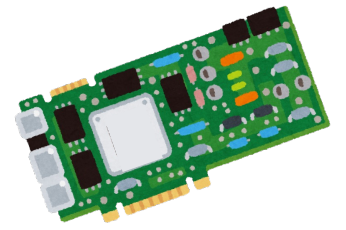
TCP/IP処理



RSSによる
振り分け

拡張

処理完了まで
時間がかかる
リクエスト



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

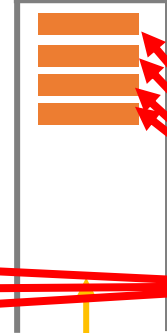
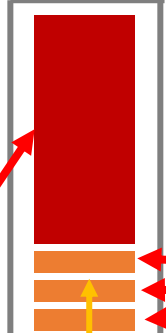
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

アプリスレッド
on CPU Core0

アプリスレッド
on CPU Core1

CPU Core0 用

CPU Core1 用



TCP/IP処理

TCP/IP処理

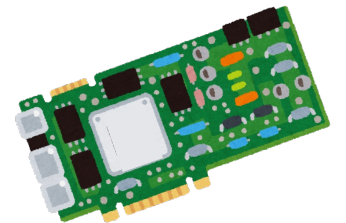
アプリのリクエストキュー

短時間で処理が完了できる
リクエスト

拡張

処理完了まで
時間がかかる
リクエスト

RSSによる
振り分け



TCP/IP スタック設計の再考

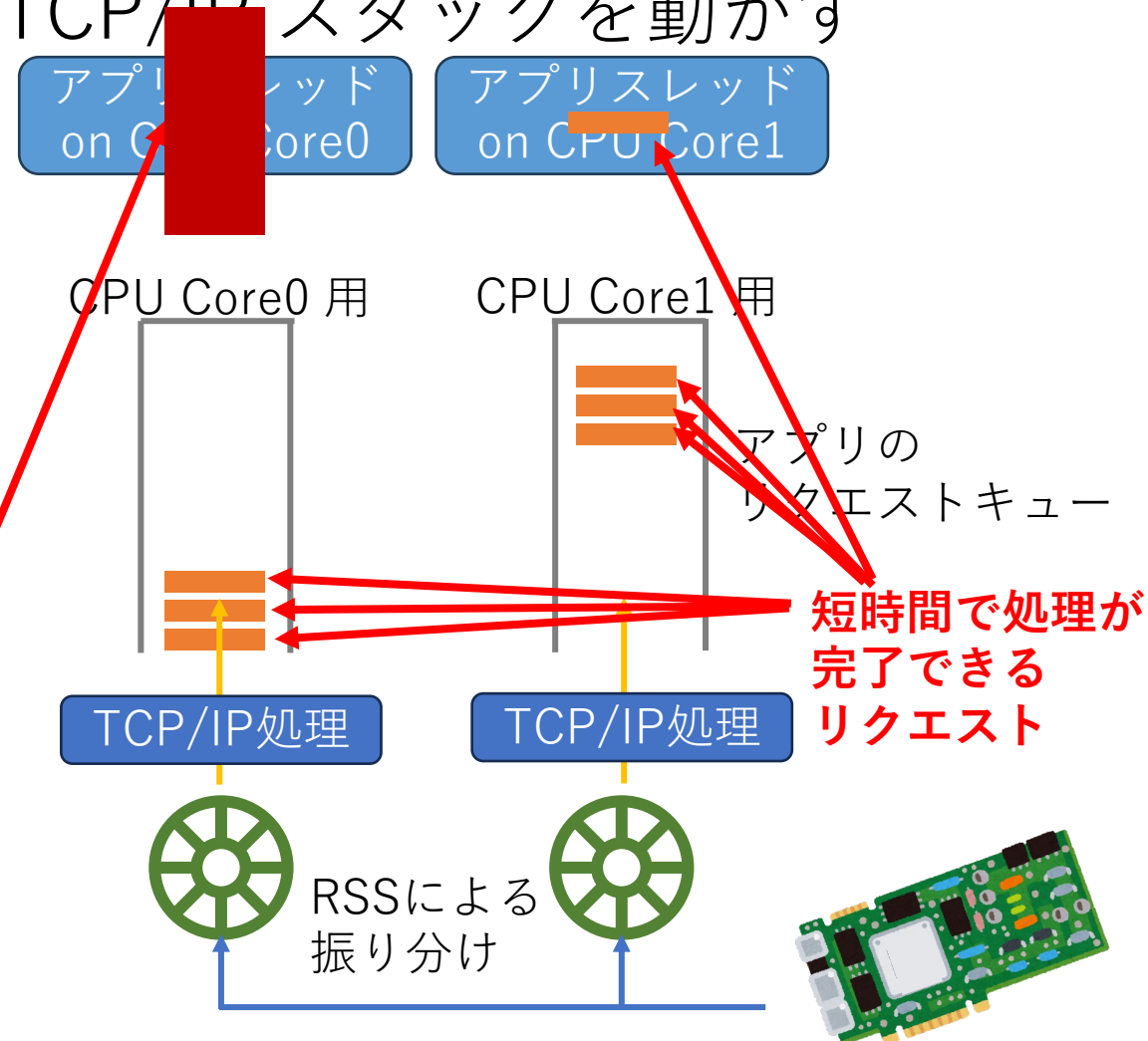
順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト



TCP/IP スタック設計の再考

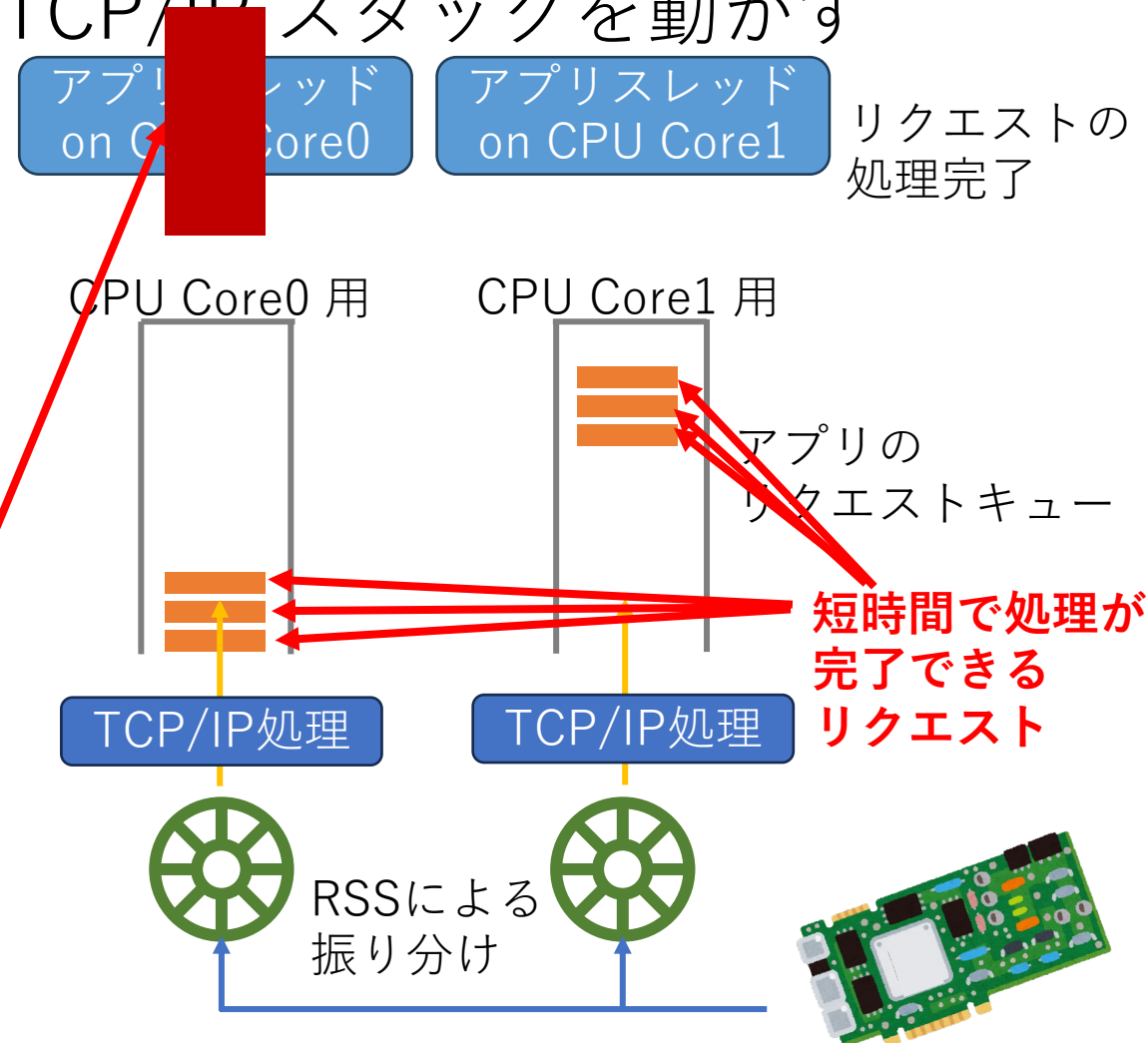
順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト



TCP/IP スタック設計の再考

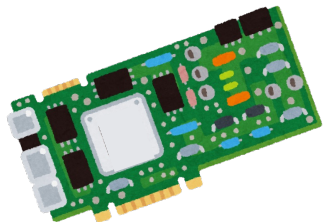
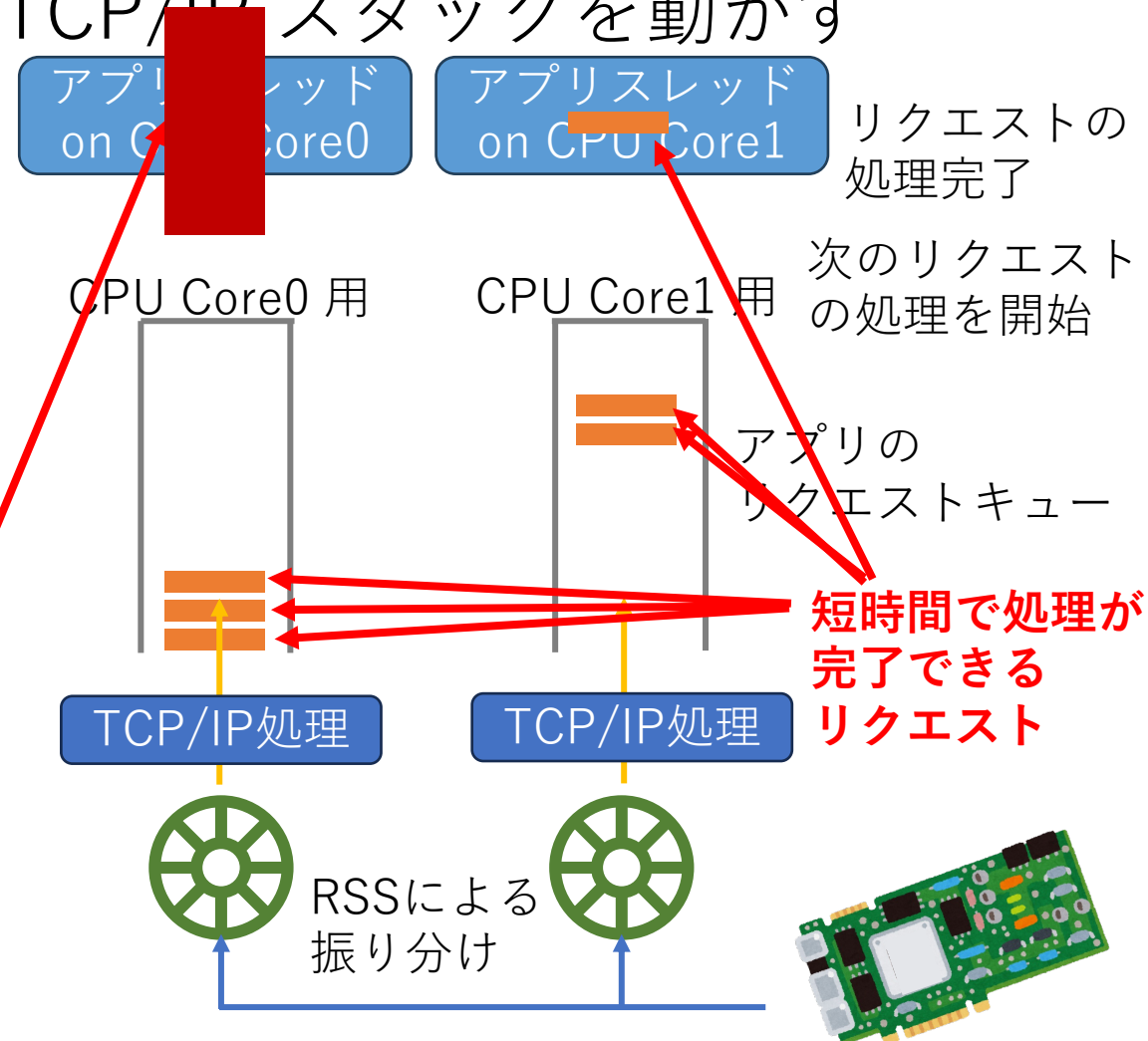
順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト



TCP/IP スタック設計の再考

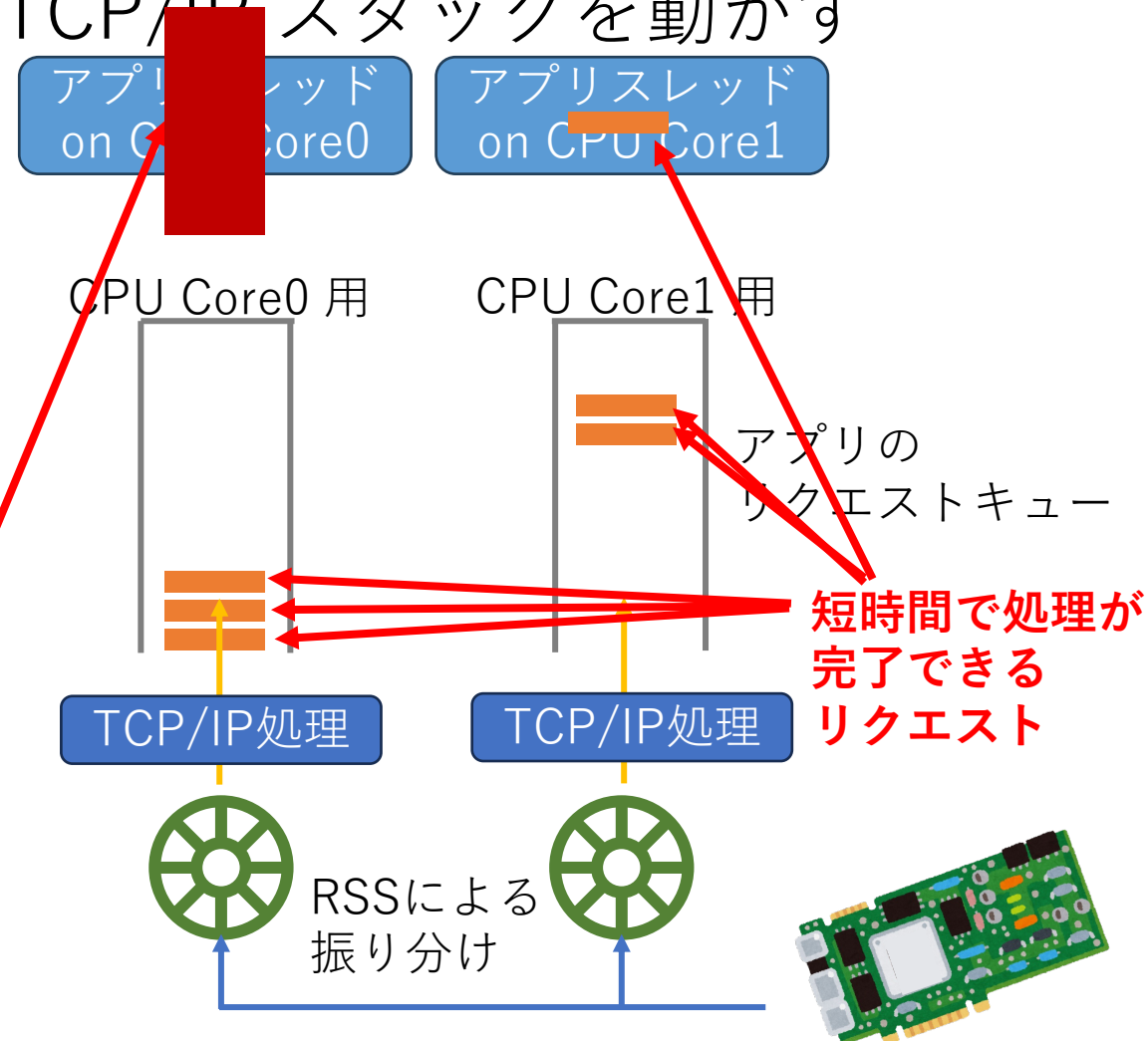
順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト

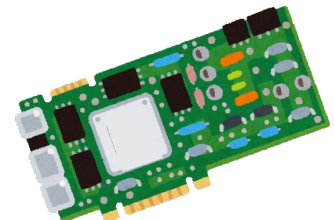
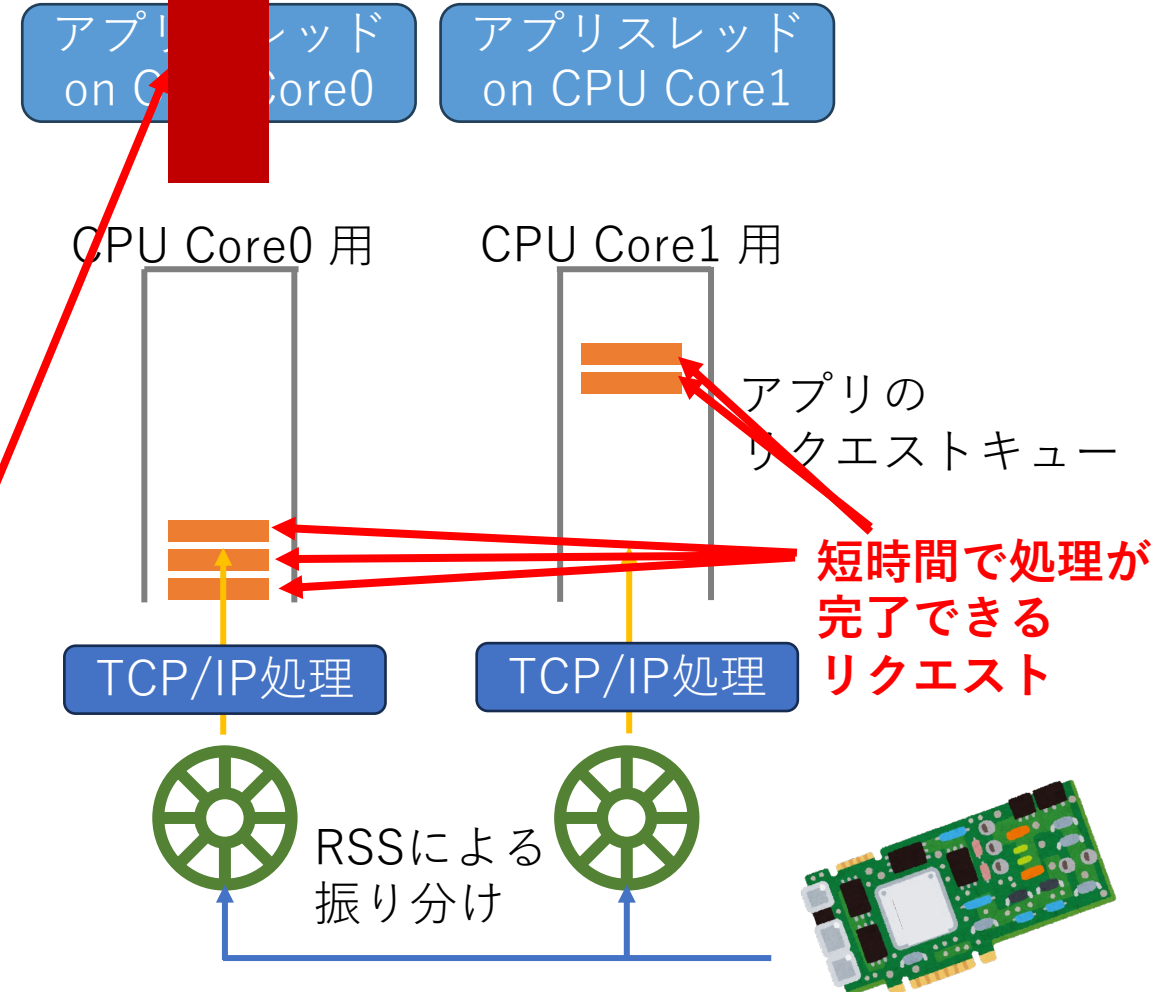


TCP/IP スタック設計の再考

順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

拡張
処理完了まで
時間がかかる
リクエスト



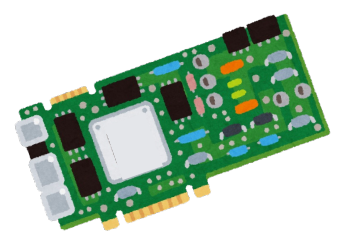
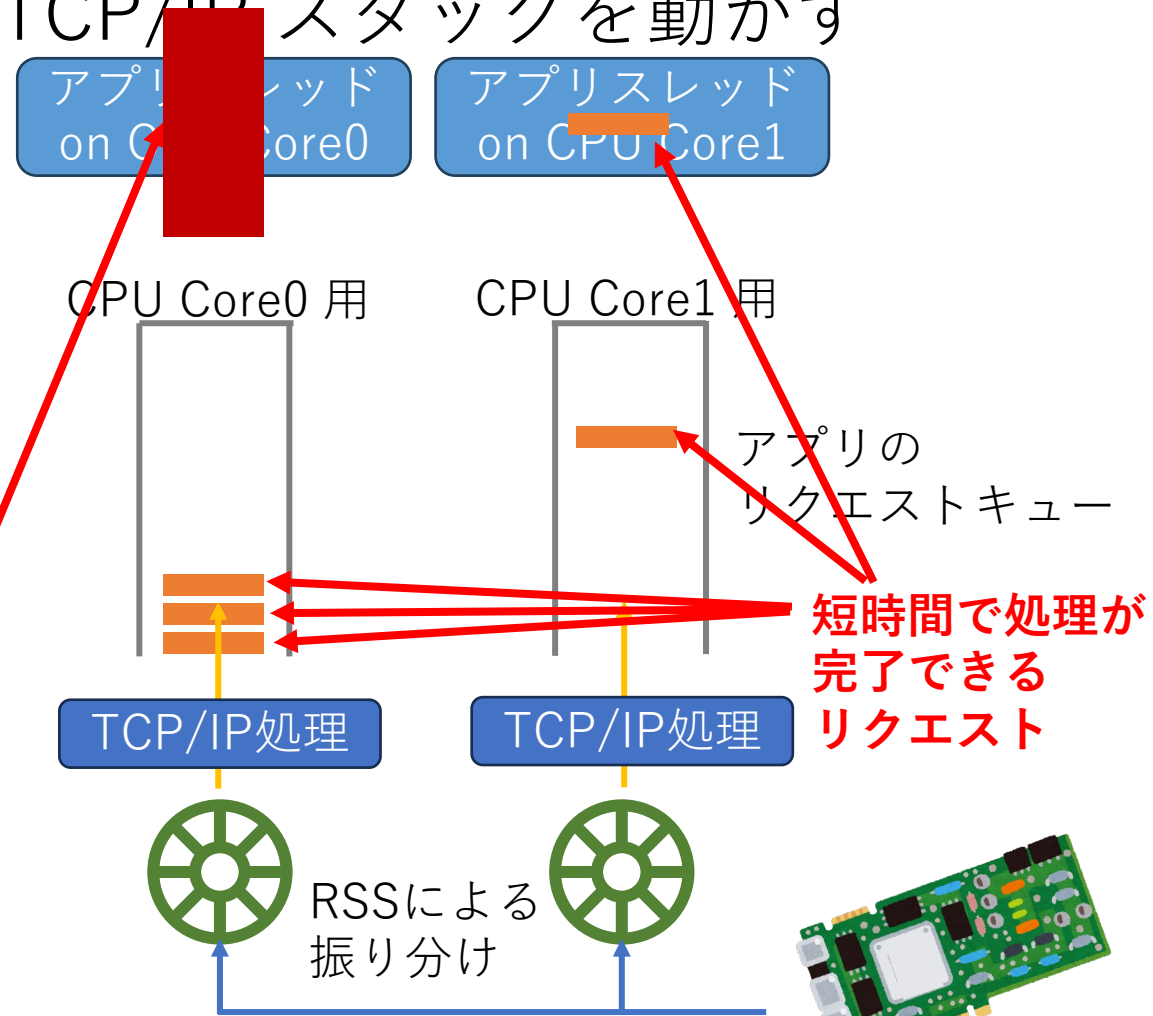
TCP/IP スタック設計の再考

順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト

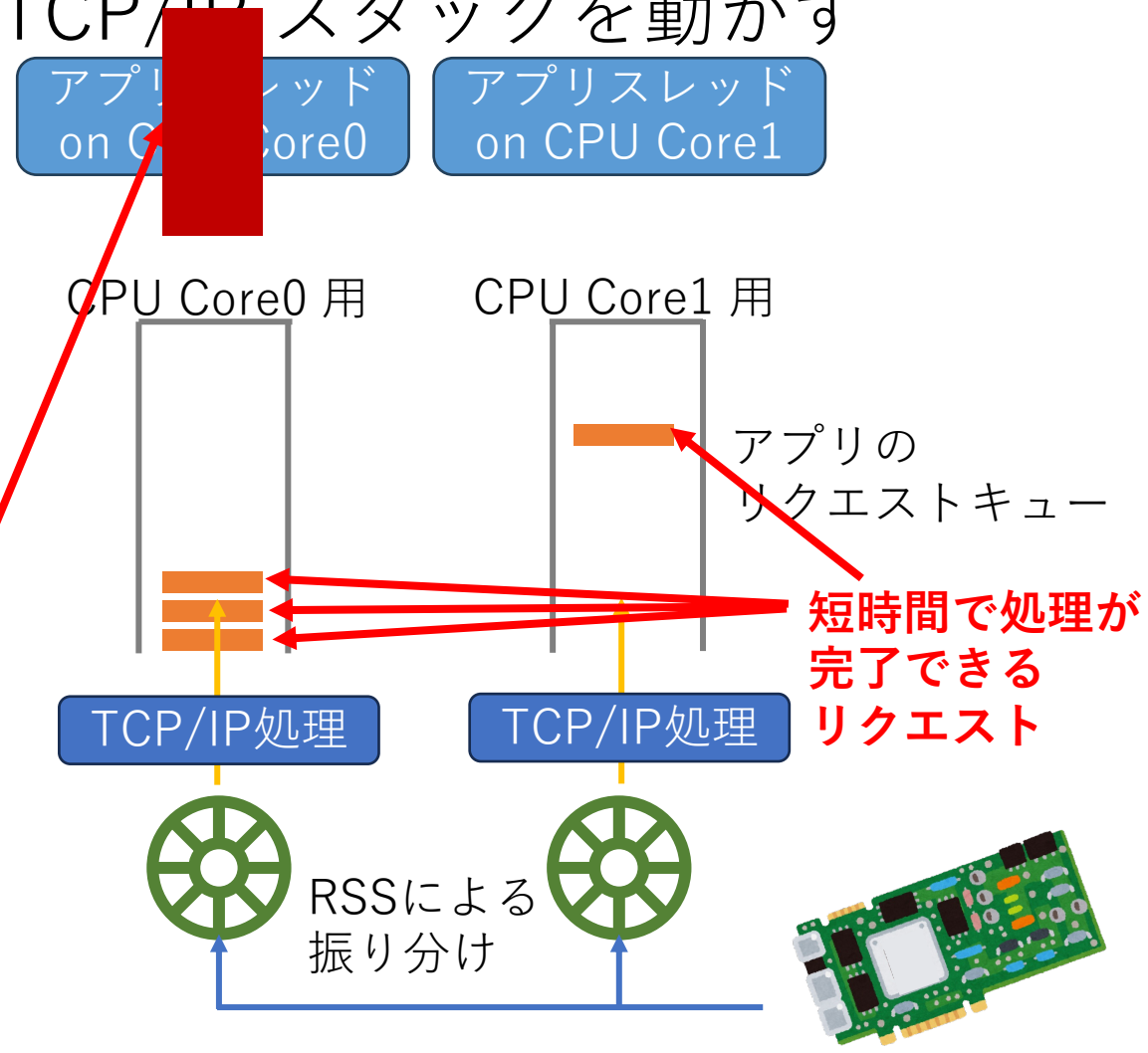


TCP/IP スタック設計の再考

順番に処理していくと

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

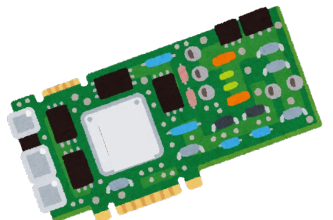
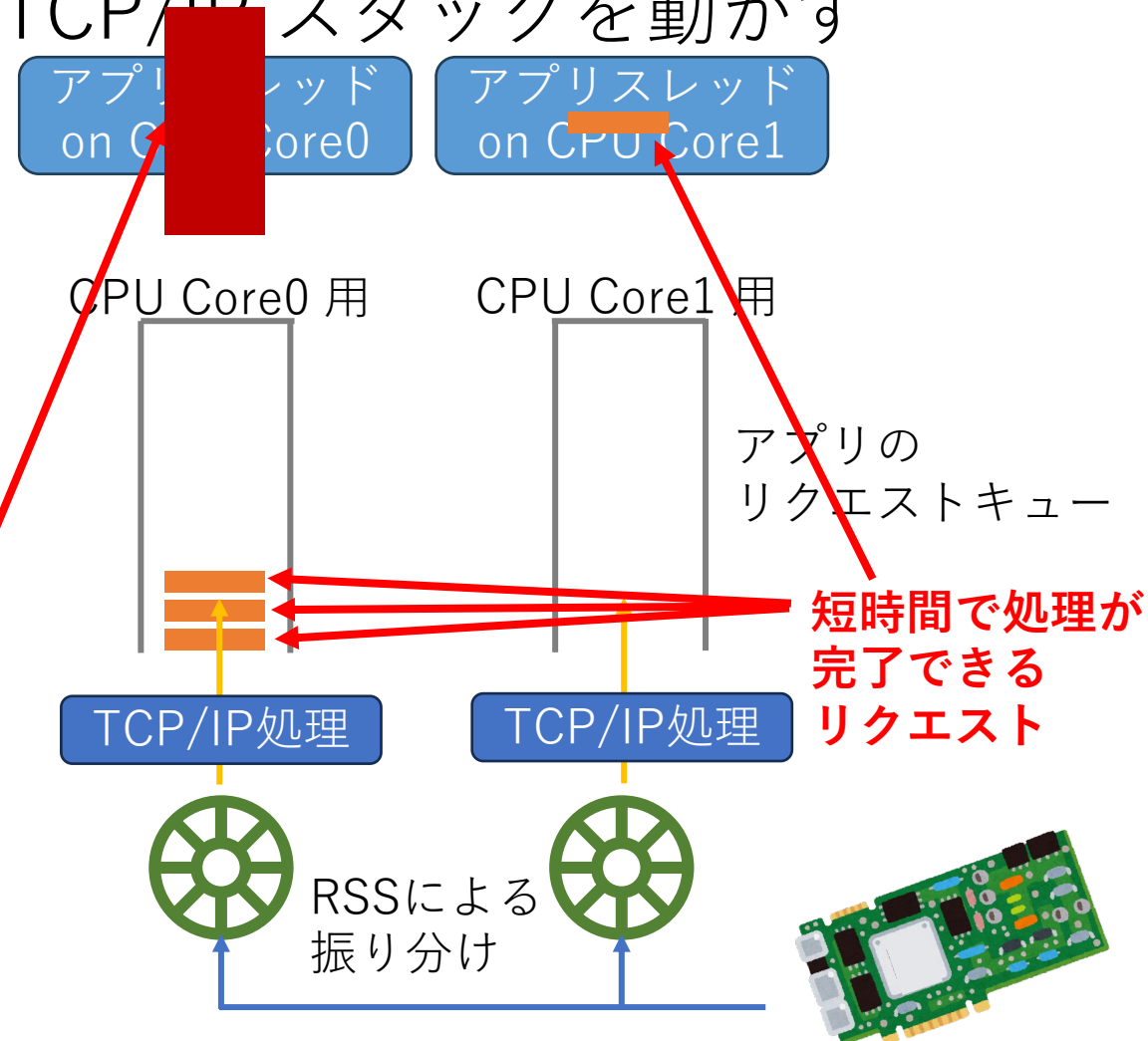
順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

処理完了まで
時間がかかる
リクエスト

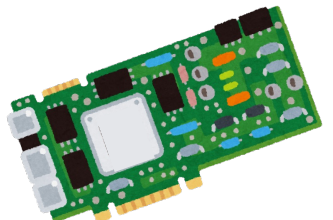
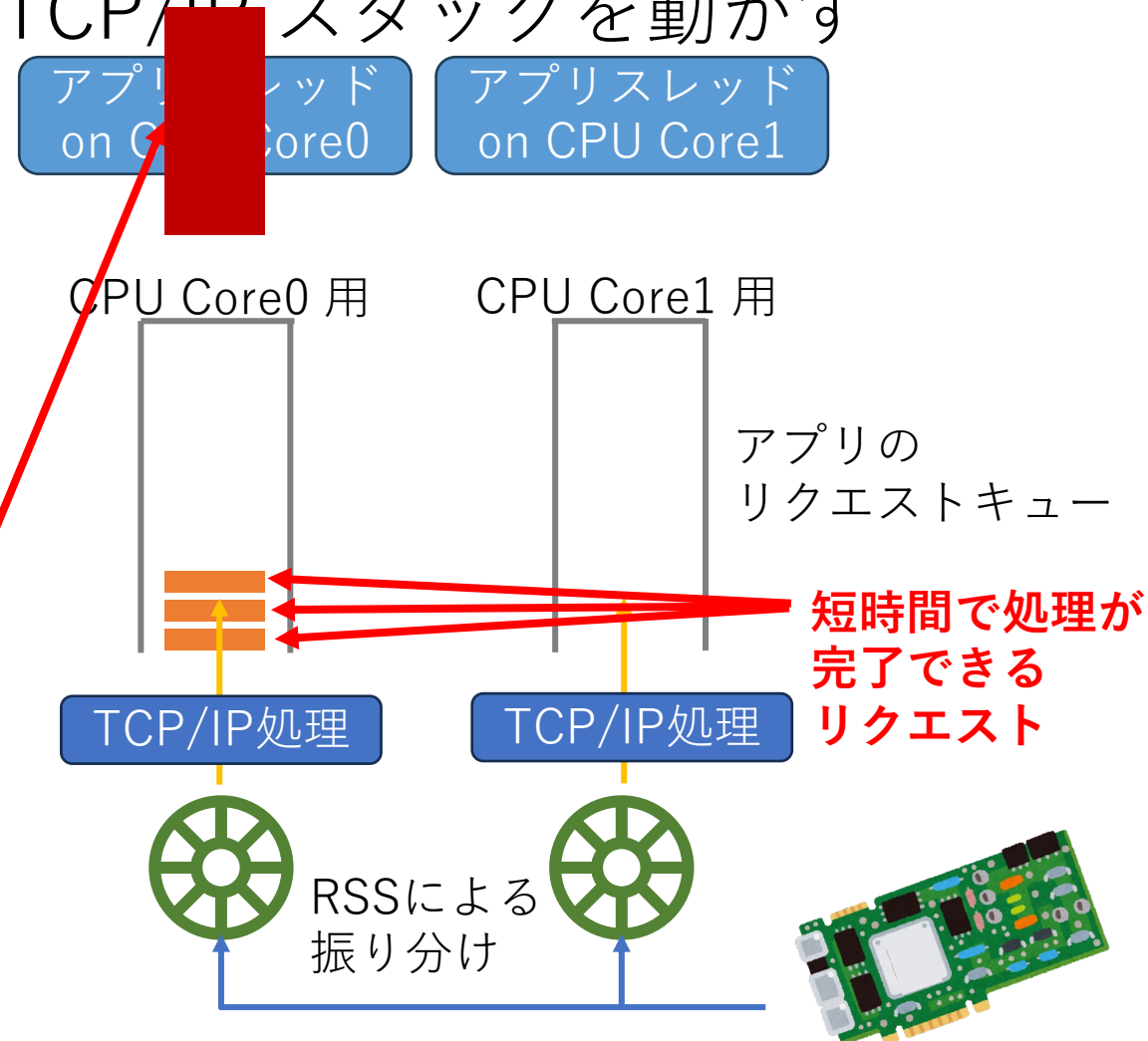


TCP/IP スタック設計の再考

順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

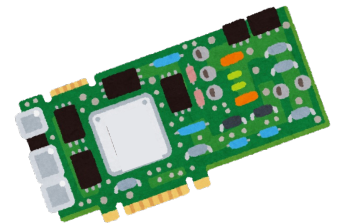
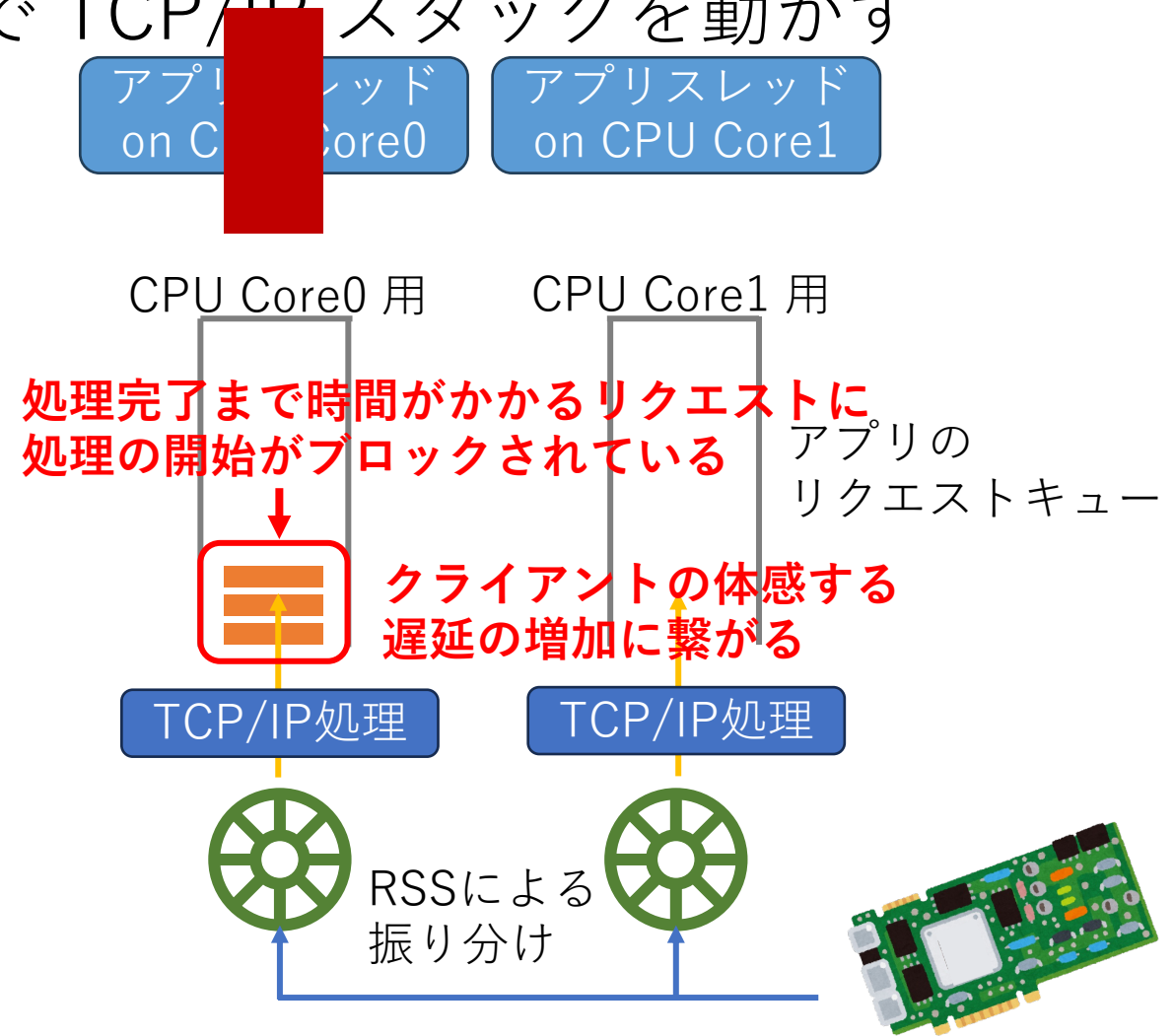
拡張
処理完了まで
時間がかかる
リクエスト



TCP/IP スタック設計の再考

順番に処理していくと

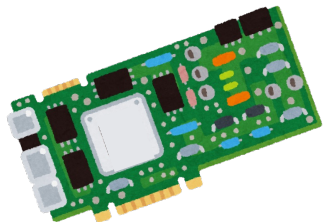
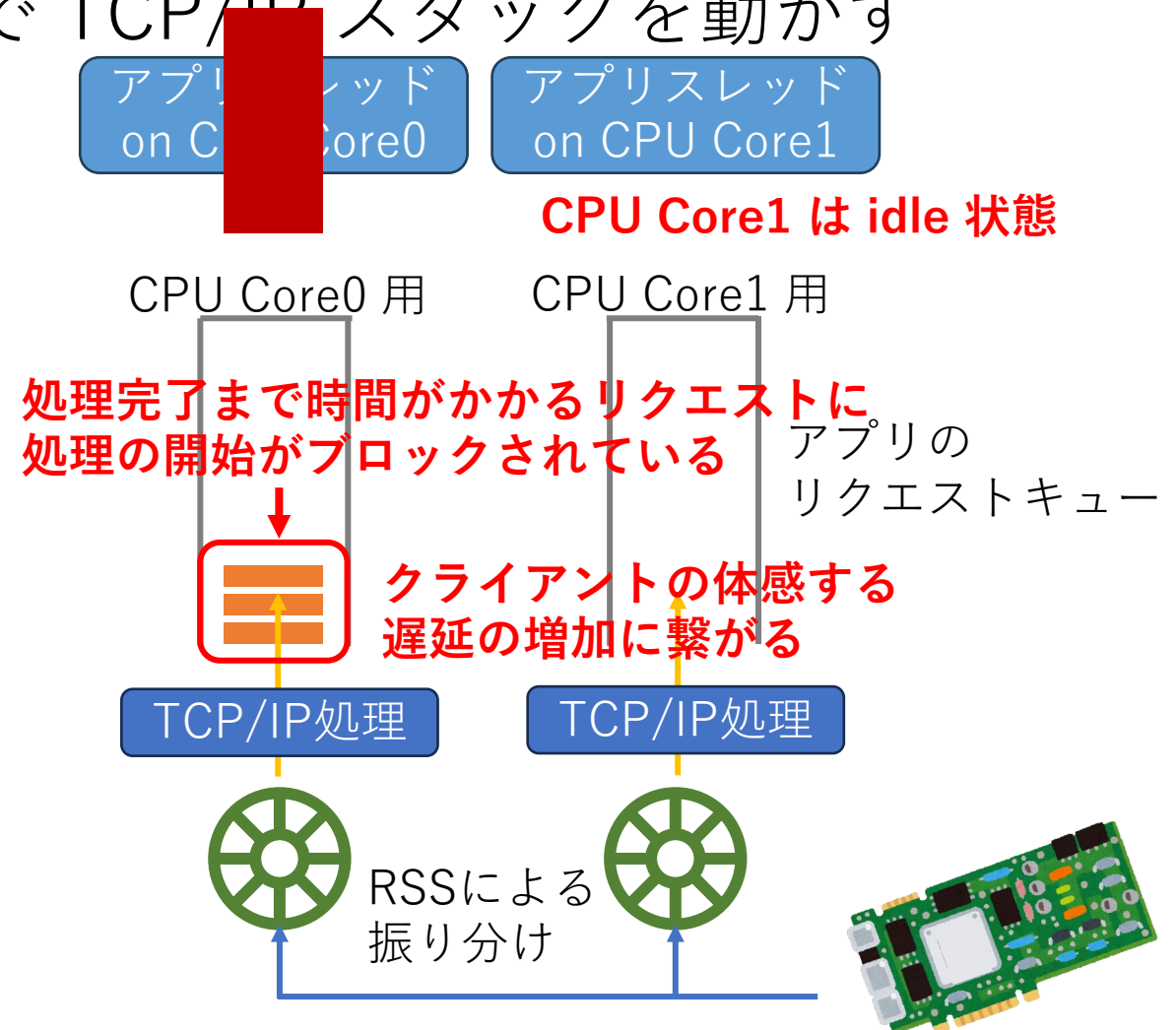
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

順番に処理していくと

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

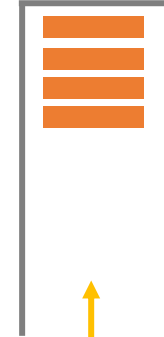
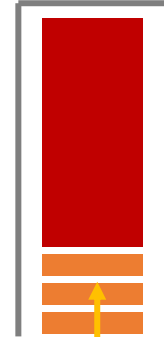
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

アプリスレッド
on CPU Core0

アプリスレッド
on CPU Core1

CPU Core0 用

CPU Core1 用



アプリの
リクエストキュー

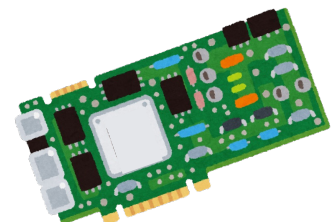
TCP/IP処理

TCP/IP処理



RSSによる
振り分け

提案手法：
Shuffle 層を追加

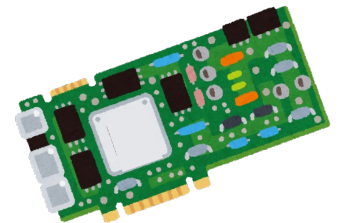
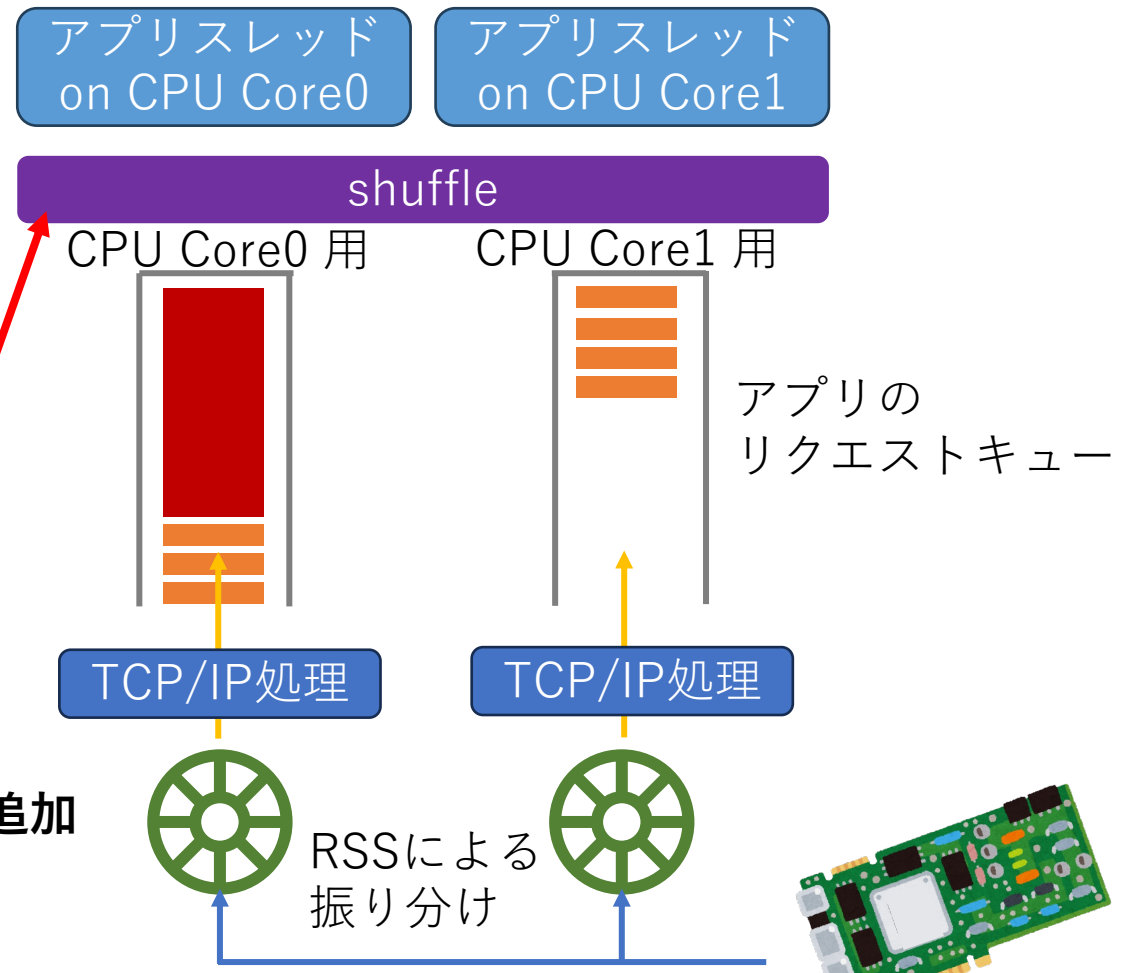


TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

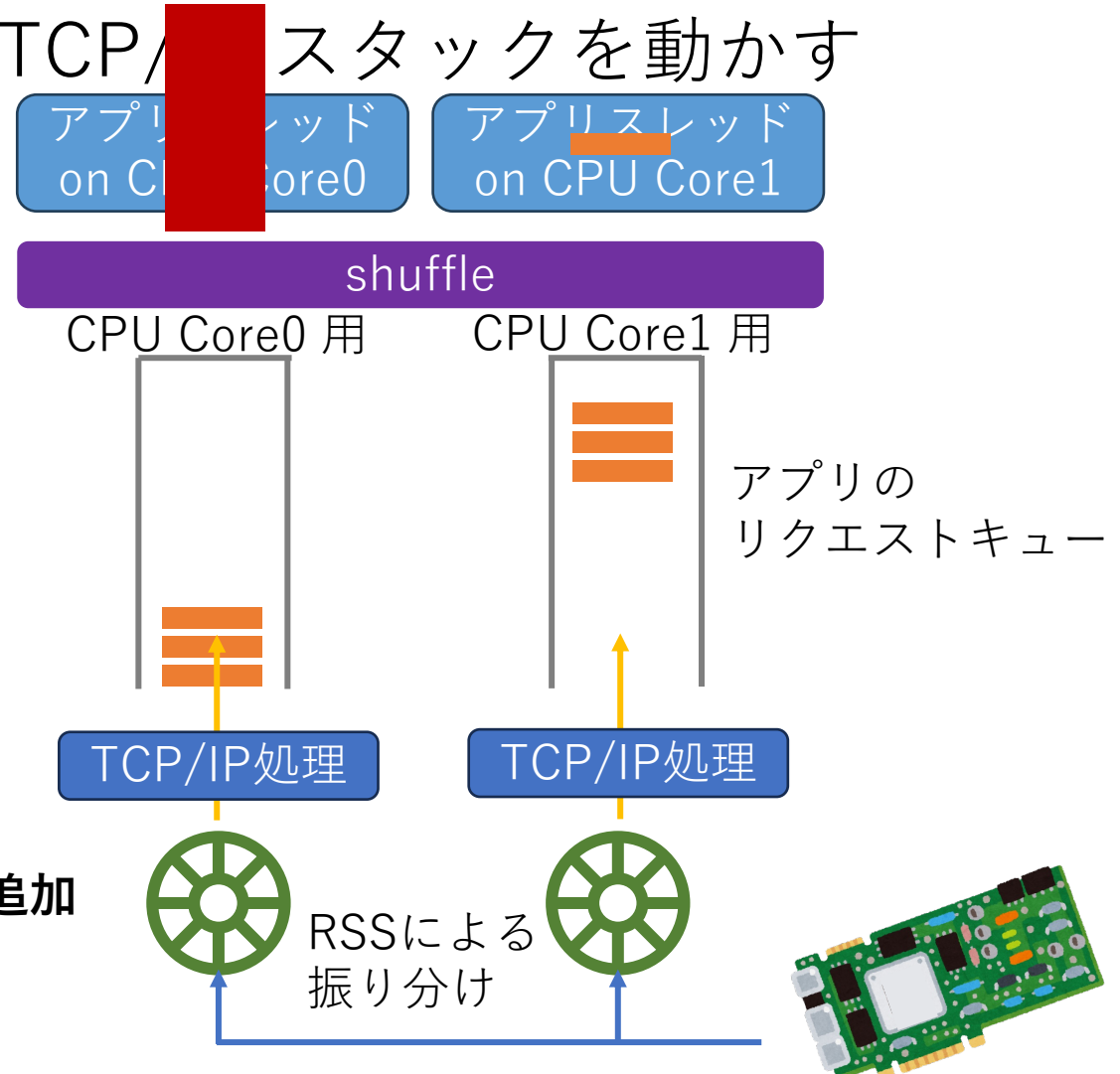
拡張

提案手法：
Shuffle 層を追加



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)** ← 拡張
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



提案手法：
Shuffle 層を追加

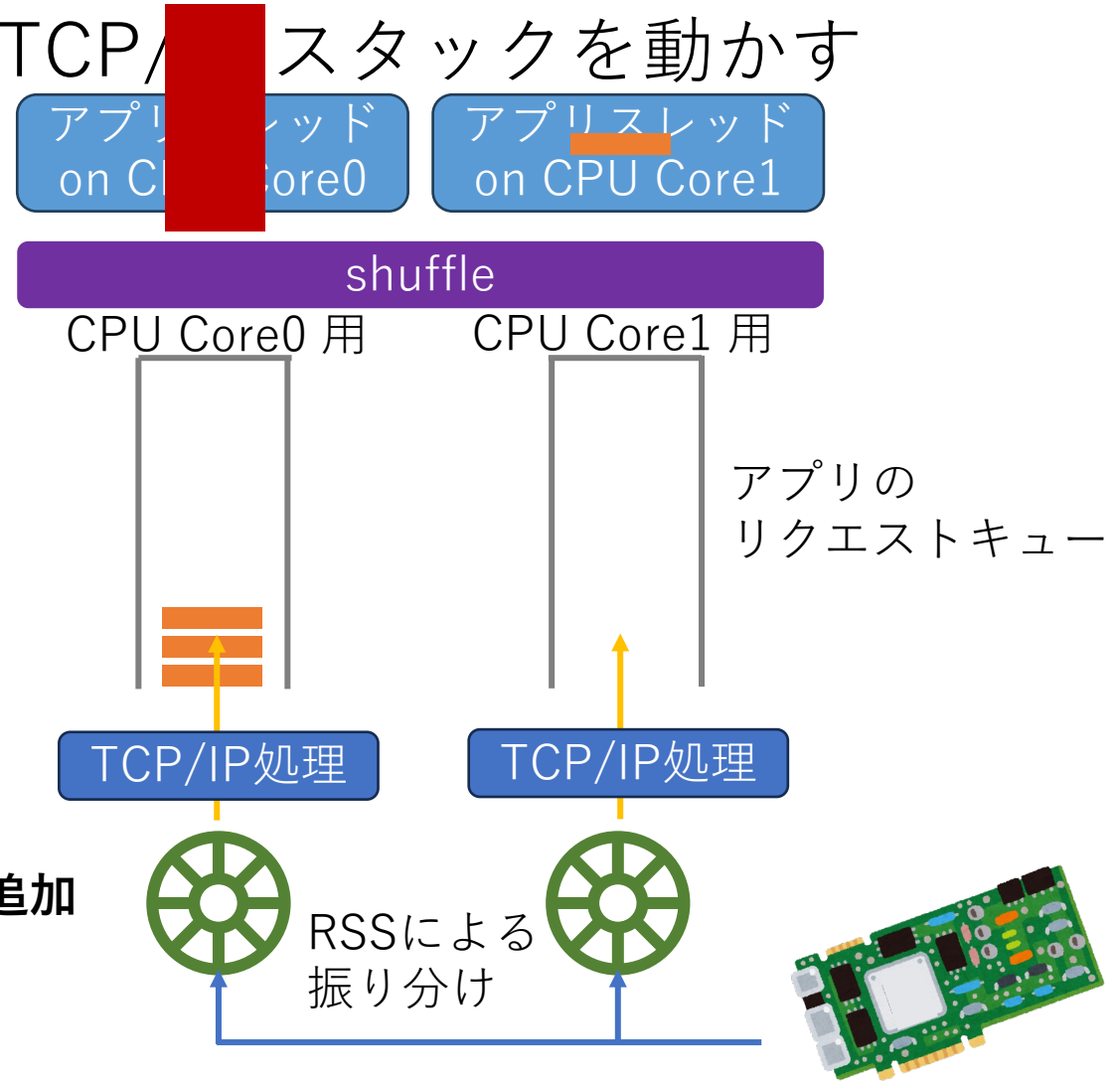
RSSによる
振り分け

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



提案手法：
Shuffle 層を追加



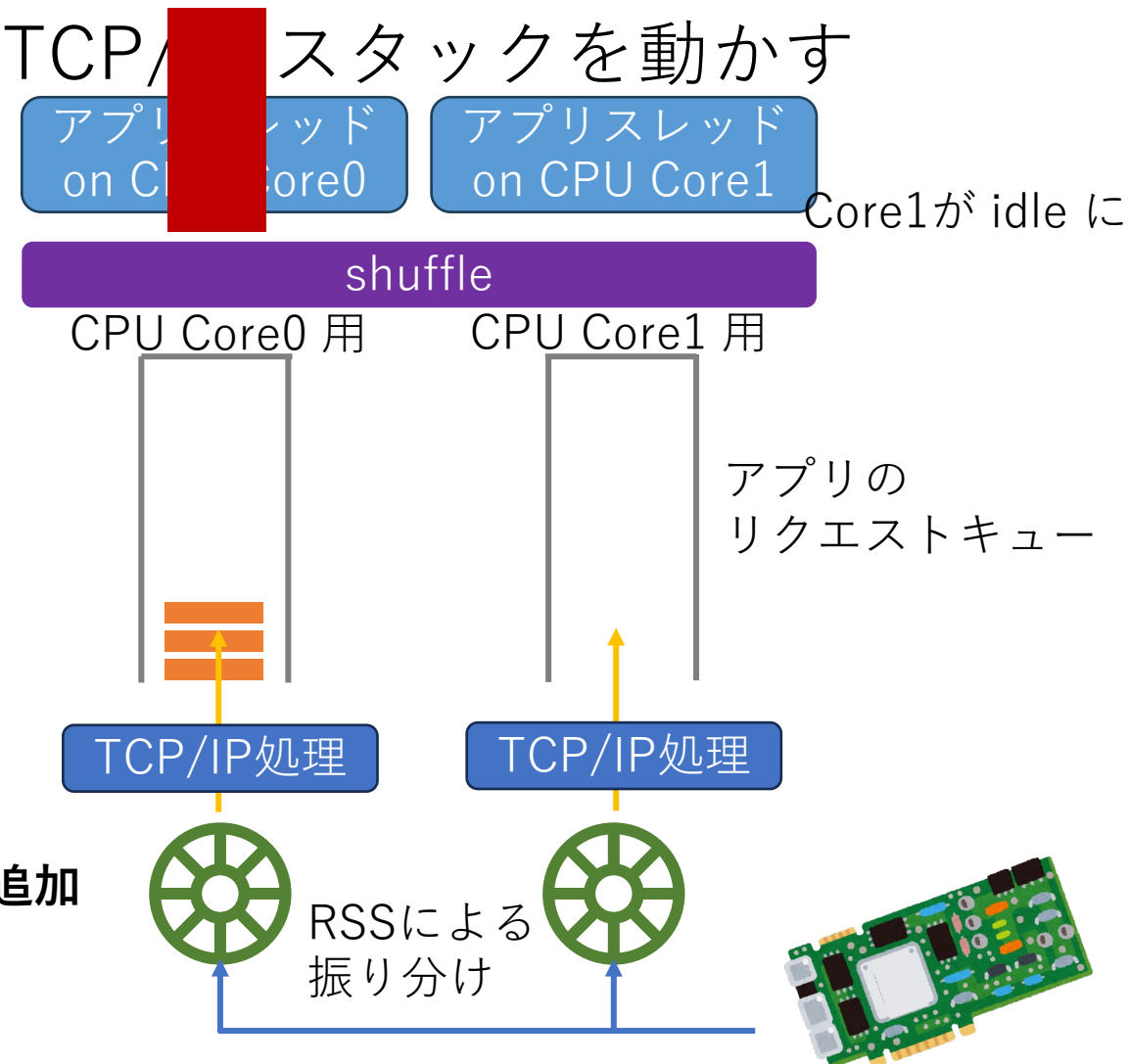
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

提案手法：
Shuffle 層を追加

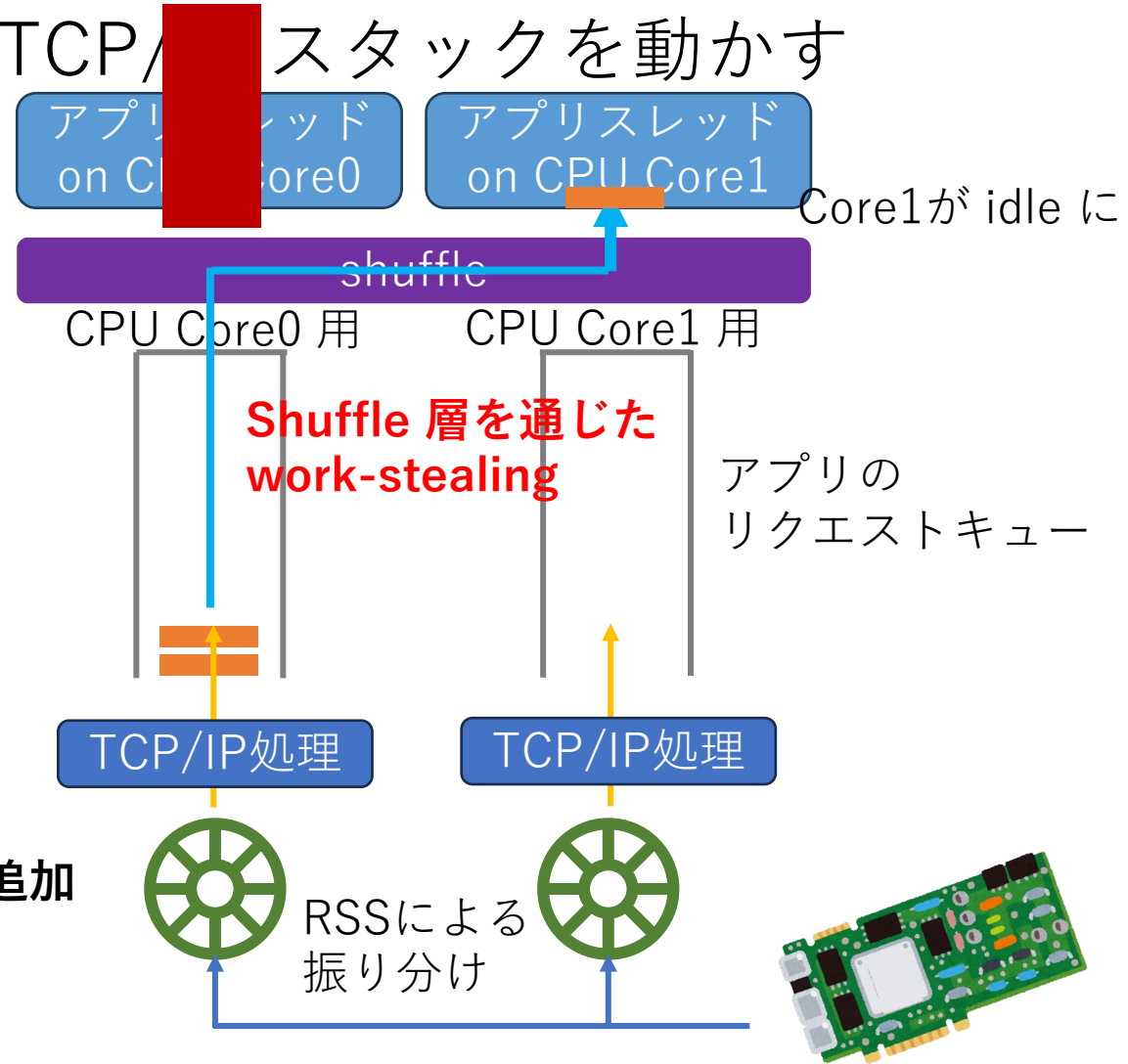


TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

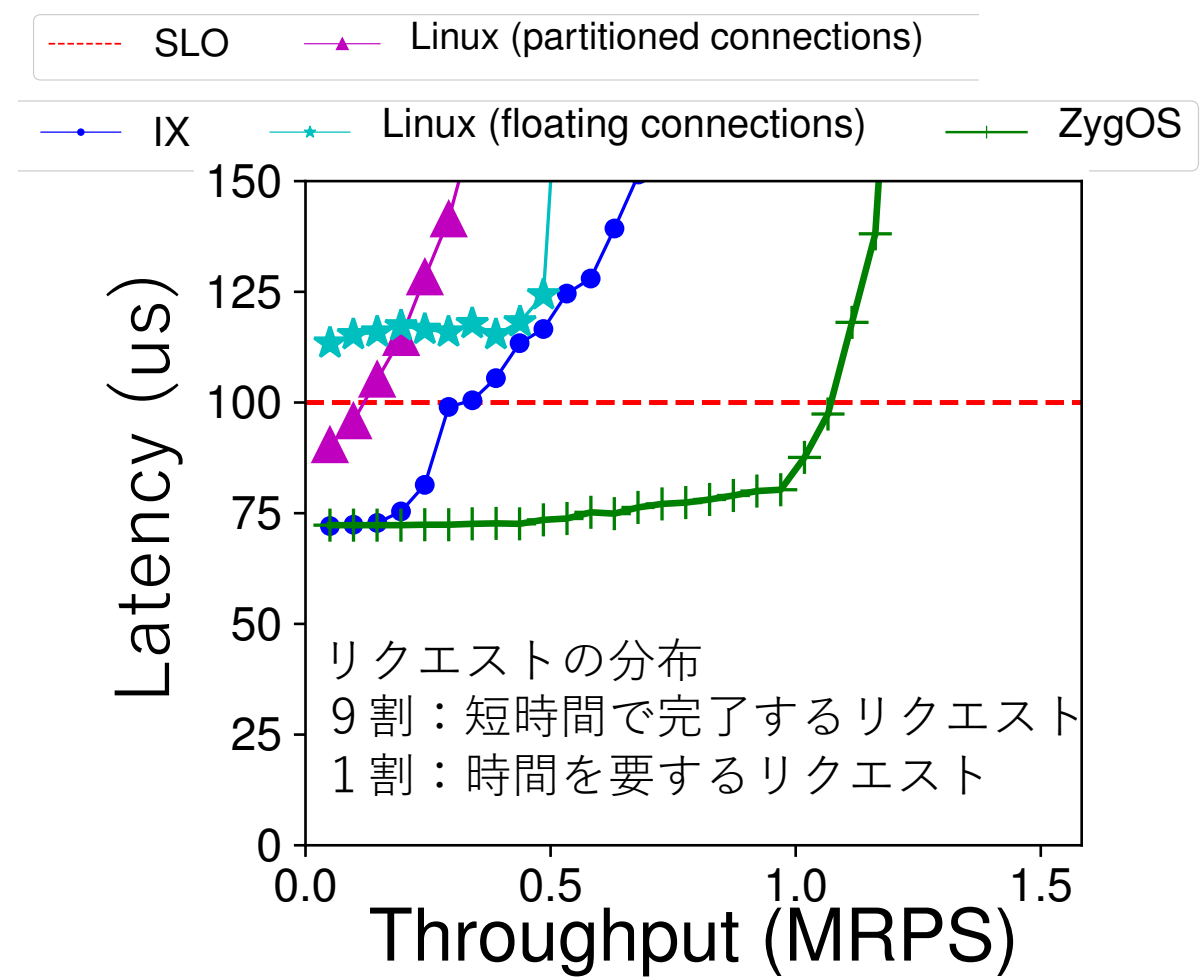
拡張

提案手法：
Shuffle 層を追加



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

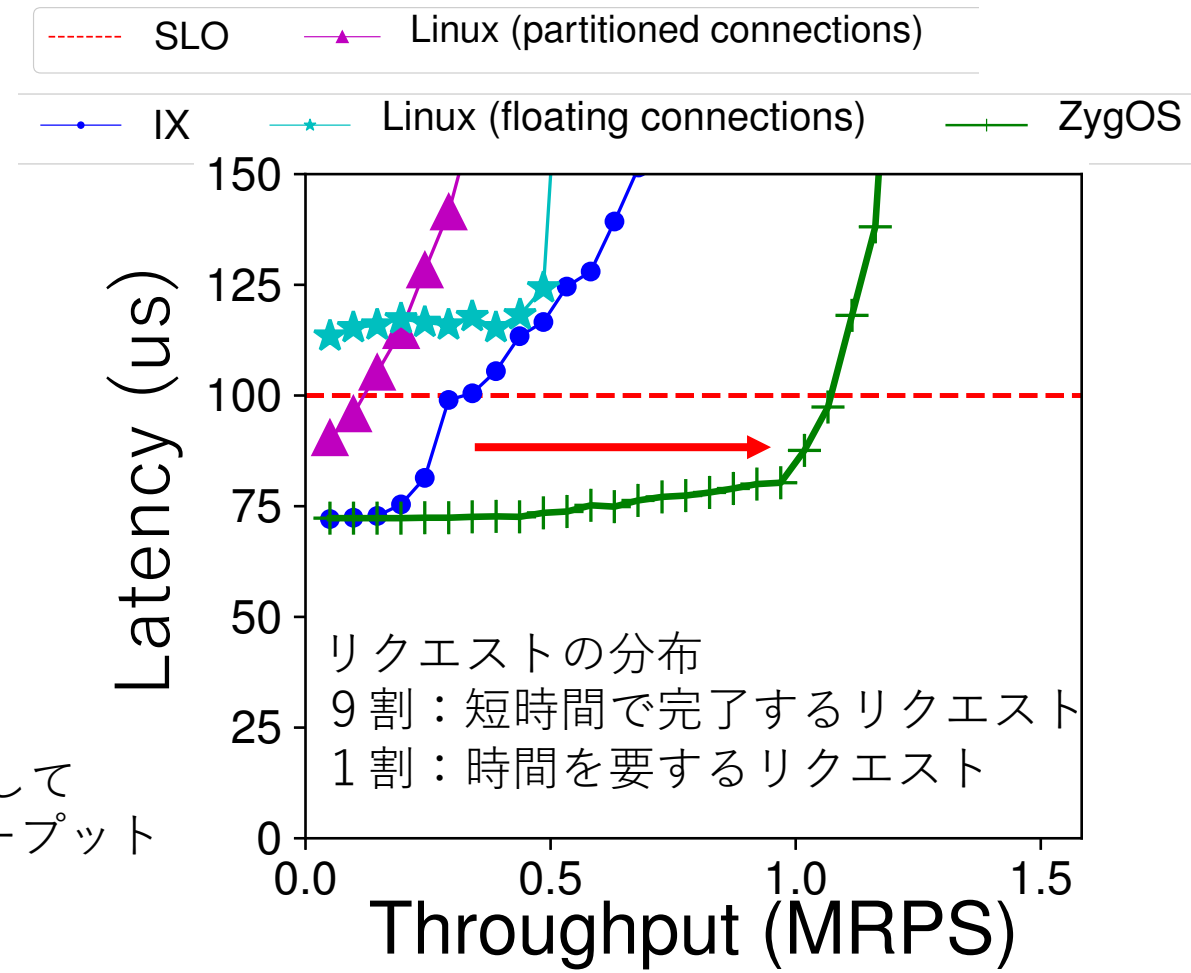


TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



IX と比較して
高いスループット



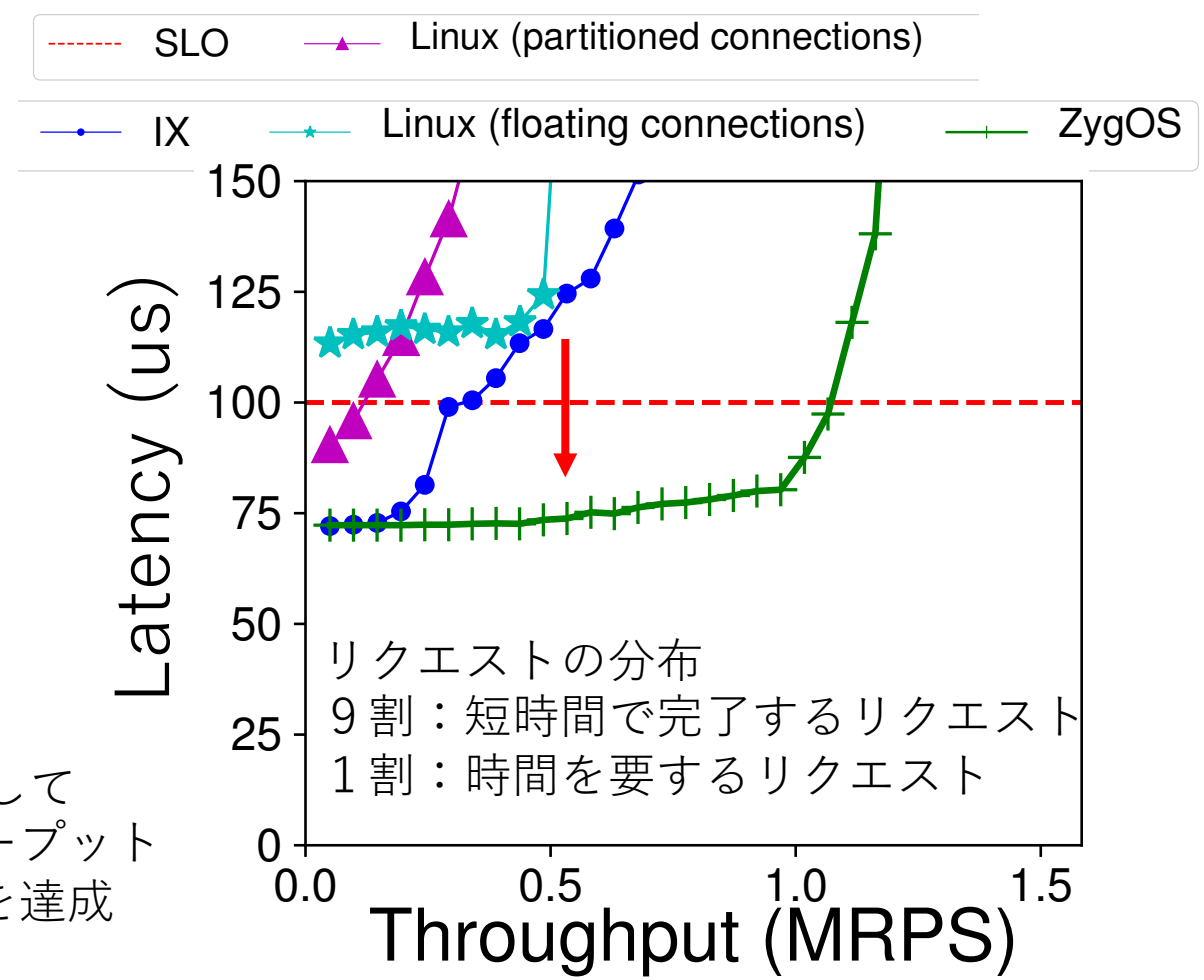
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - **ZygOS (SOSP 2017)**
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)




拡張

IX と比較して
高いスループット
低い遅延を達成



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - **Shinjuku (NSDI 2019)**
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)
- 

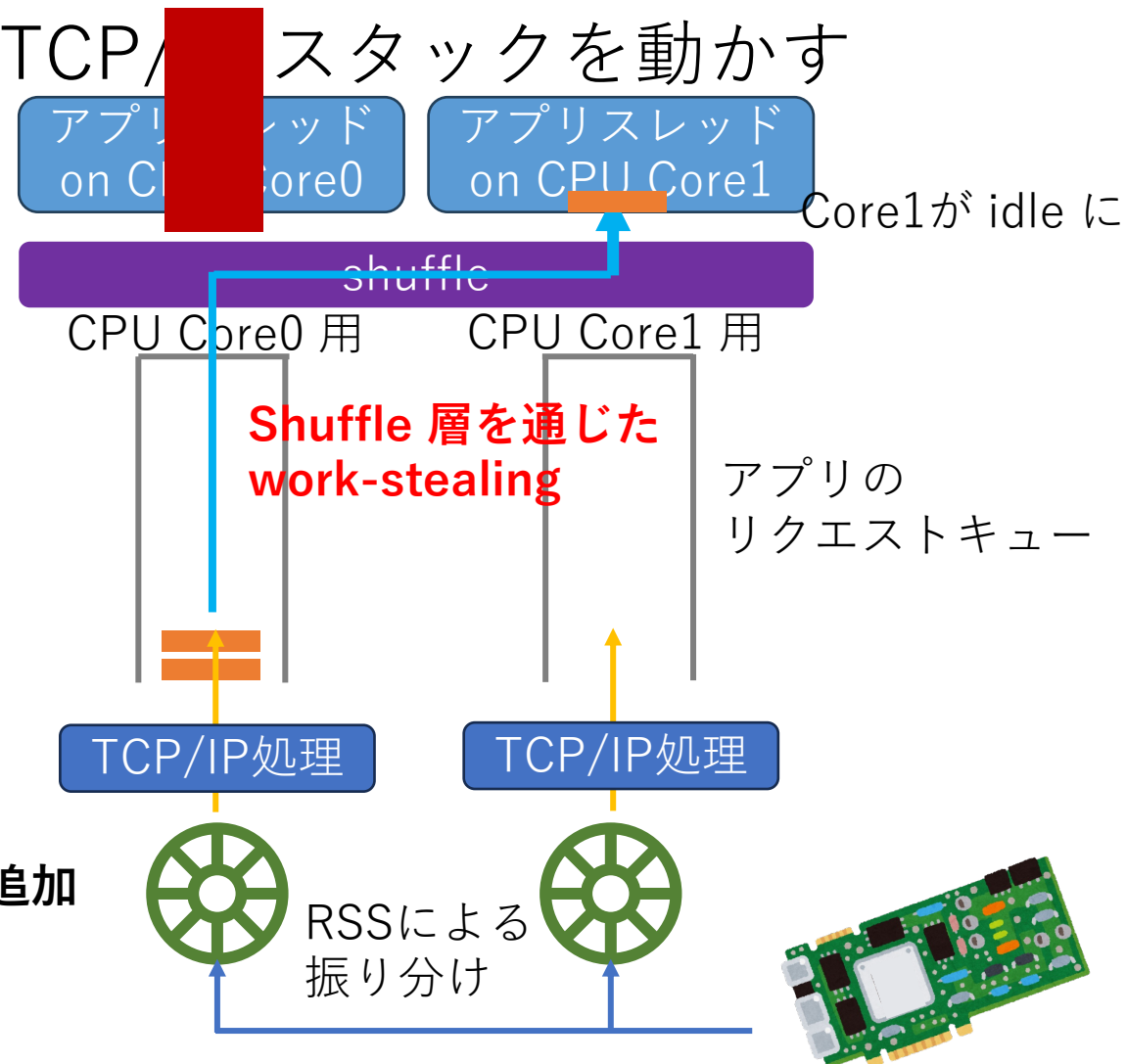
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- **ZygOS (SOSP 2017)**
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

提案手法：
Shuffle 層を追加



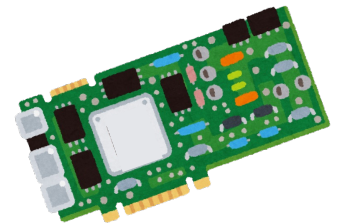
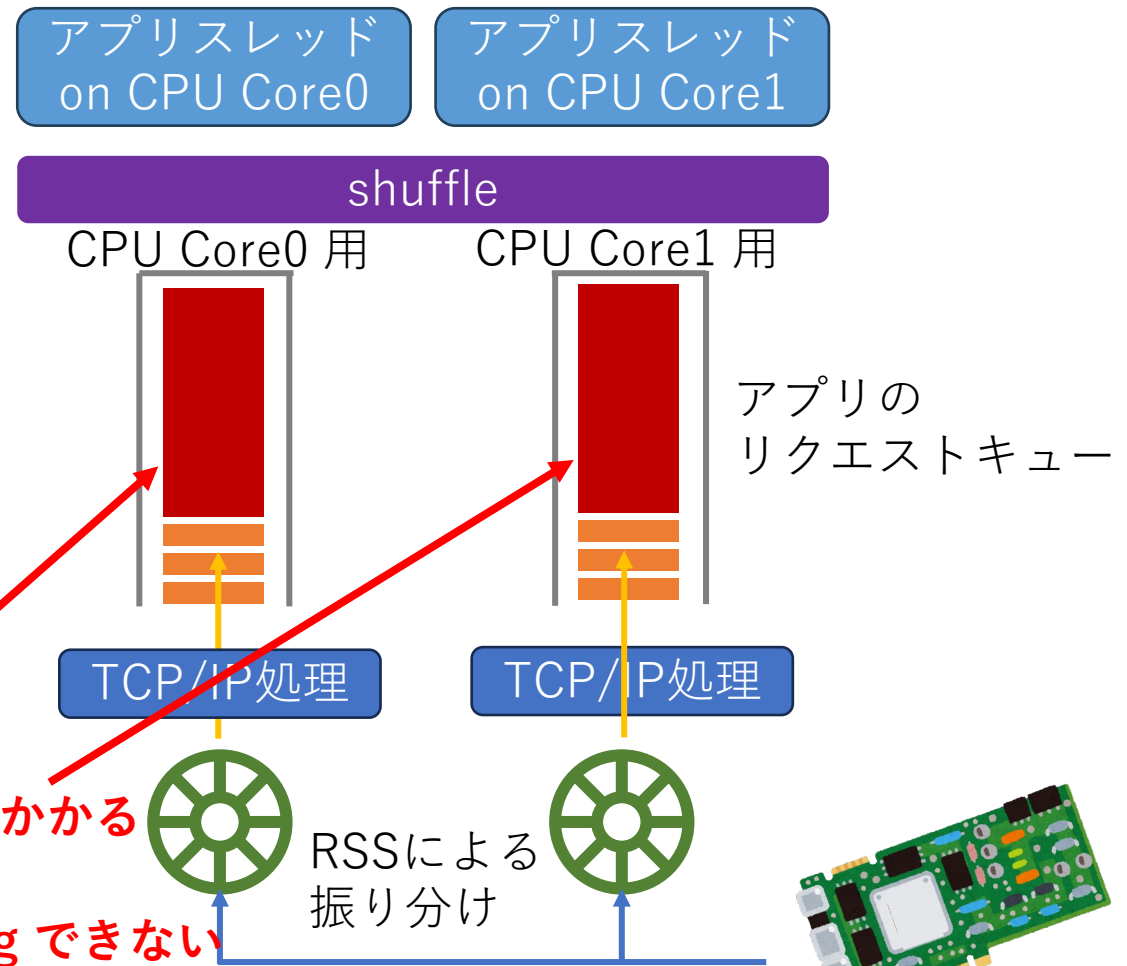
TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

拡張

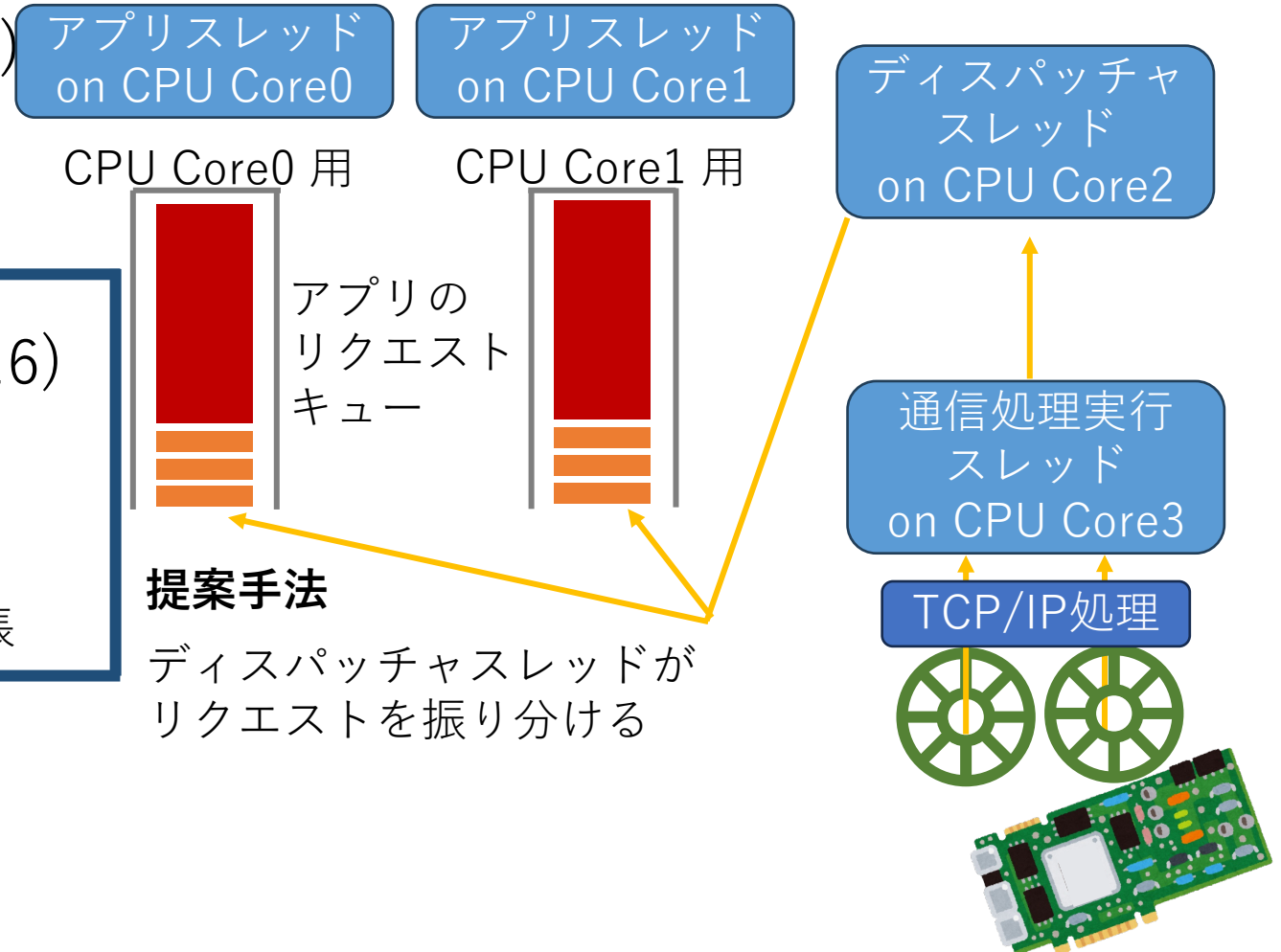
問題：
 全部のコアに時間のかかる
 リクエストが来ると
 Work-stealing できない



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

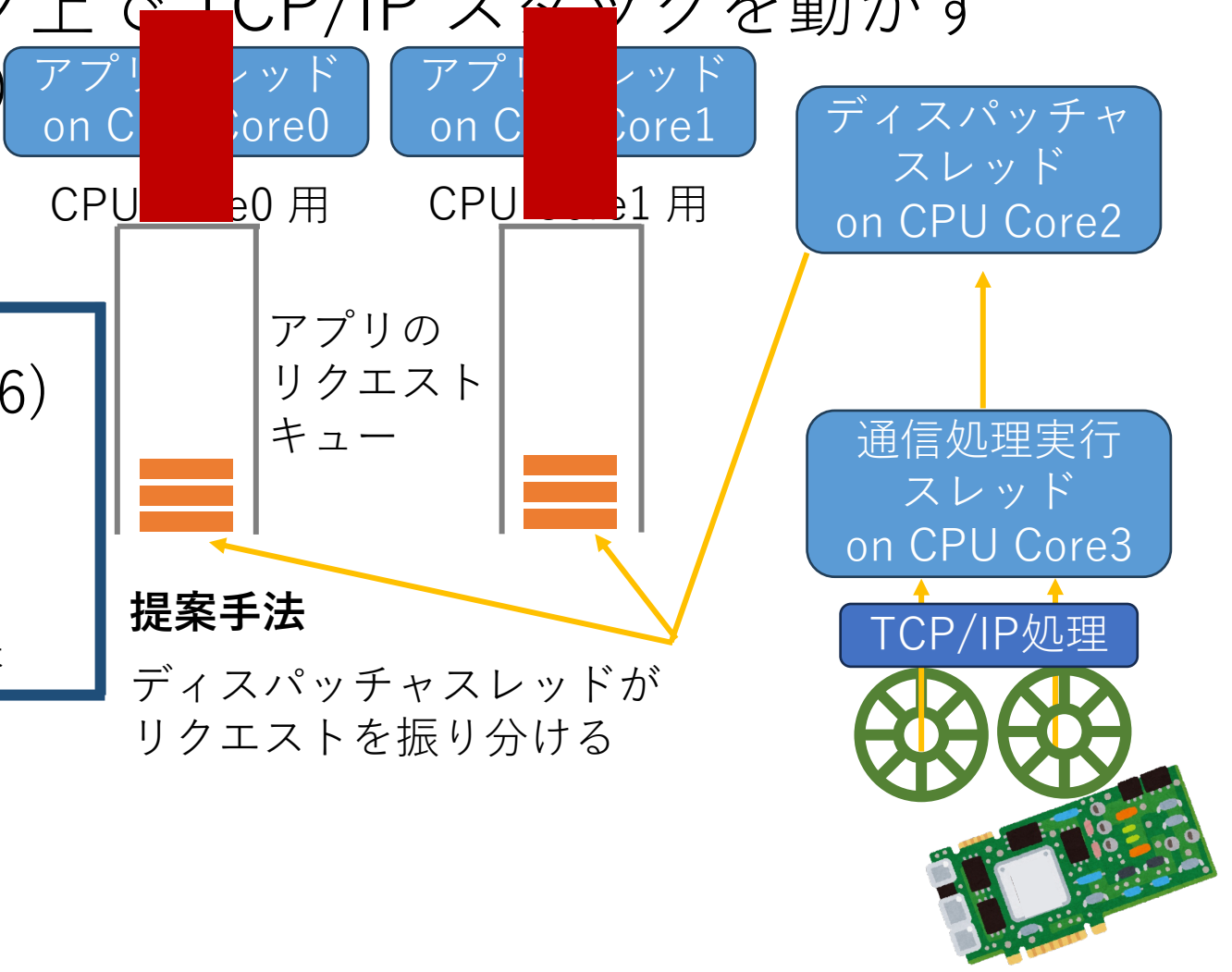
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

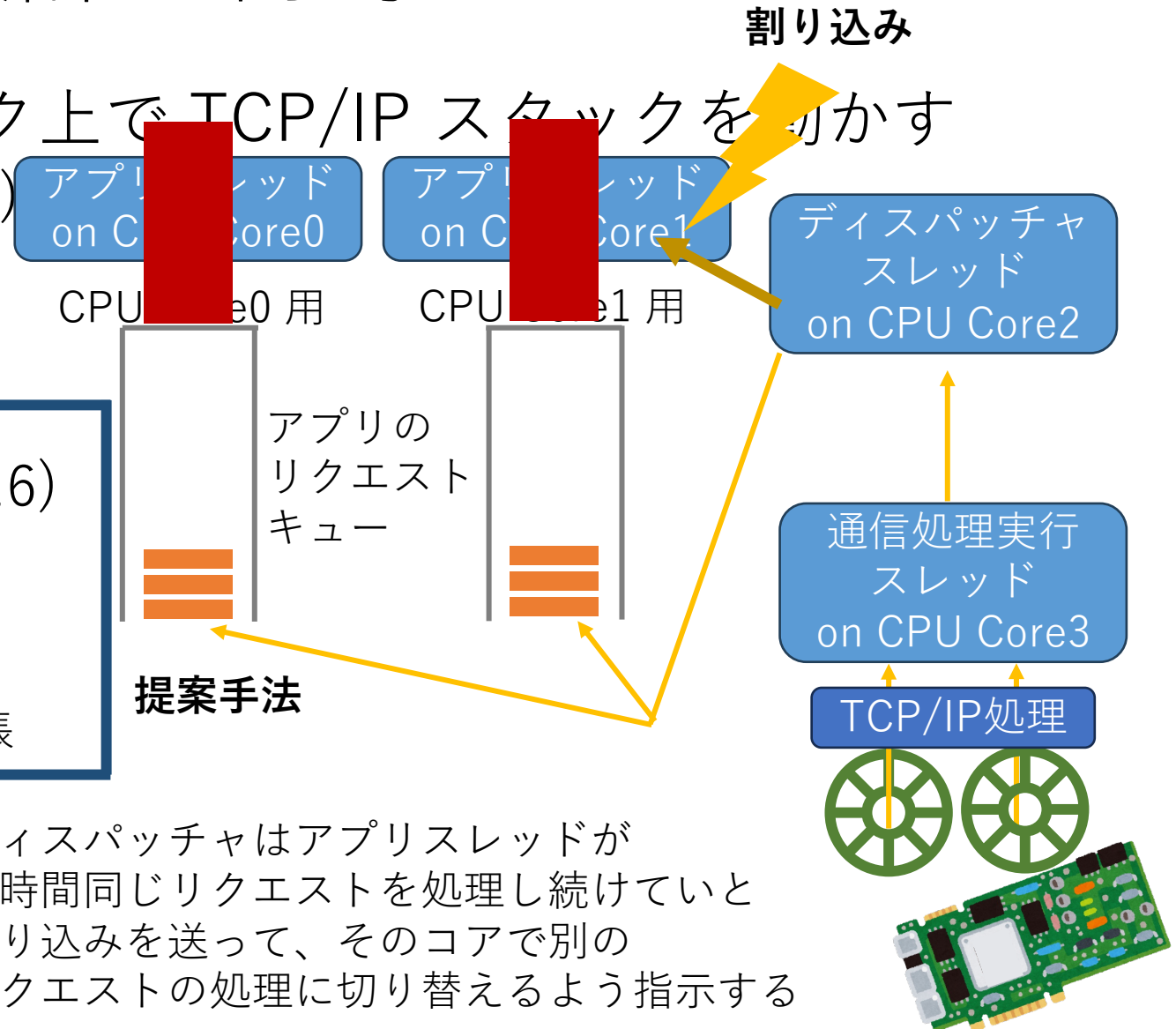
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

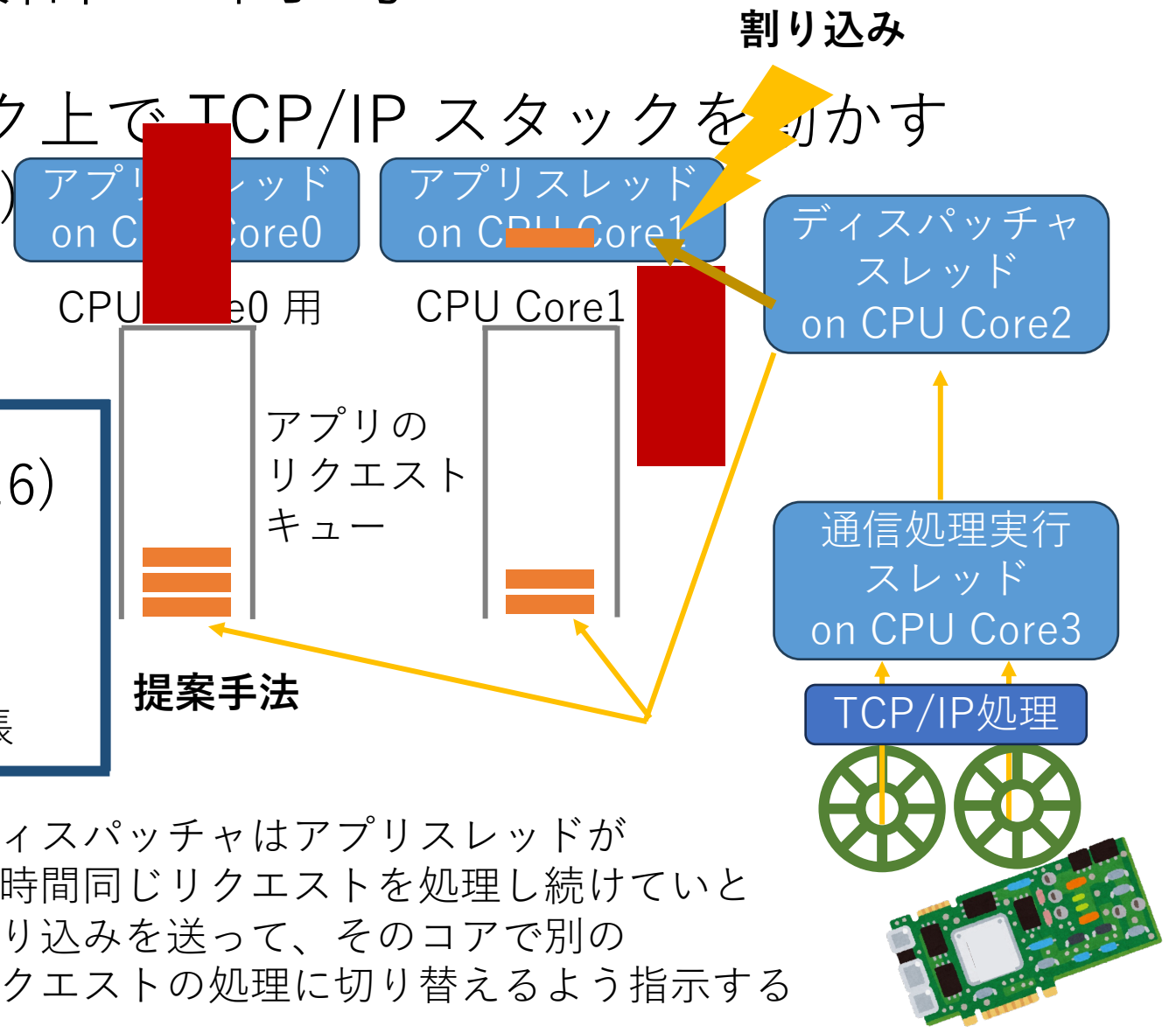
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

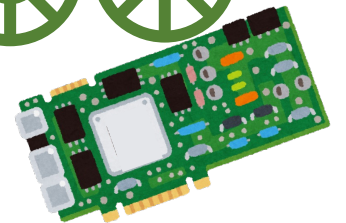
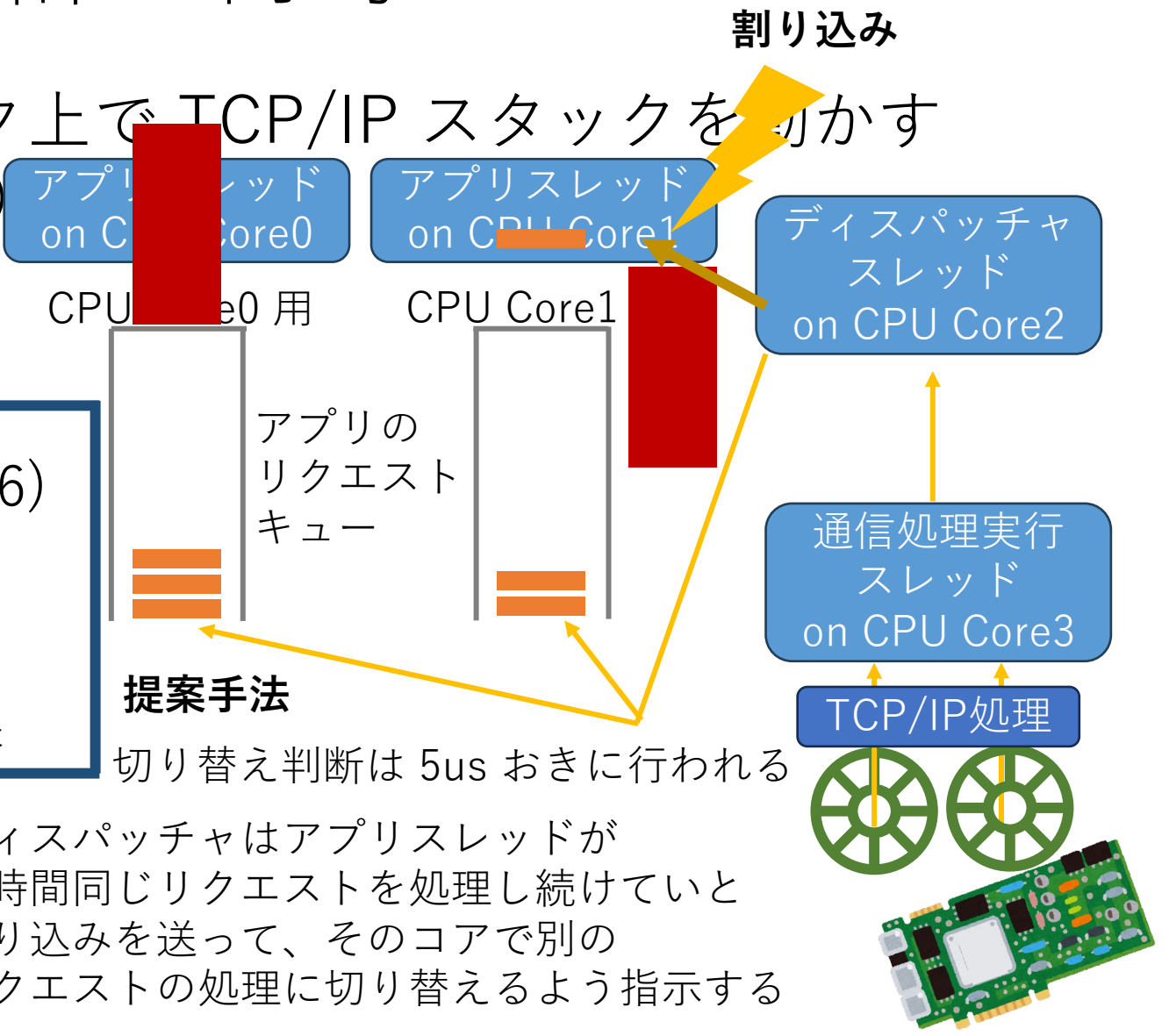
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

• パケット I/O フレームワーク上で TCP/IP スタックを動かす

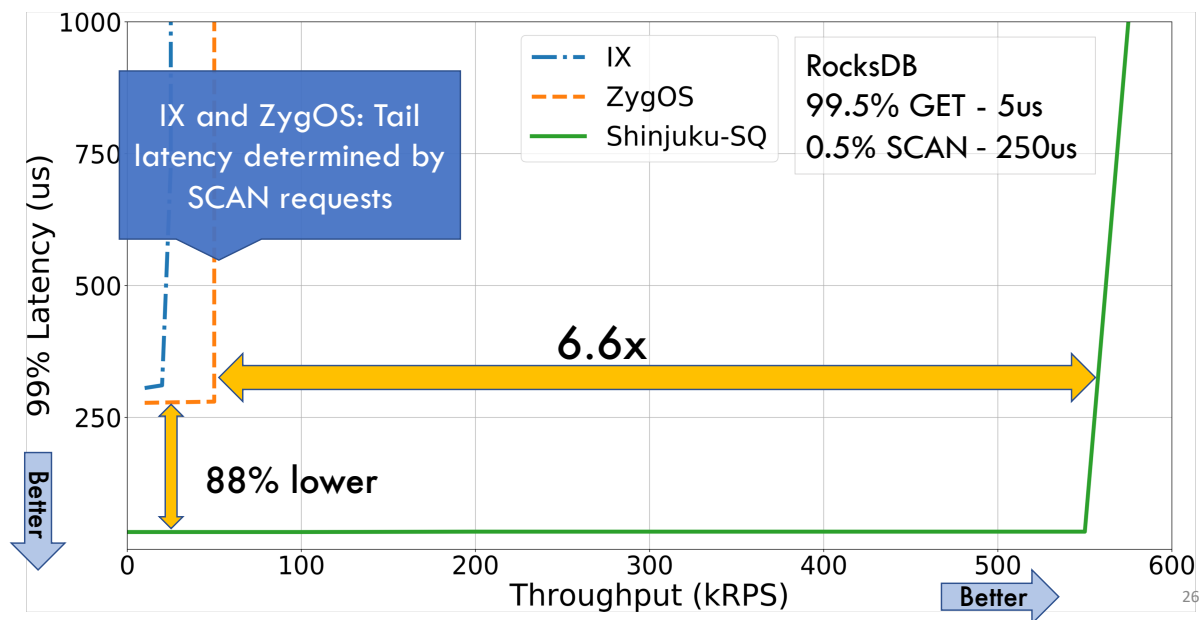
- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- **Shinjuku (NSDI 2019)**
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - **Shinjuku (NSDI 2019)**
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

拡張



研究紹介

TCP/IP スタック設計

パケット I/O フレームワークを適用する

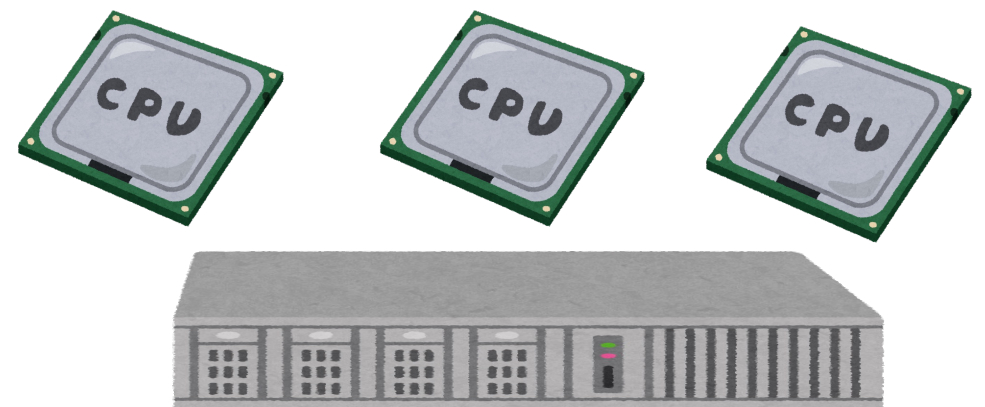
様々なワークロードを一つのサーバーで動かしつつ
遅延が重要なアプリが目標の応答性能を達成できるようにする

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

要件

- 色々なワークロードを**一つのサーバー**で動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
 - CPU を消費するワークロード (Hadoop, Spark 等)

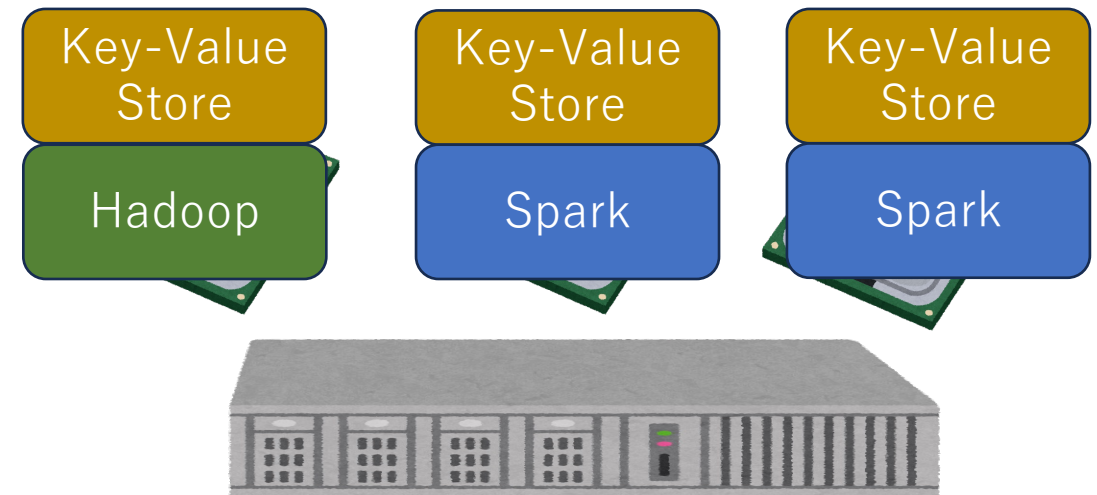


TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

要件

- 色々なワークロードを**一つのサーバー**で動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
 - CPU を消費するワークロード (Hadoop, Spark 等)



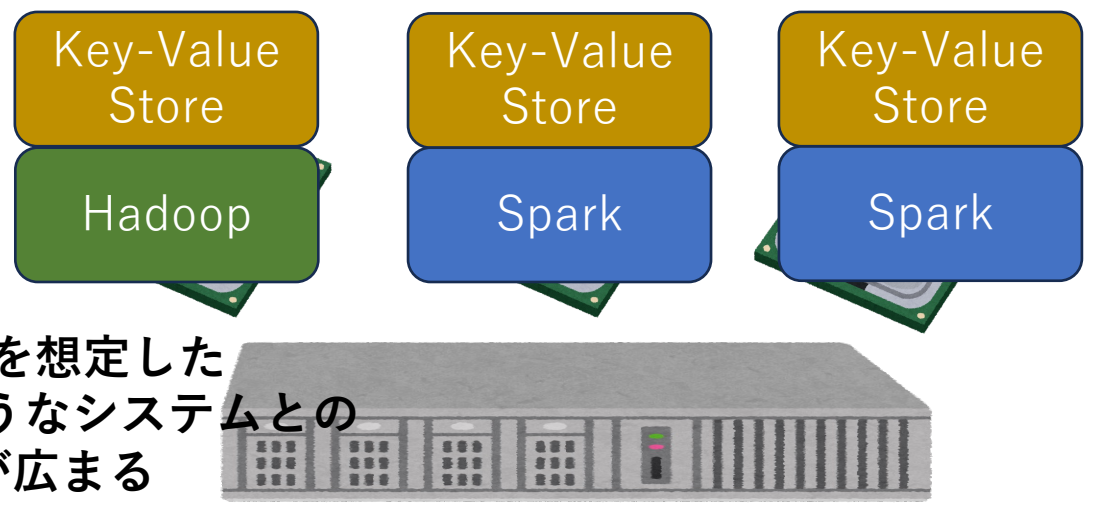
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

要件

- 色々なワークロードを一つのサーバーで動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
- CPU を消費するワークロード (Hadoop, Spark 等)

CPU の専有を想定した DPDK のようなシステムとの組み合わせが広まる



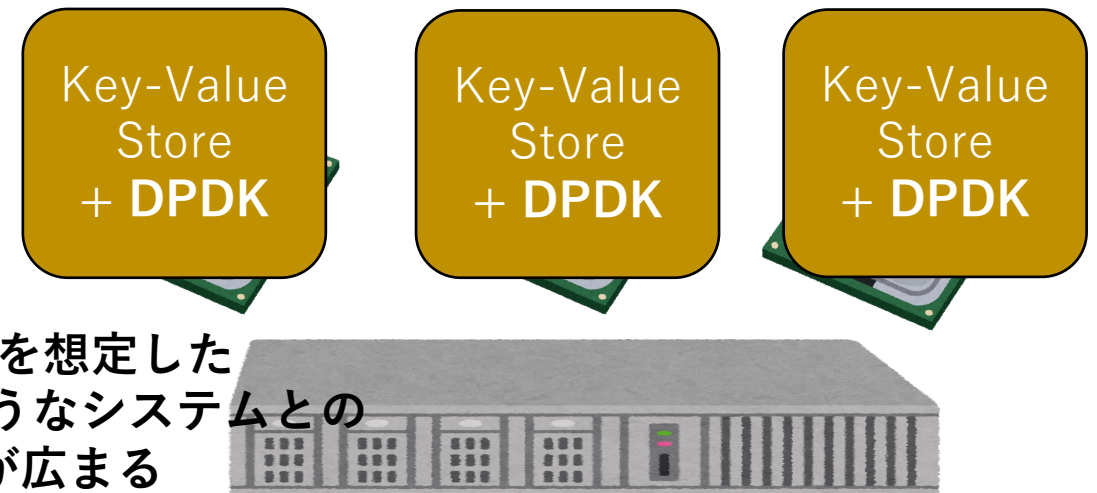
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

要件

- 色々なワークロードを一つのサーバーで動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
 - CPU を消費するワークロード (Hadoop, Spark 等)

CPU の専有を想定した
DPDK のようなシステムとの
組み合わせが広まる



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- **Shenango (NSDI 2019)**
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- Demikernel (SOSP 2021)

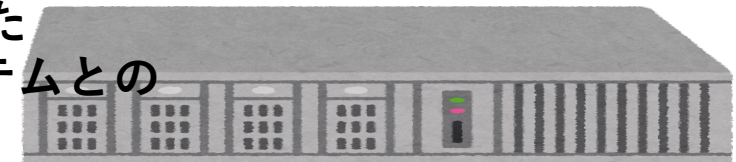
要件

- 色んなワークロードを一つのサーバーで動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
- CPU を消費するワークロード (Hadoop, Spark 等)

問題：DPDK 等に CPU を占有させると Hadoop, Spark 等のワークロードへ割り当てる CPU サイクルがなくなる



CPU の専有を想定した DPDK のようなシステムとの組み合わせが広まる



TCP/IP スタック設計の再考

- パケット I/O フレームワーク
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

目的

低遅延が重要なワークロードに適切な数の CPU コアを割り当てられるようにしたい

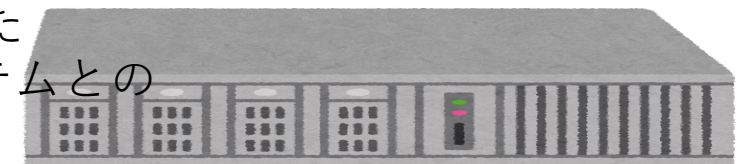
要件

- 色々なワークロードを**一つのサーバー**で動かしたい
- 低遅延が重要なワークロード (Key-Value Store 等)
 - CPU を消費するワークロード (Hadoop, Spark 等)

問題：DPDK 等に CPU を占有させると Hadoop, Spark 等のワークロードへ割り当てる CPU サイクルがなくなる



CPU の専有を想定した DPDK のようなシステムとの組み合わせが広まる



TCP/IP スタック設計の再考

- パケット I/O フレームワーク
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

目的
 低遅延が重要なワークロードに適切な数の CPU コアを割り当てられるようにしたい

要件
 Key-Value Store へのリクエストが少ない時

問題：DPDK 等に CPU を占有させると Hadoop, Spark 等のワークロードへ割り当てる CPU サイクルがなくなる



CPU の専有を想定した DPDK のようなシステムとの組み合わせが広まる



TCP/IP スタック設計の再考

- パケット I/O フレームワーク
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

目的
 低遅延が重要なワークロードに適切な数の CPU コアを割り当てられるようにしたい

要件
 Key-Value Store へのリクエストが多い時

問題：DPDK 等に CPU を占有させると Hadoop, Spark 等のワークロードへ割り当てる CPU サイクルがなくなる

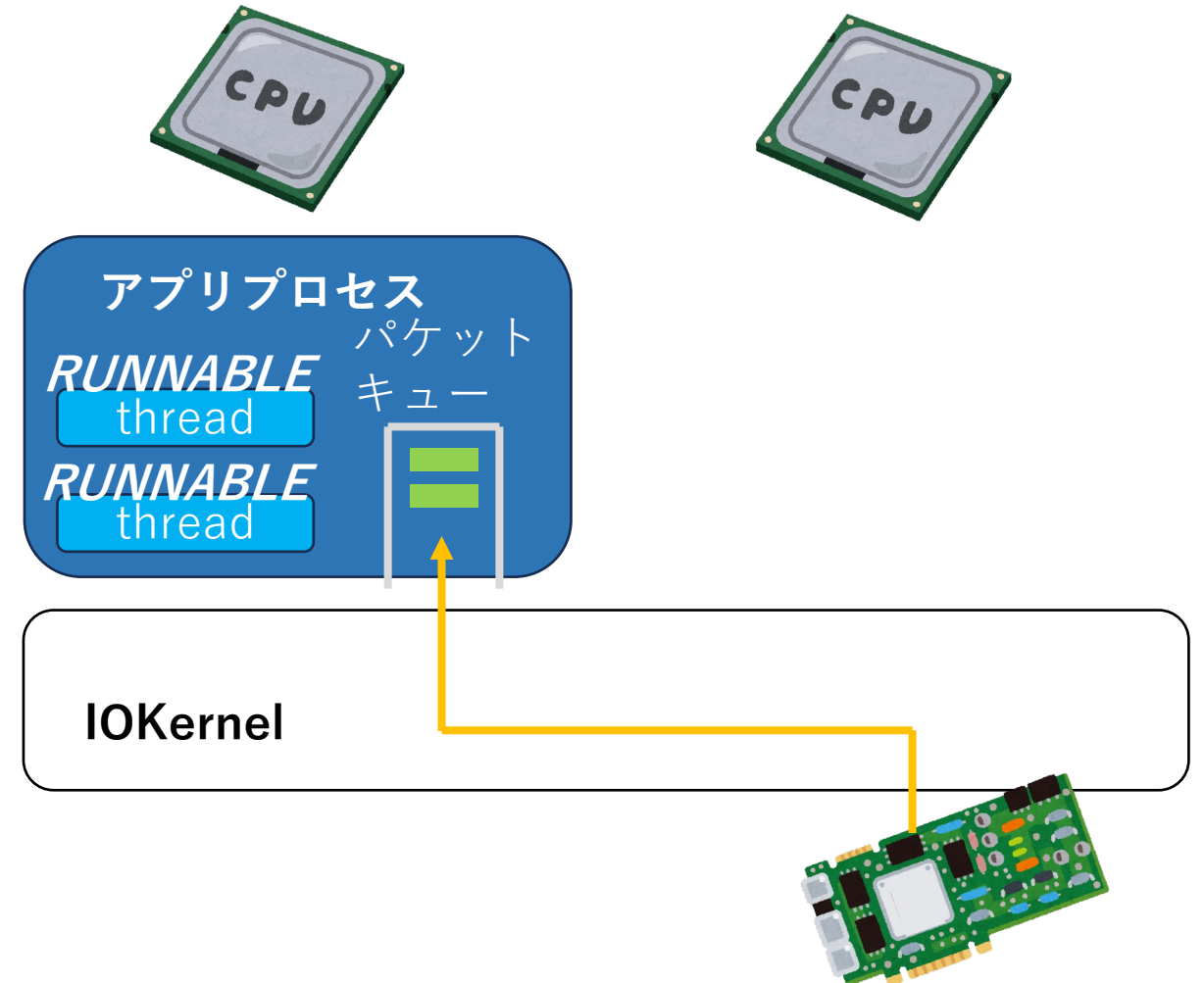


CPU の専有を想定した DPDK のようなシステムとの組み合わせが広まる



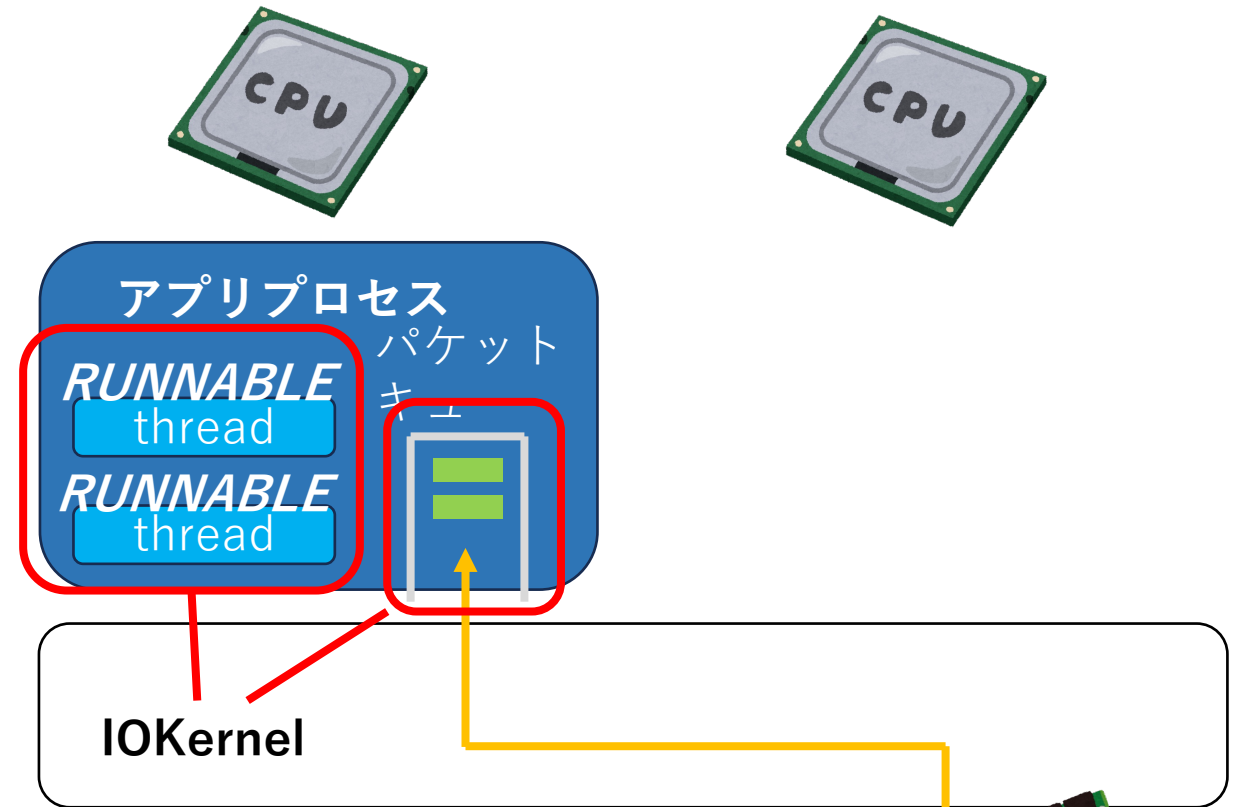
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

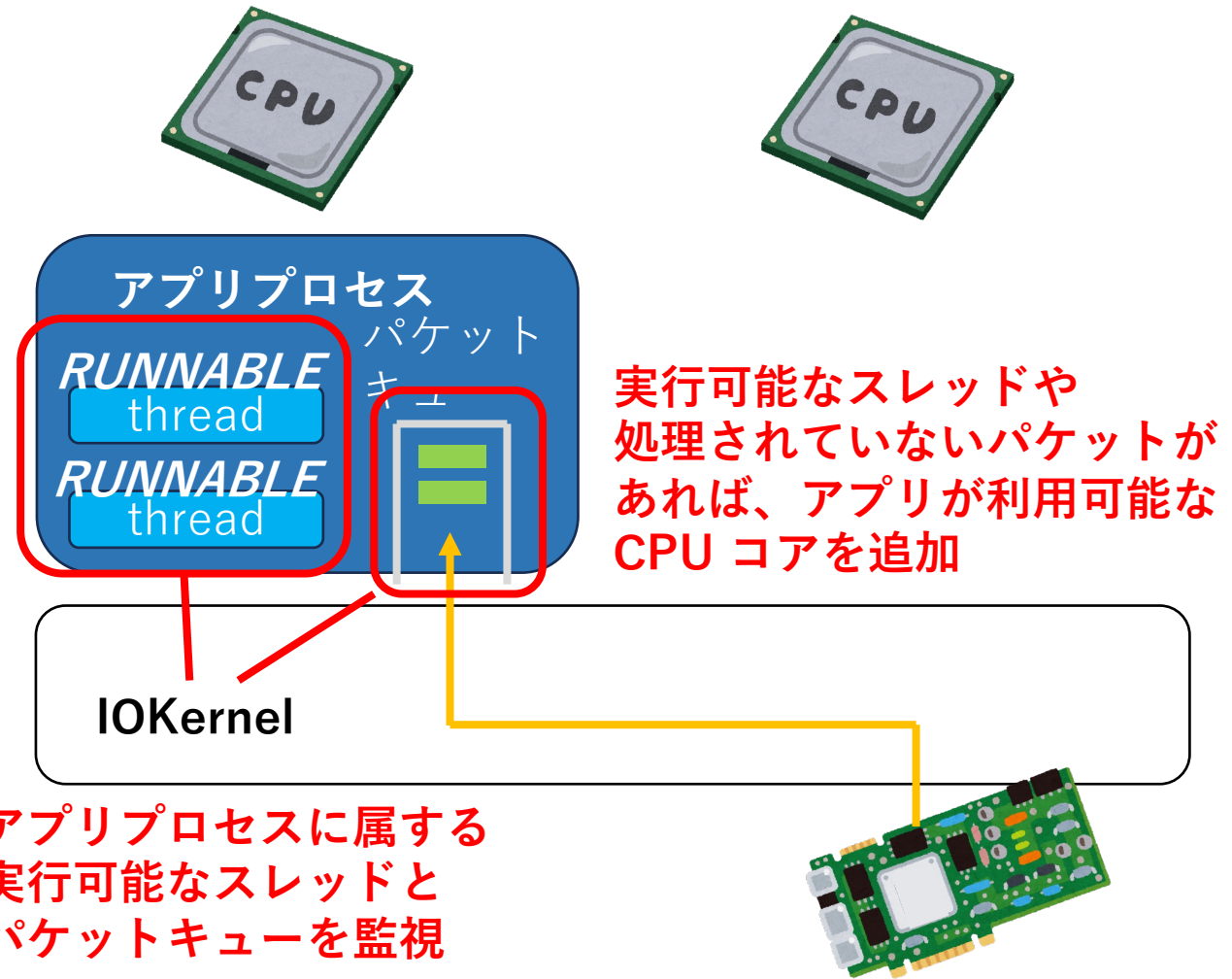
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



アプリプロセスに属する
実行可能なスレッドと
パケットキューを監視

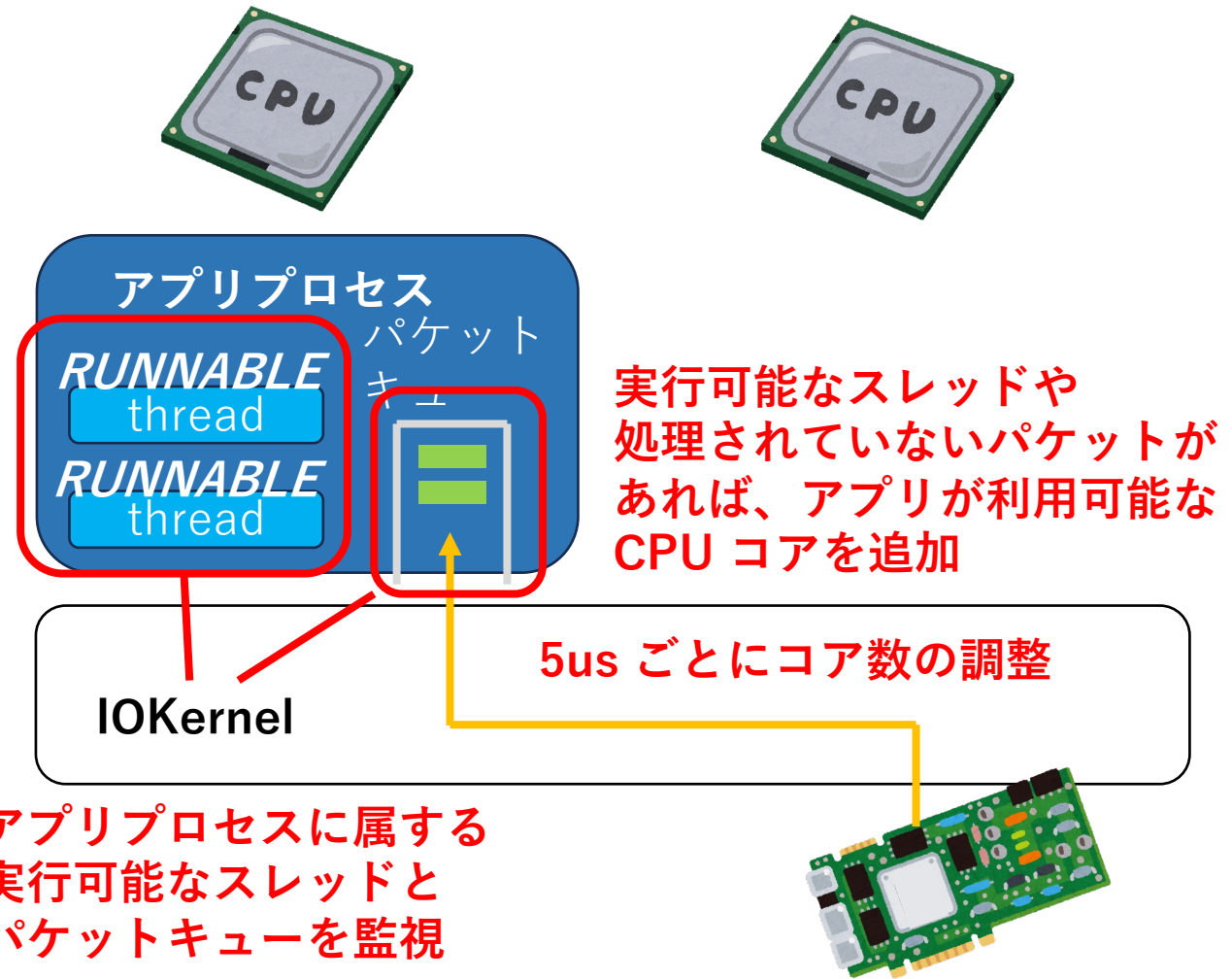
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



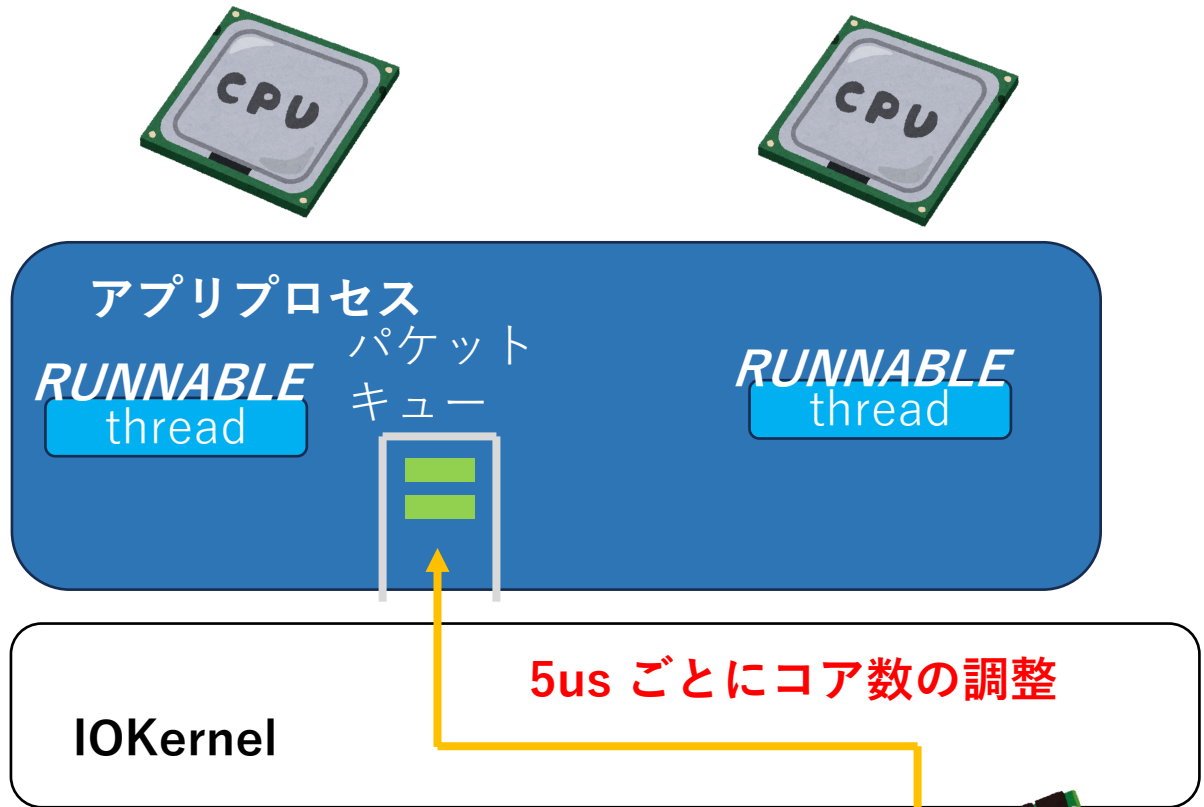
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

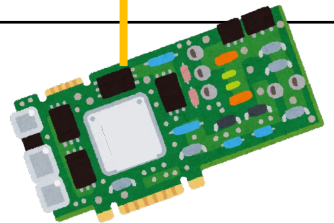


TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

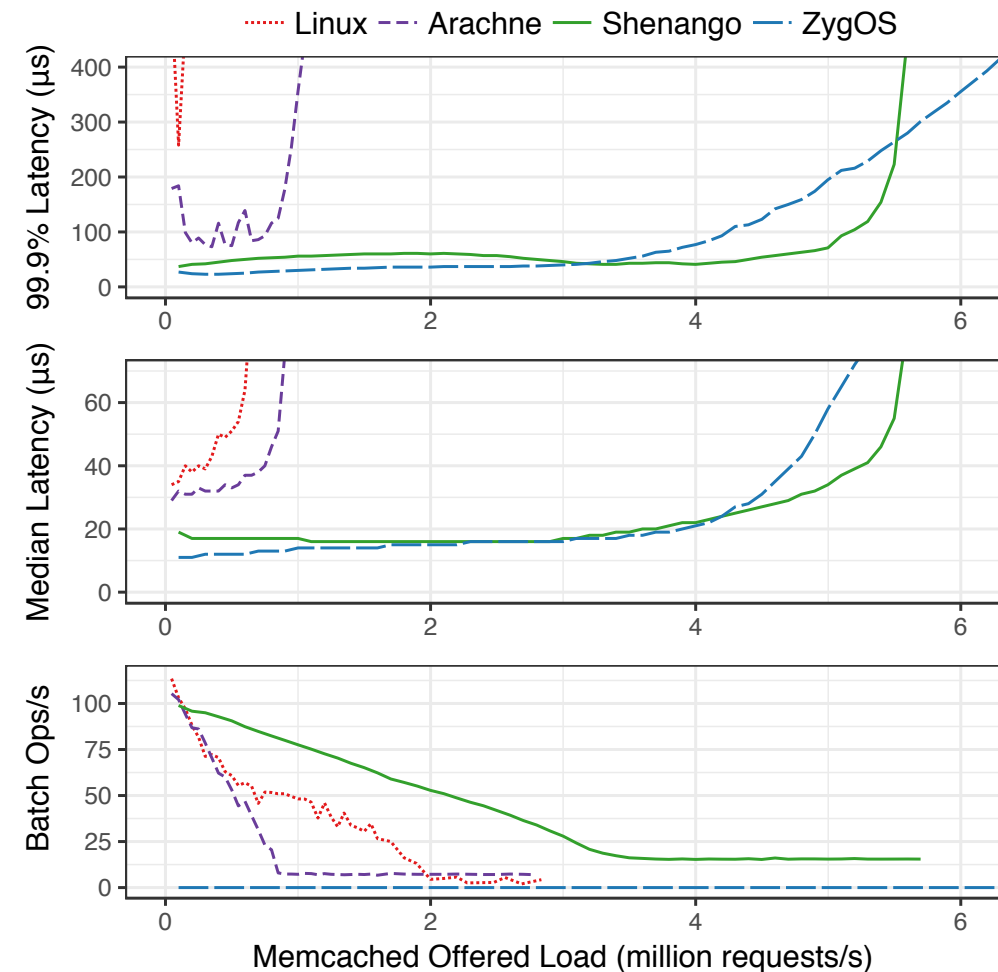


アプリプロセスに属する
実行可能なスレッドと
パケットキューを監視



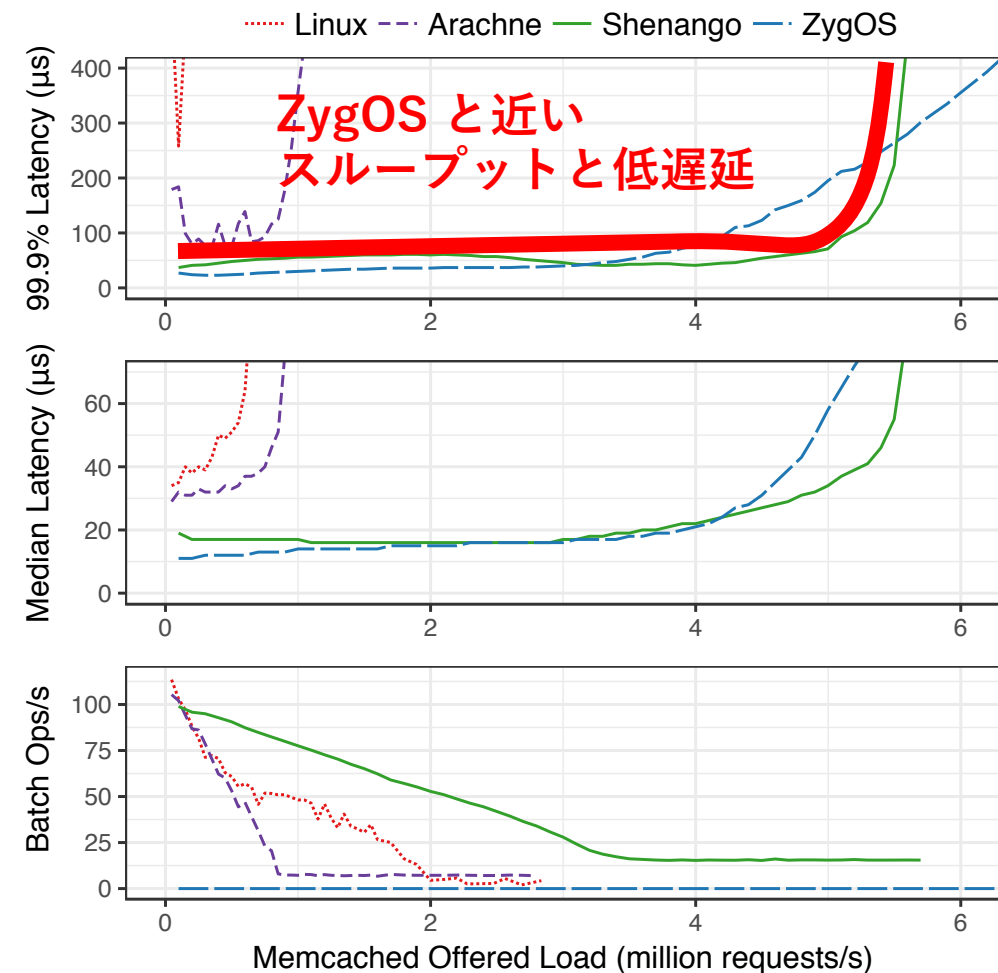
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019) Memcached と PARSEC swaptions を実行
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



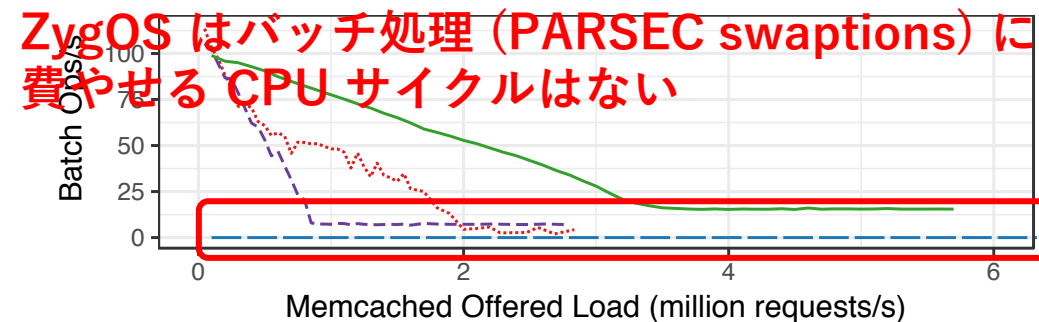
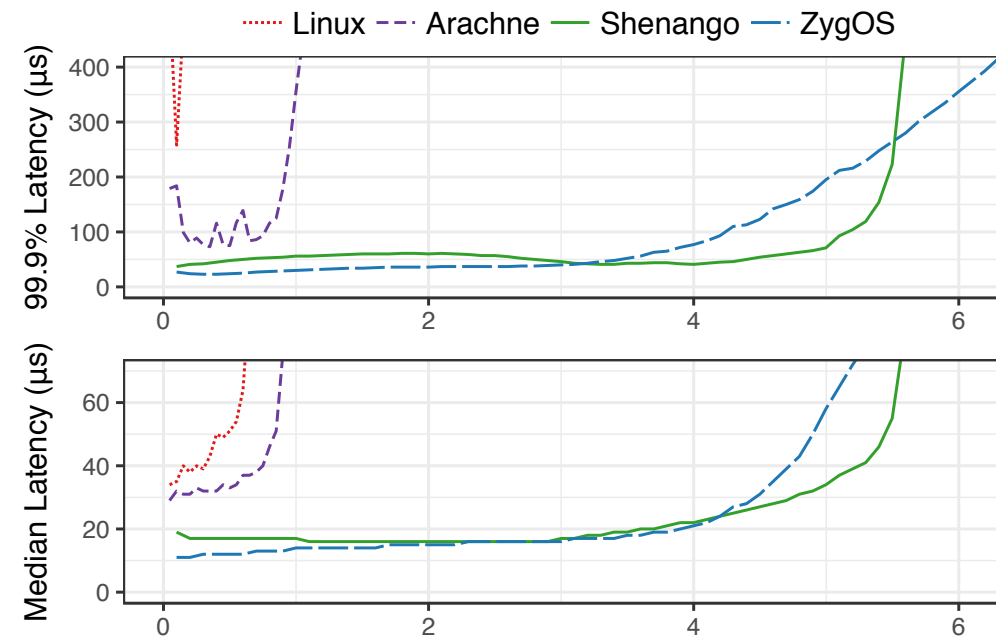
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019) Memcached と PARSEC swaptions を実行
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

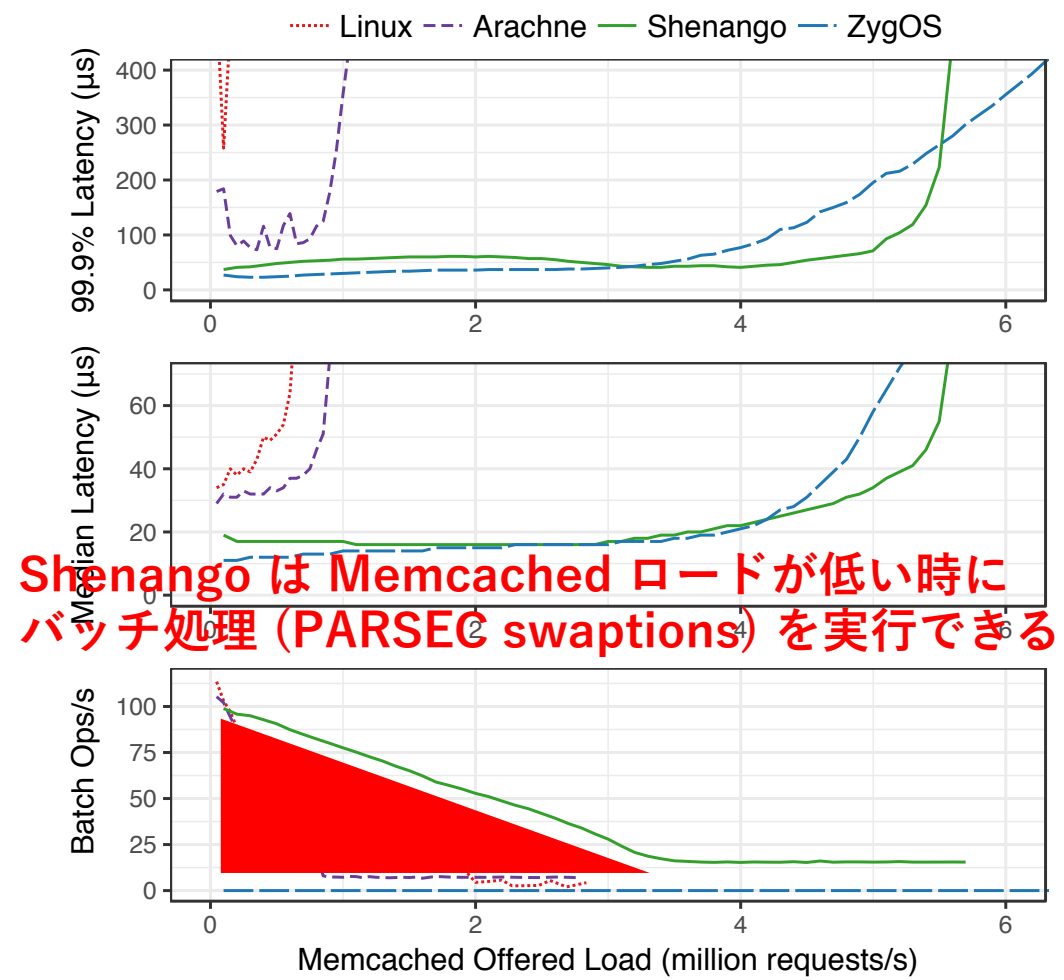
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019) Memcached と PARSEC swaptions を実行
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - **Shenango (NSDI 2019)**
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - Demikernel (SOSP 2021)

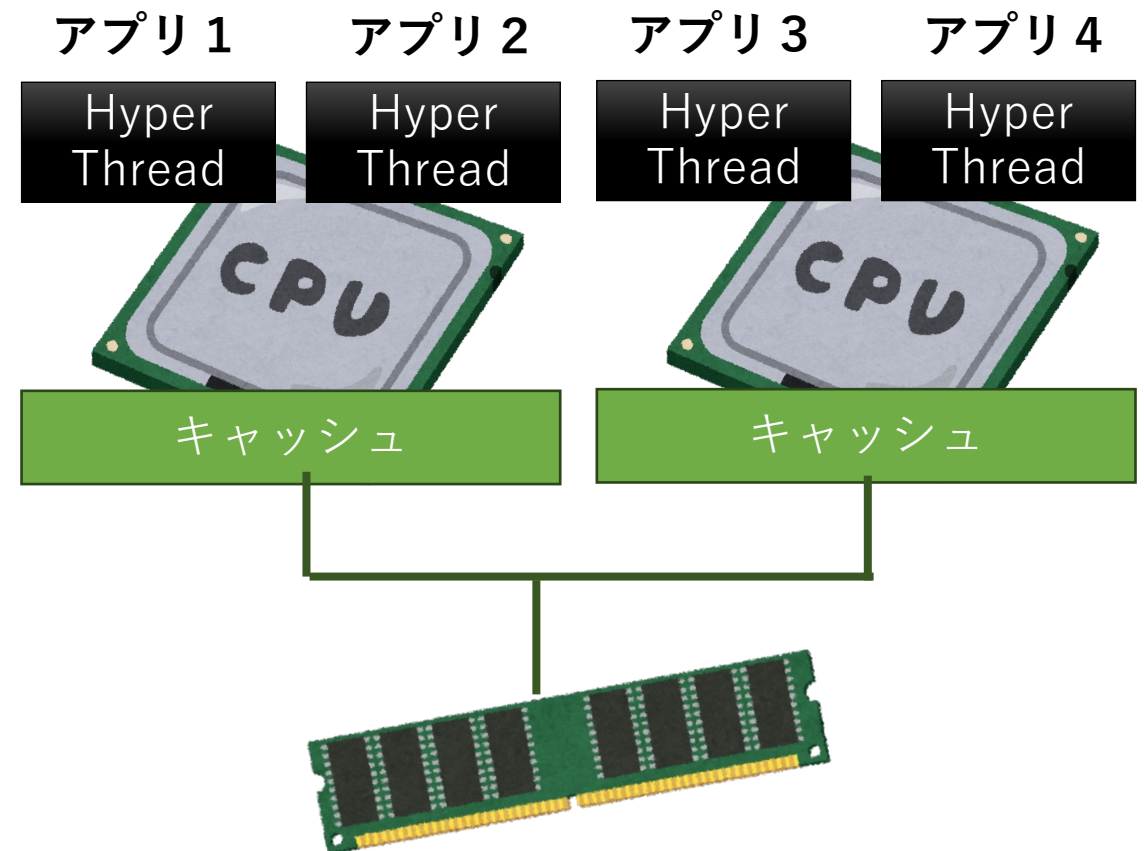
Memcached と PARSEC swaptions を実行



Shenango は Memcached ロードが低い時にバッチ処理 (PARSEC swaptions) を実行できる

TCP/IP スタック設計の再考

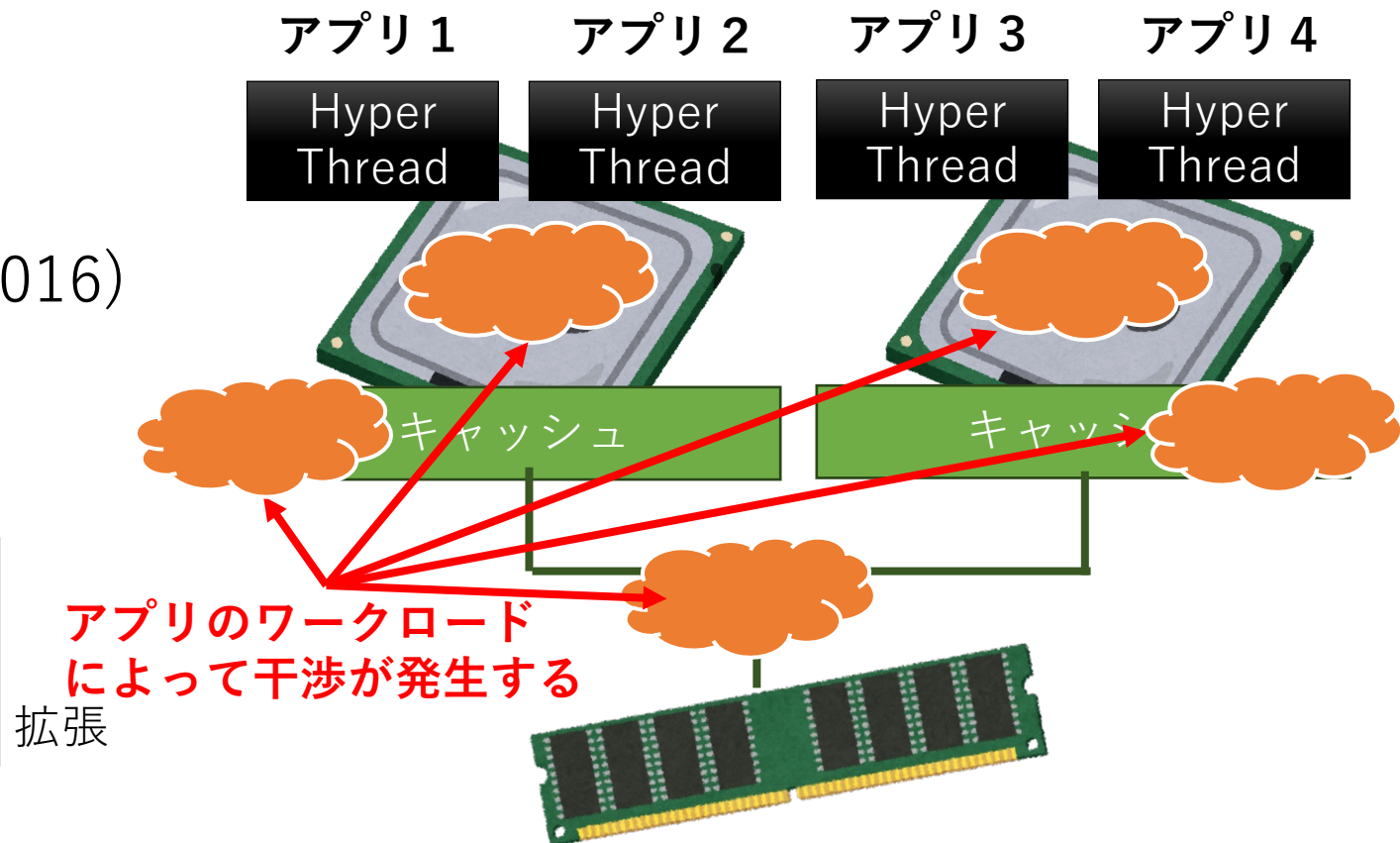
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)



拡張

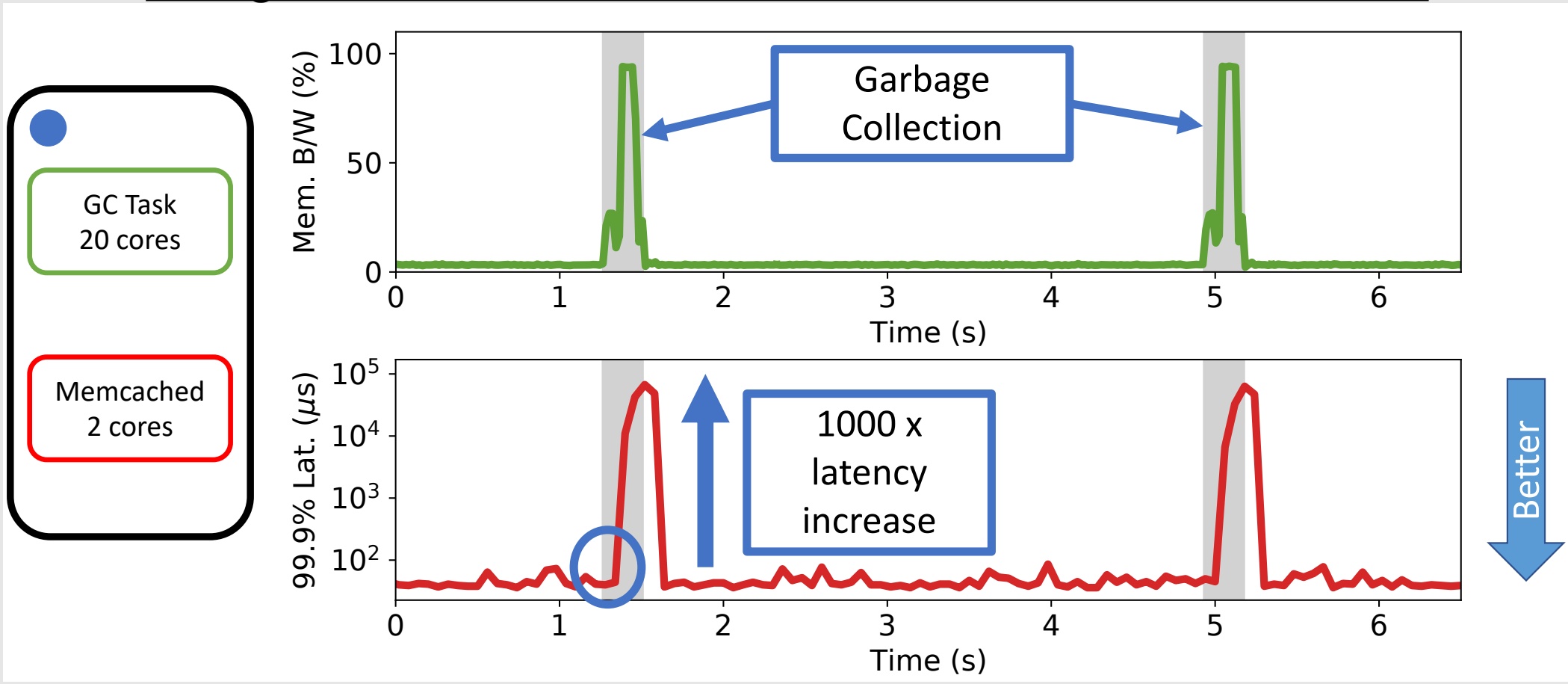
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)



TCP/IP スタック設計の再考

Garbage Collection を行うタスクと、Memcached を同じマシンで動かす場合

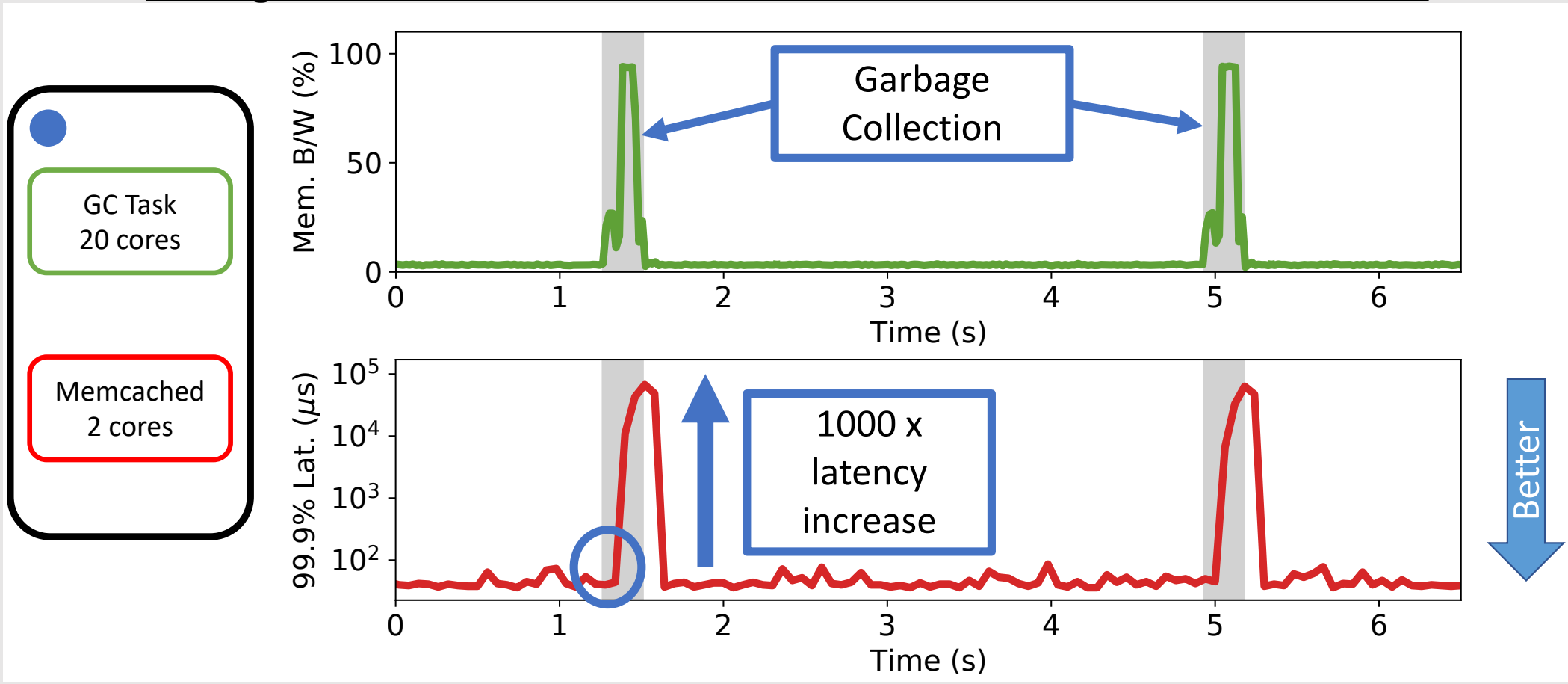


Garbage Collection が動くとメモリ帯域が干渉して Memcached の応答遅延が増加する

TCP/IP スタック設計の再考

目的：このような干渉を避けて、遅延が重要なシステムの遅延が一定以上を越えないようにしたい

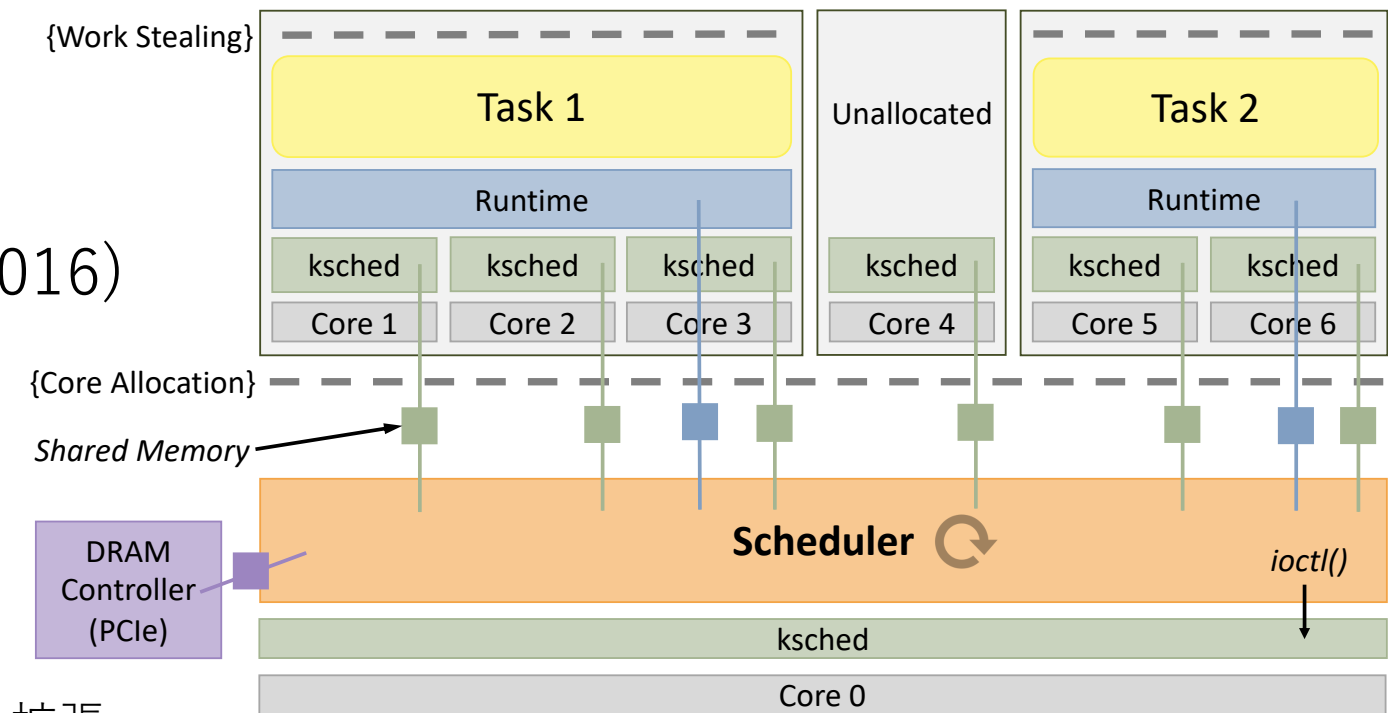
Garbage Collection を行うタスクと、Memcached を同じマシンで動かす場合



Garbage Collection が動くとメモリ帯域が干渉して Memcached の応答遅延が増加する

TCP/IP スタック設計の再考

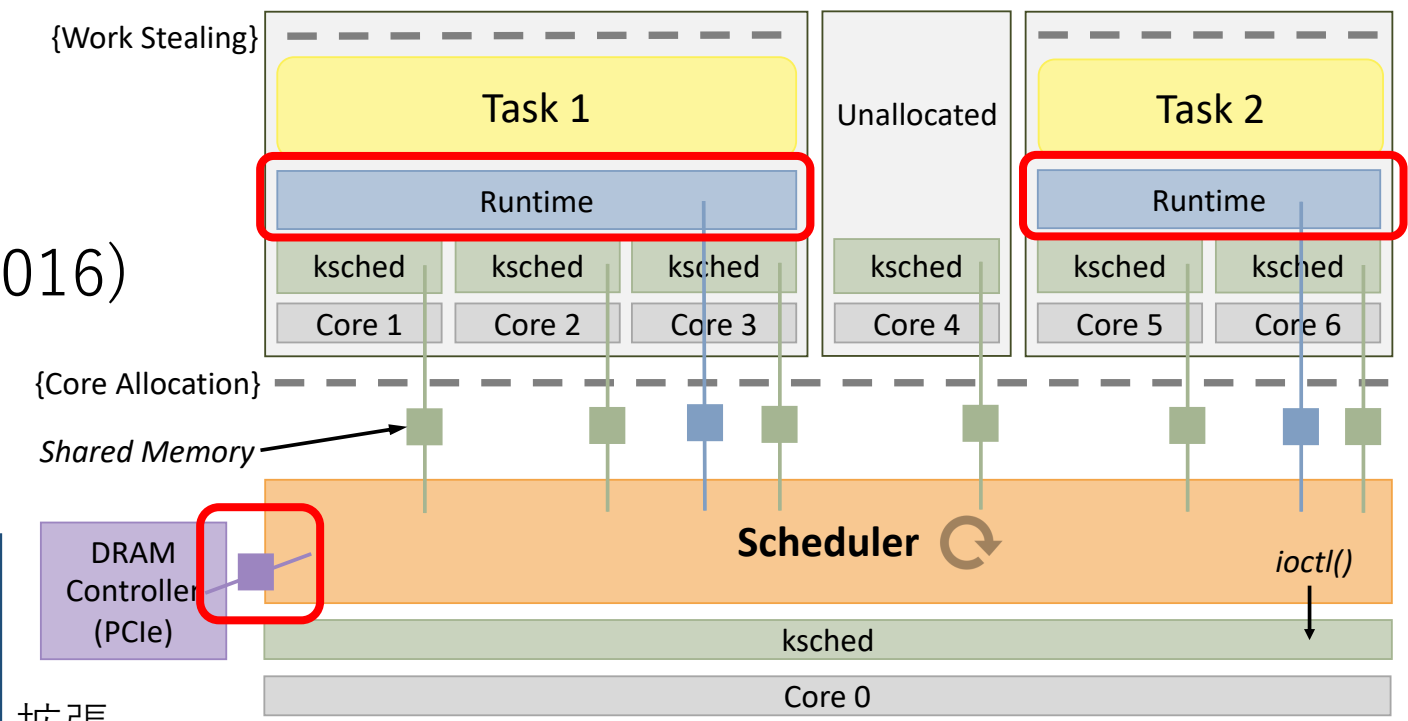
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)



拡張

TCP/IP スタック設計の再考

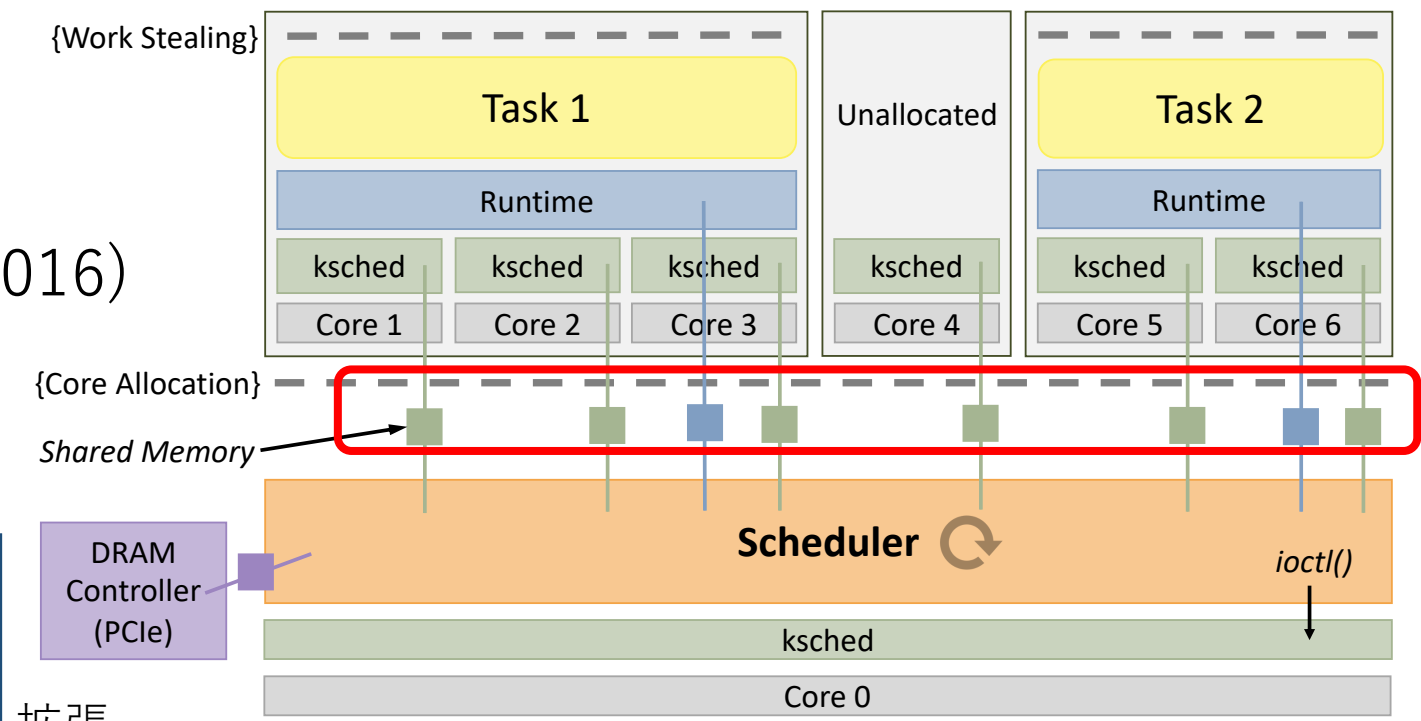
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)



Caladan scheduler が Caladan ランタイム環境と DRAM コントローラーのカウンタを通して干渉を検知

TCP/IP スタック設計の再考

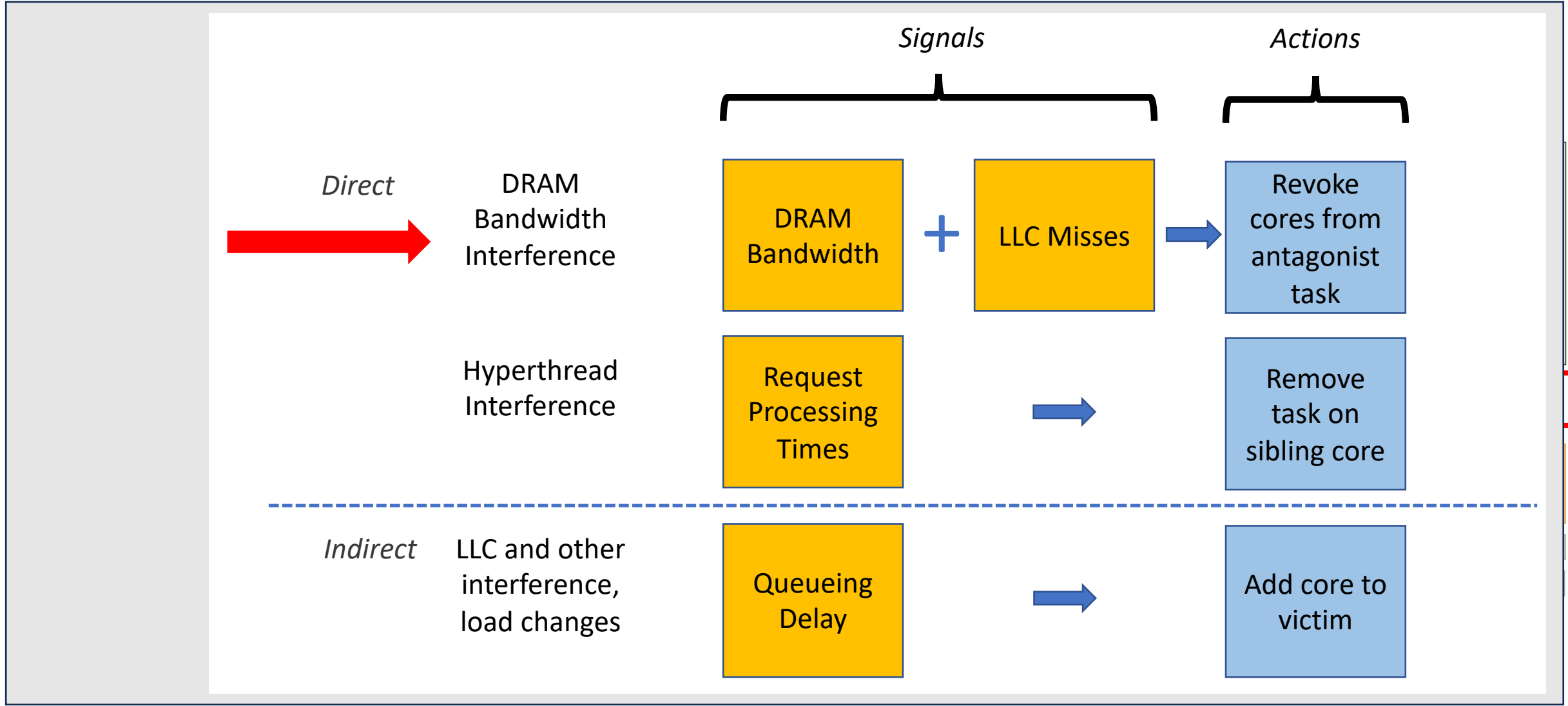
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)



拡張

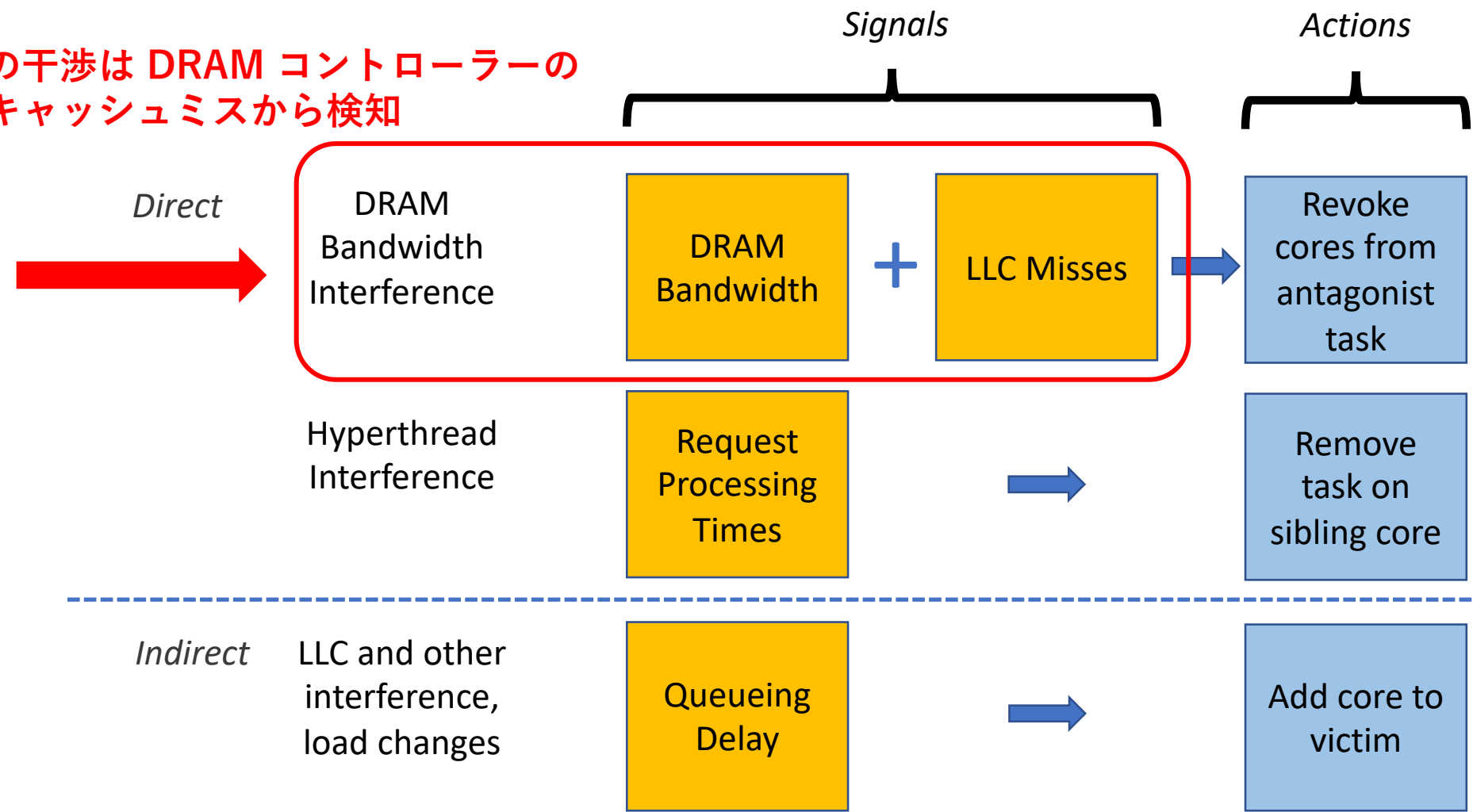
Caladan scheduler と Caladan ランタイム環境は共有メモリを通じて情報をやりとりする

TCP/IP スタック設計の再考

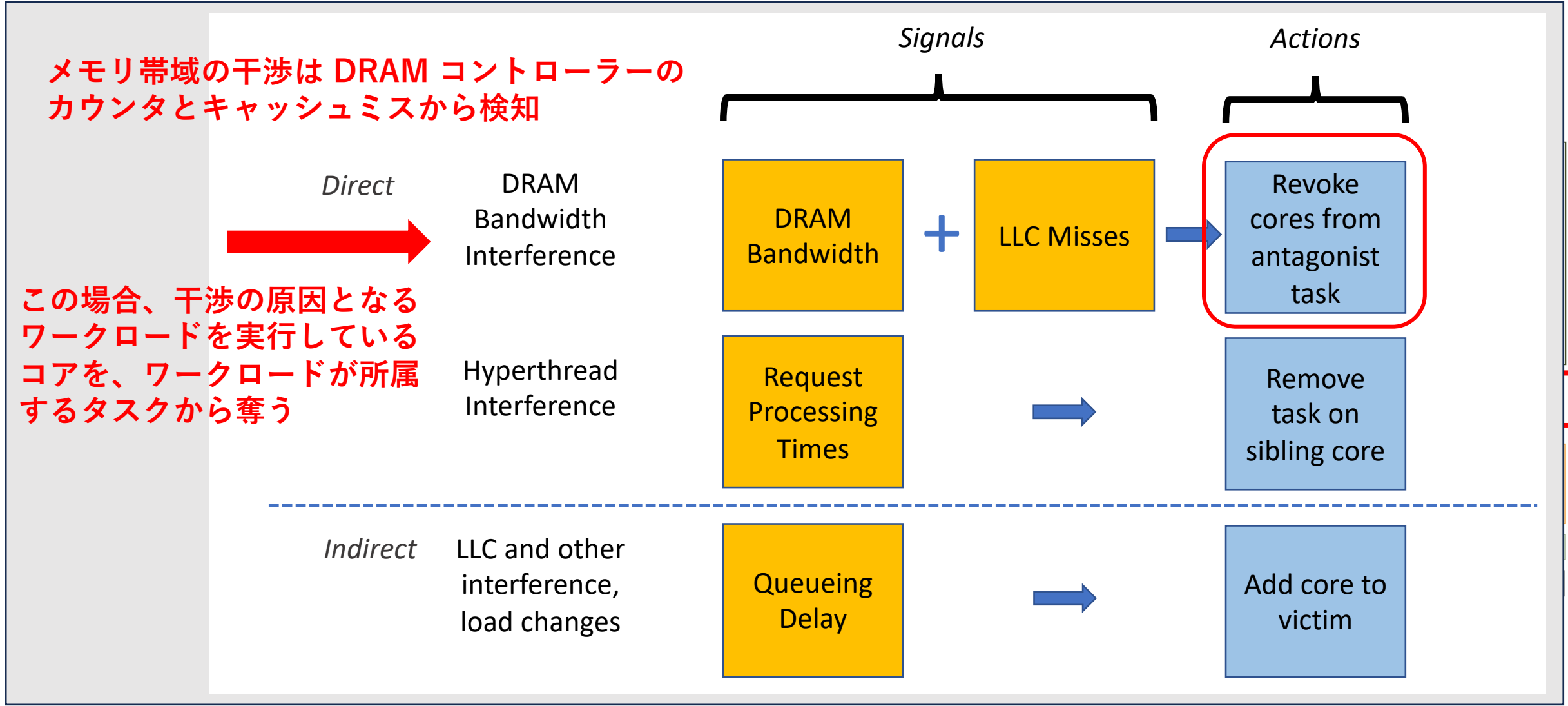


TCP/IP スタック設計の再考

メモリ帯域の干渉は DRAM コントローラーの
カウンタとキャッシュミスから検知

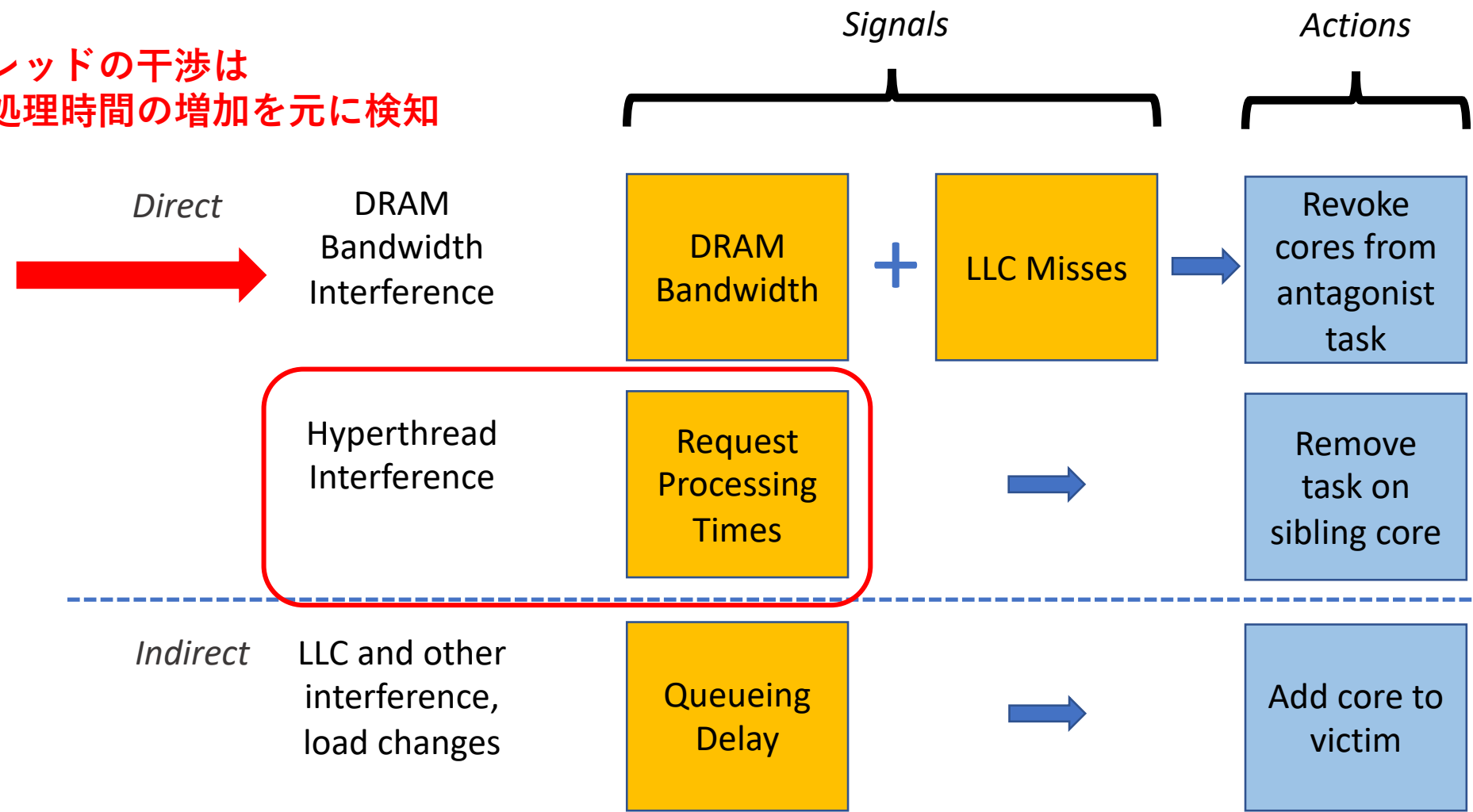


TCP/IP スタック設計の再考

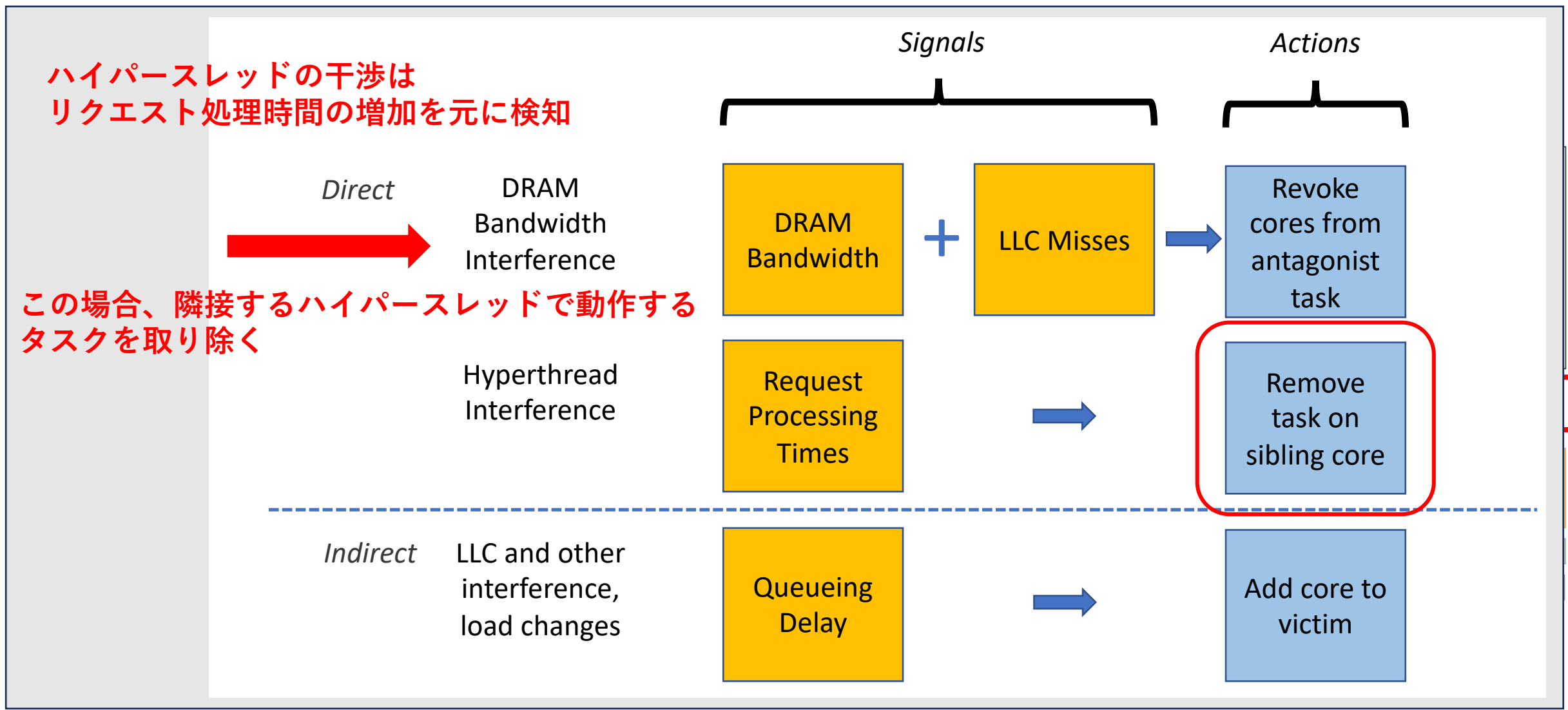


TCP/IP スタック設計の再考

ハイパースレッドの干渉は
リクエスト処理時間の増加を元に検知

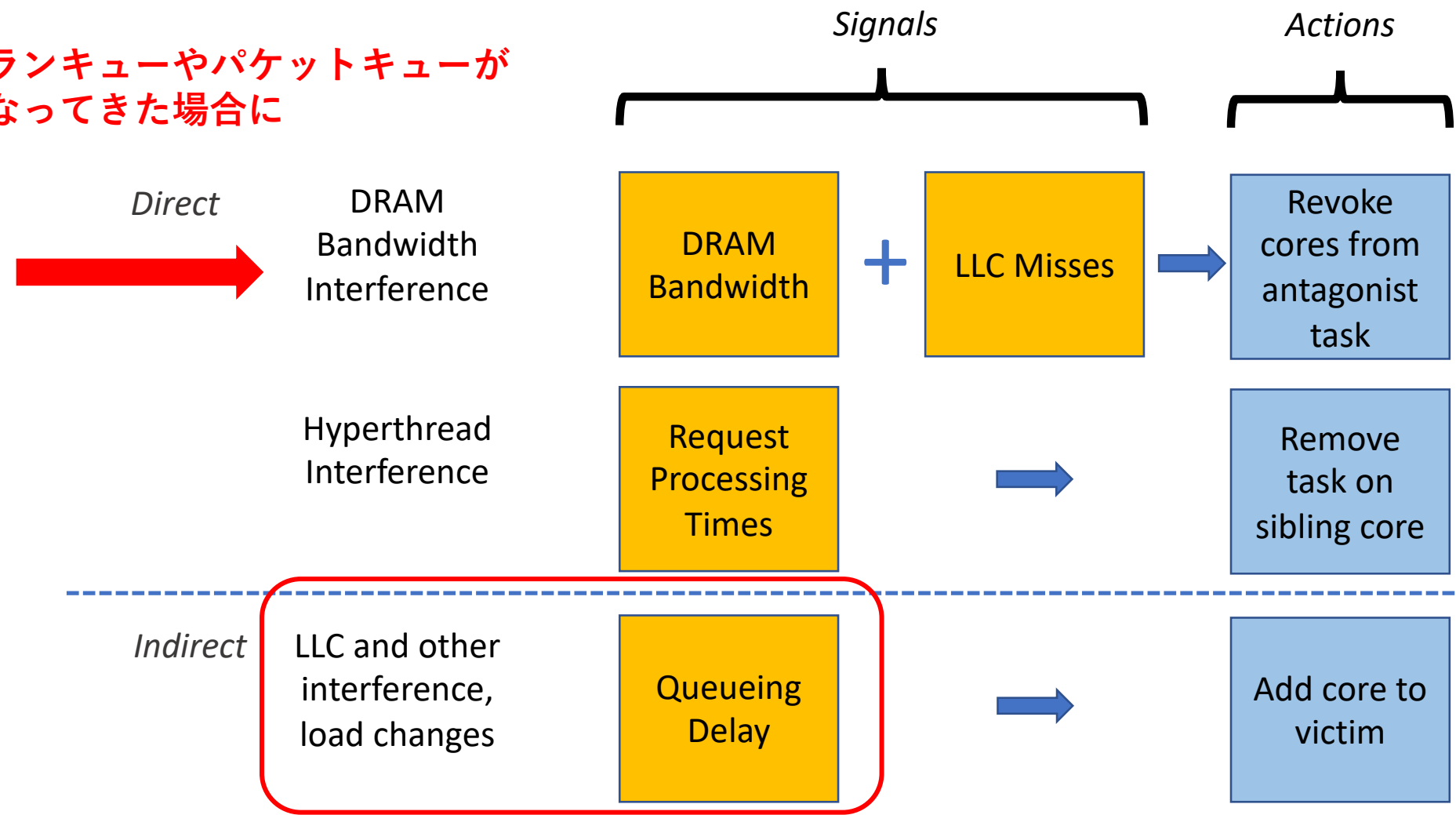


TCP/IP スタック設計の再考

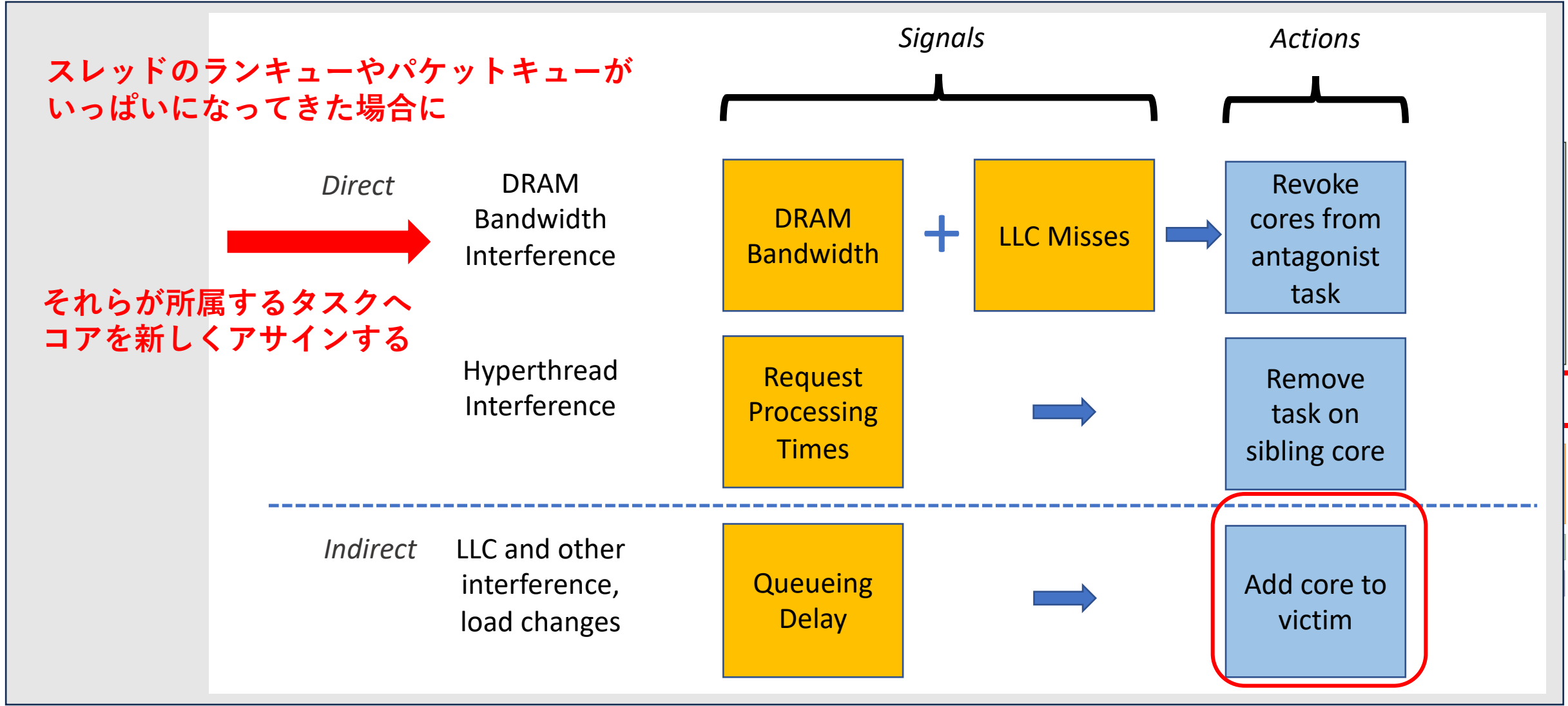


TCP/IP スタック設計の再考

スレッドのランキューやパケットキューがいっぱいになってきた場合に



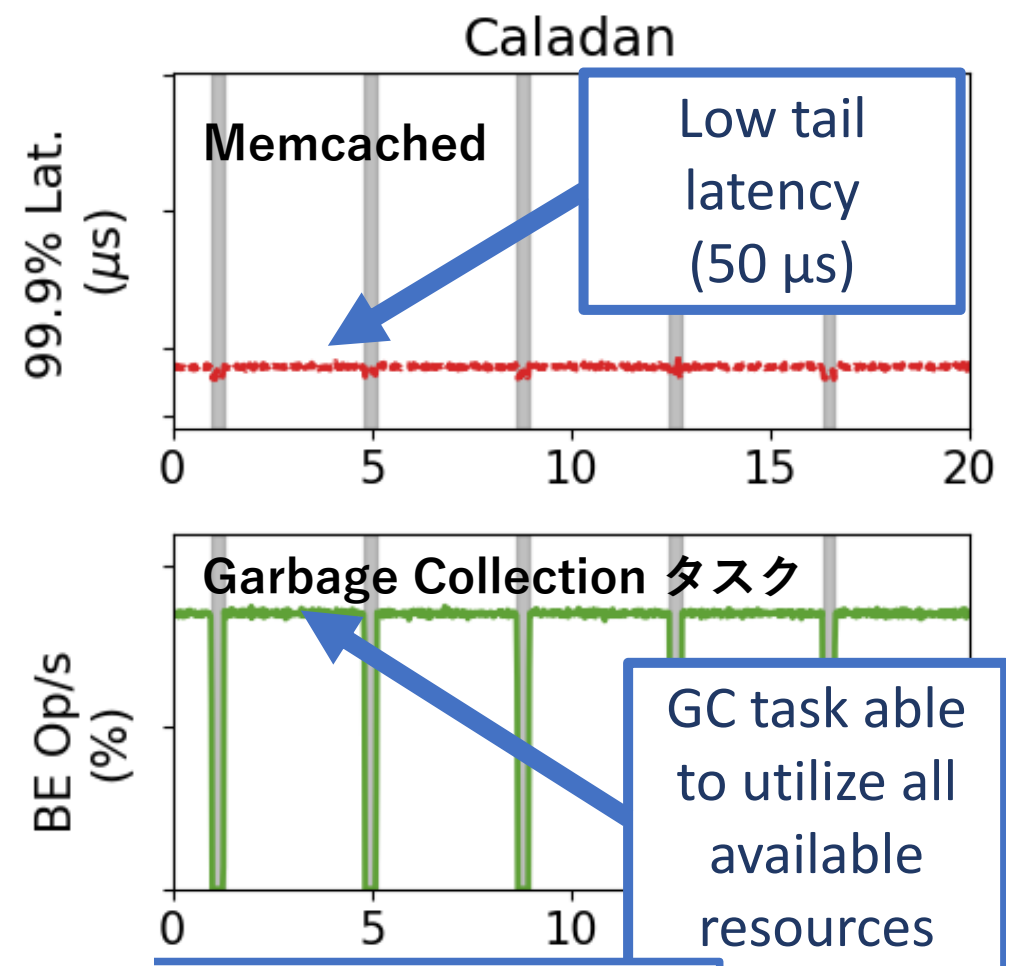
TCP/IP スタック設計の再考



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)

拡張

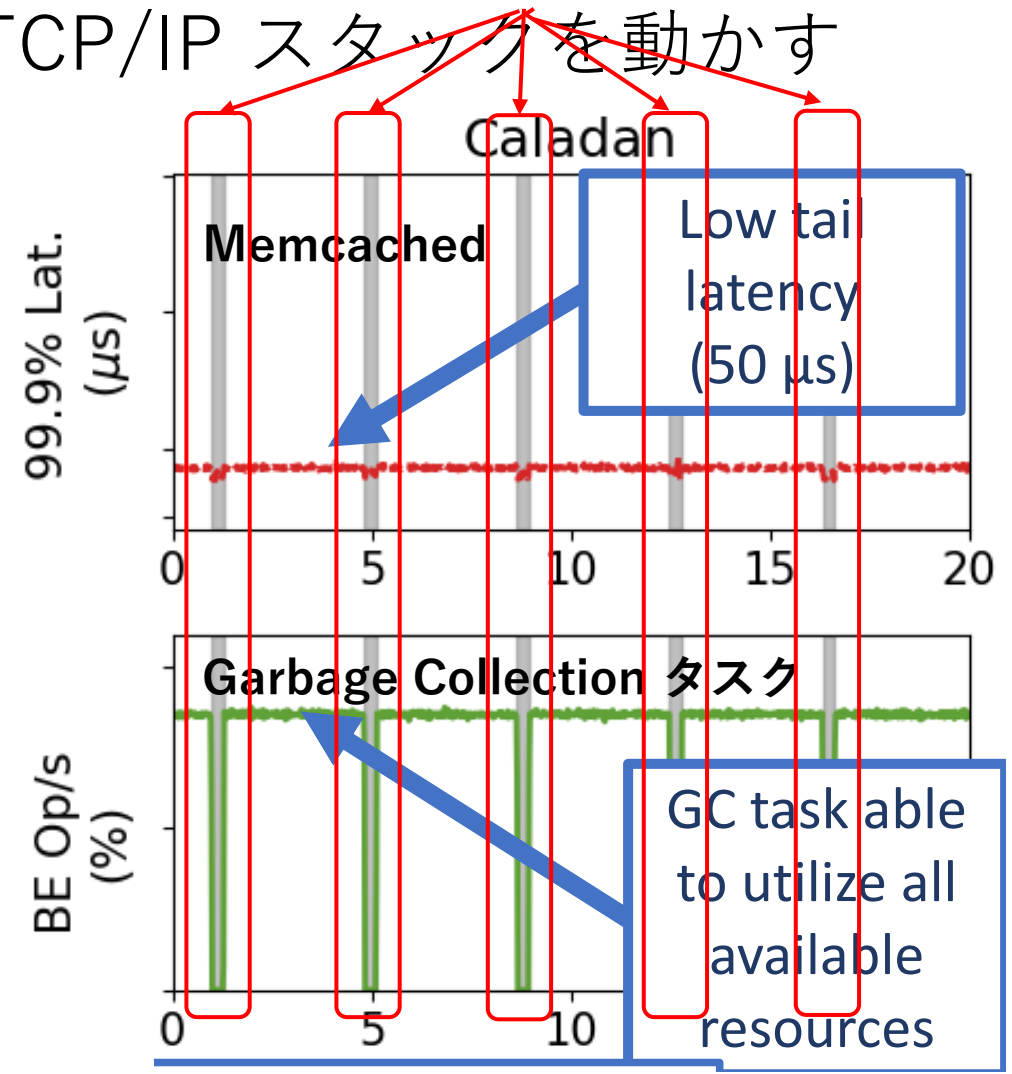


TCP/IP スタック設計の再考

灰色の箇所がGarbage Collection が実行されている時間

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)

拡張

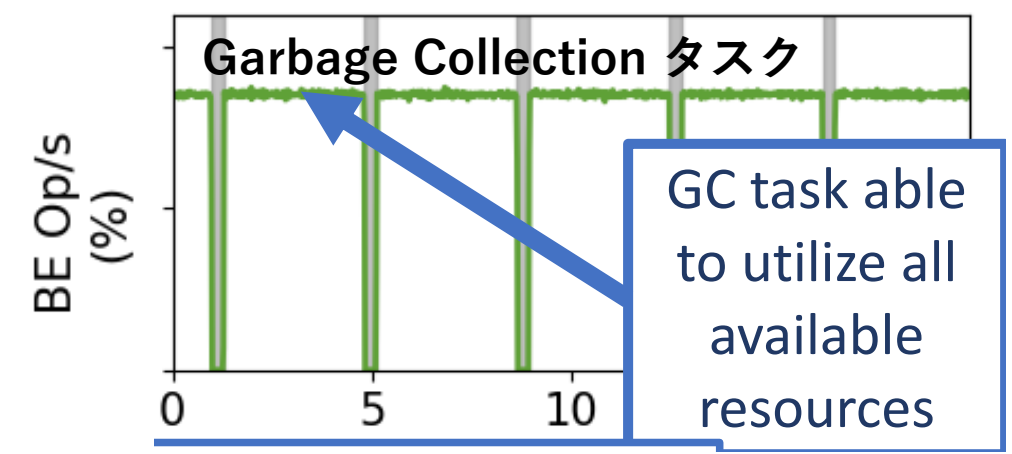
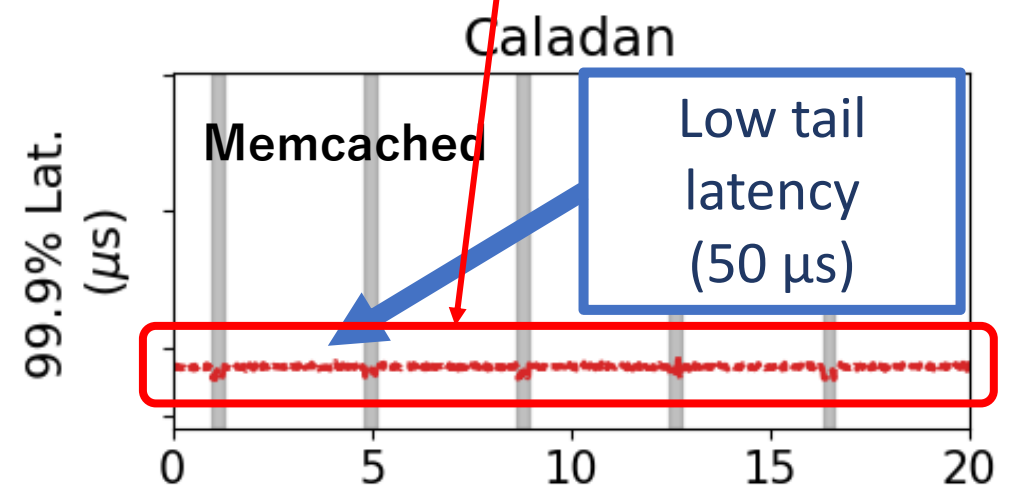


TCP/IP スタック設計の再考

Garbage Collection にかかわらず低い遅延を達成

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - **Caladan (OSDI 2020)**
 - Demikernel (SOSP 2021)

拡張



研究紹介

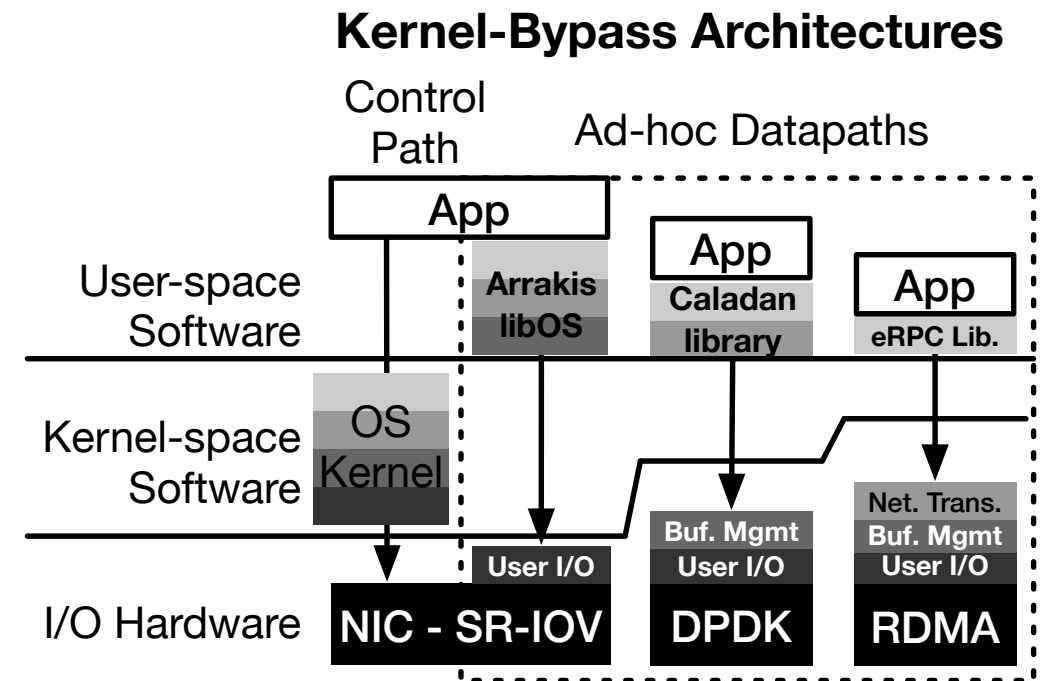
TCP/IP スタック設計

パケット I/O フレームワークを適用する

インターフェース設計

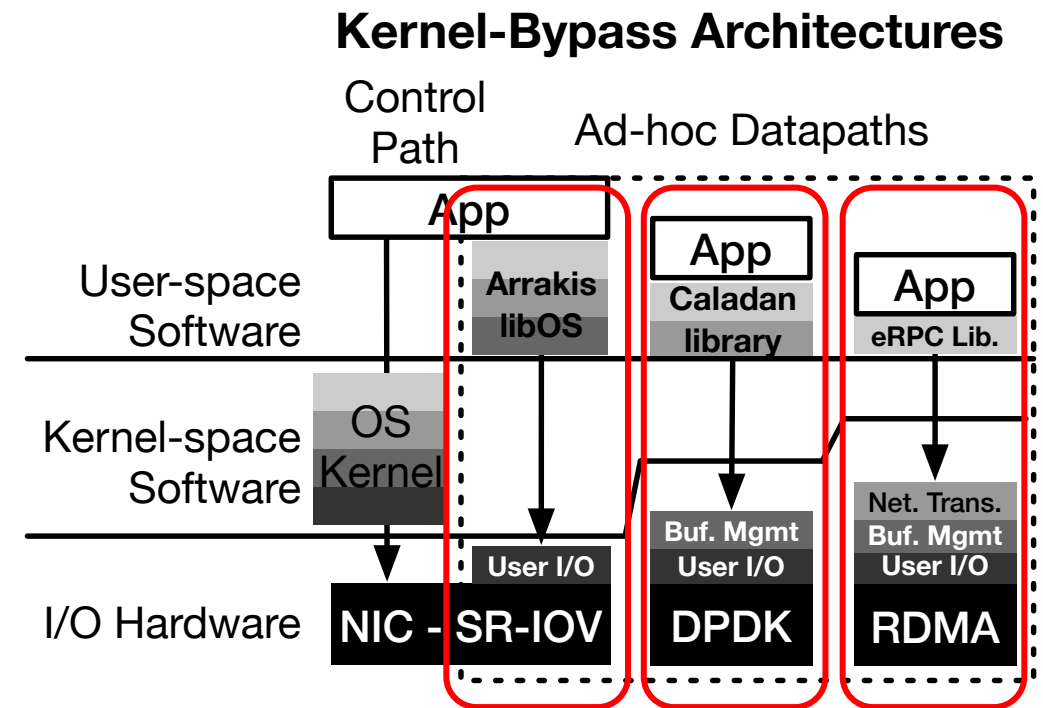
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**



TCP/IP スタック設計の再考

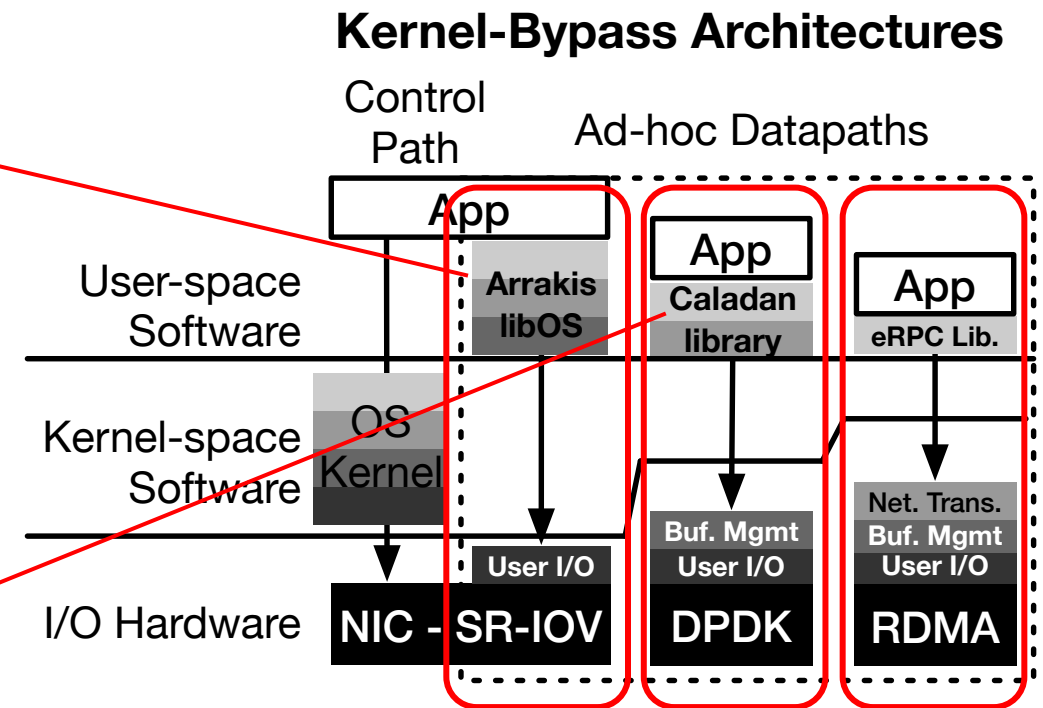
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**



様々なカーネルをバイパスするシステムが提案された

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**



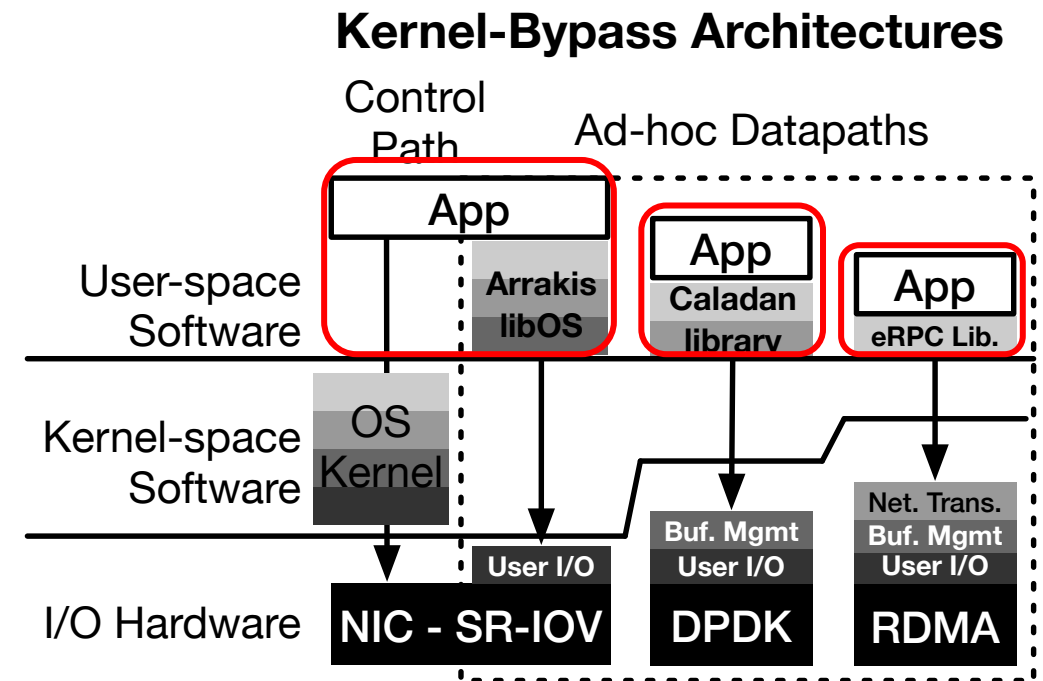
様々なカーネルをバイパスするシステムが提案された

TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす

- Sandstorm (SIGCOMM 2014)
- mTCP (NSDI 2014)
- Arrakis (OSDI 2014)
- IX (OSDI 2014)
- StackMap (USENIX ATC 2016)
- Atlas (SIGCOMM 2017)
- ZygOS (SOSP 2017)
- Shenango (NSDI 2019)
- Shinjuku (NSDI 2019)
- TAS (EuroSys 2019)
- Caladan (OSDI 2020)
- **Demikernel (SOSP 2021)**

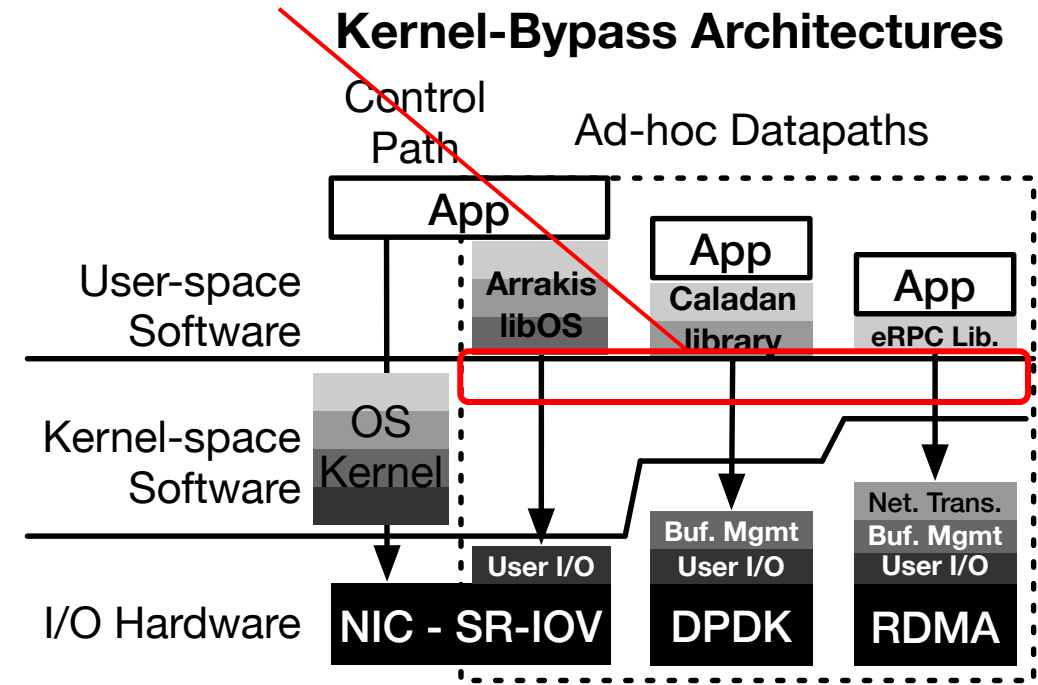
問題：アプリが利用を想定するデバイスに依存



様々なカーネルをバイパスするシステムが提案された

TCP/IP スタック設計の再考

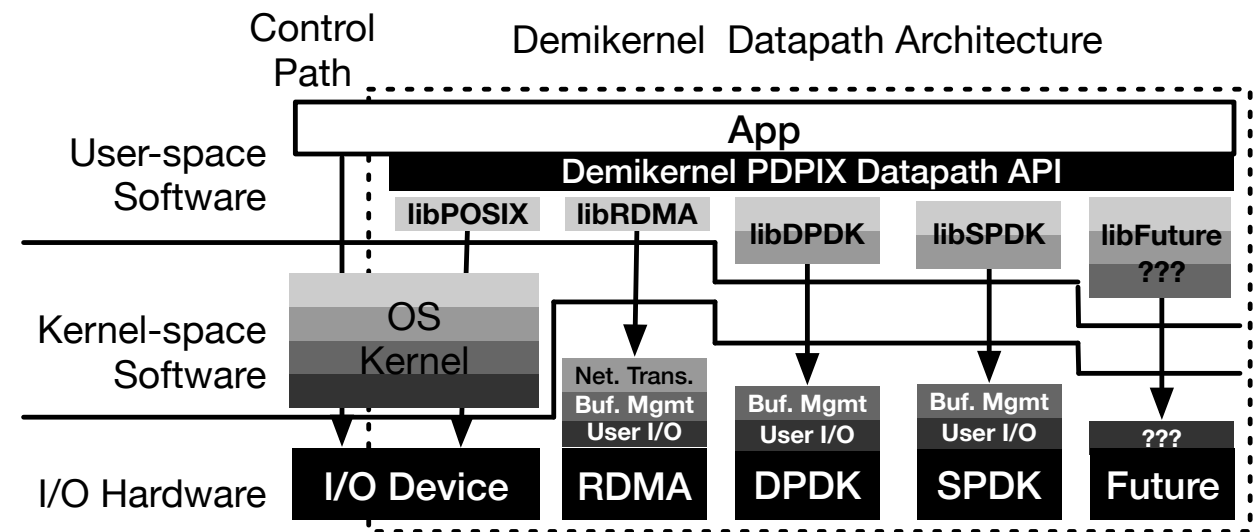
- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014) 問題：アプリが利用を想定するデバイスに依存
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**
- モチベーション：汎用的なインターフェースがあった方が良い



様々なカーネルをバイパスするシステムが提案された

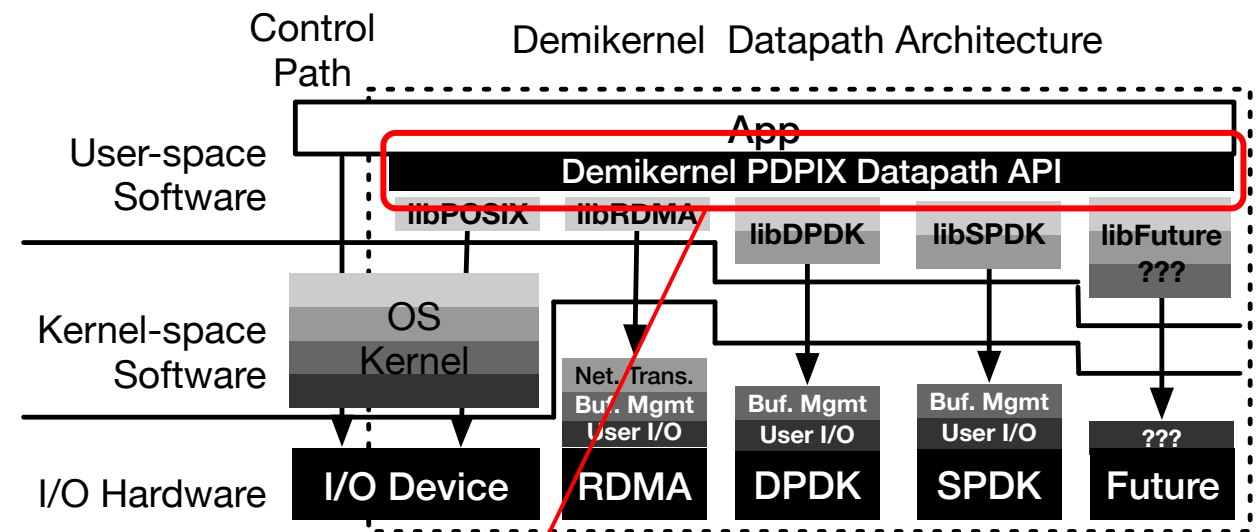
TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**



TCP/IP スタック設計の再考

- パケット I/O フレームワーク上で TCP/IP スタックを動かす
 - Sandstorm (SIGCOMM 2014)
 - mTCP (NSDI 2014)
 - Arrakis (OSDI 2014)
 - IX (OSDI 2014)
 - StackMap (USENIX ATC 2016)
 - Atlas (SIGCOMM 2017)
 - ZygOS (SOSP 2017)
 - Shenango (NSDI 2019)
 - Shinjuku (NSDI 2019)
 - TAS (EuroSys 2019)
 - Caladan (OSDI 2020)
 - **Demikernel (SOSP 2021)**



提案：異なるタイプのデバイスへ
共通のインターフェースからアクセス可能にする

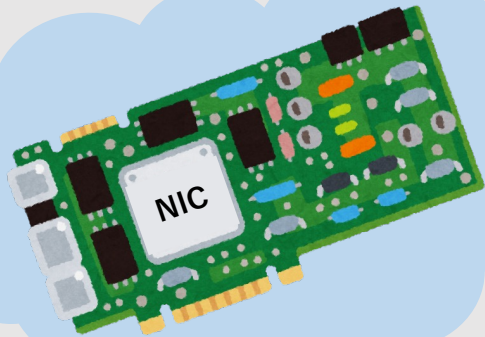
研究紹介

仮想マシン通信について

基本的な仕組みの説明

通信関連のシステムソフトウェア

データセンター内のサーバー



10 ~ Gbps



ハードウェア (NIC) は速くなったので
ソフトウェアの効率が重要になる

通信関連ソフトウェア

ユーザー空間

アプリケーション

仮想マシン

カーネル

TCP/IP スタック

NIC デバイスドライバ

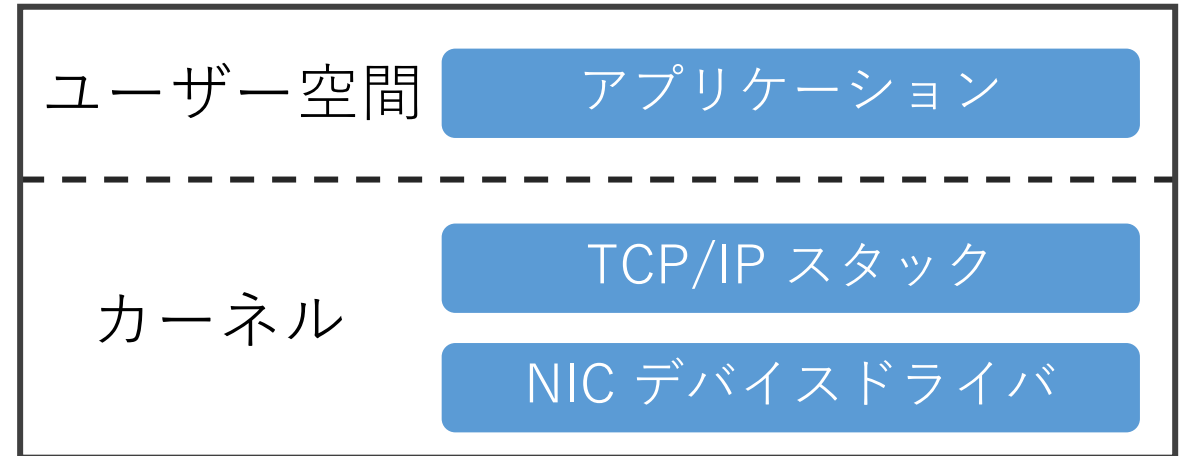
ホスト

仮想 NIC バックエンド

仮想スイッチ

NIC デバイスドライバ

仮想マシン環境



仮想マシン環境

これまでの話はこの辺りの改善

ユーザー空間

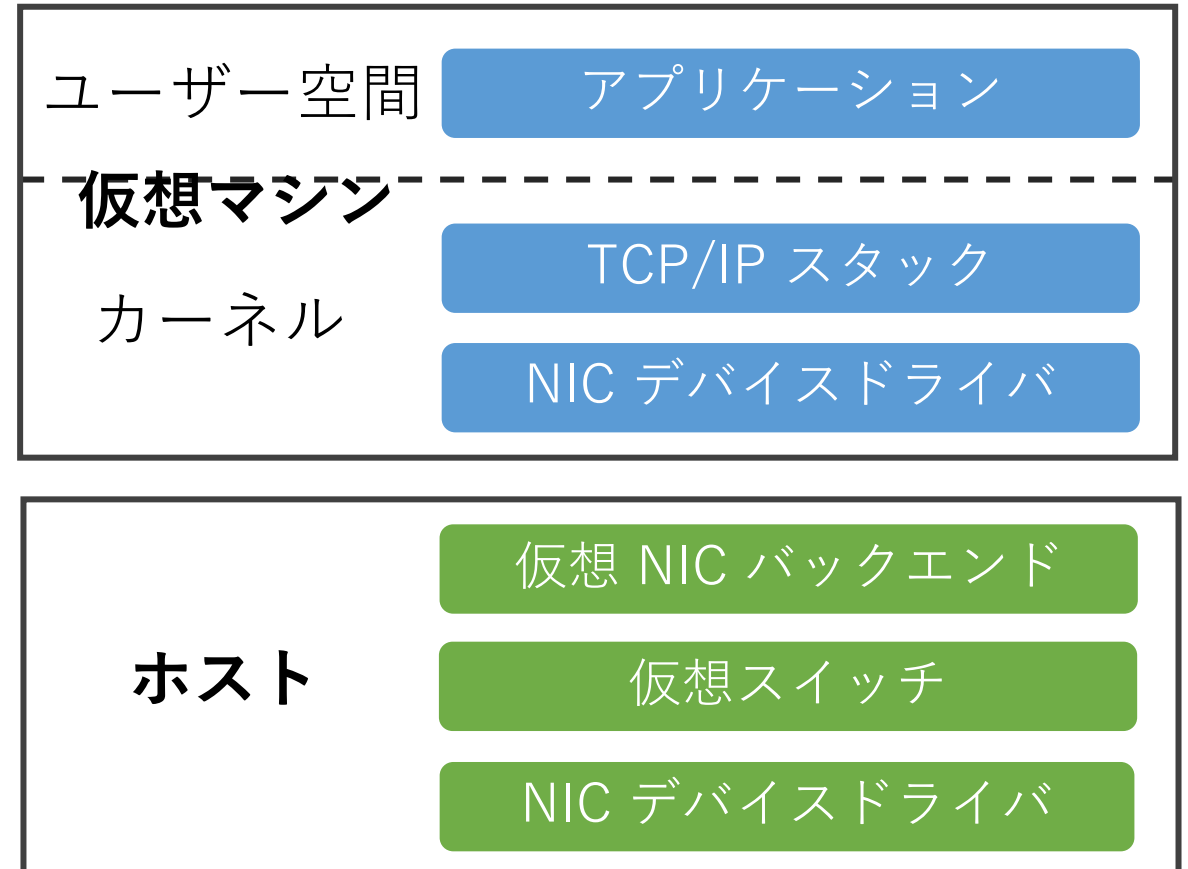
アプリケーション

カーネル

TCP/IP スタック

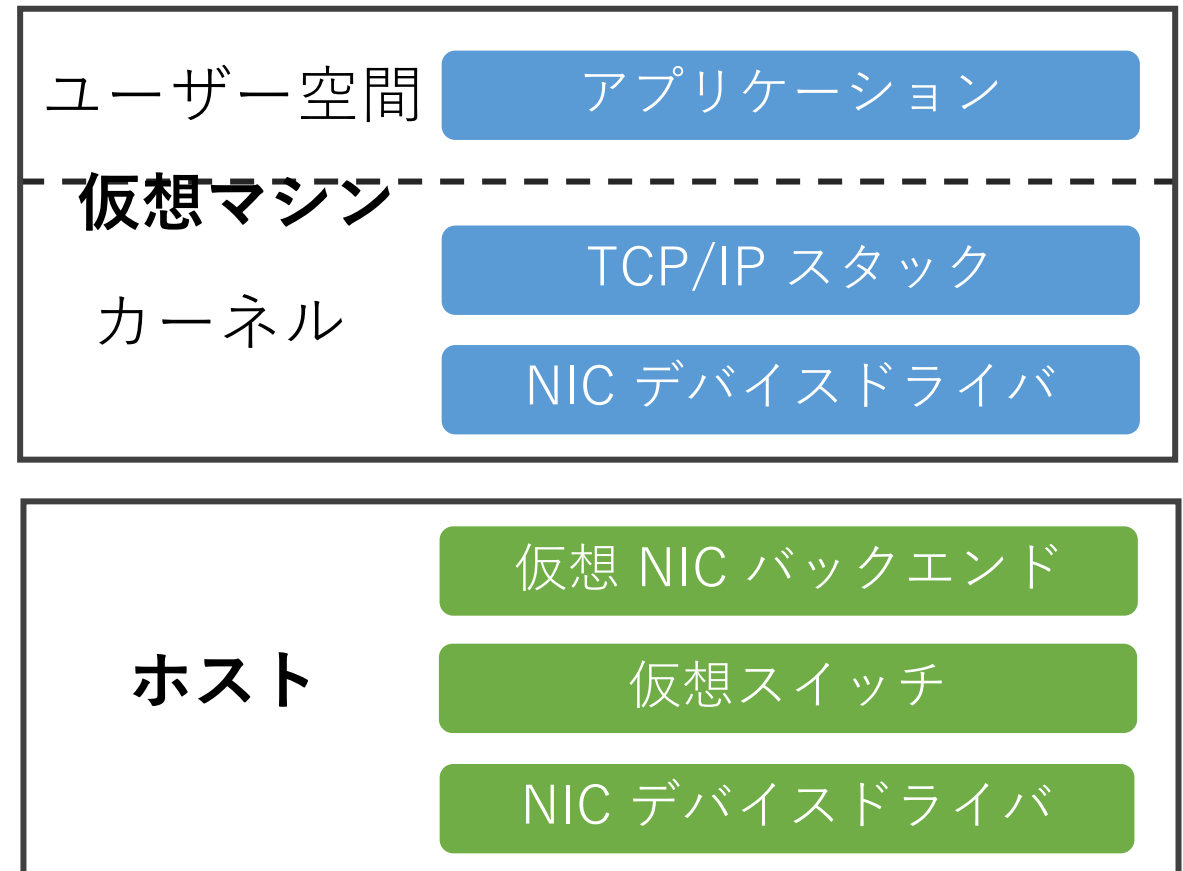
NIC デバイスドライバ

仮想マシン環境



仮想マシン環境

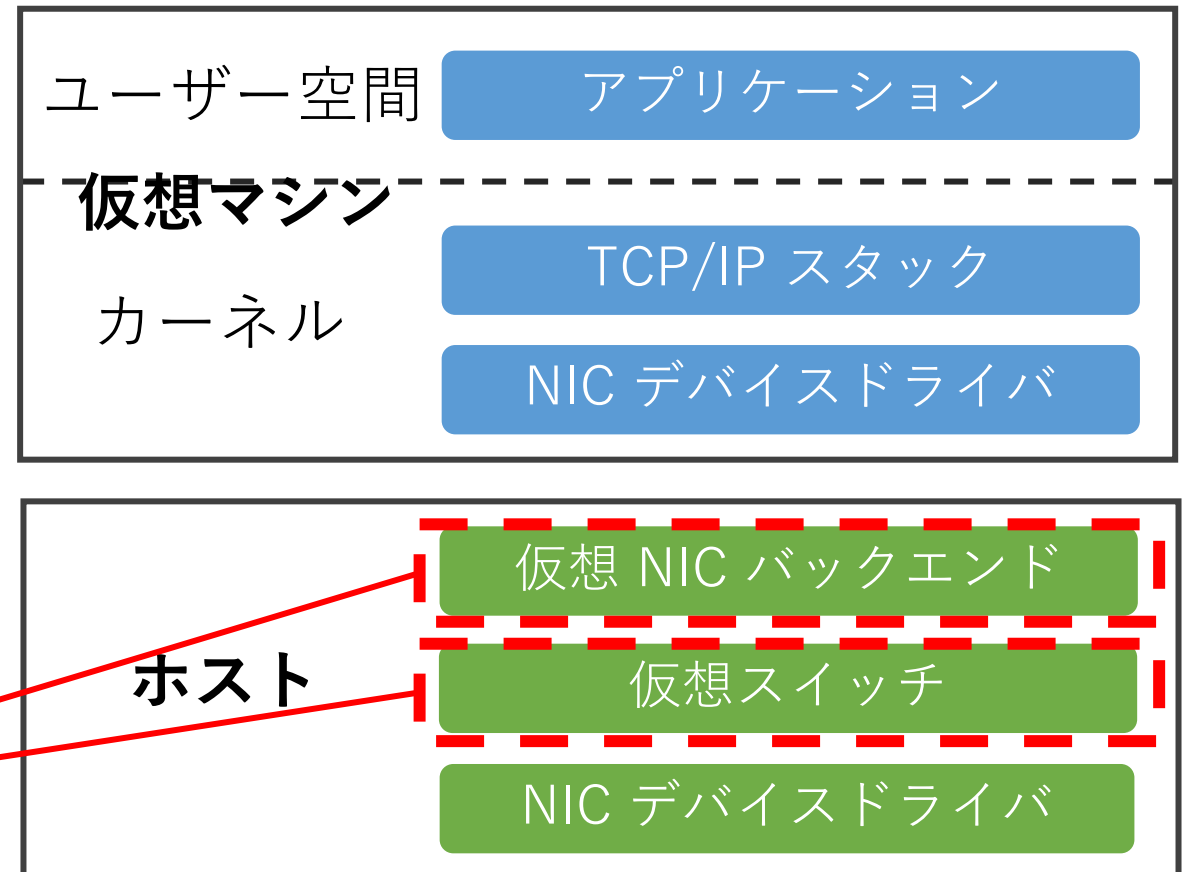
仮想マシン環境ではバックエンドも速くないと性能が制限されてしまう



仮想マシン環境

仮想マシン環境ではバックエンドも速くないと性能が制限されてしまう

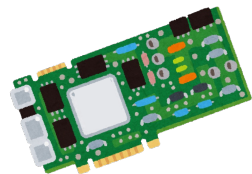
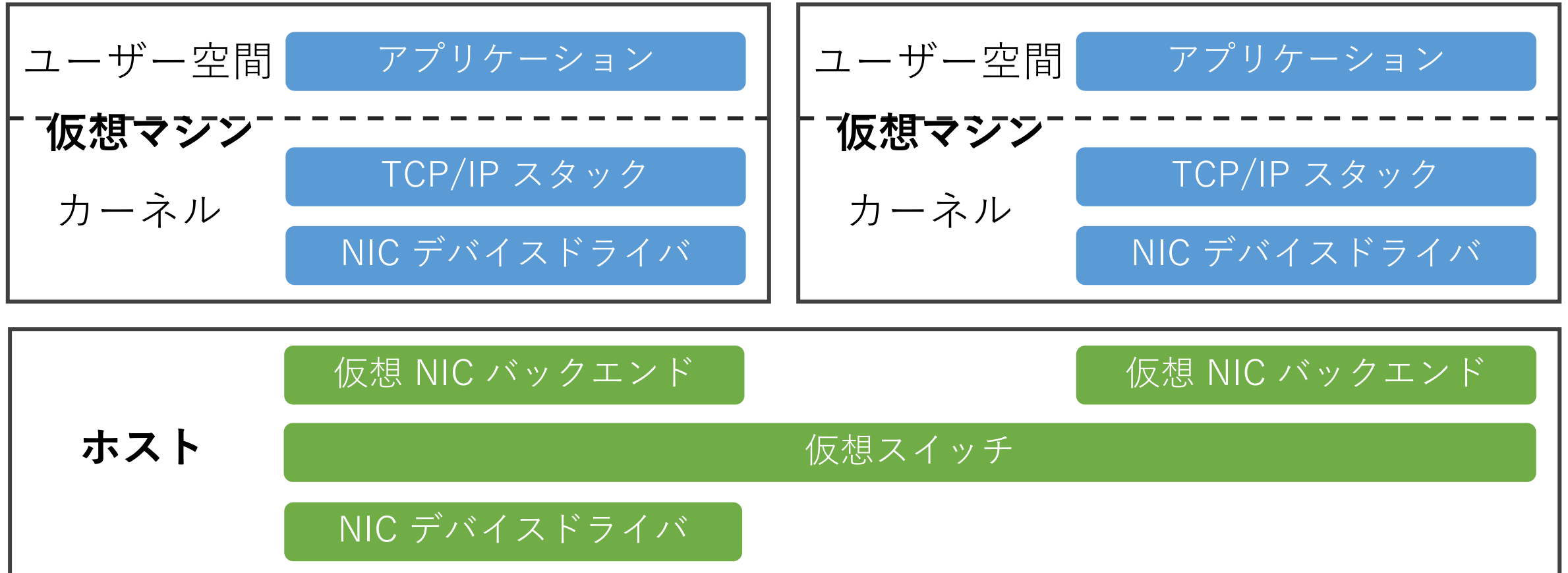
高速化ポイント



仮想スイッチ

なぜ必要？

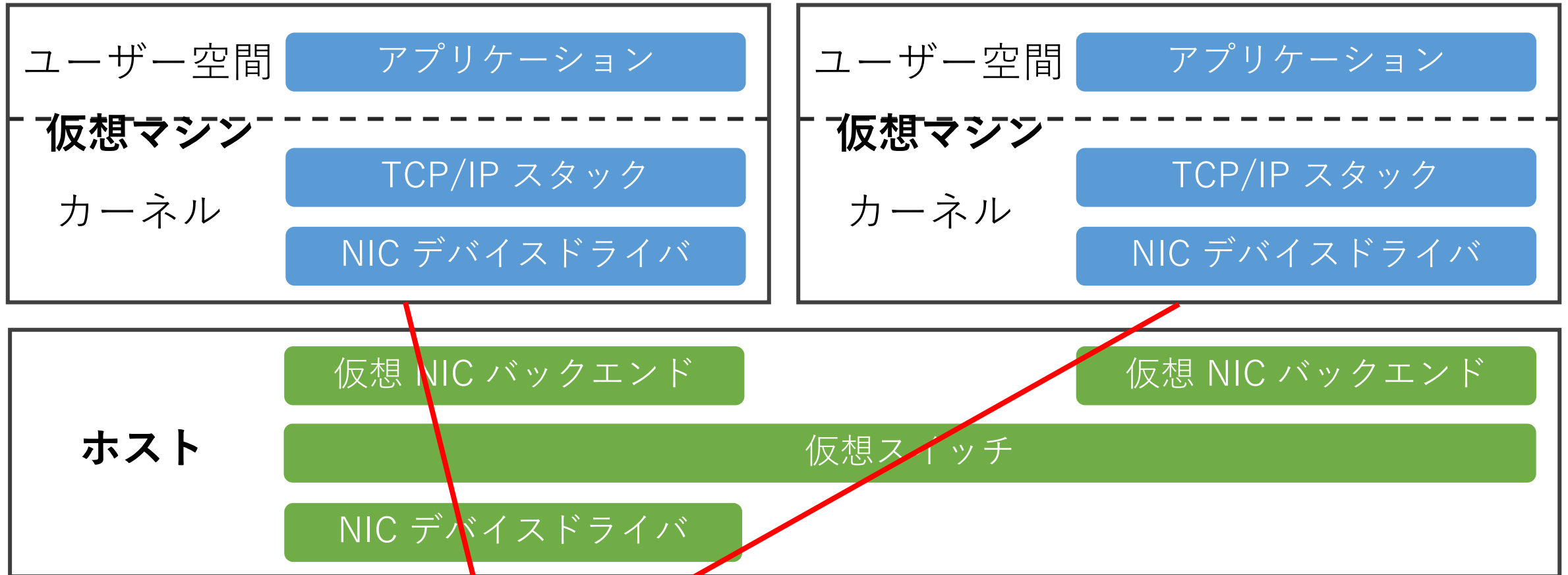
物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



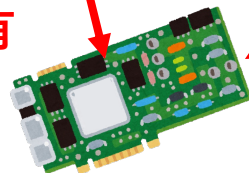
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



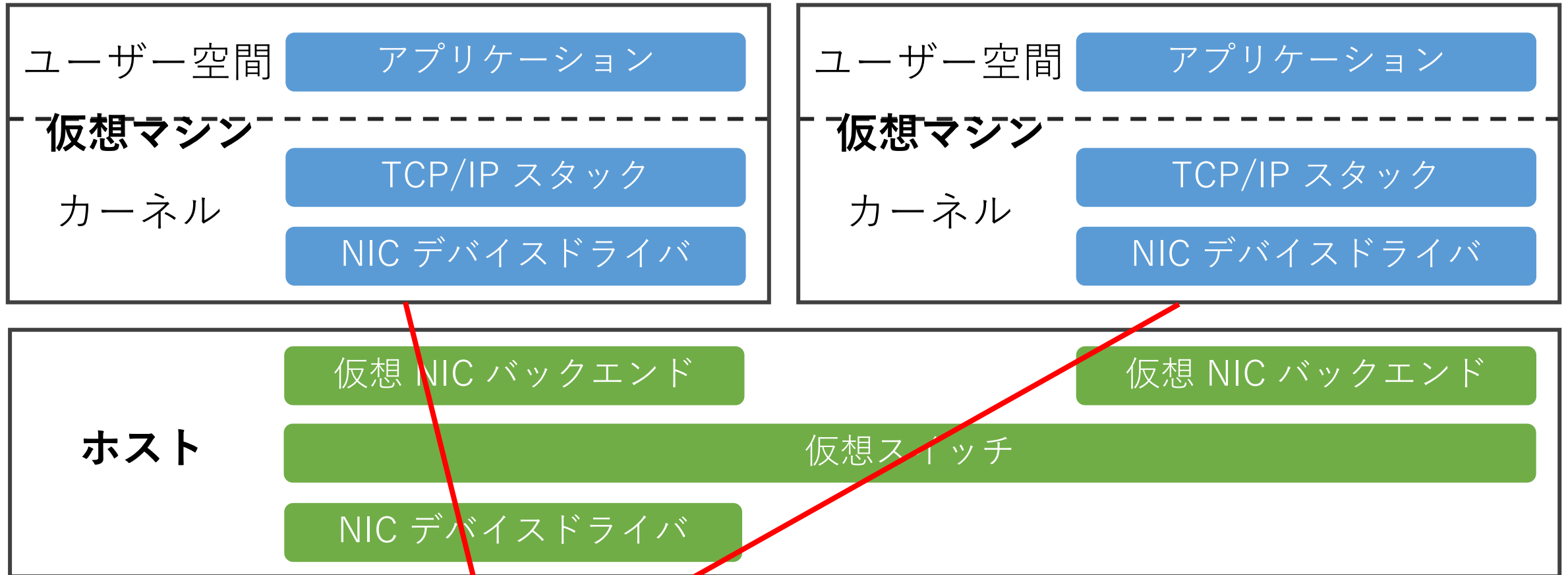
共有



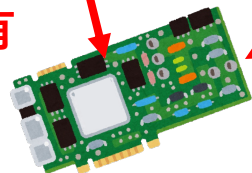
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



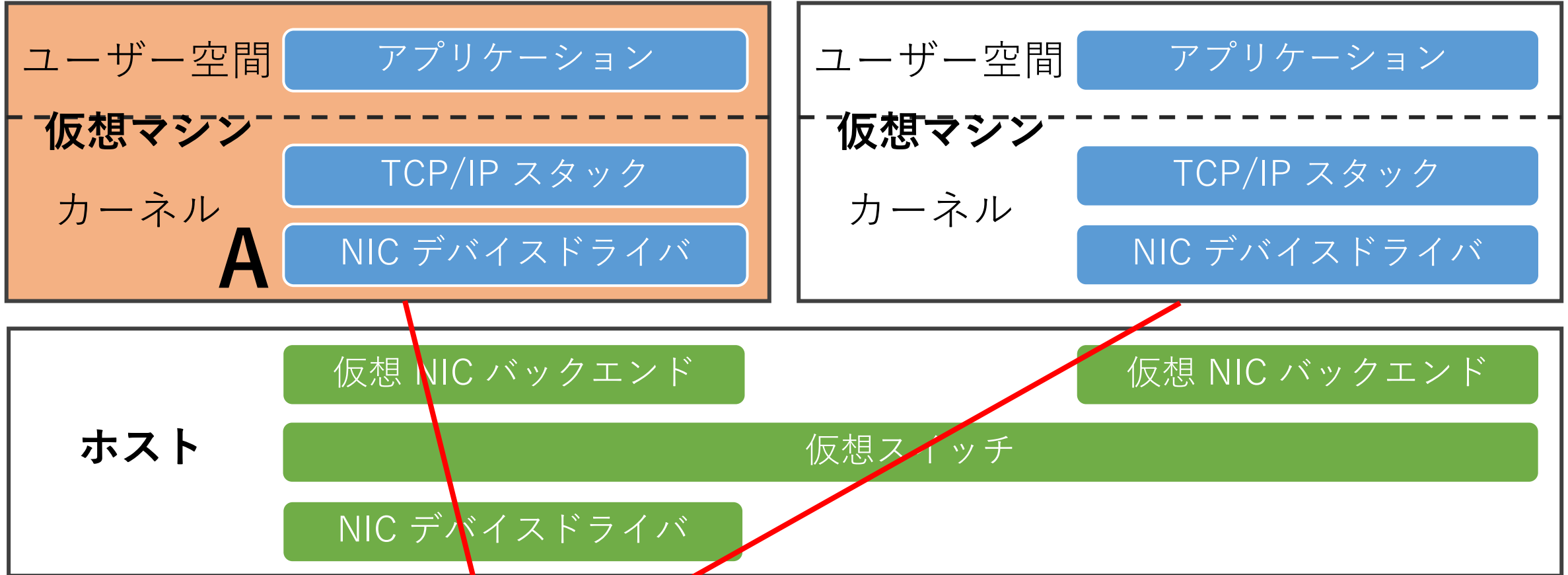
共有



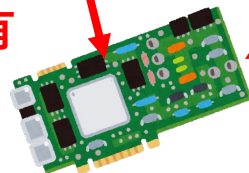
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



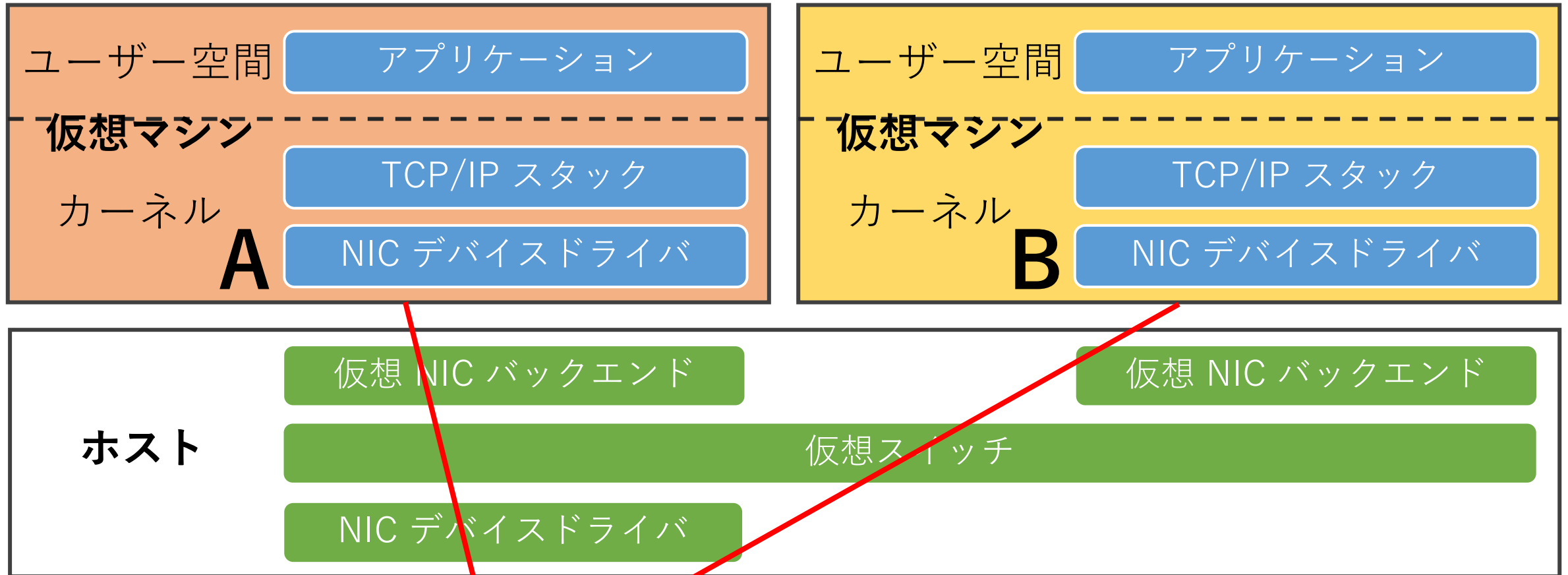
共有



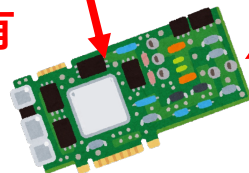
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



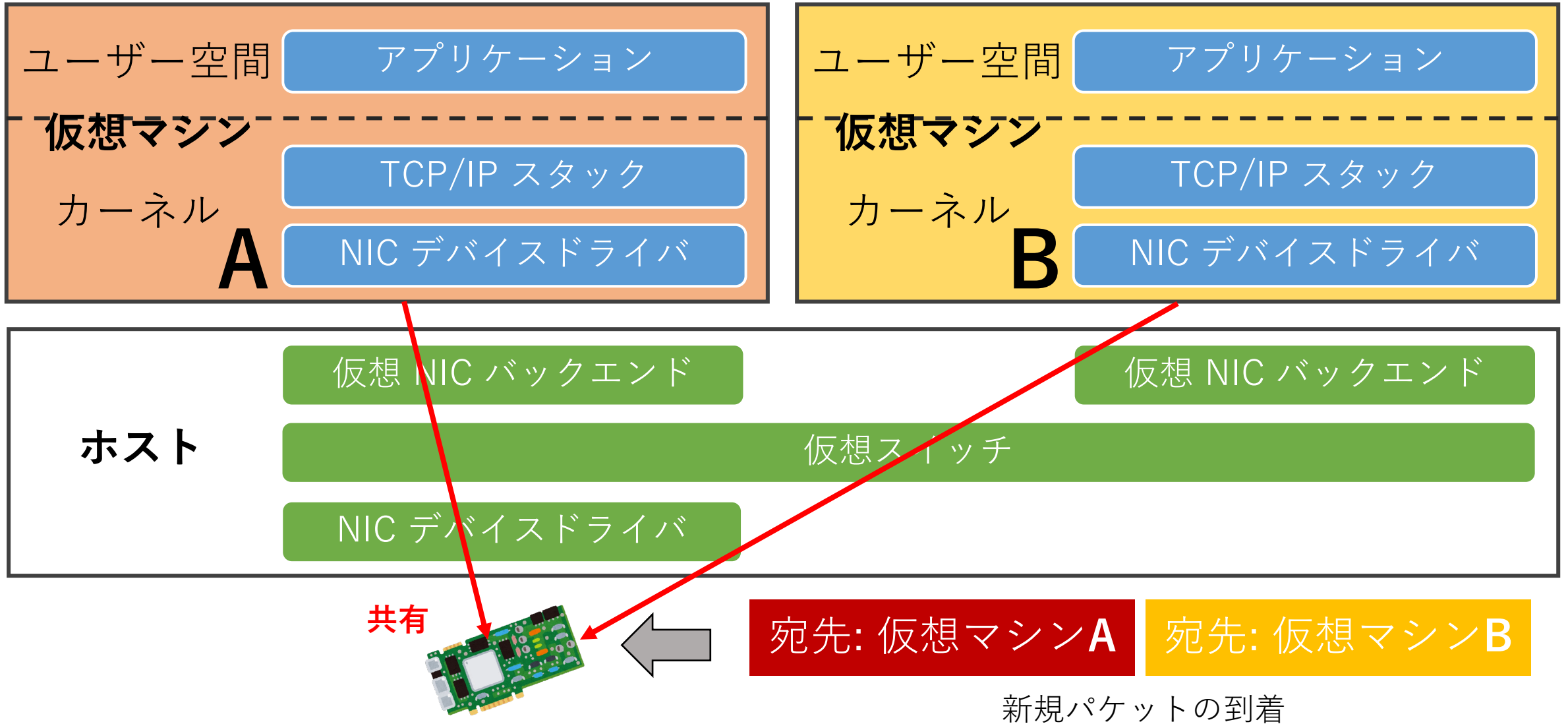
共有



仮想スイッチ

なぜ必要？

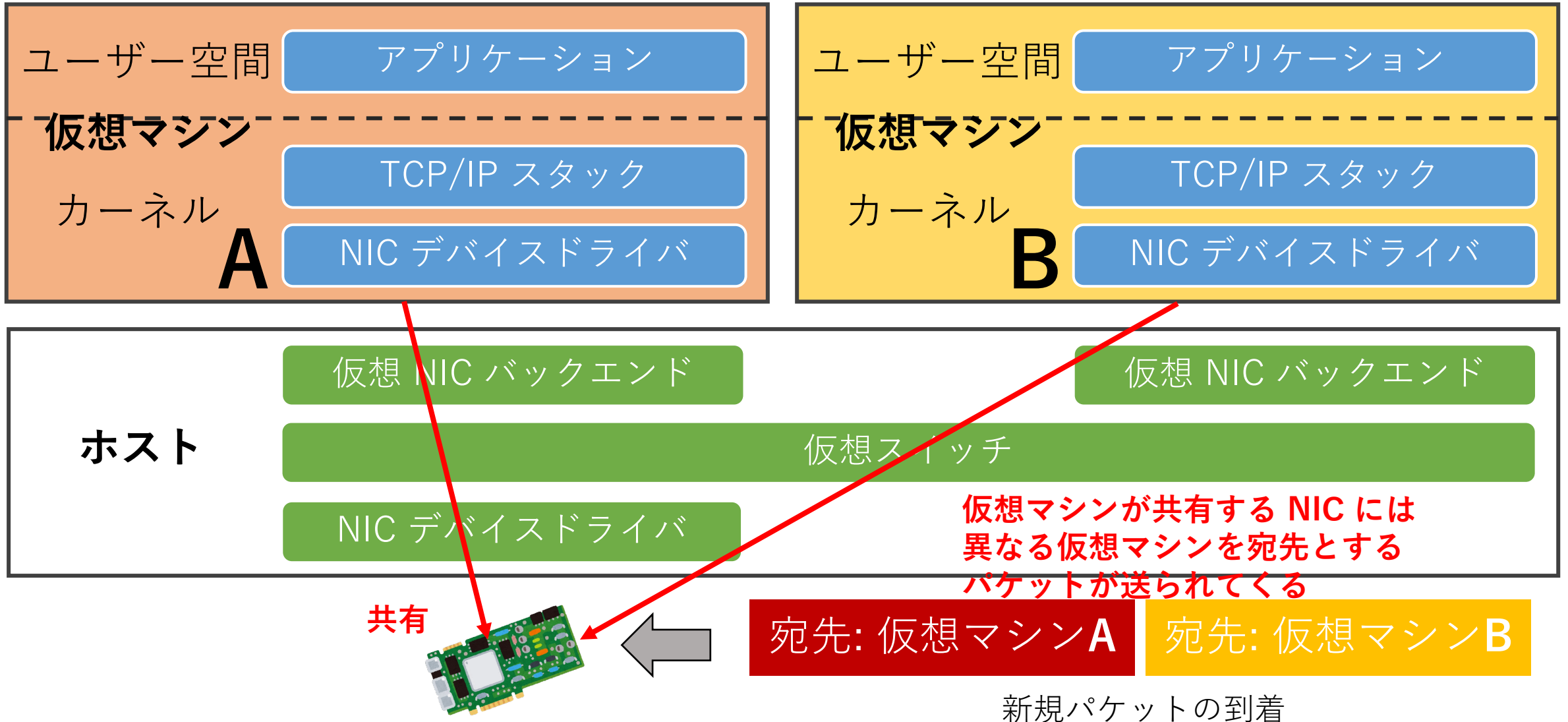
物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



仮想スイッチ

なぜ必要？

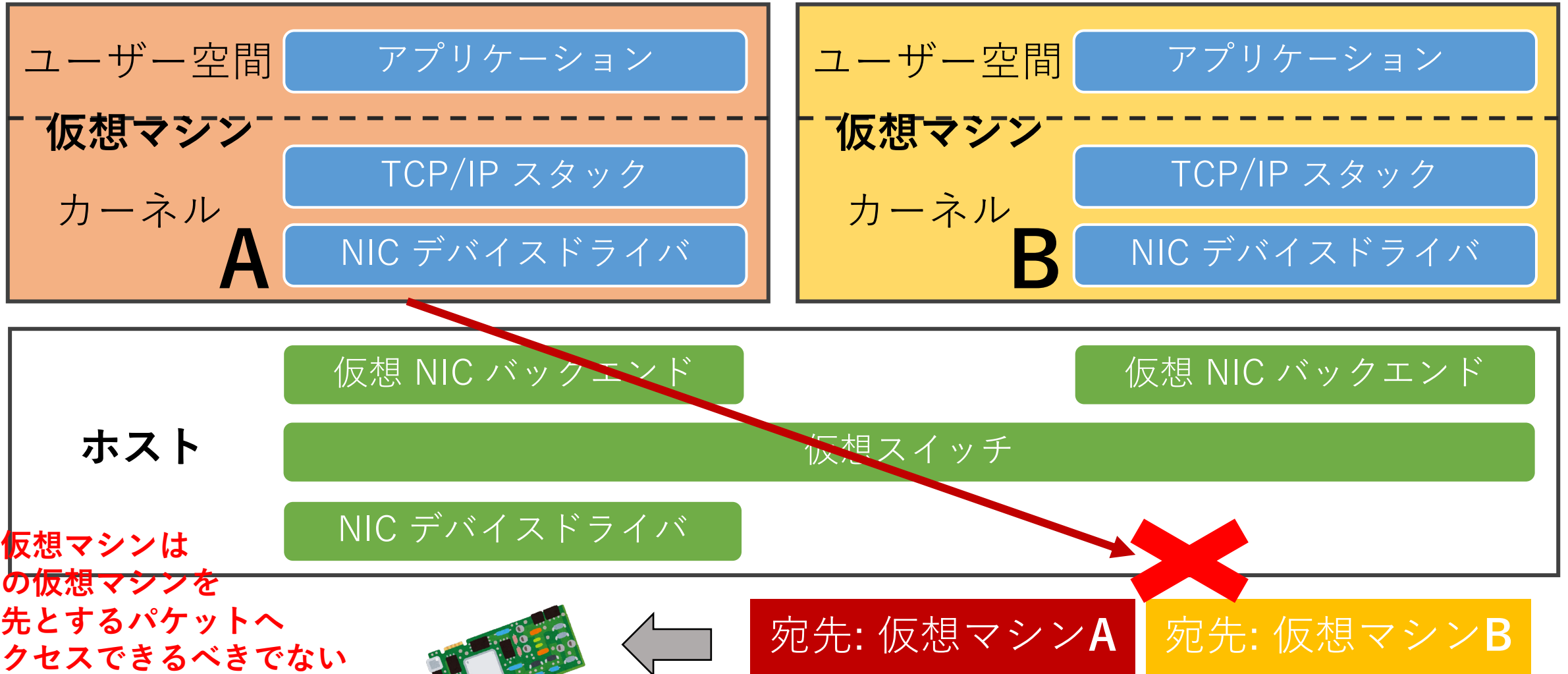
物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



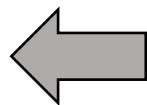
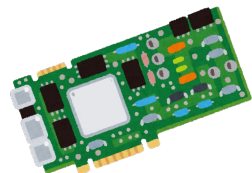
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



各仮想マシンは別の仮想マシンを宛先とするパケットへアクセスできるべきでない



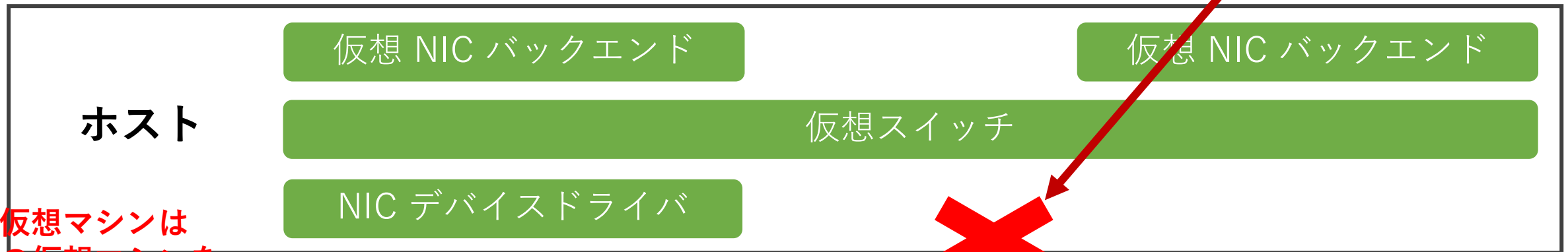
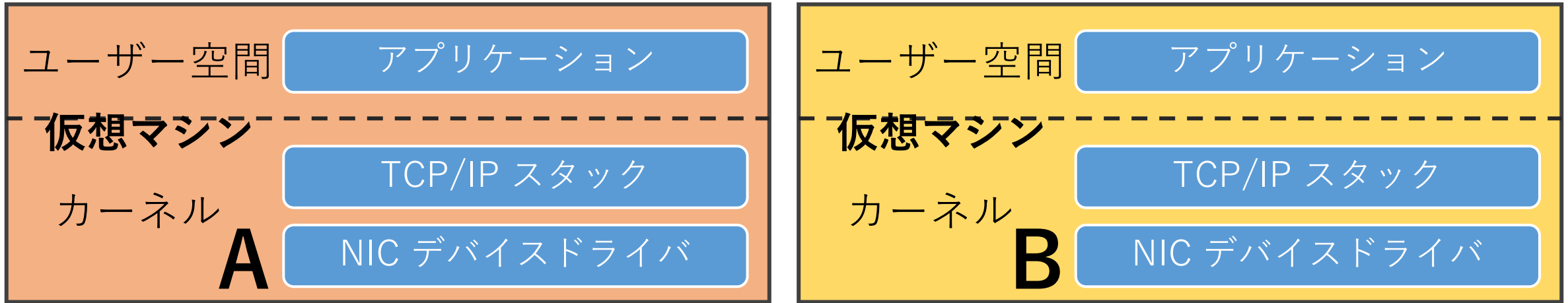
宛先: 仮想マシンA 宛先: 仮想マシンB

新規パケットの到着

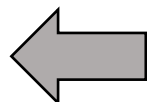
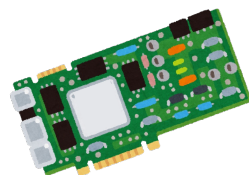
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



各仮想マシンは別の仮想マシンを宛先とするパケットへアクセスできるべきでない

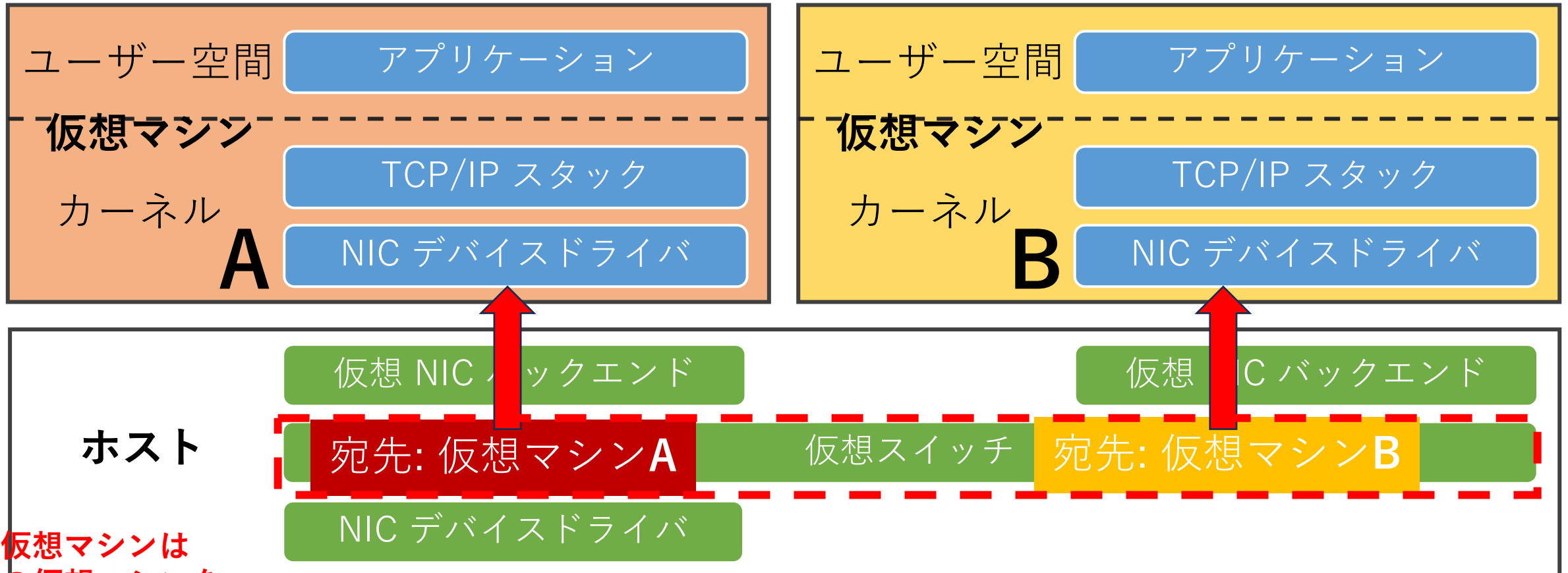


新規パケットの到着

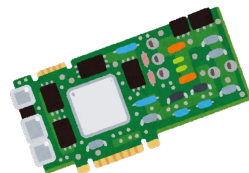
仮想スイッチ

なぜ必要？

物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



各仮想マシンは別の仮想マシンを宛先とするパケットへアクセスできるべきでない

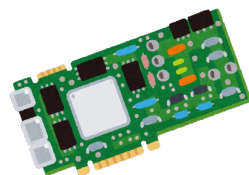
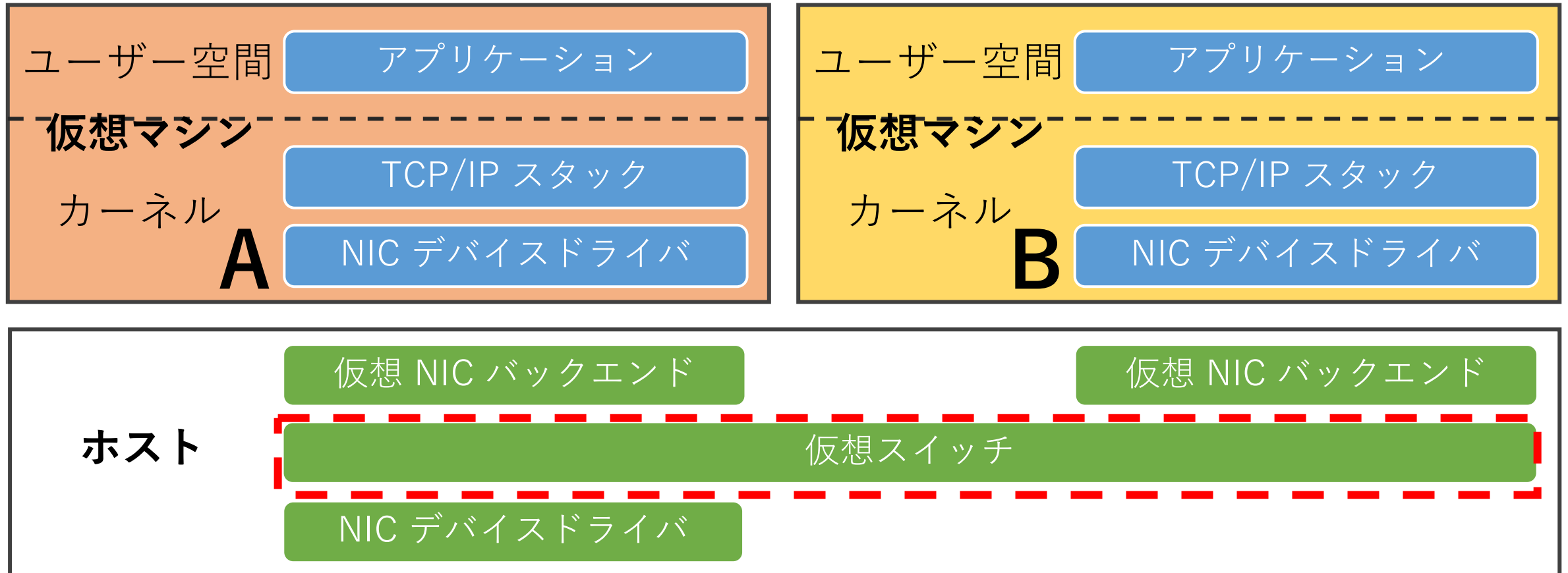


仮想スイッチがパケットの宛先 (MAC アドレス) を元に仮想マシンへパケットをフォワードするようにすることで各仮想マシンは自分へ宛てられたパケット以外見えなくできる

仮想スイッチ

なぜ必要？

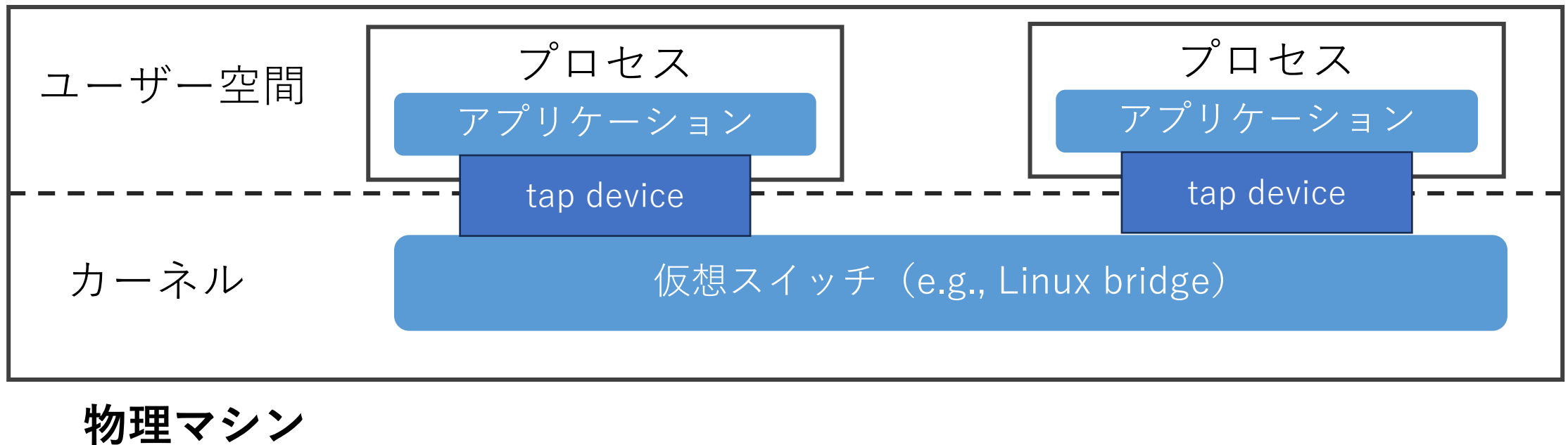
物理 NIC を複数の仮想マシンで**分離**を維持しつつ**共有**するため



仮想スイッチの処理は仮想マシン通信において頻繁に実行されるため、高い性能を発揮するためには効率が重要

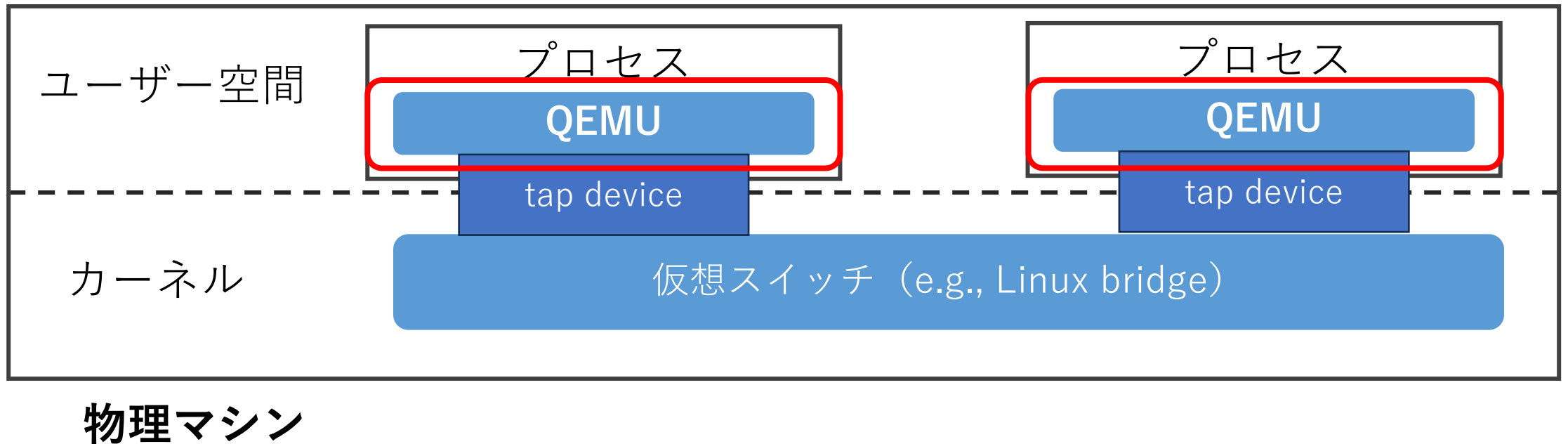
(比較的一般的な) 仮想スイッチ利用法

- ユーザー空間プロセスは tap デバイスを通じてパケットを送受信する



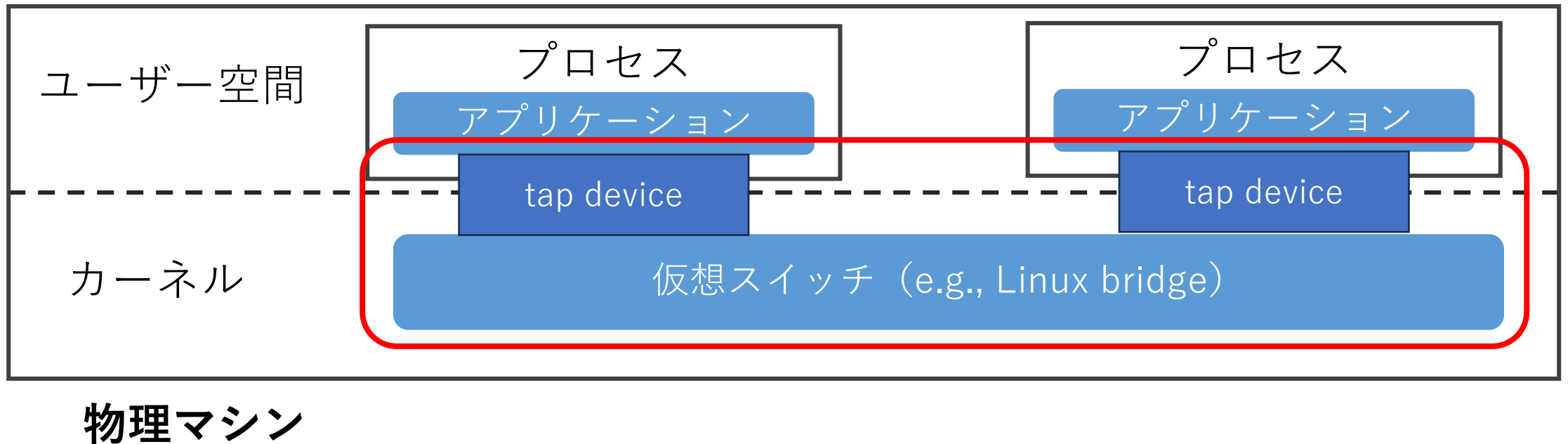
(比較的一般的な) 仮想スイッチ利用法

- ユーザー空間プロセスは tap デバイスを通じてパケットを送受信する
- QEMU/KVM ベースの仮想マシンの場合は図中のプロセスが実行するアプリケーションが QEMU になる



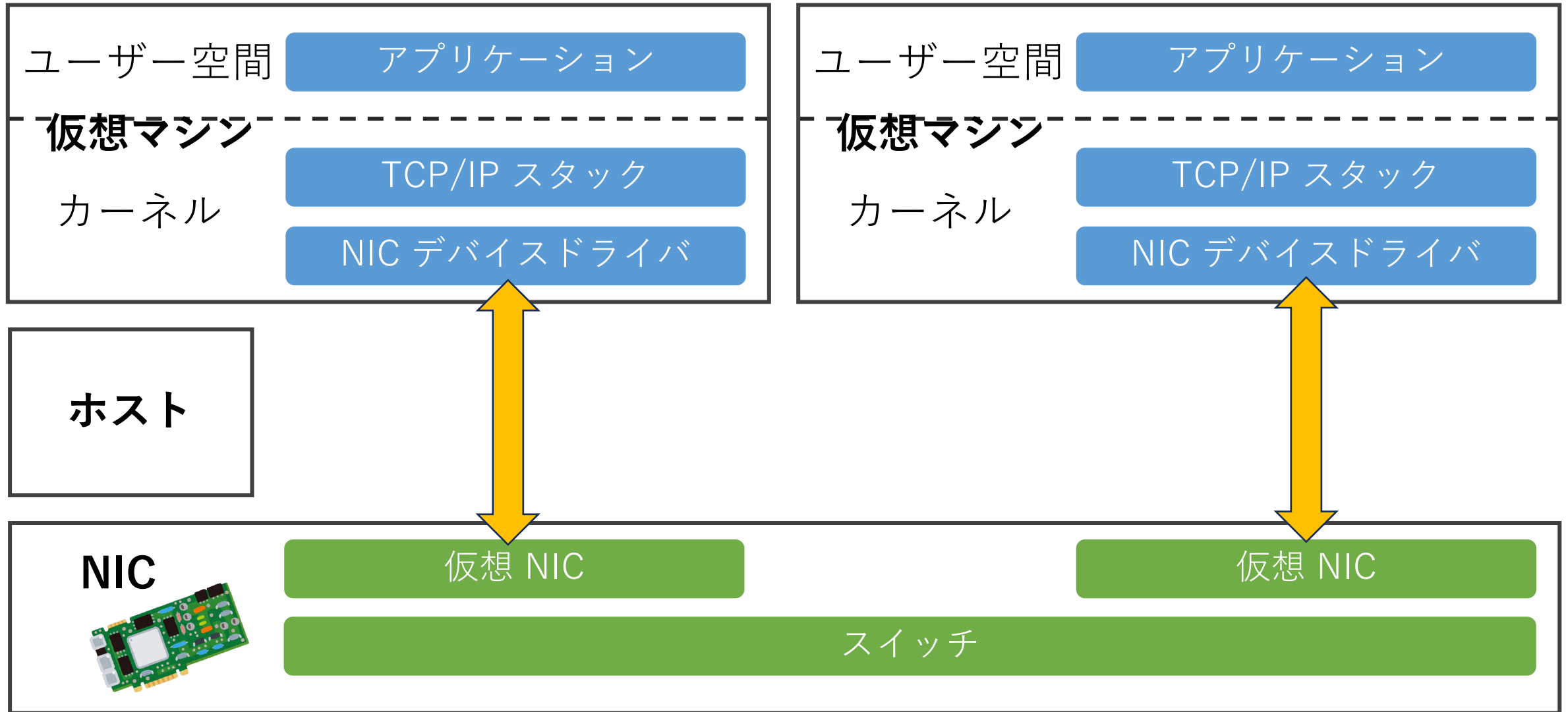
(比較的一般的な) 仮想スイッチ利用法

- 問題：既存の tap デバイスと仮想スイッチが高速にパケットをフォワードできない



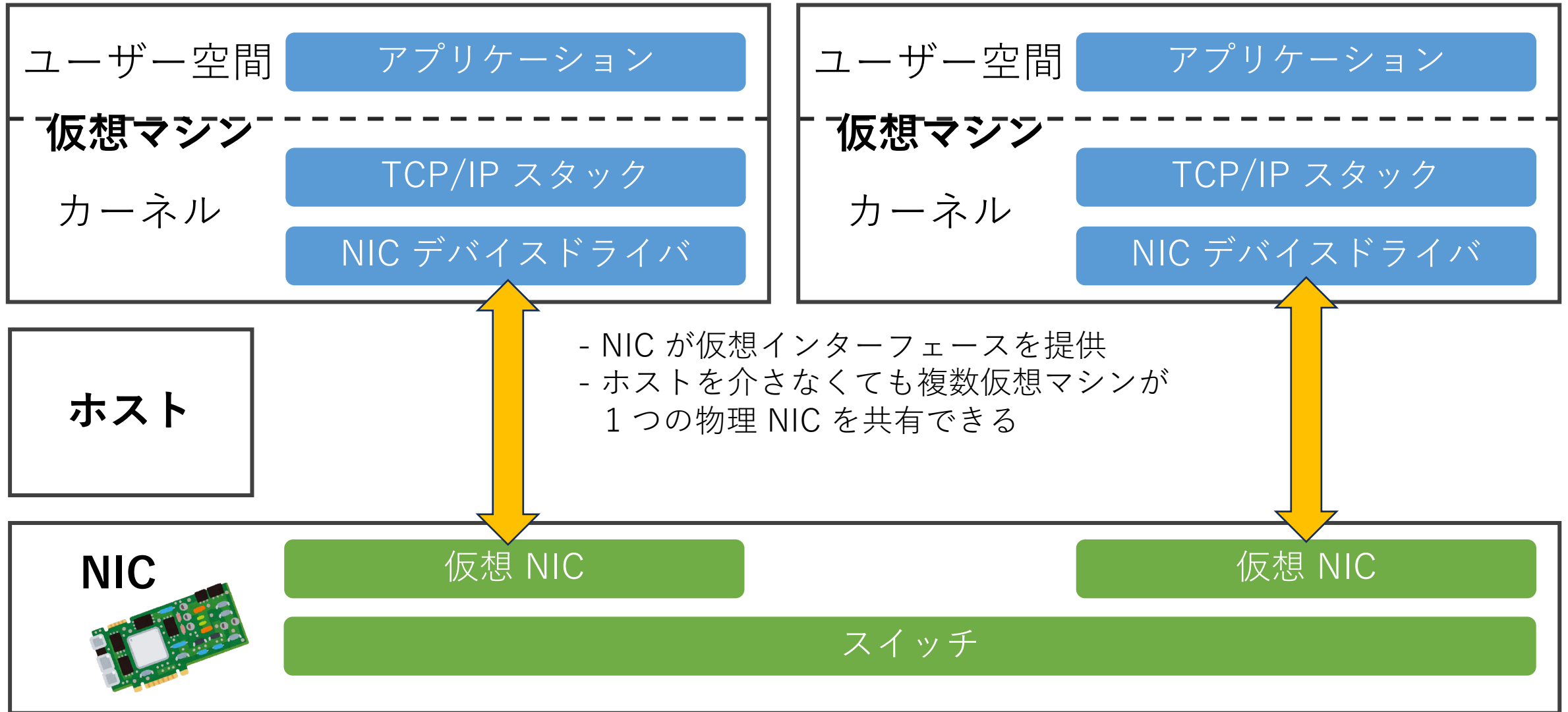
他の仮想 I/O 機構：SR-IOV

* Single Root I/O Virtualization



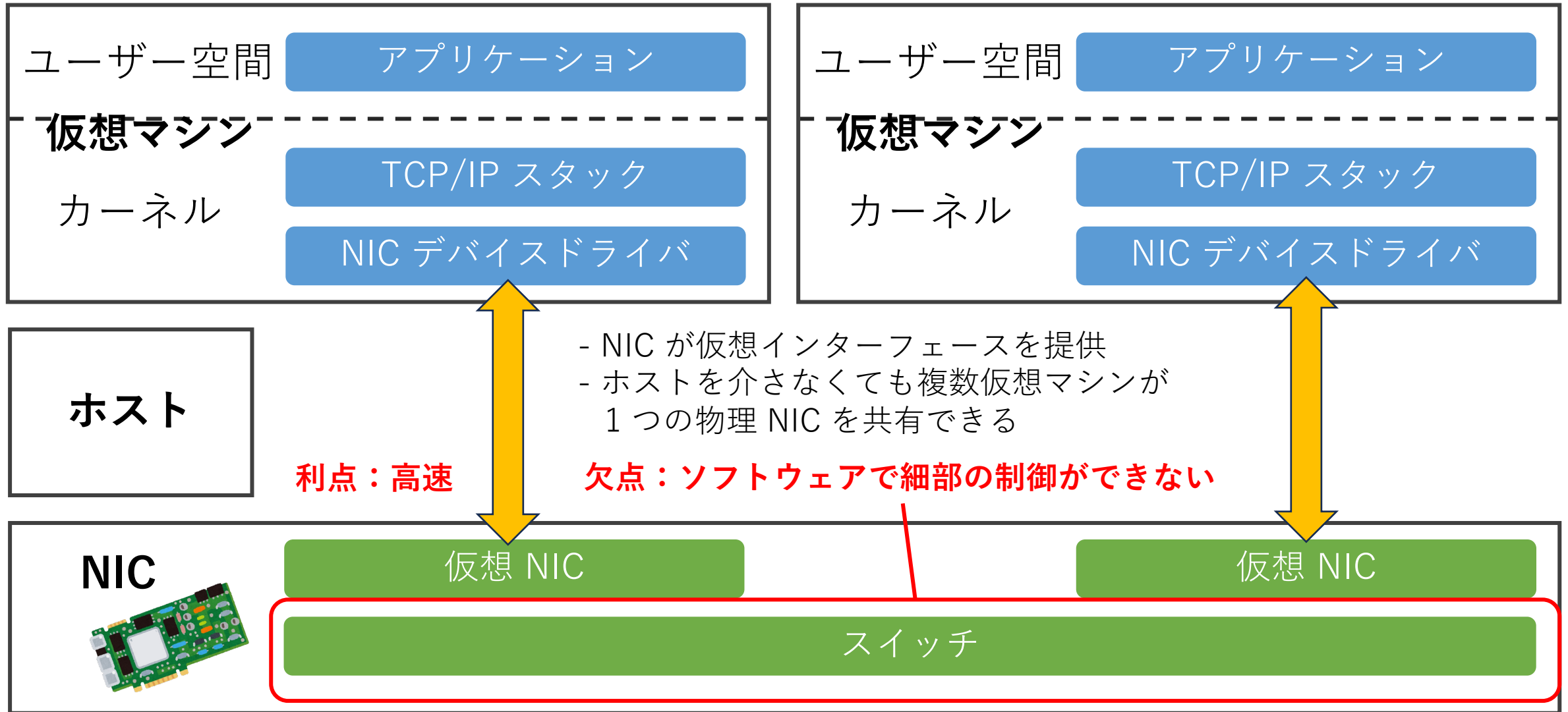
他の仮想 I/O 機構：SR-IOV

* Single Root I/O Virtualization



他の仮想 I/O 機構：SR-IOV

* Single Root I/O Virtualization



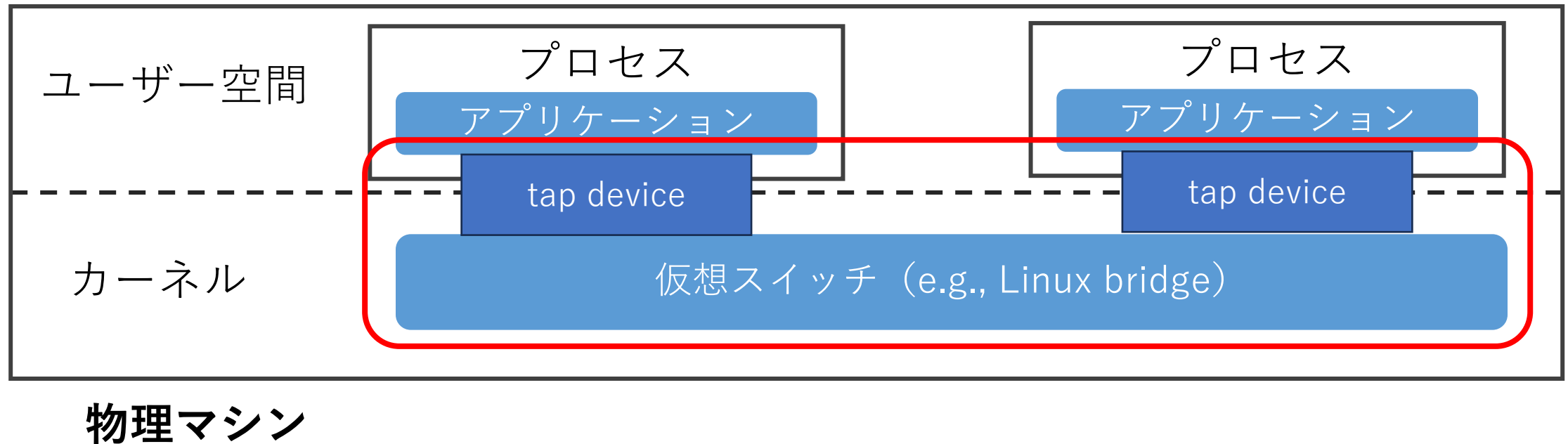
研究紹介

仮想マシン通信について

仮想スイッチの高速化

仮想マシン通信の高速化

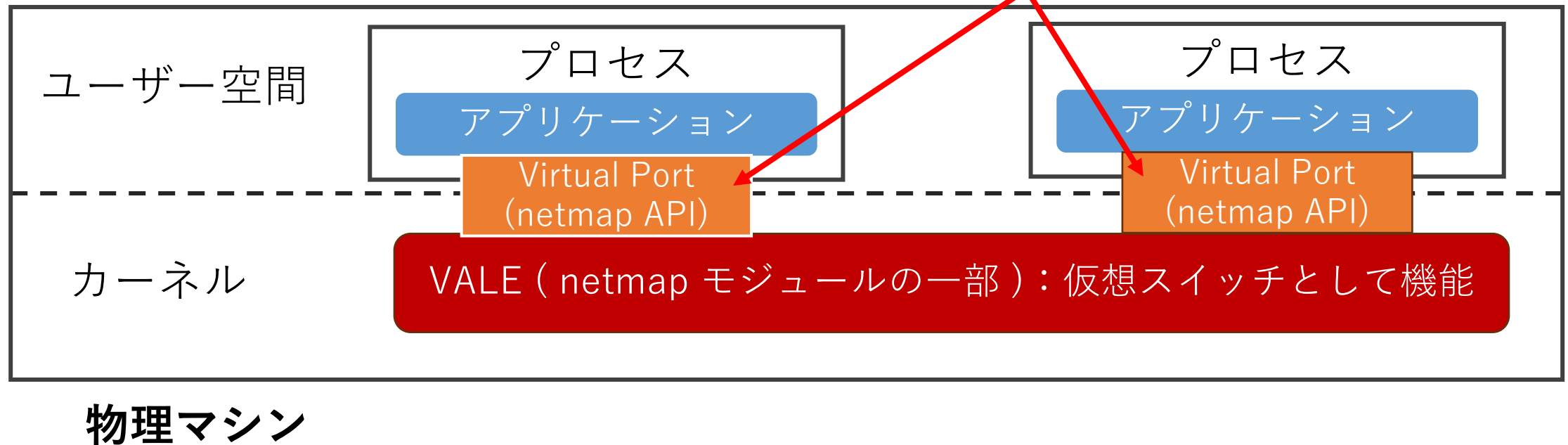
- 仮想スイッチへパケット I/O フレームワークを適用
 - VALE (CoNEXT 2012)
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)



仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)

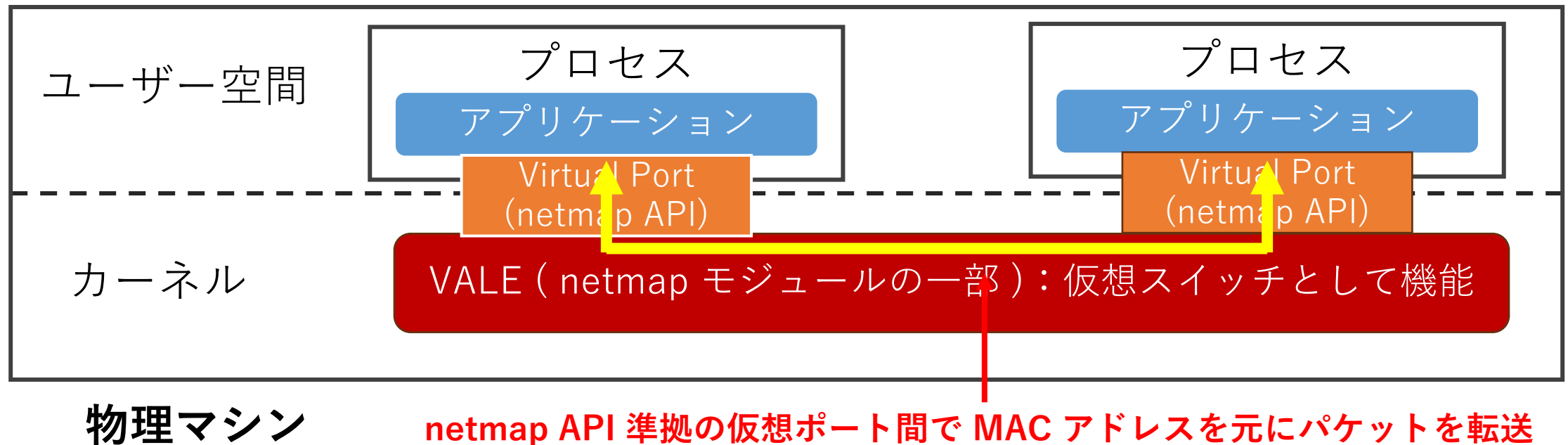
tap デバイスの代わりに netmap API 準拠の仮想ポート



仮想マシン通信の高速化

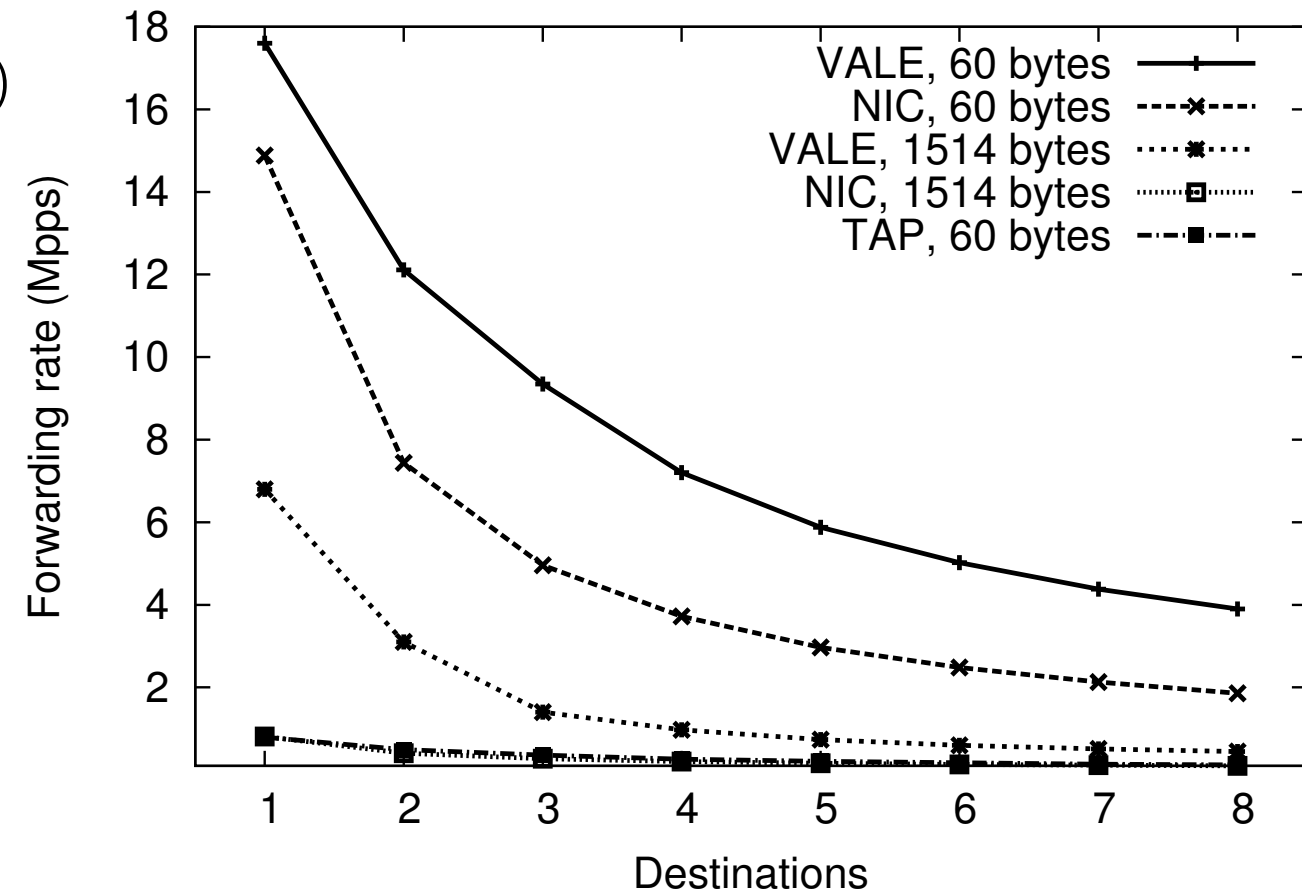
- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)

tap デバイスの代わりに netmap API 準拠の仮想ポート



仮想マシン通信の高速化

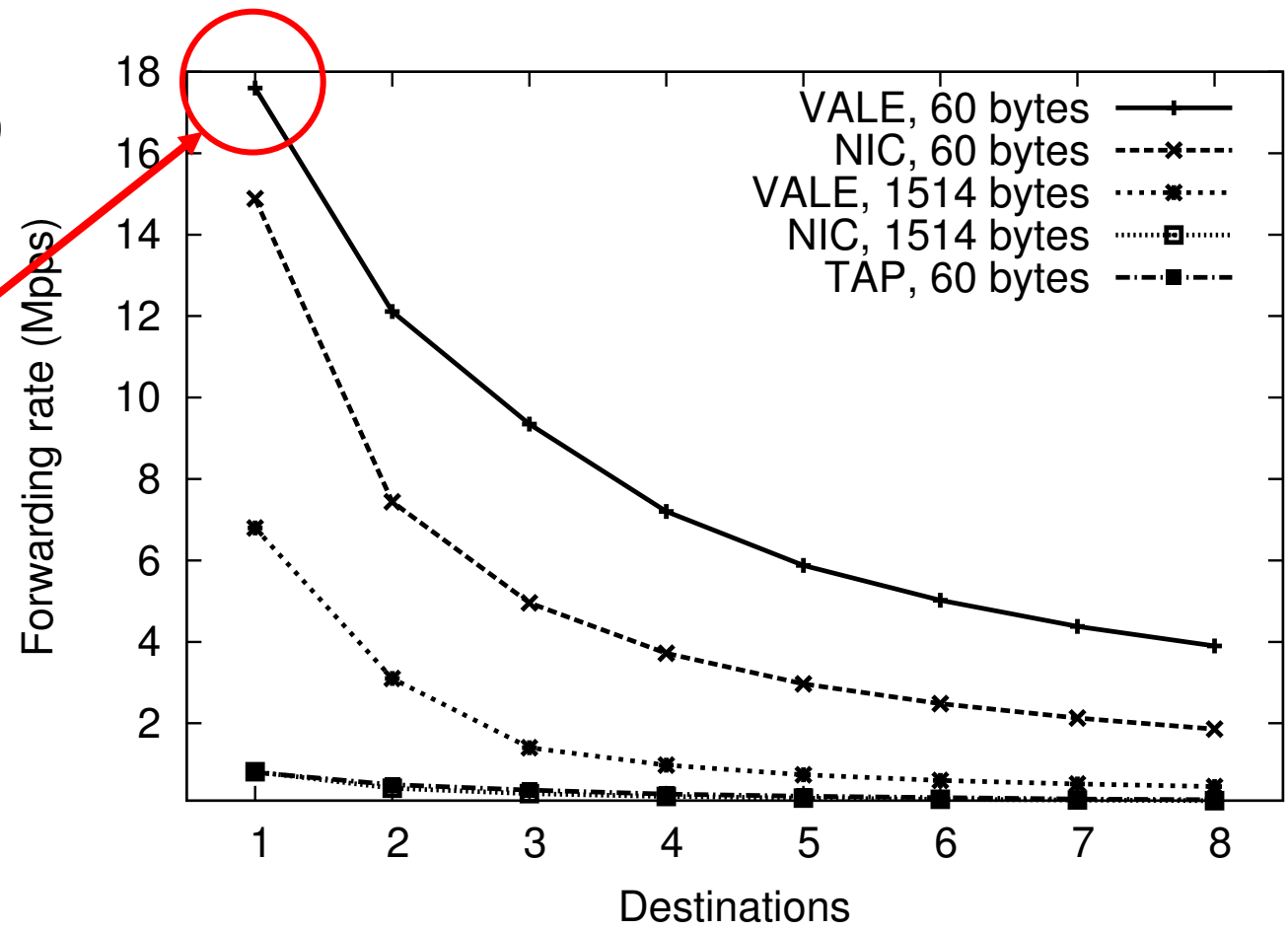
- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)



仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)

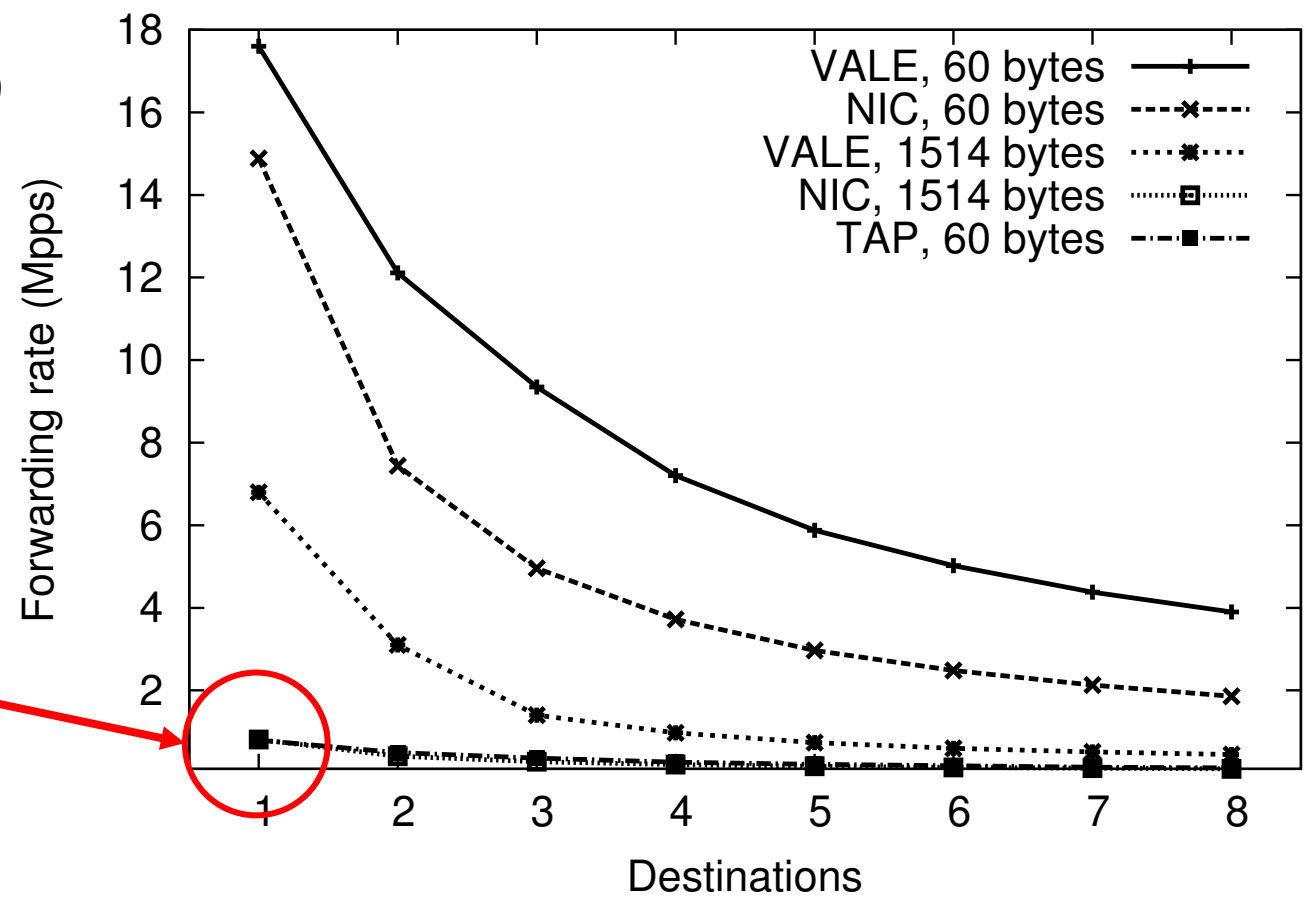
- パケットサイズが 60 バイトで宛先が 1 つの場合
 - **VALE: 17.6 Mpps**
 - tap デバイス: 1 Mpps 以下



仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)

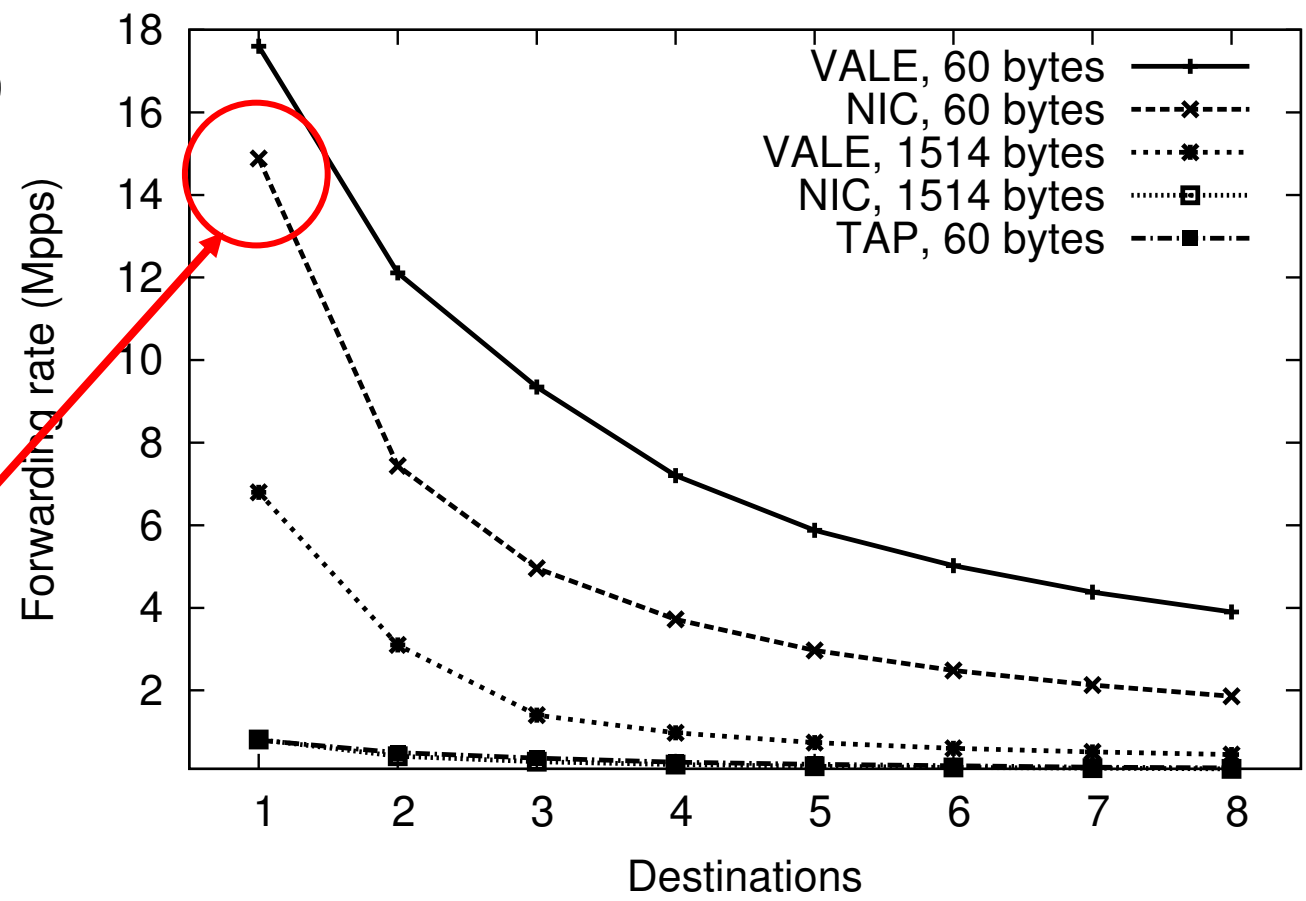
- パケットサイズが 60 バイトで宛先が 1 つの場合
 - VALE: 17.6 Mpps
 - **tap デバイス: 1 Mpps 以下**



仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用
 - **VALE (CoNEXT 2012)**
 - CuckooSwitch (CoNEXT 2013)
 - mSwitch (SOSR 2015)

- パケットサイズが 60 バイトで宛先が 1 つの場合
 - VALE: 17.6 Mpps
 - tap デバイス: 1 Mpps 以下



SR-IOV はデータが PCI バスを経由するため仮想ポート間の通信は VALE より遅くなる

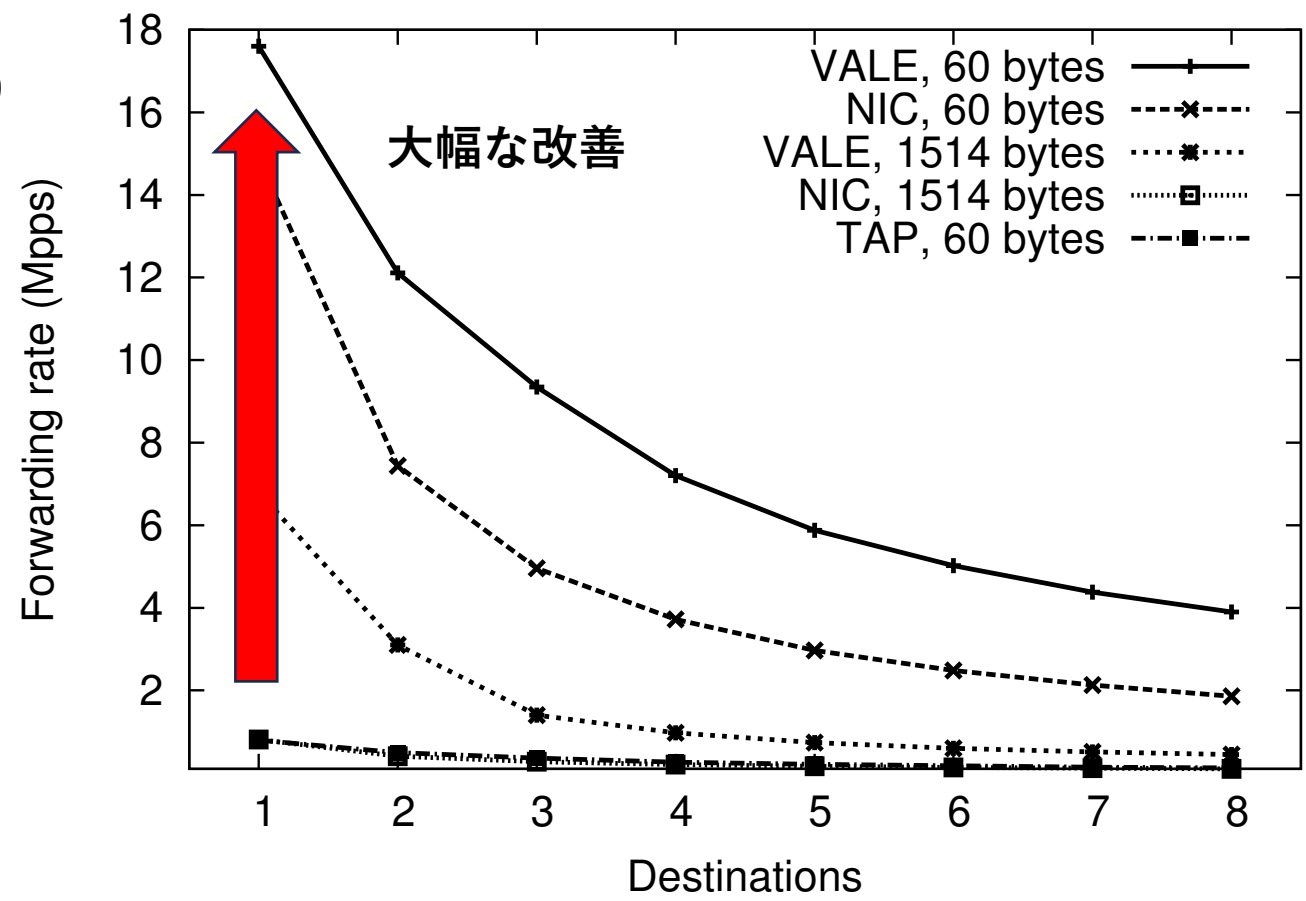
仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用

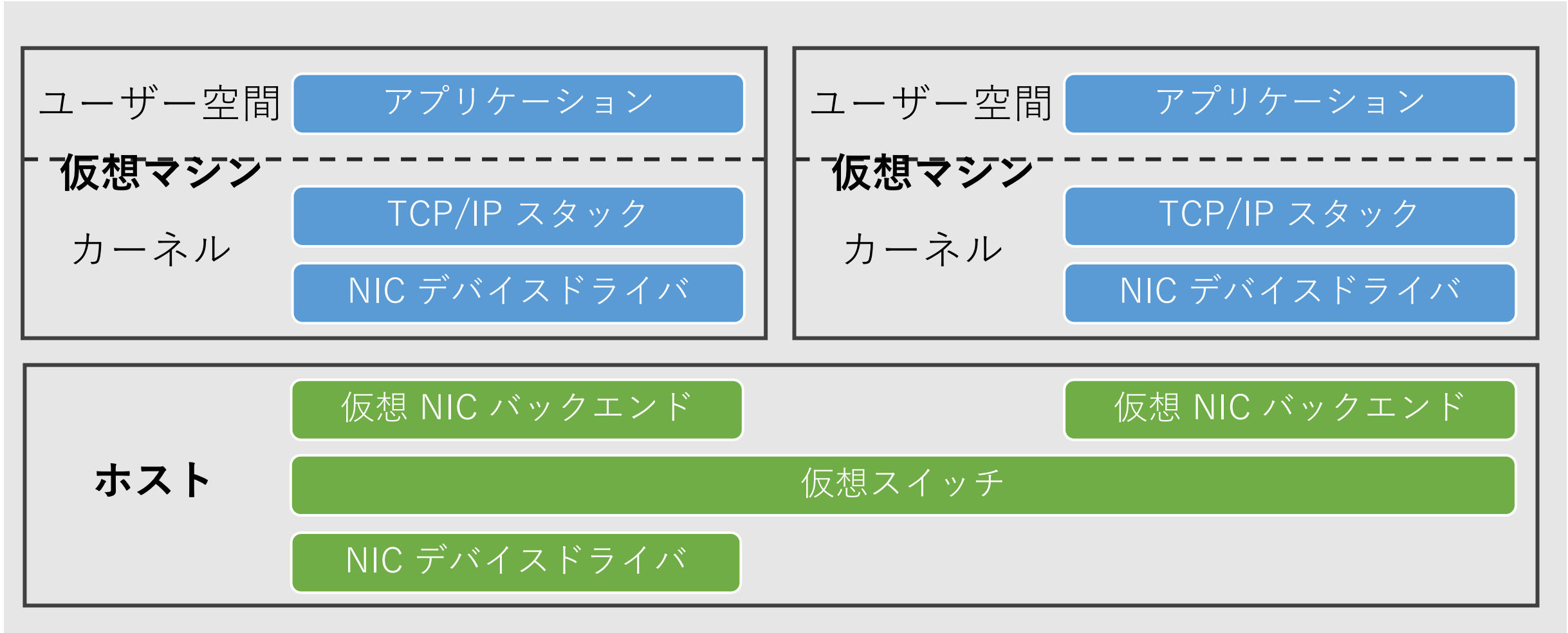
- **VALE (CoNEXT 2012)**
- CuckooSwitch (CoNEXT 2013)
- mSwitch (SOSR 2015)

- パケットサイズが 60 バイトで宛先が 1 つの場合

- VALE: 17.6 Mpps
- tap デバイス: 1 Mpps 以下

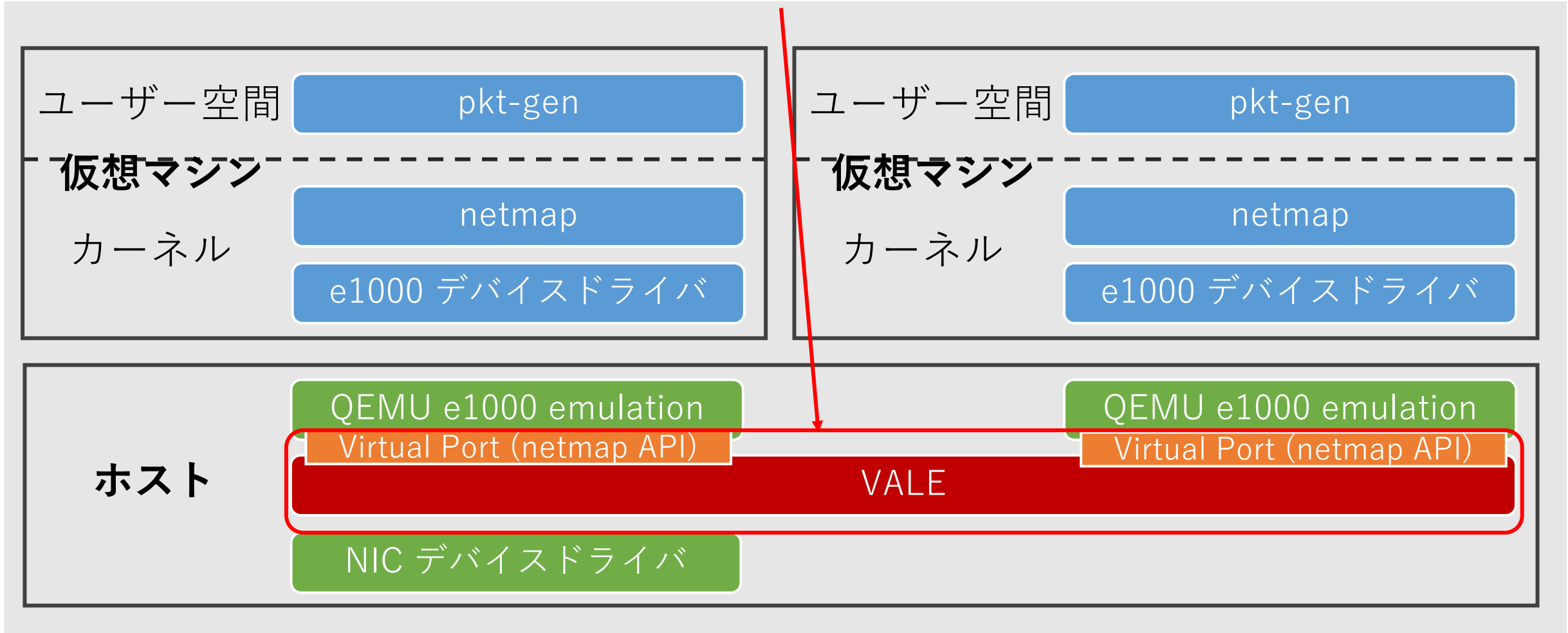


仮想マシン通信の高速化



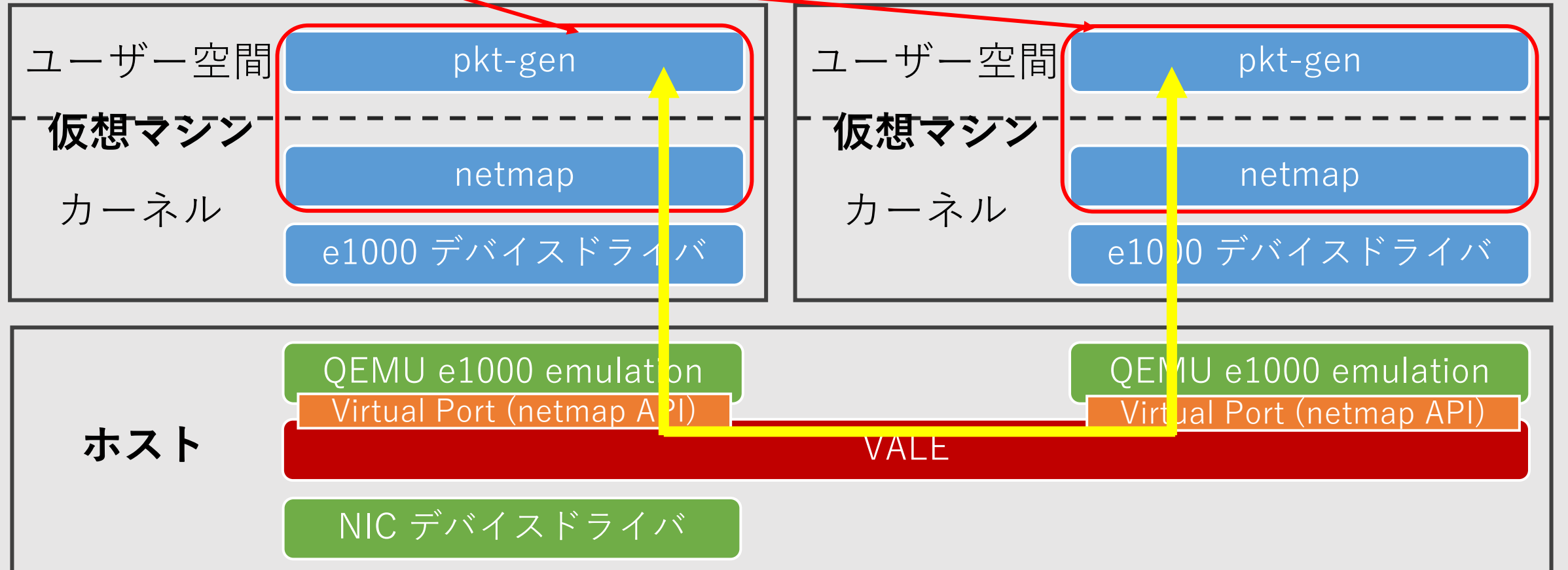
仮想マシン通信の高速化

VALE を QEMU/KVM へ適用して実験



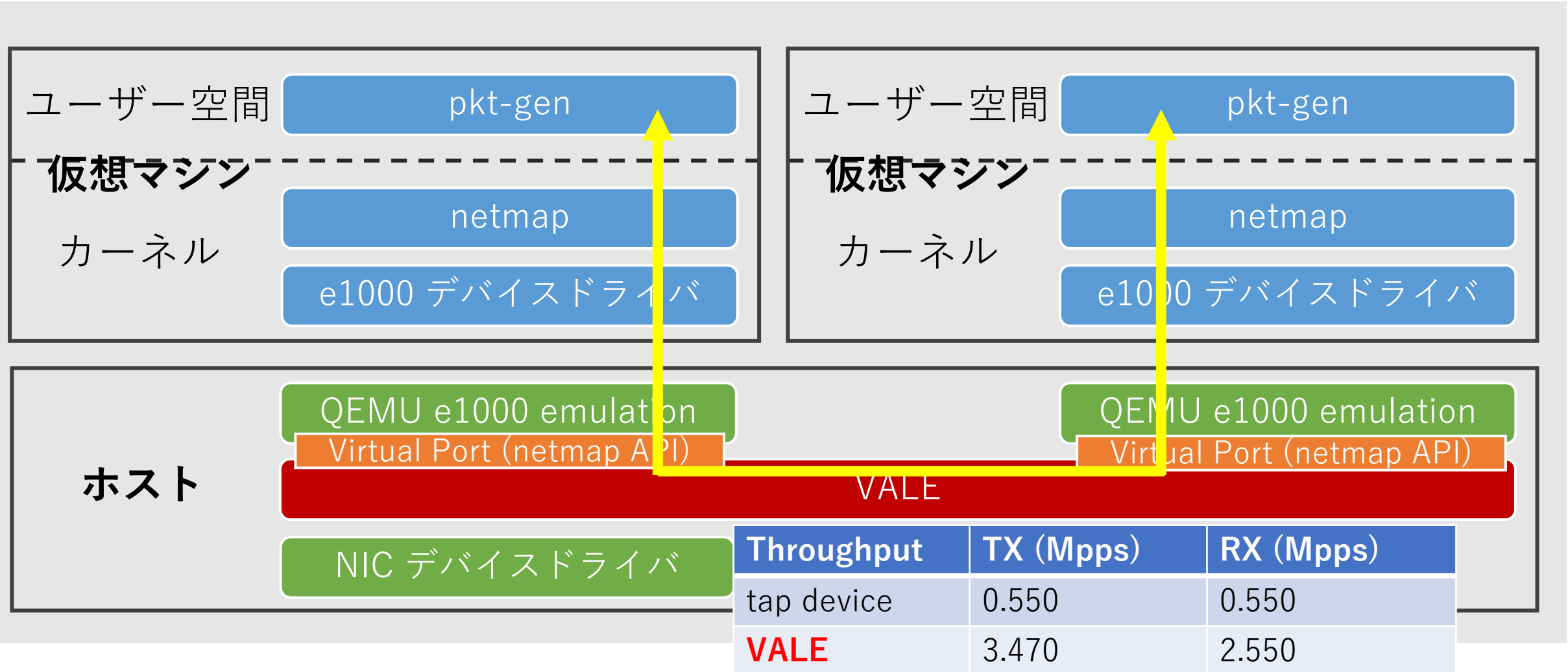
仮想マシン通信の高速化

仮想マシン内では netmap ベースの pkt-gen アプリを実行 VALE を QEMU/KVM へ適用して実験



仮想マシン通信の高速化

VALE を QEMU/KVM へ適用して実験



仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用

- VALE (CoNEXT 2012)
- CuckooSwitch (CoNEXT 2013)
- **mSwitch (SOSR 2015)**

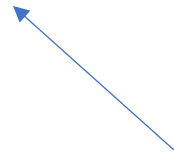
改善

- 利用可能ポート数がスケールできるようにする
- パケット転送ロジックをカーネルモジュールで実装できるようにする

仮想マシン通信の高速化

- 仮想スイッチへパケット I/O フレームワークを適用
 - VALE (CoNEXT 2012)
 - **CuckooSwitch (CoNEXT 2013)**
 - mSwitch (SOSR 2015)

DPDK ベース実装



研究紹介

仮想マシン通信について

高速な仮想スイッチを仮想マシン通信へ適用

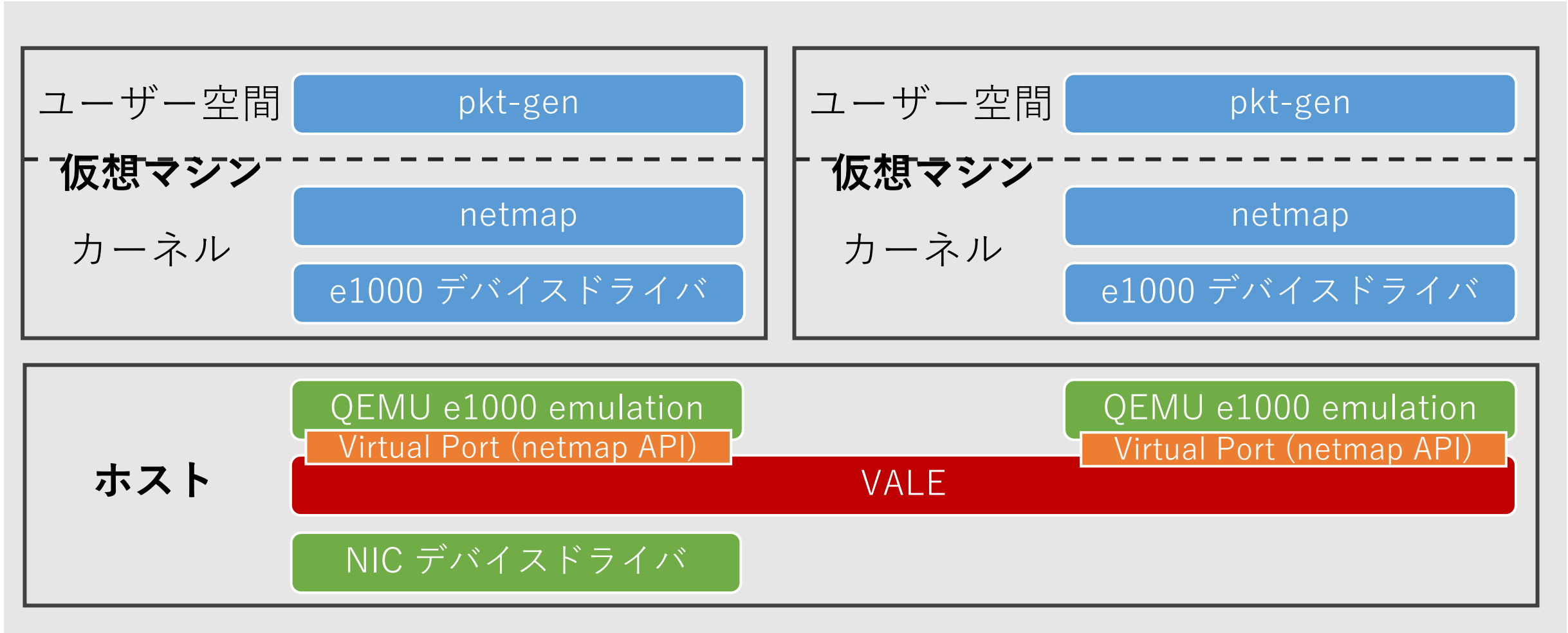
仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - ELISA (ASPLOS 2023)

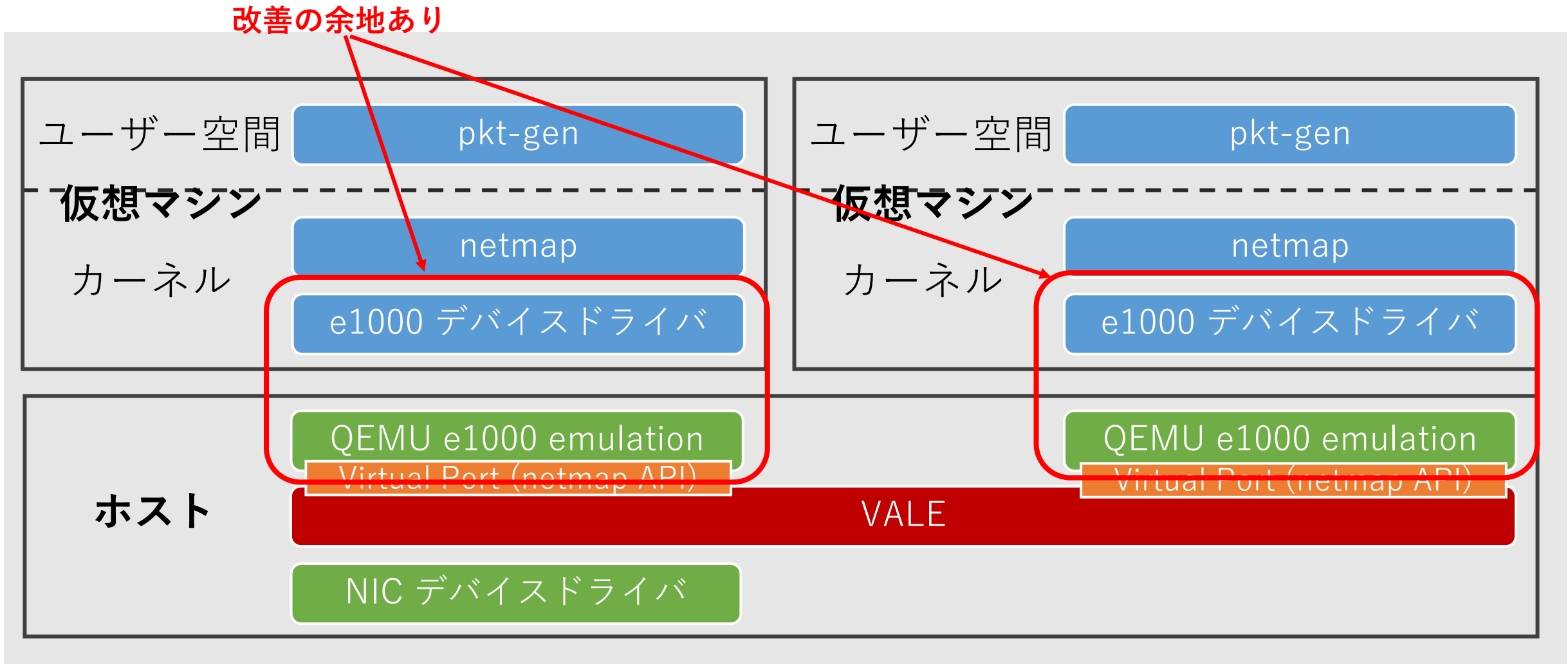
仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - **ptnetmap (ANCS 2015, LANMAN 2016)**
 - HyperNF (SoCC 2017)
 - ELISA (ASPLOS 2023)

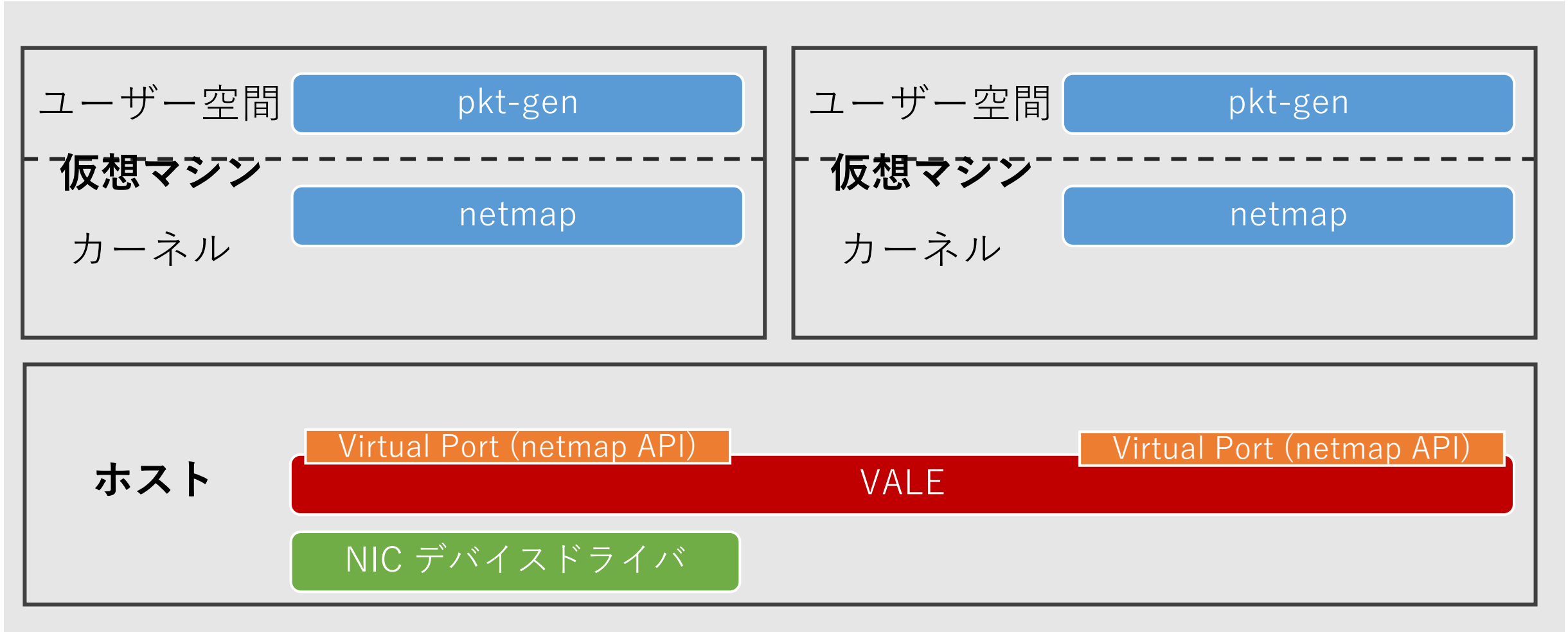
仮想マシン通信の高速化



仮想マシン通信の高速化



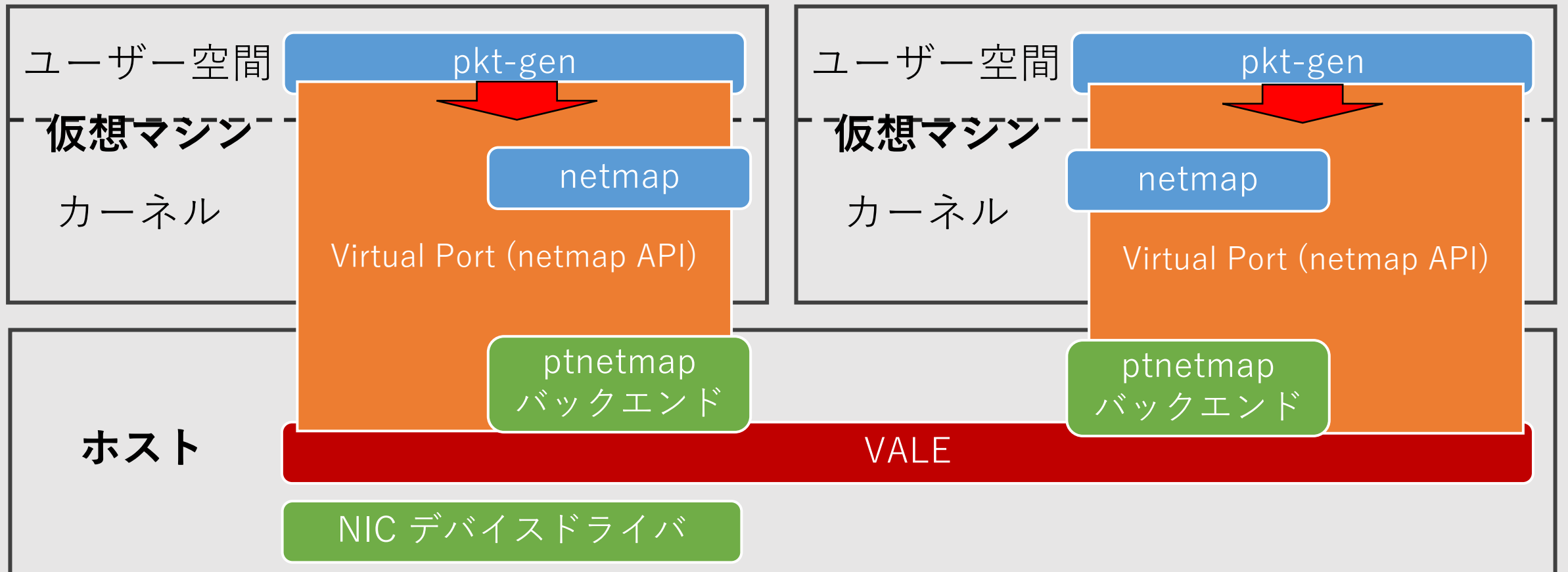
仮想マシン通信の高速化



仮想マシン通信の高速化

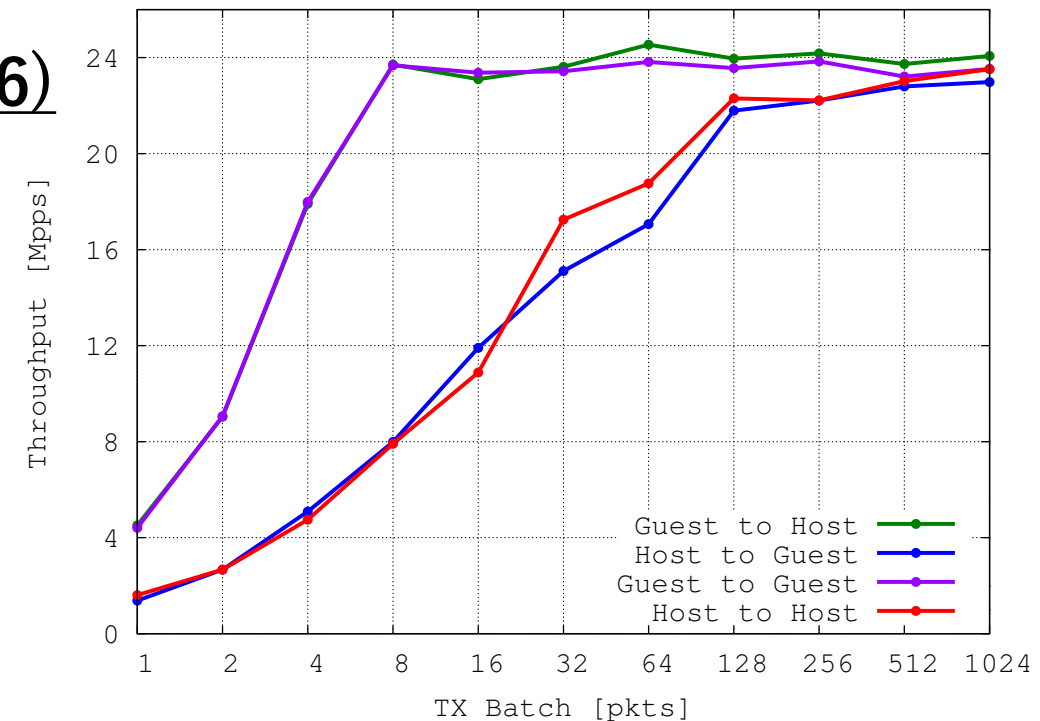
ptnetmap (pt: passthrough) :

ホストが作成した仮想 (netmap) ポートへ、仮想マシン内のアプリが直接アクセスできるようにする



仮想マシン通信の高速化

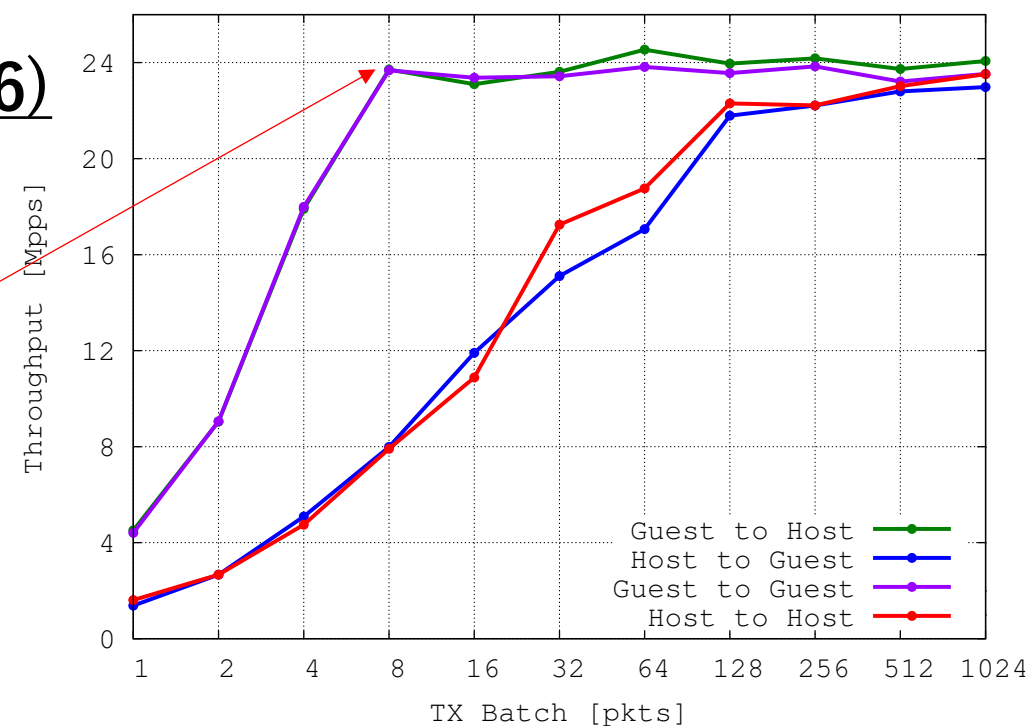
- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - **ptnetmap (ANCS 2015, LANMAN 2016)**
 - HyperNF (SoCC 2017)
 - ELISA (ASPLOS 2023)



仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - **ptnetmap (ANCS 2015, LANMAN 2016)**
 - HyperNF (SoCC 2017)
 - ELISA (ASPLOS 2023)

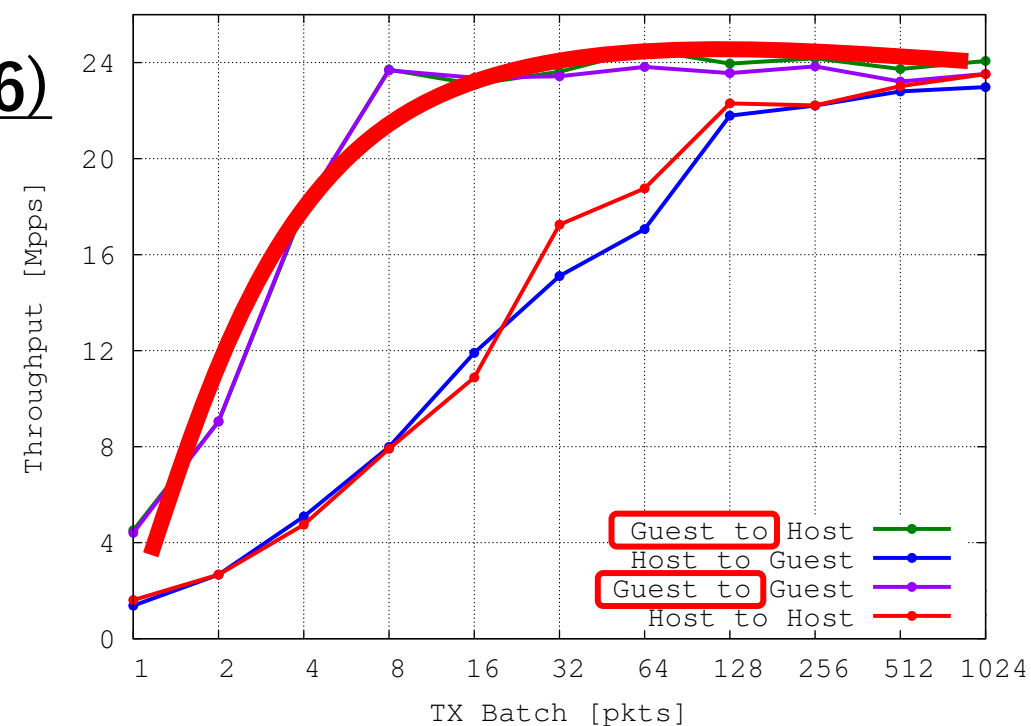
24 Mpps



仮想マシン通信の高速化

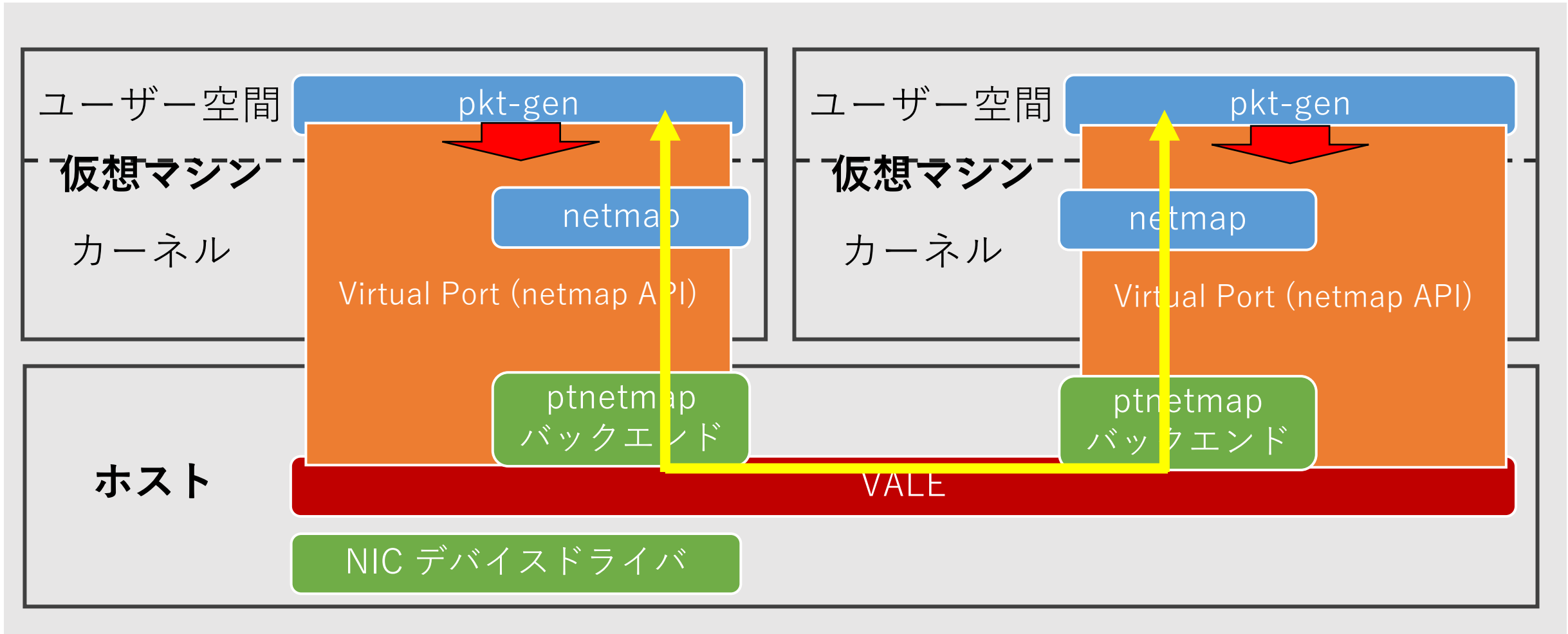
- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - **ptnetmap (ANCS 2015, LANMAN 2016)**
 - HyperNF (SoCC 2017)
 - ELISA (ASPLOS 2023)

Guest が送信側の時が速い



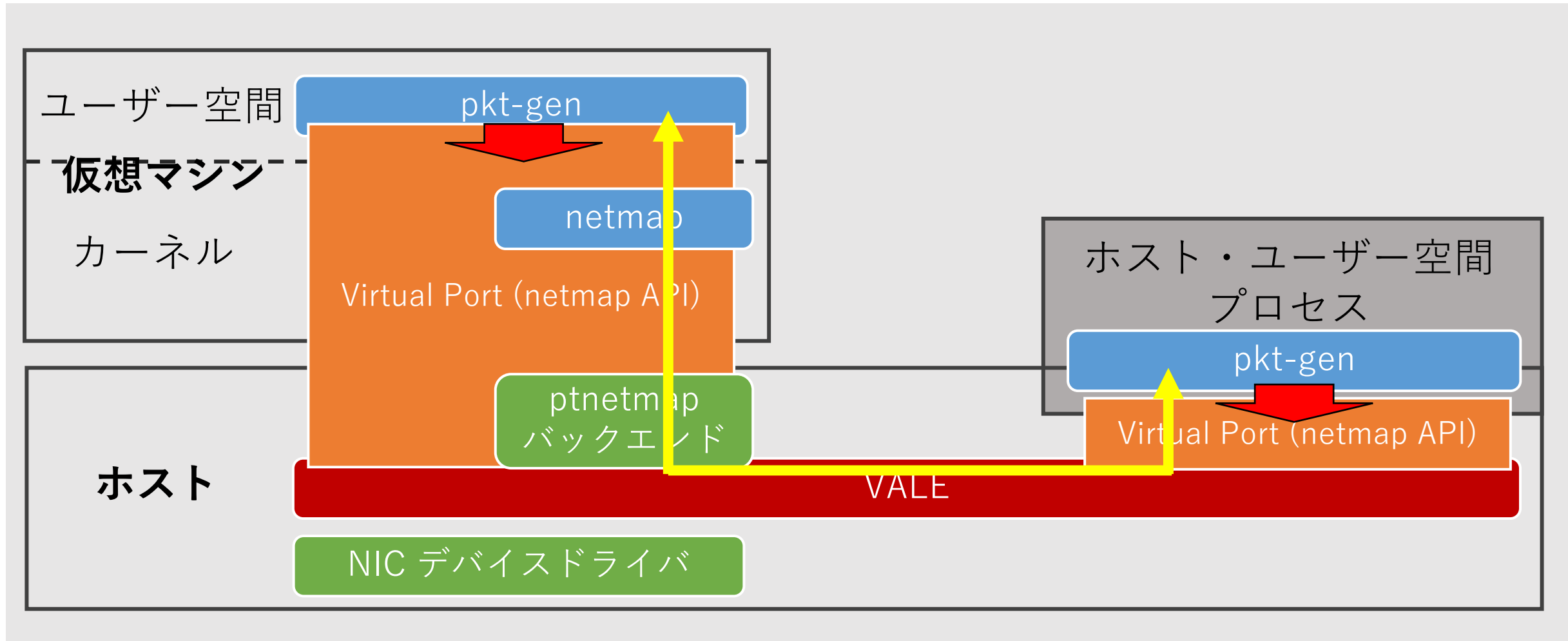
仮想マシン通信の高速化

Guest to Guest



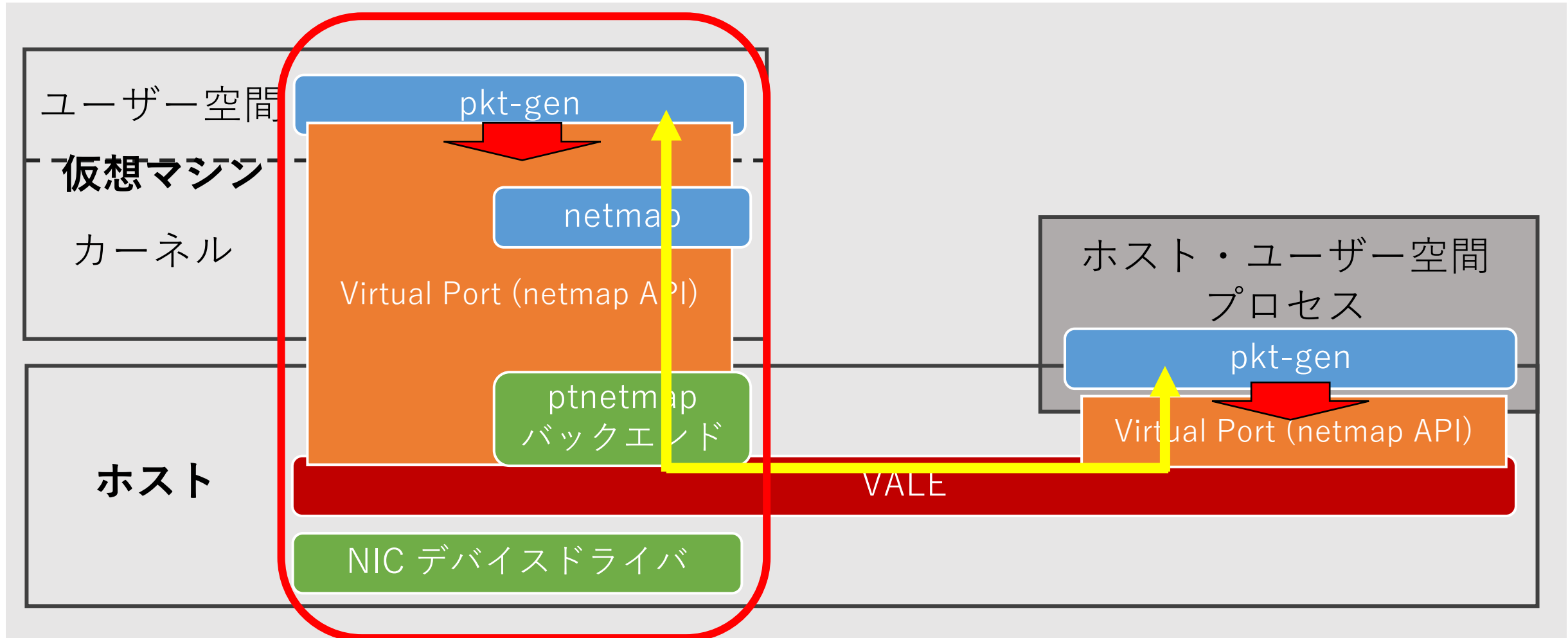
仮想マシン通信の高速化

Guest to Host



仮想マシン通信の高速化

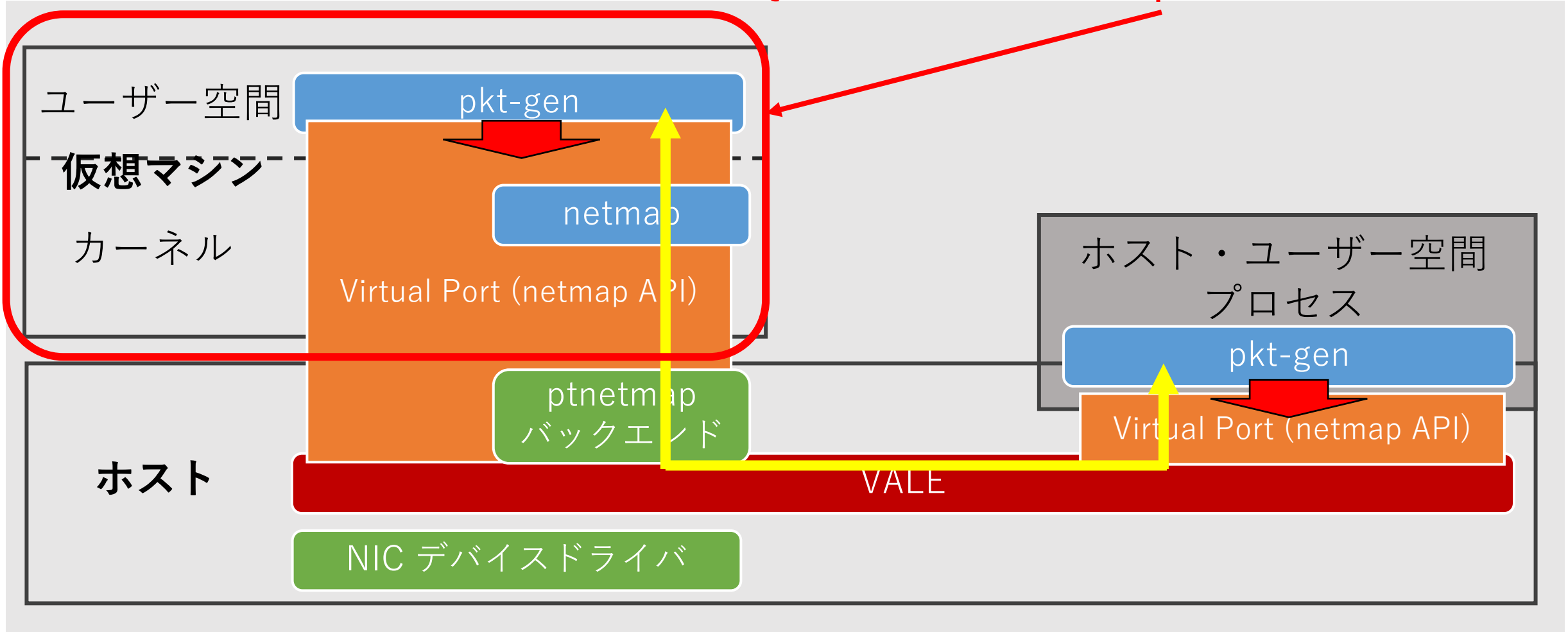
Guest to Host



仮想マシン通信の高速化

Guest to Host

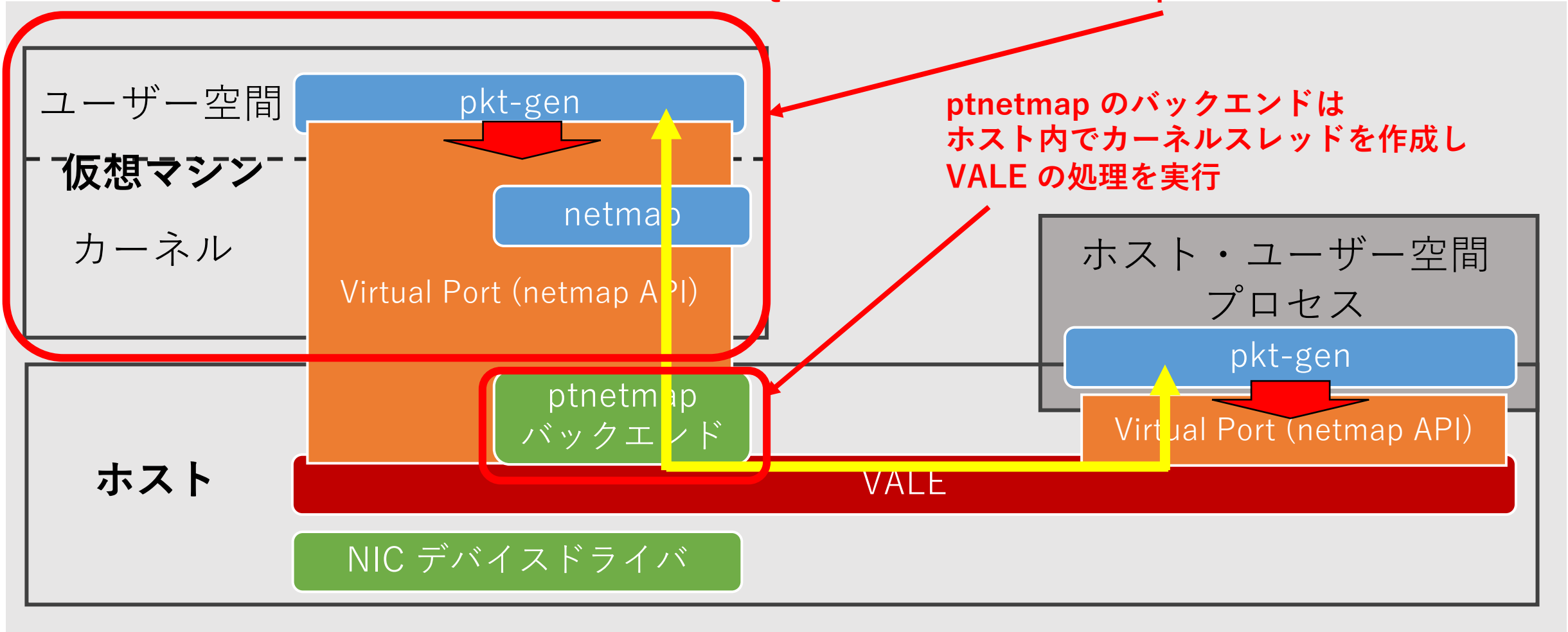
QEMU/KVM では vCPU は pthread として実装される



仮想マシン通信の高速化

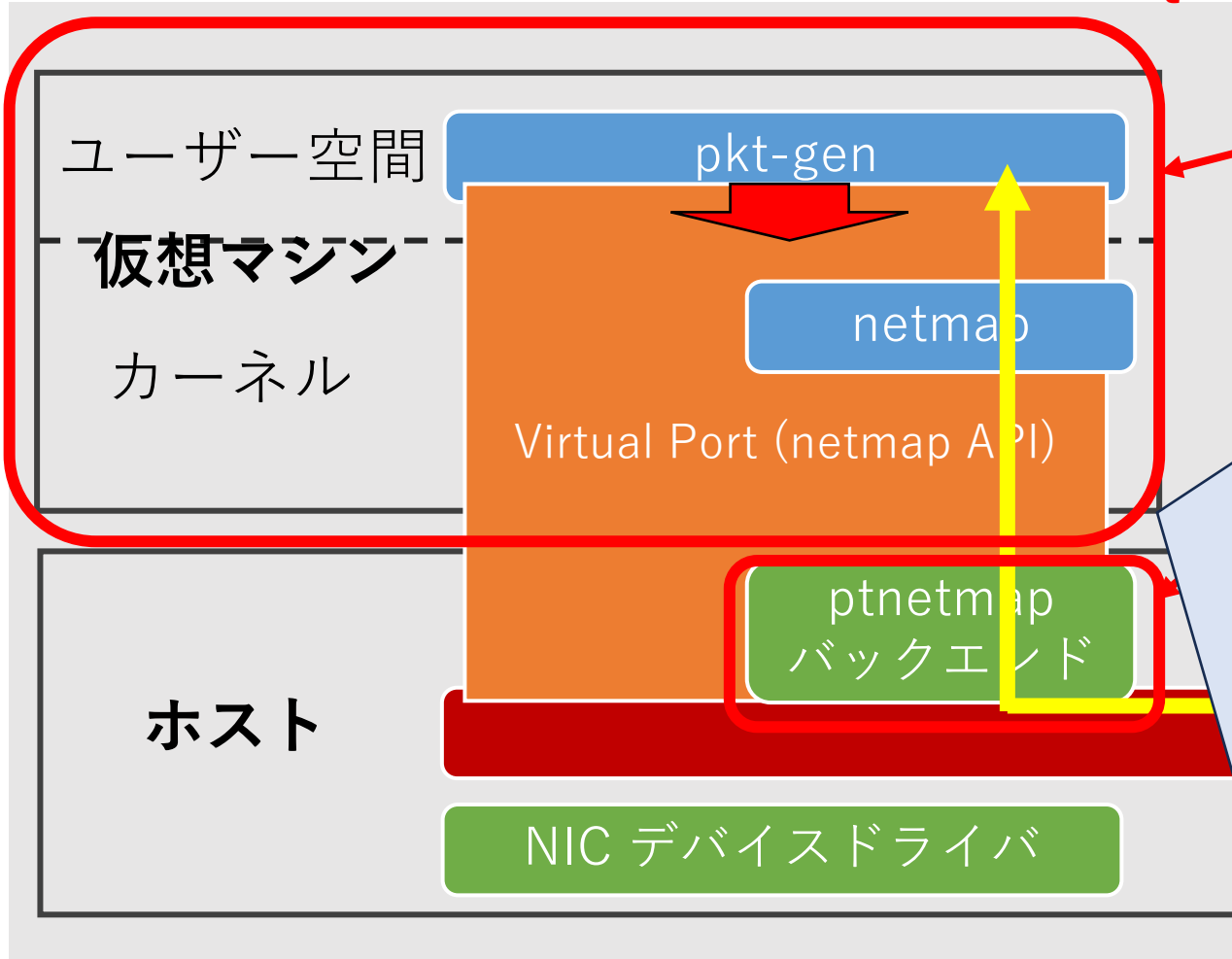
Guest to Host

QEMU/KVM では vCPU は pthread として実装される

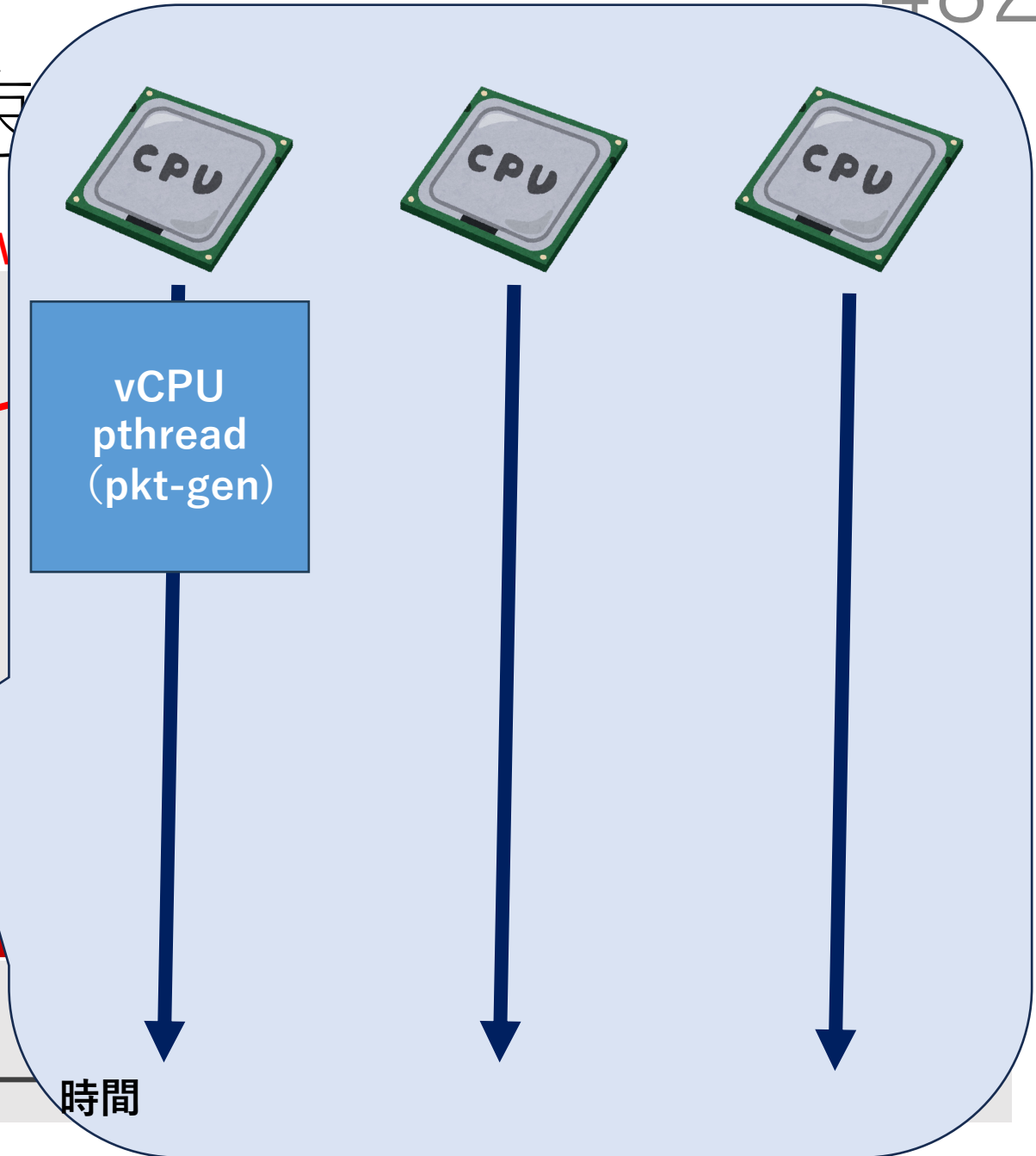


仮想マシン通信の高速

Guest to Host

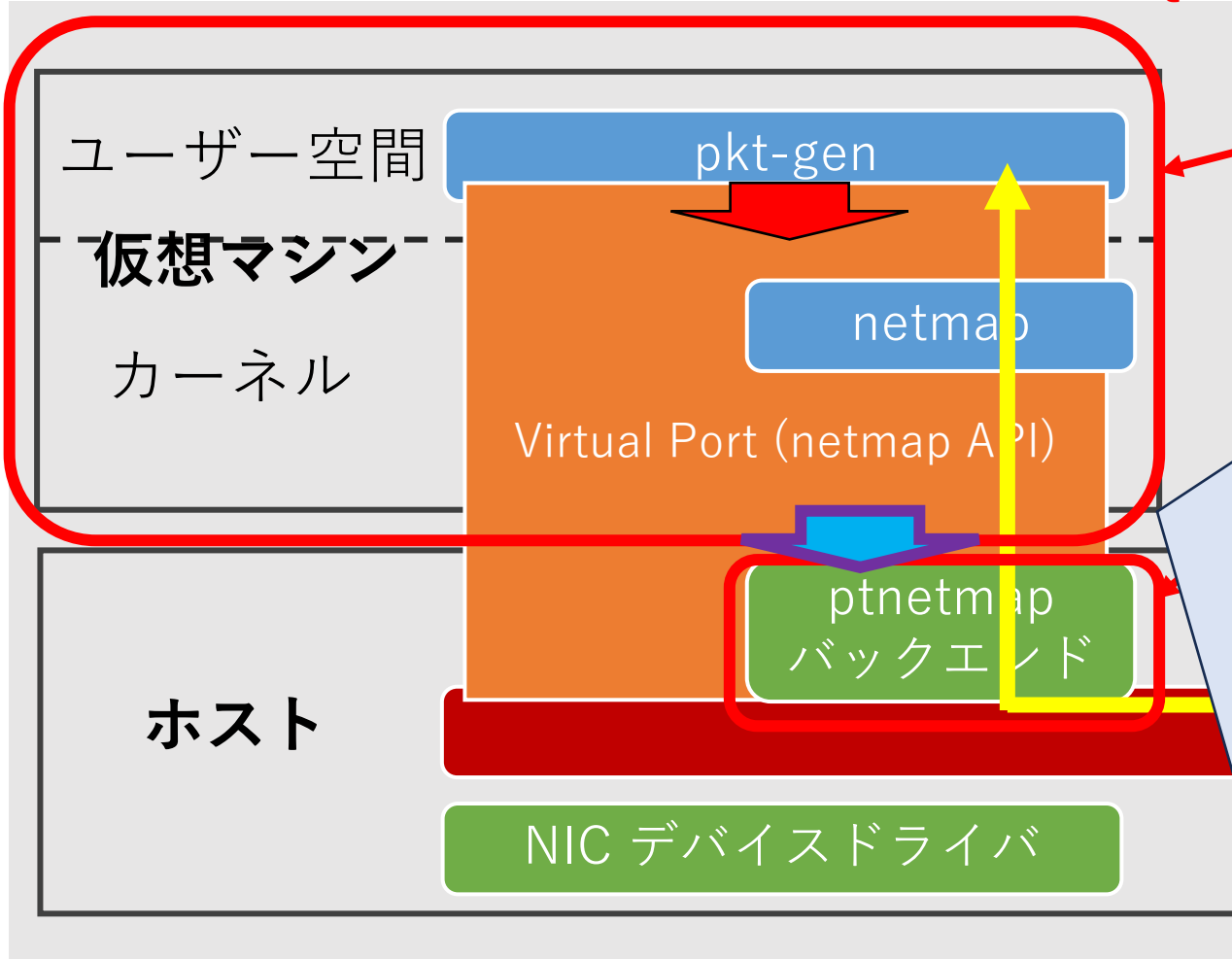


QEM

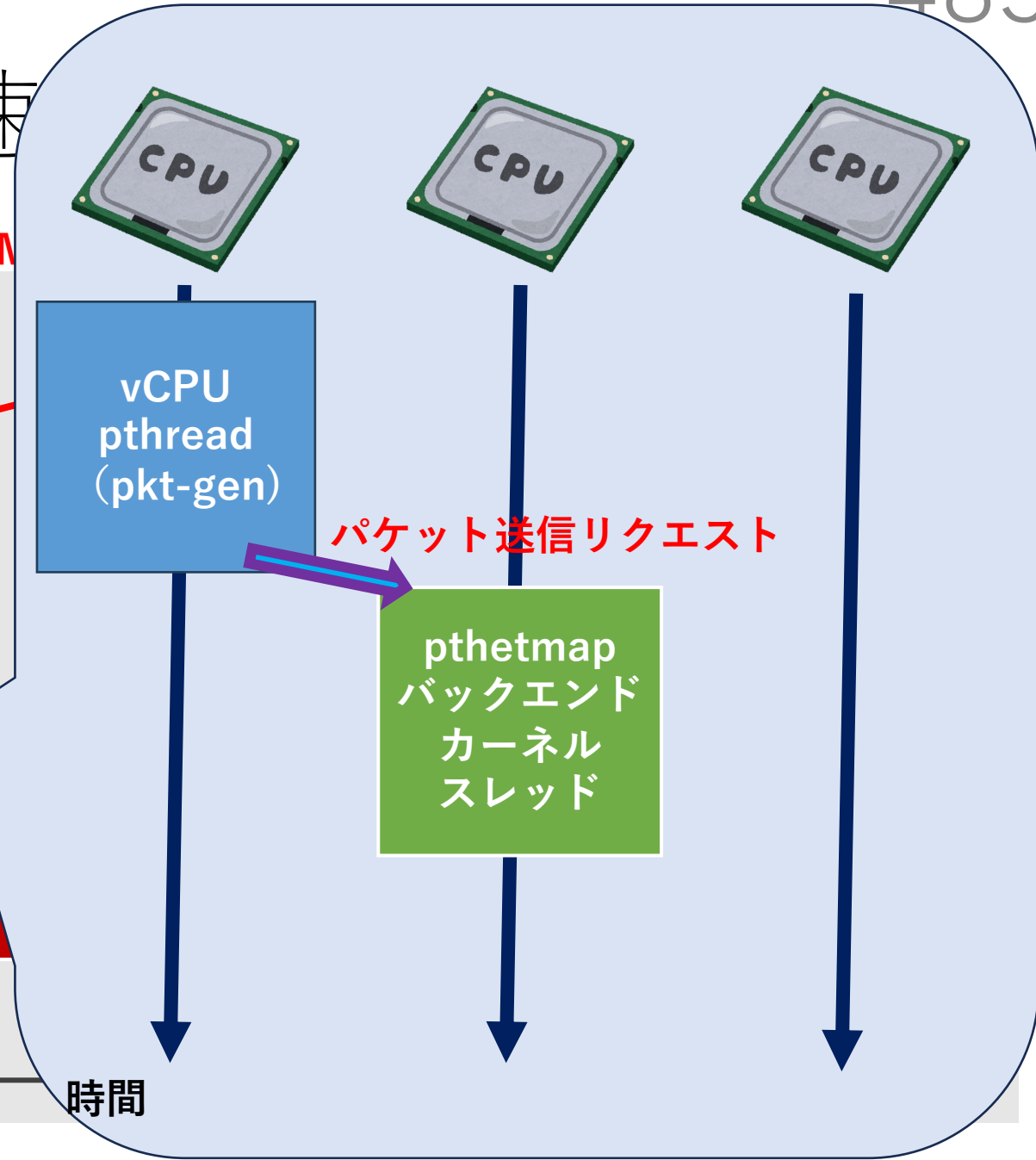


仮想マシン通信の高速

Guest to Host

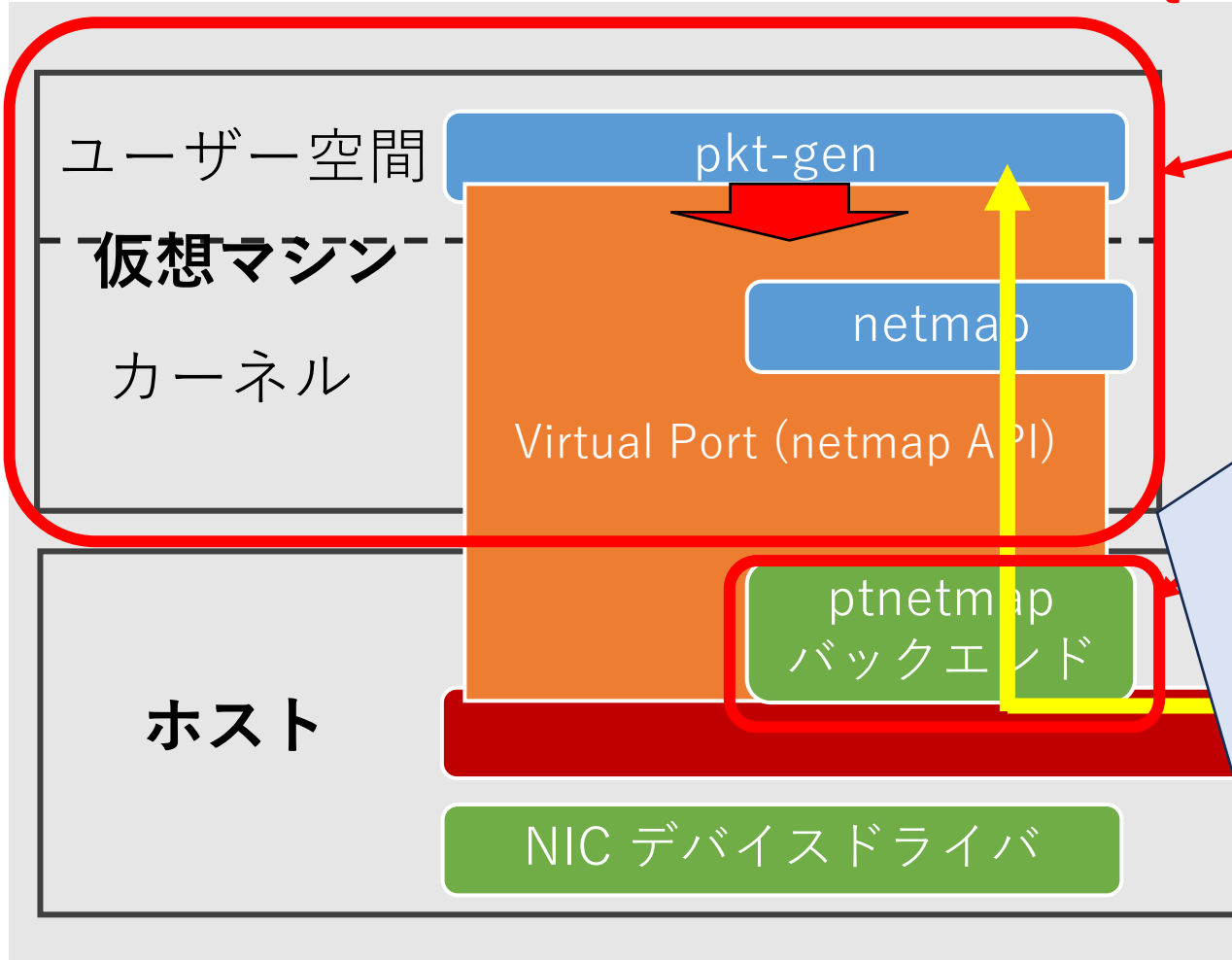


QEM

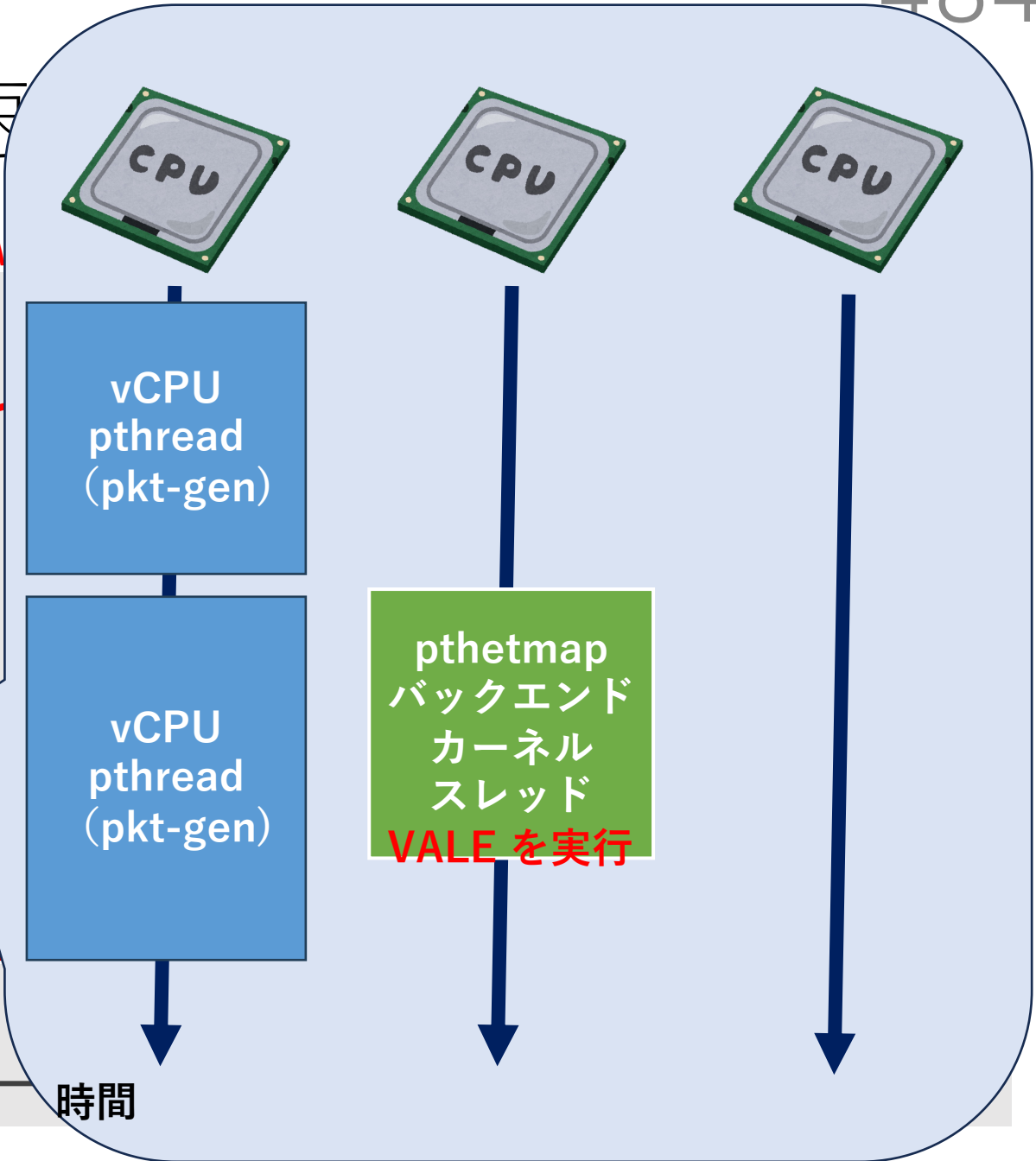


仮想マシン通信の高速

Guest to Host

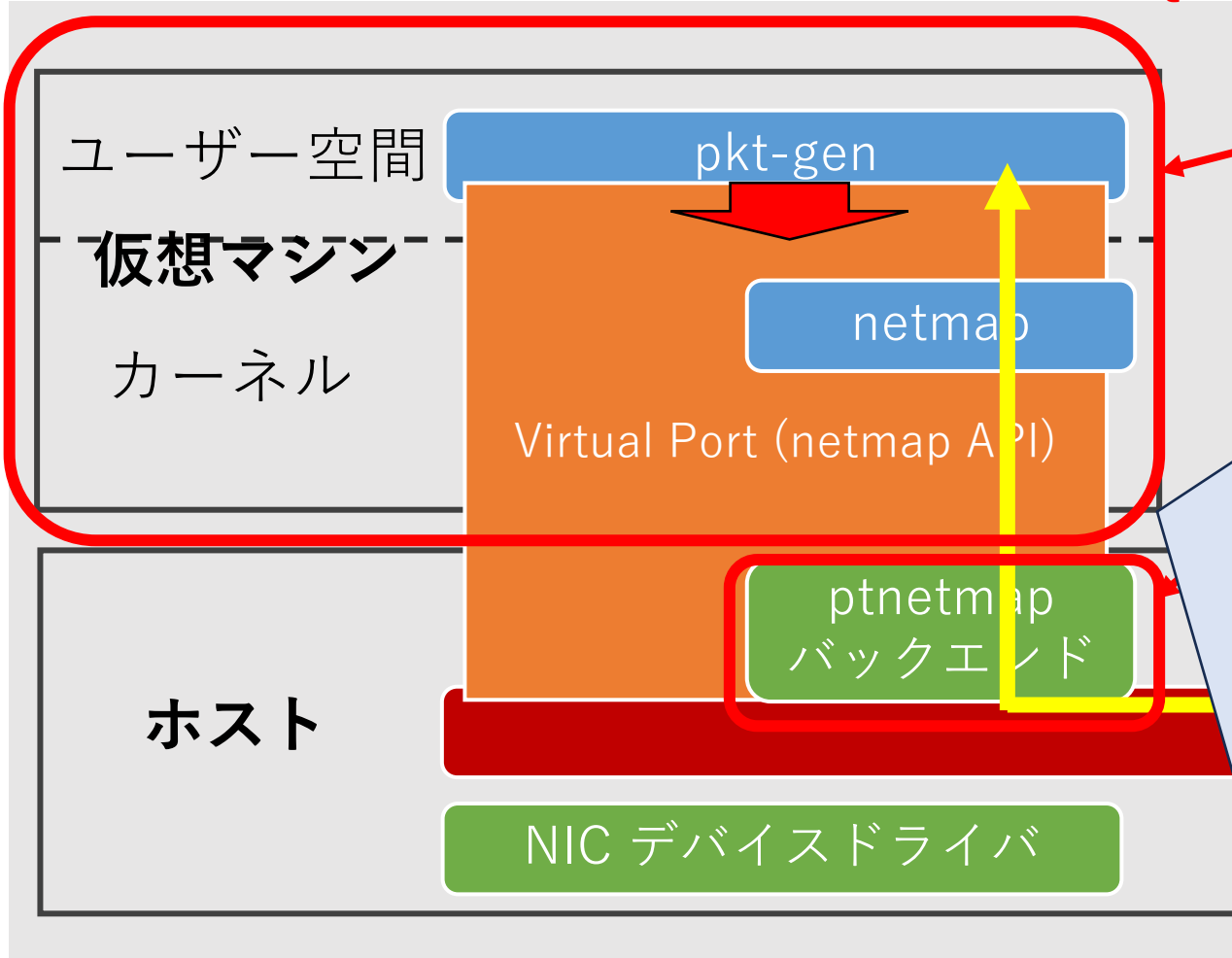


QEM

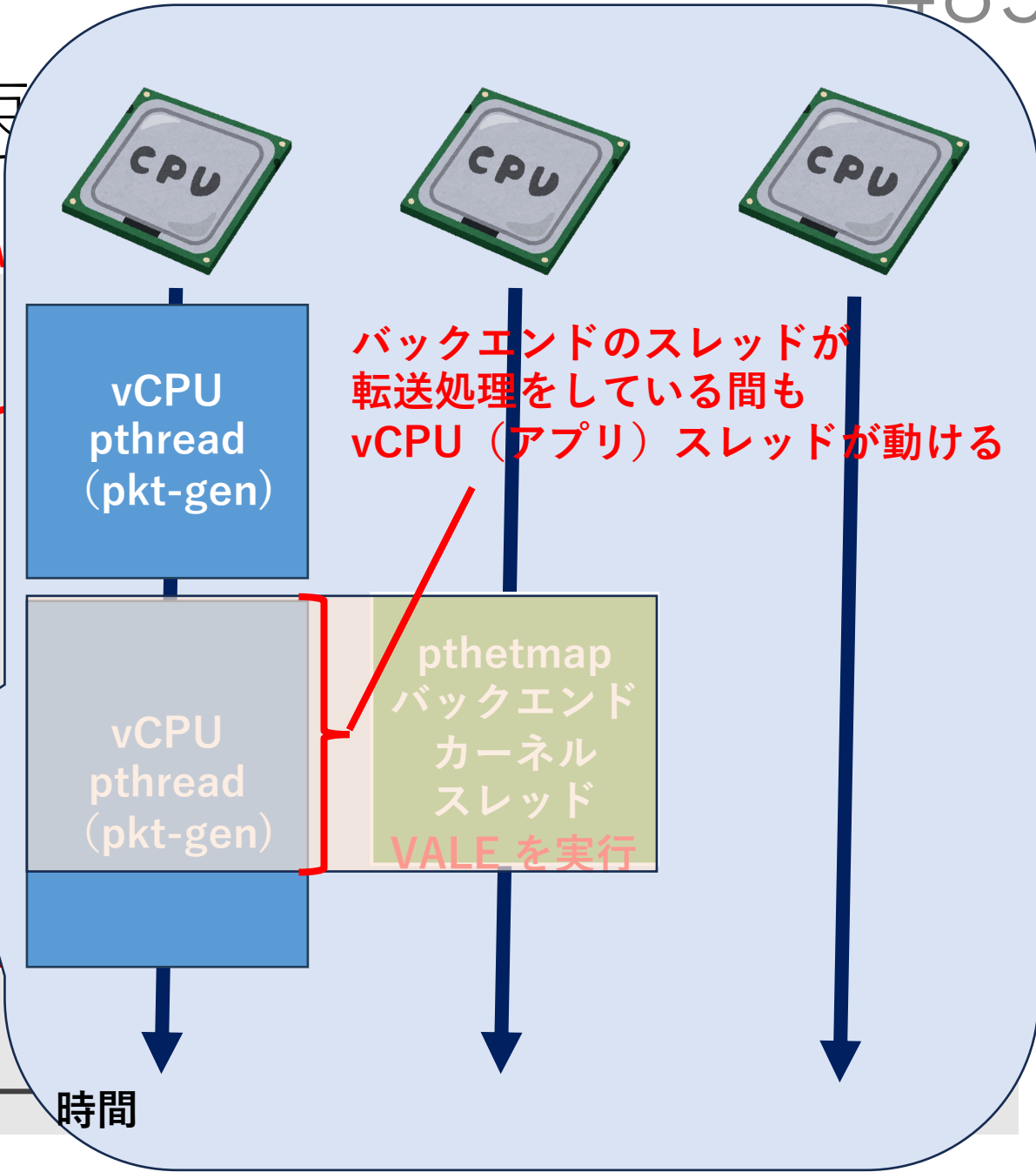


仮想マシン通信の高速

Guest to Host



QEM

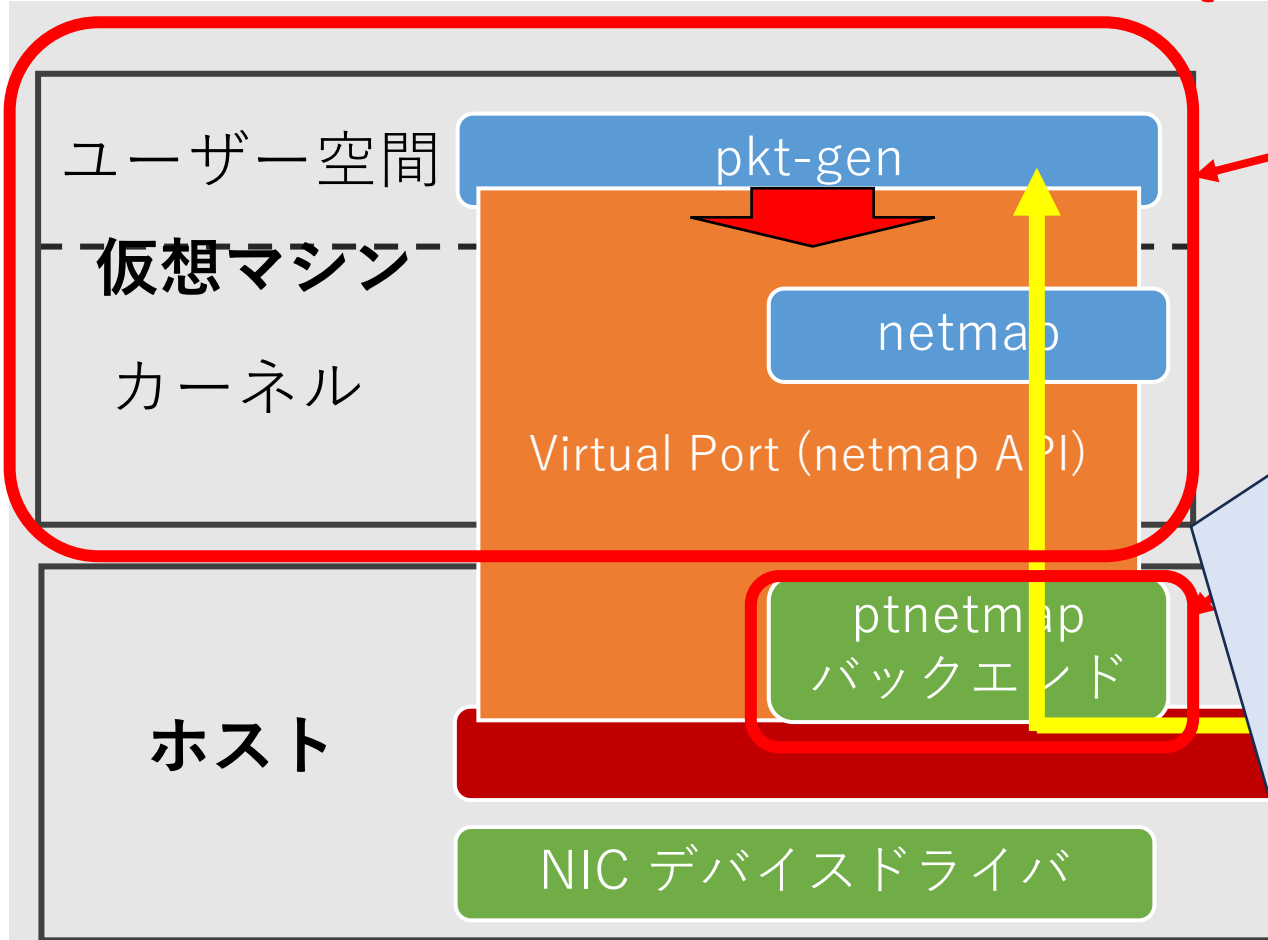


バックエンドのスレッドが
転送処理をしている間も
vCPU (アプリ) スレッドが動ける

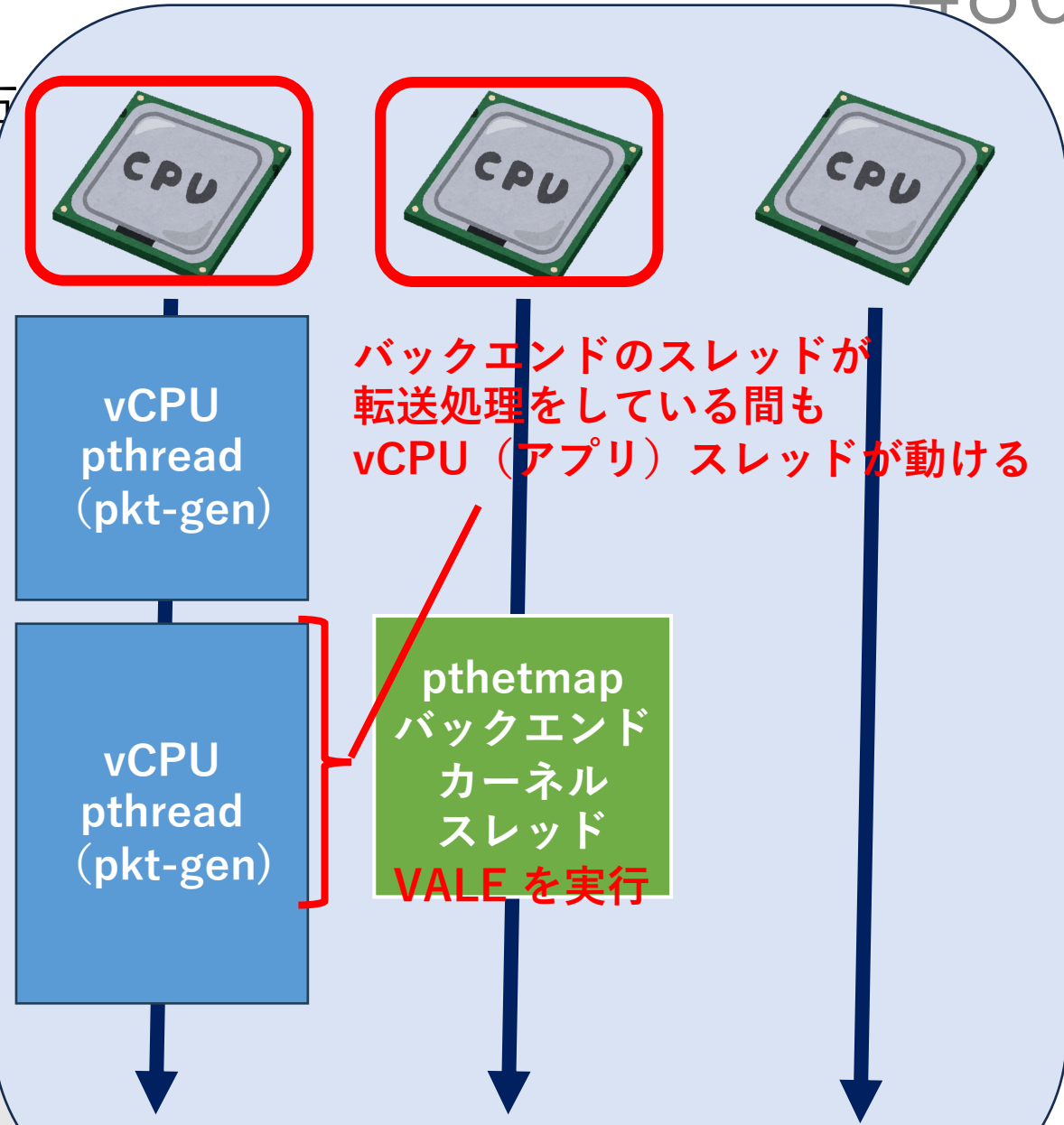
時間

仮想マシン通信の高速

Guest to Host



QEM



バックエンドのスレッドが転送処理をしている間も vCPU (アプリ) スレッドが動ける

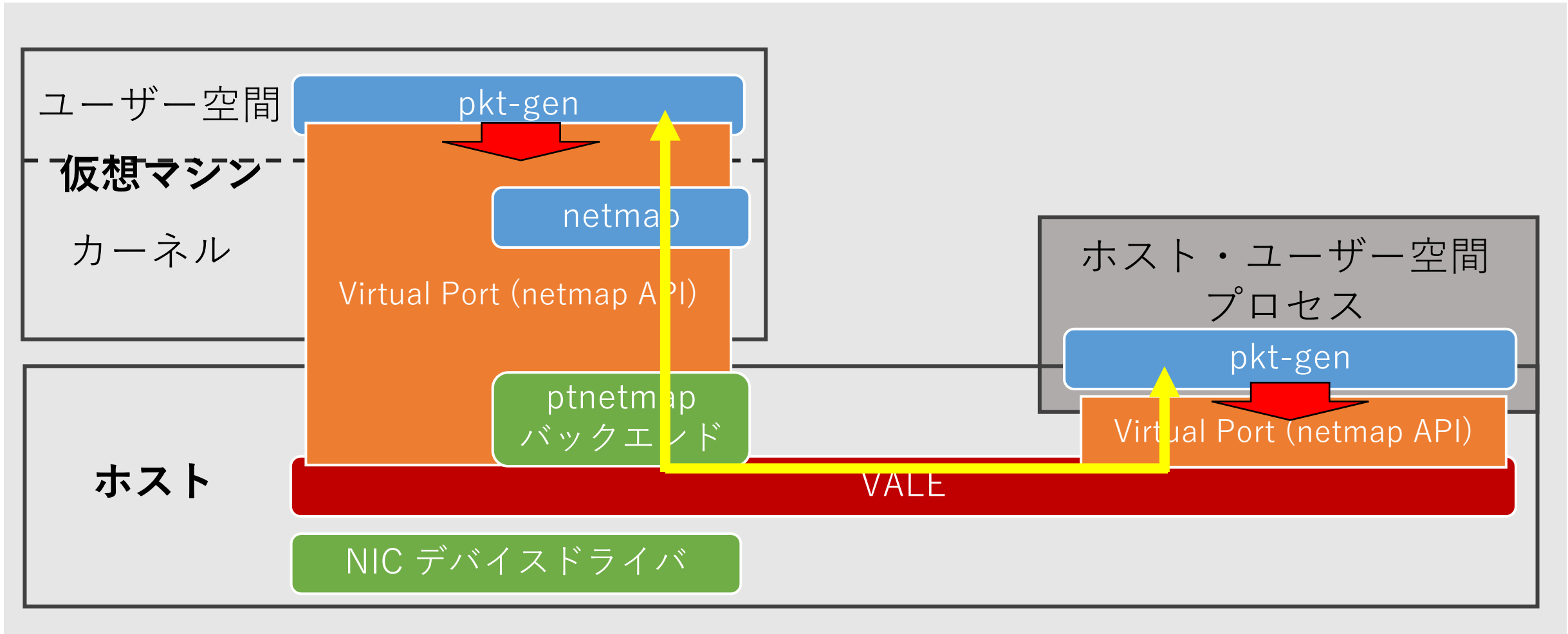
pthreadmap
バックエンド
カーネル
スレッド
VALE を実行

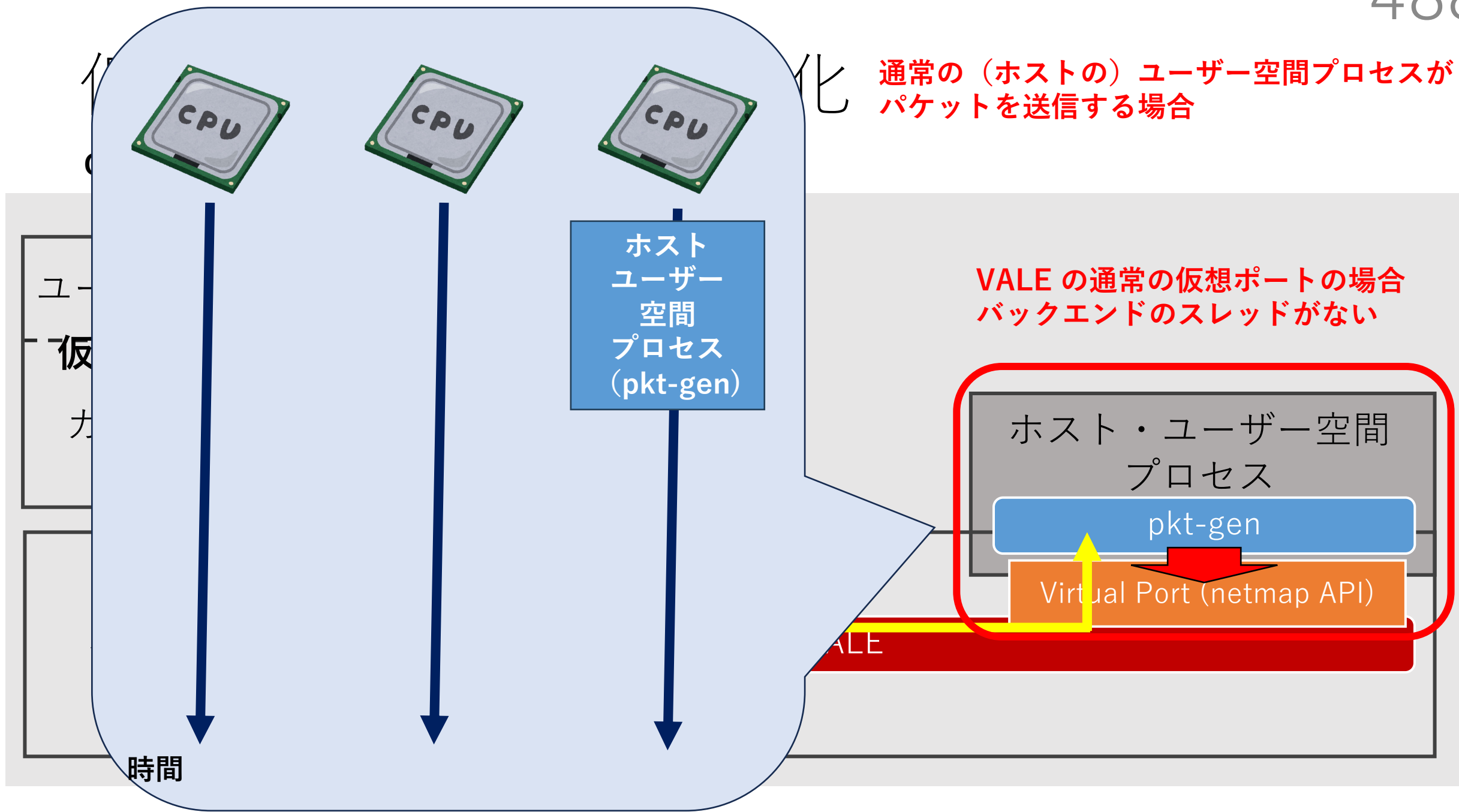
時間

バックエンドのカーネルスレッドのおかげで pkt-gen のために最大 2 CPU コア同時に動く

仮想マシン通信の高速化

Guest to Host





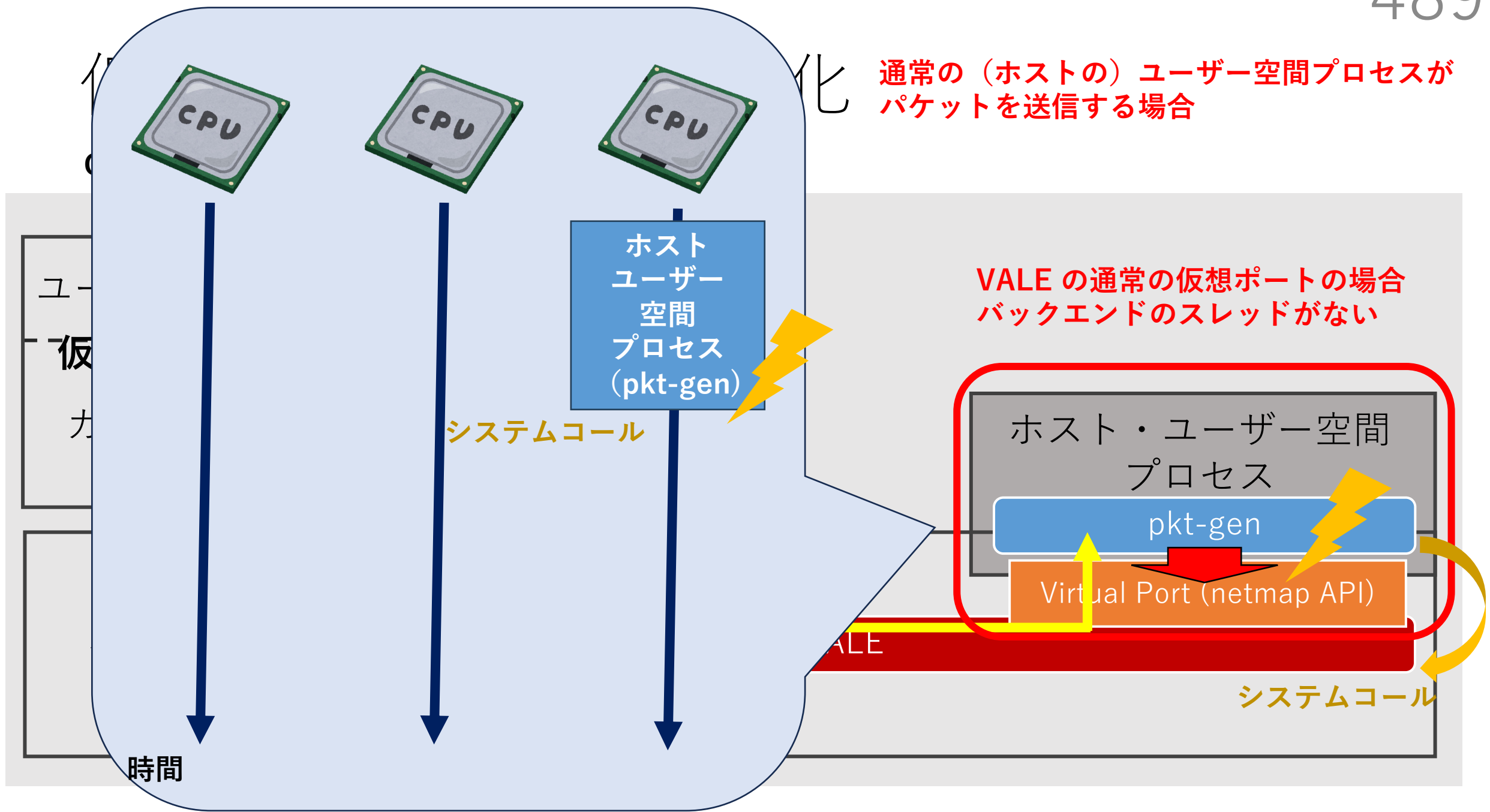
通常の（ホストの）ユーザー空間プロセスが
パケットを送信する場合

VALE の通常の仮想ポートの場合
バックエンドのスレッドがない

時間

ユー
仮
カ

VALE



通常の (ホストの) ユーザー空間プロセスが
パケットを送信する場合

VALE の通常の仮想ポートの場合
バックエンドのスレッドがない

システムコール

ホスト・ユーザー空間
プロセス

pkt-gen

Virtual Port (netmap API)

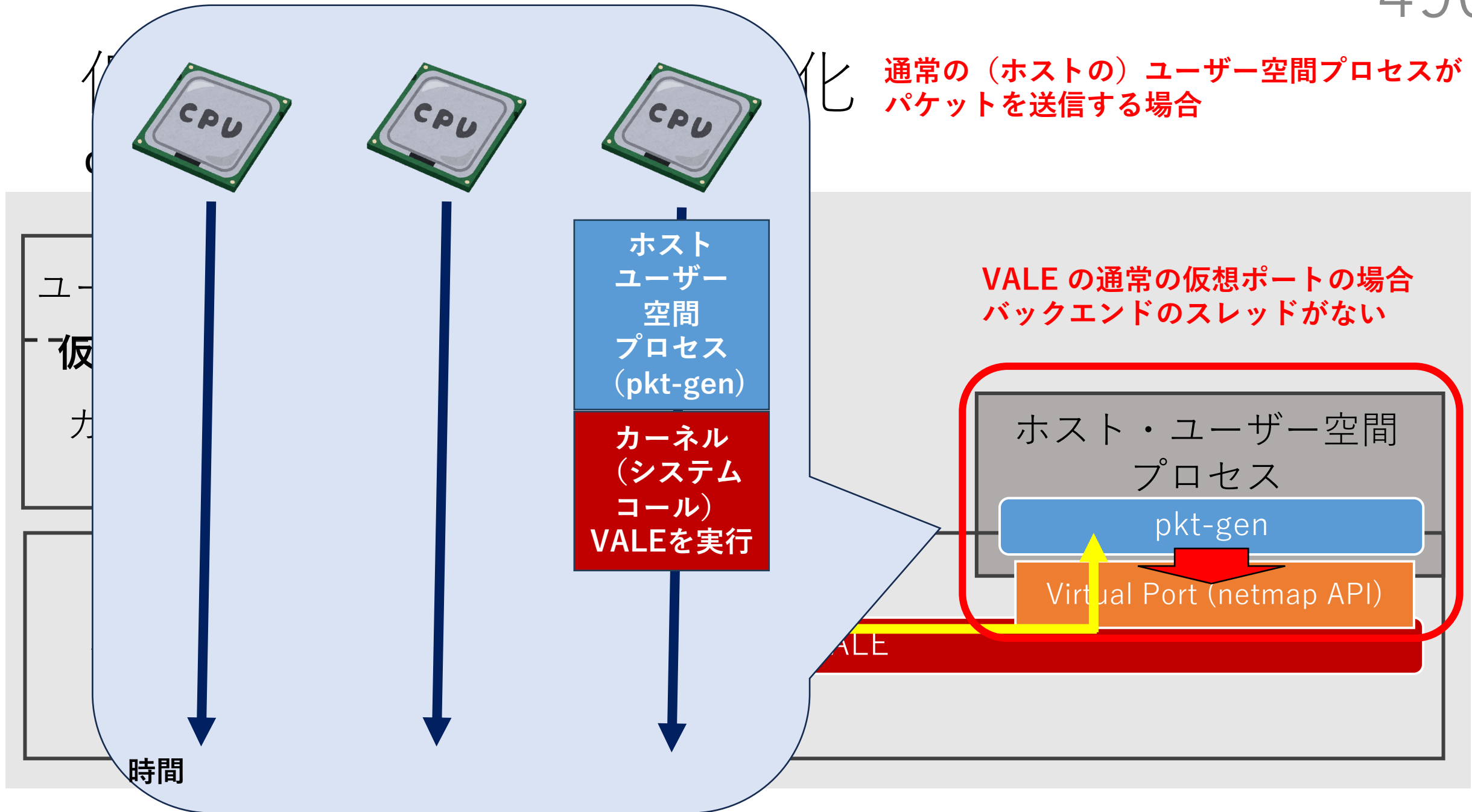
システムコール

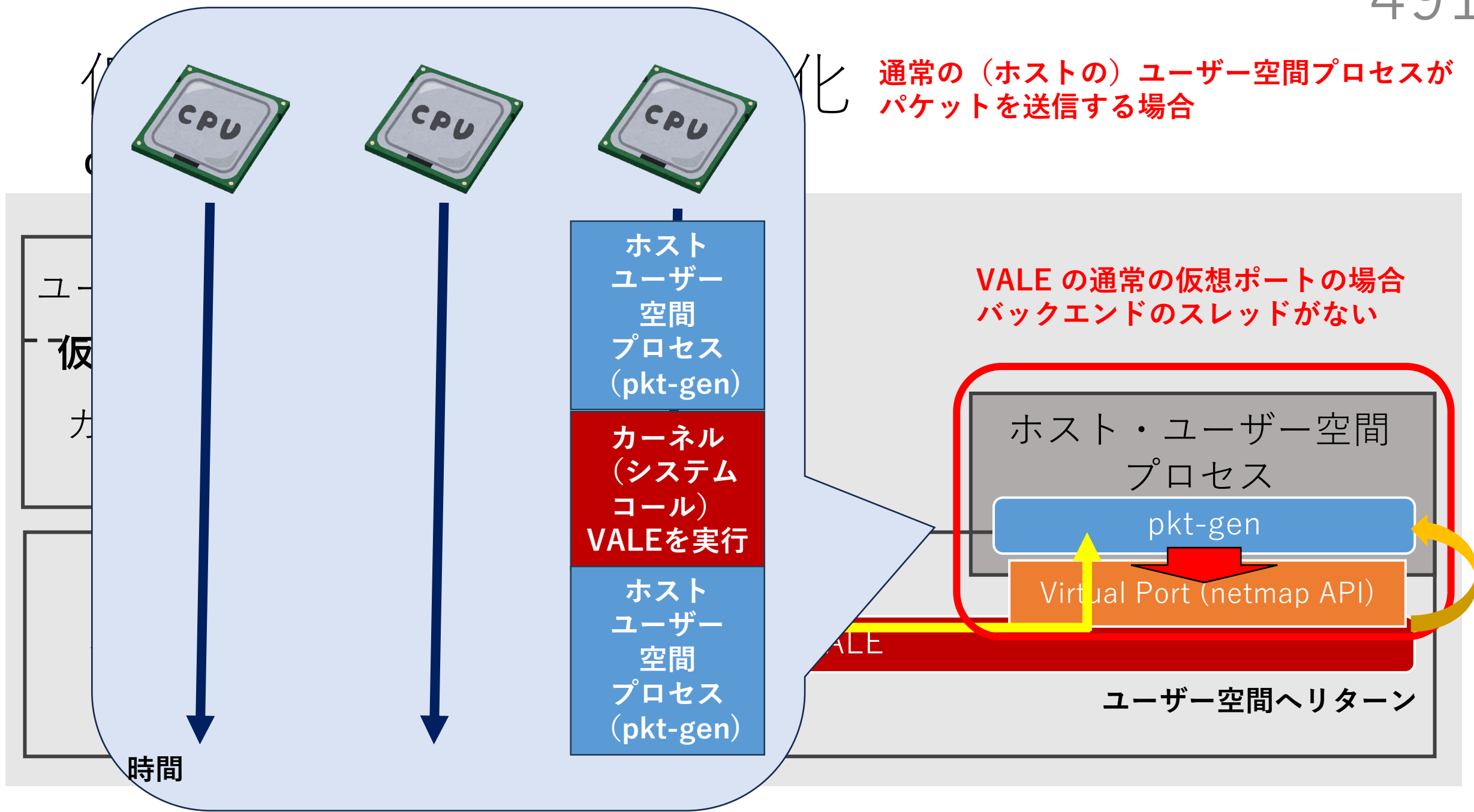
時間

ユー
仮
ナ

化

VALE





通常の（ホストの）ユーザー空間プロセスが
パケットを送信する場合

VALE の通常の仮想ポートの場合
バックエンドのスレッドがない

時間

ホスト
ユーザー
空間
プロセス
(pkt-gen)

カーネル
(システム
コール)
VALEを実行

ホスト
ユーザー
空間
プロセス
(pkt-gen)

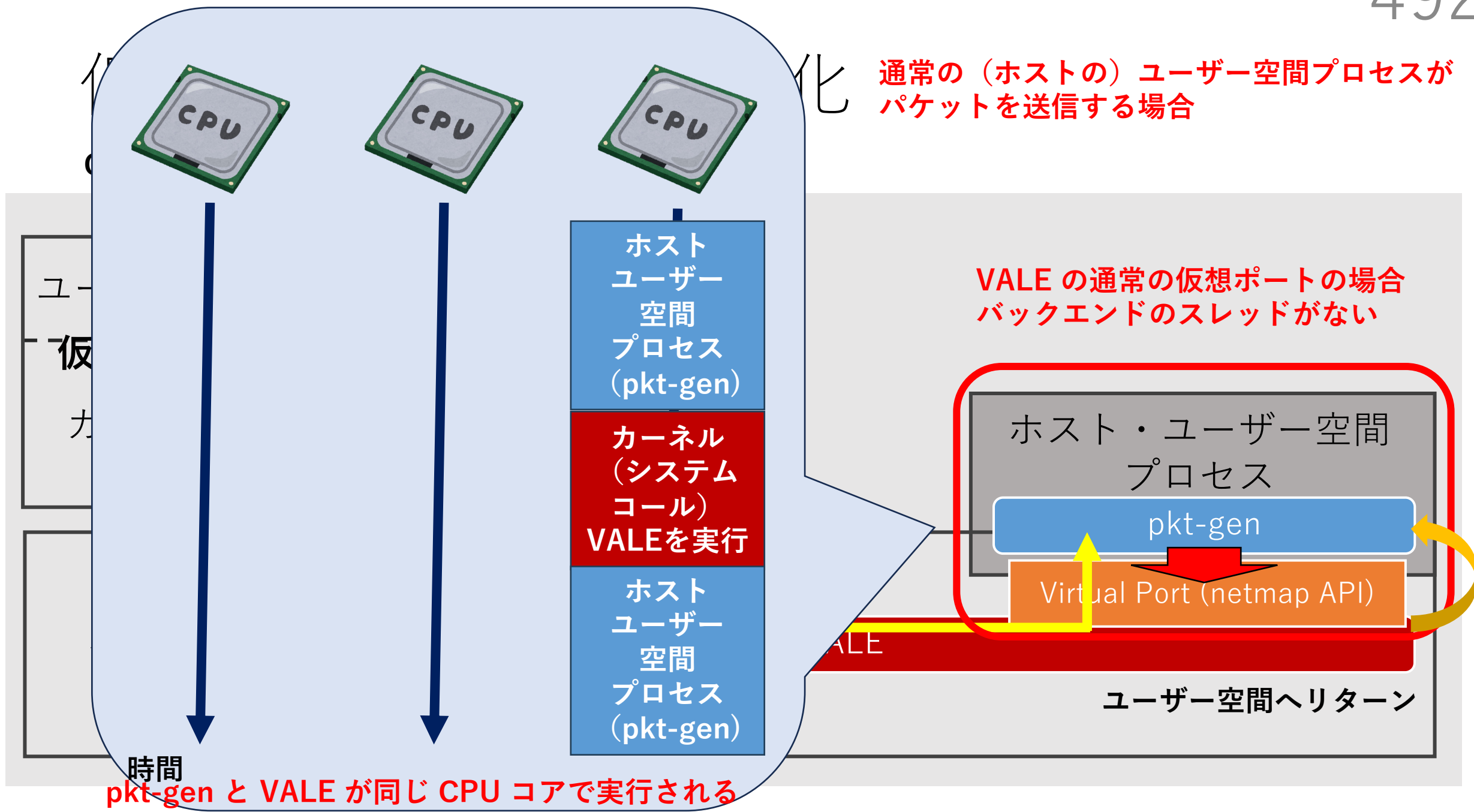
ホスト・ユーザー空間
プロセス

pkt-gen

Virtual Port (netmap API)

ユーザー空間へリターン

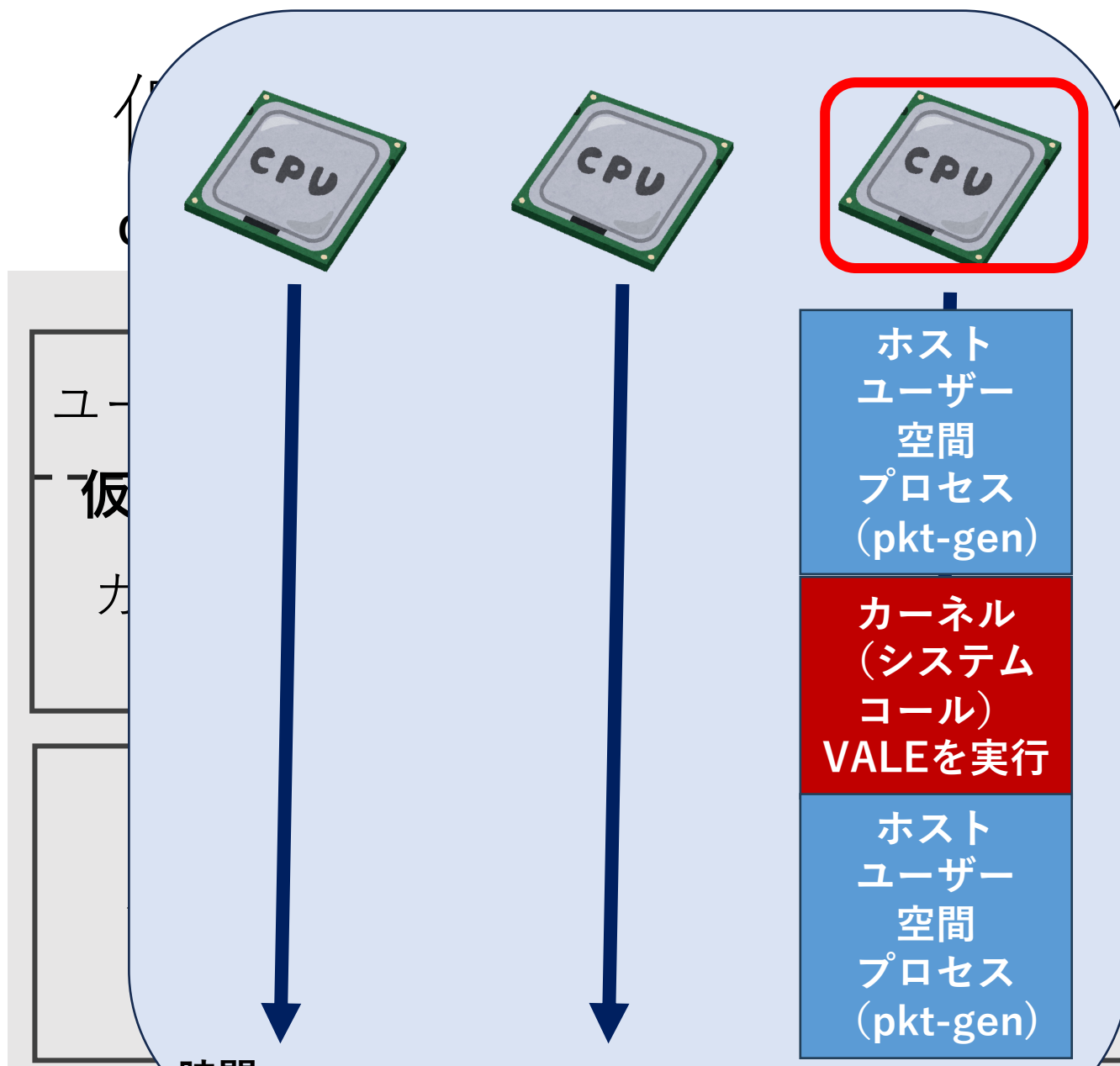
VALE



ユー
仮
カ

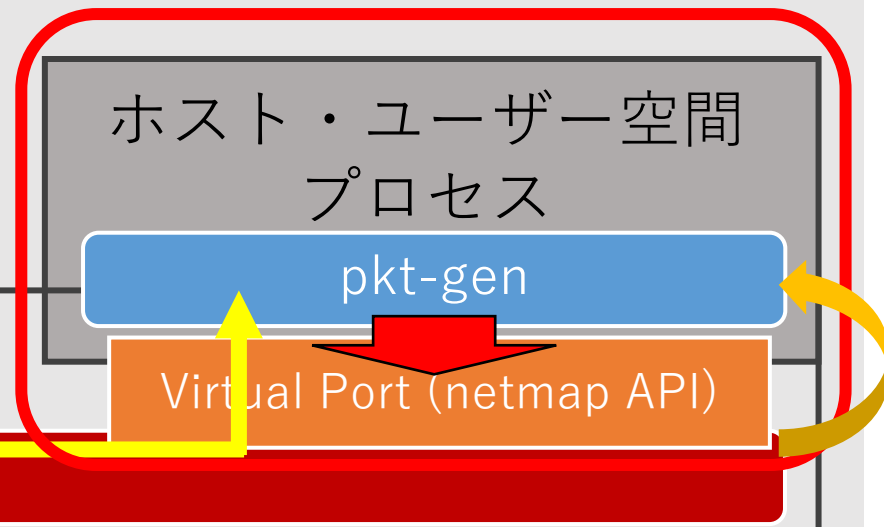
化

VALE



通常の（ホストの）ユーザー空間プロセスが
パケットを送信する場合

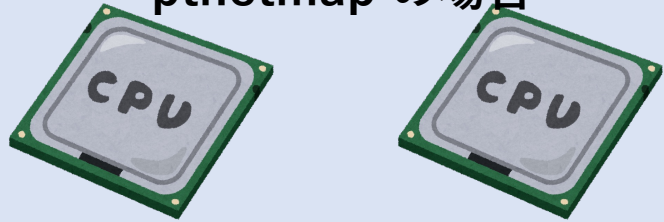
VALE の通常の仮想ポートの場合
バックエンドのスレッドがない



ユーザー空間へリターン

時間
pkt-gen と VALE が同じ CPU コアで実行される：pkt-gen のために最大 1 CPU コアが同時に動く

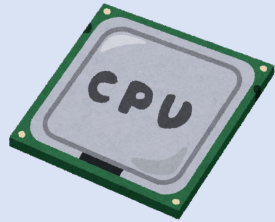
ptnetmap の場合



vCPU pthread (pkt-gen)

vCPU pthread (pkt-gen)

pthreadmap
バックエンド
カーネル
スレッド



ホスト
ユーザー
空間
プロセス
(pkt-gen)

カーネル
(システム
コール)
VALEを実行

ホスト
ユーザー
空間
プロセス
(pkt-gen)

VALE の通常の仮想ポートの場合
バックエンドのスレッドがない

ホスト・ユーザー空間
プロセス

pkt-gen

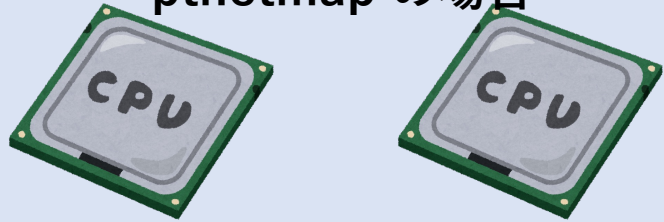
Virtual Port (netmap API)

化

ユー
- 仮
カ

時間

ptnetmap の場合

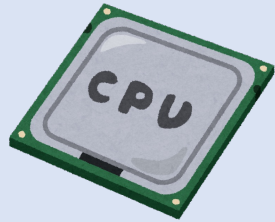


vCPU pthread (pkt-gen)

pthread (pkt-gen)

最大 2 CPUコアを同時に利用できるから速い

pthreadman スレッド



ホスト ユーザー空間 プロセス (pkt-gen)

カーネル (システムコール) VALEを実行

ホスト ユーザー空間 プロセス (pkt-gen)

VALE の通常の仮想ポートの場合 バックエンドのスレッドがない

ホスト・ユーザー空間 プロセス

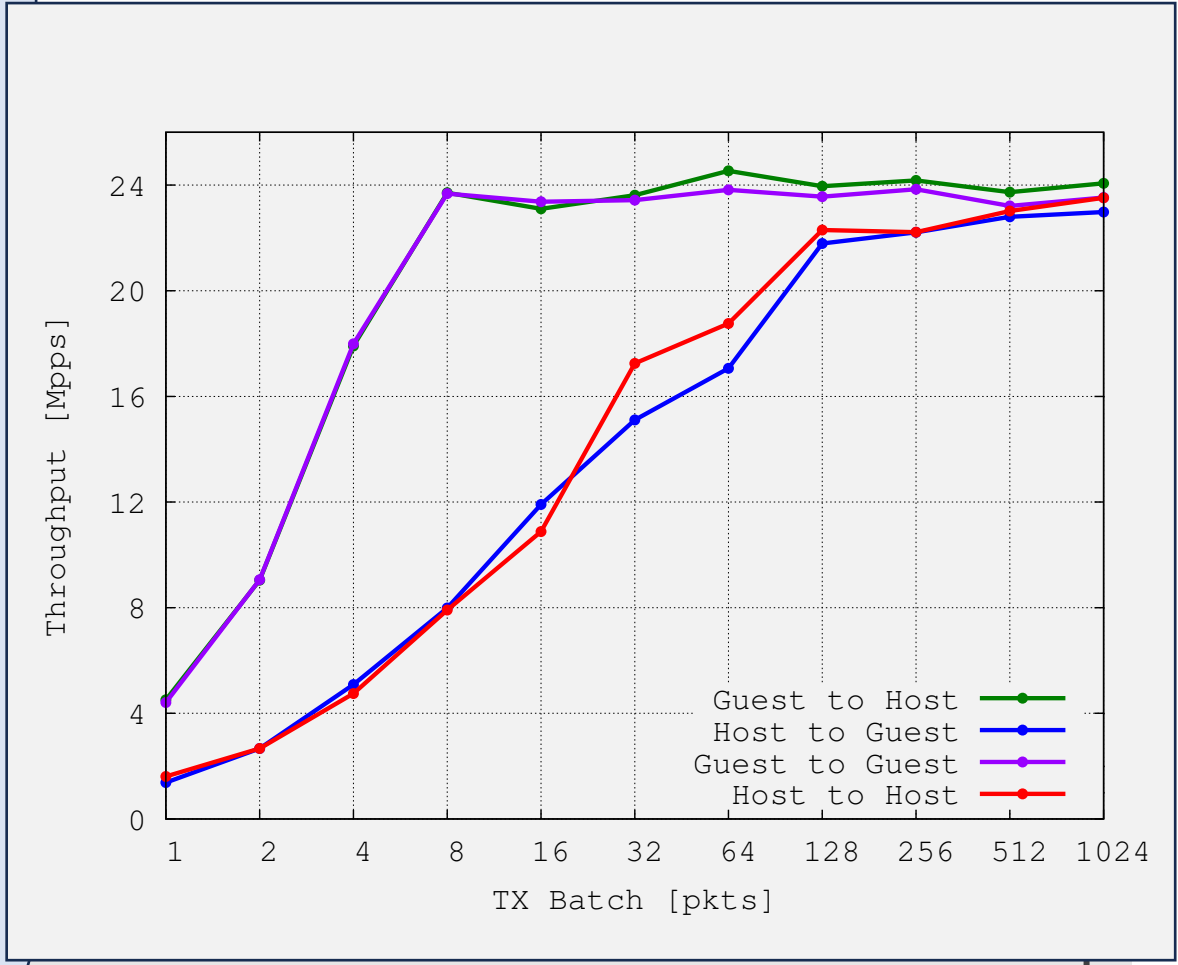
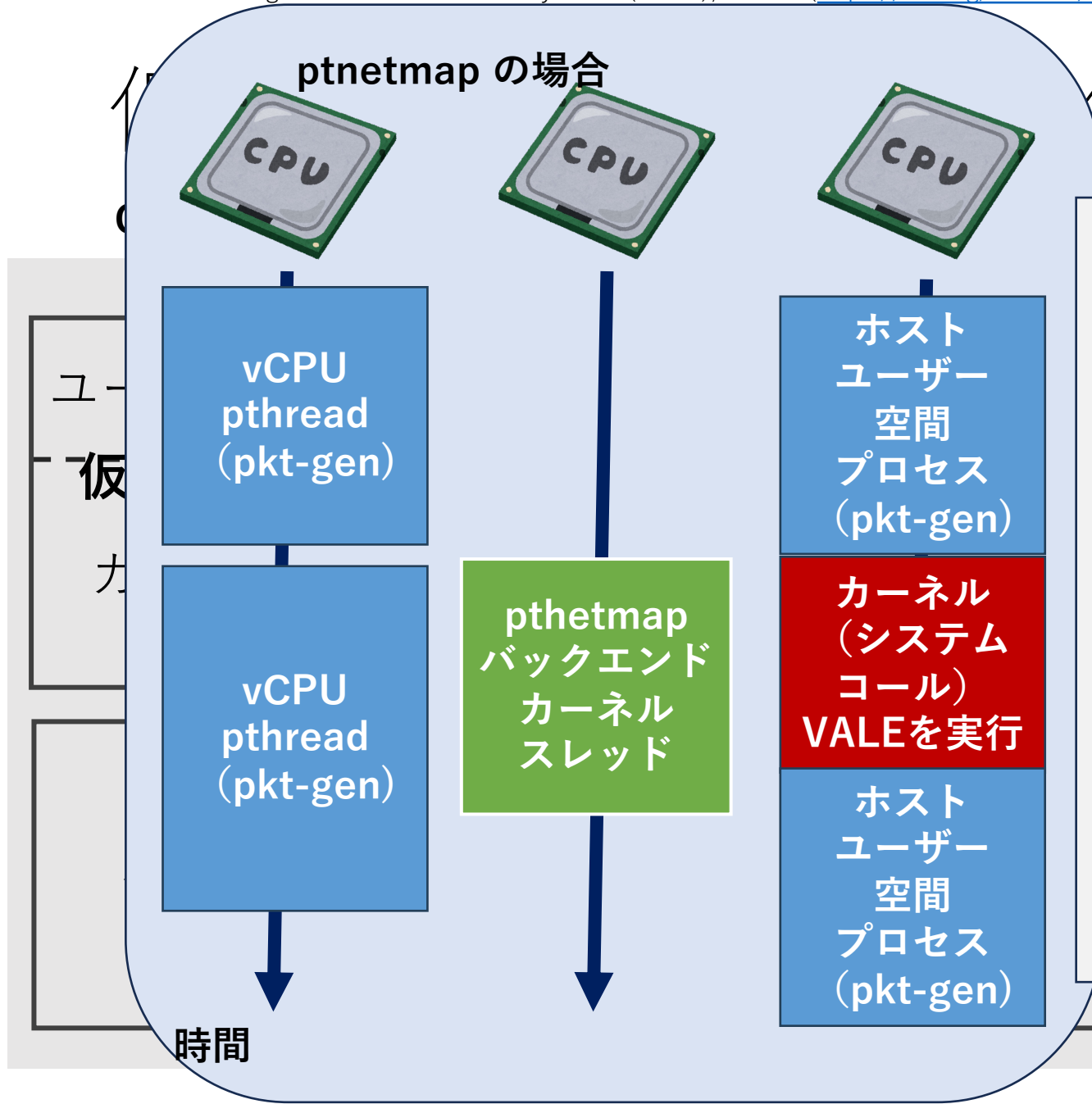
pkt-gen

Virtual Port (netmap API)

化

ユー
仮
カ

時間



研究紹介

仮想マシン通信について

仮想 I/O 実行方式の改善

仮想マシン通信の高速化

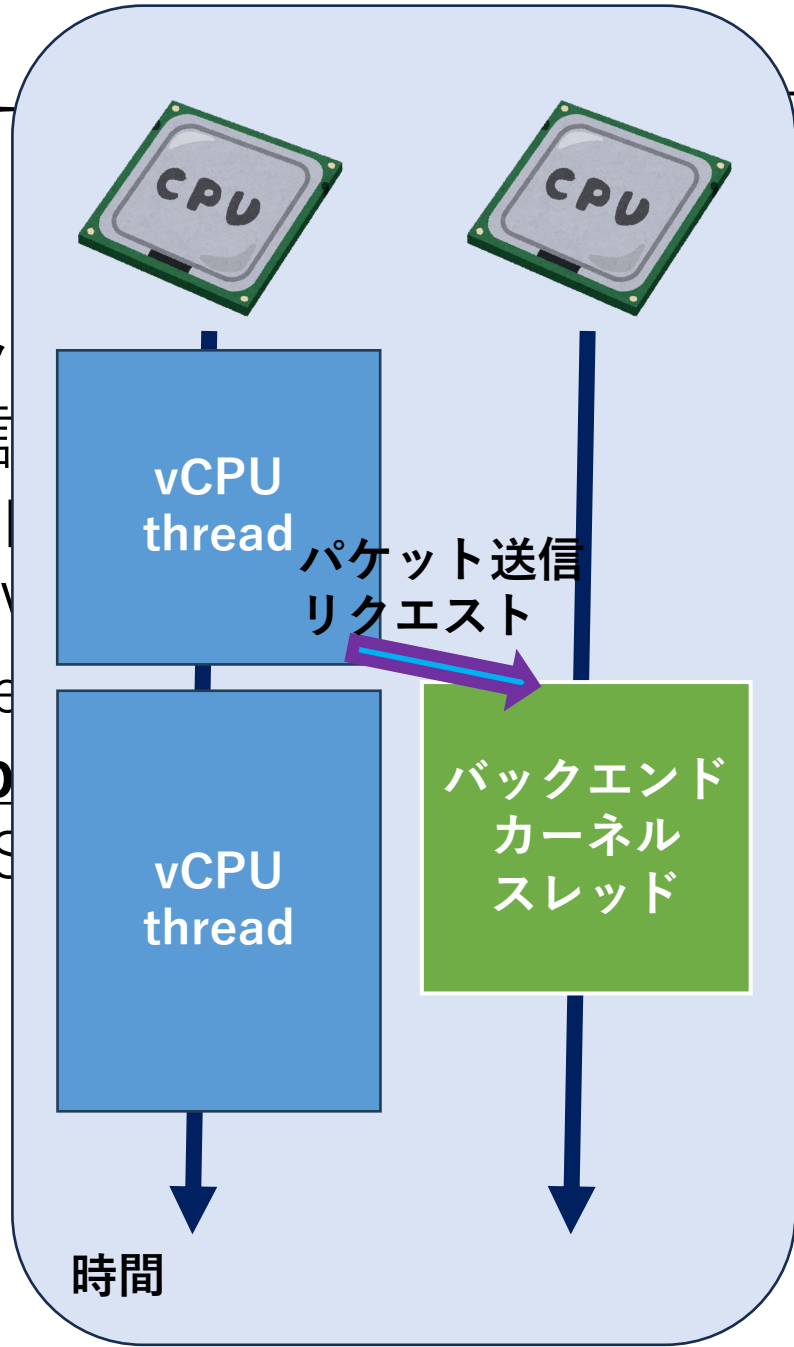
- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014) ← Xen に netmap/VALE を適用
 - NetVM (NSDI 2014) ← QEMU/KVM に DPDK を適用
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017) ← Xen に netmap/VALE を適用
 - ELISA (ASPLOS 2023)

仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014) ← Xen に netmap/VALE を適用
 - NetVM (NSDI 2014) ← QEMU/KVM に DPDK を適用
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - **HyperNF (SoCC 2017)** ← Xen に netmap/VALE を適用
 - ELISA (ASPLOS 2023) **実行のモデルを改善**

仮想マシン加速化

- パッケージ通信
 - Click
 - NetV
 - ptne
 - **Hyp**
 - ELIS



を使った仮想スイッチを仮想マシン

Xen に netmap/VALE を適用

QEMU/KVM に DPDK を適用

(AN 2016)

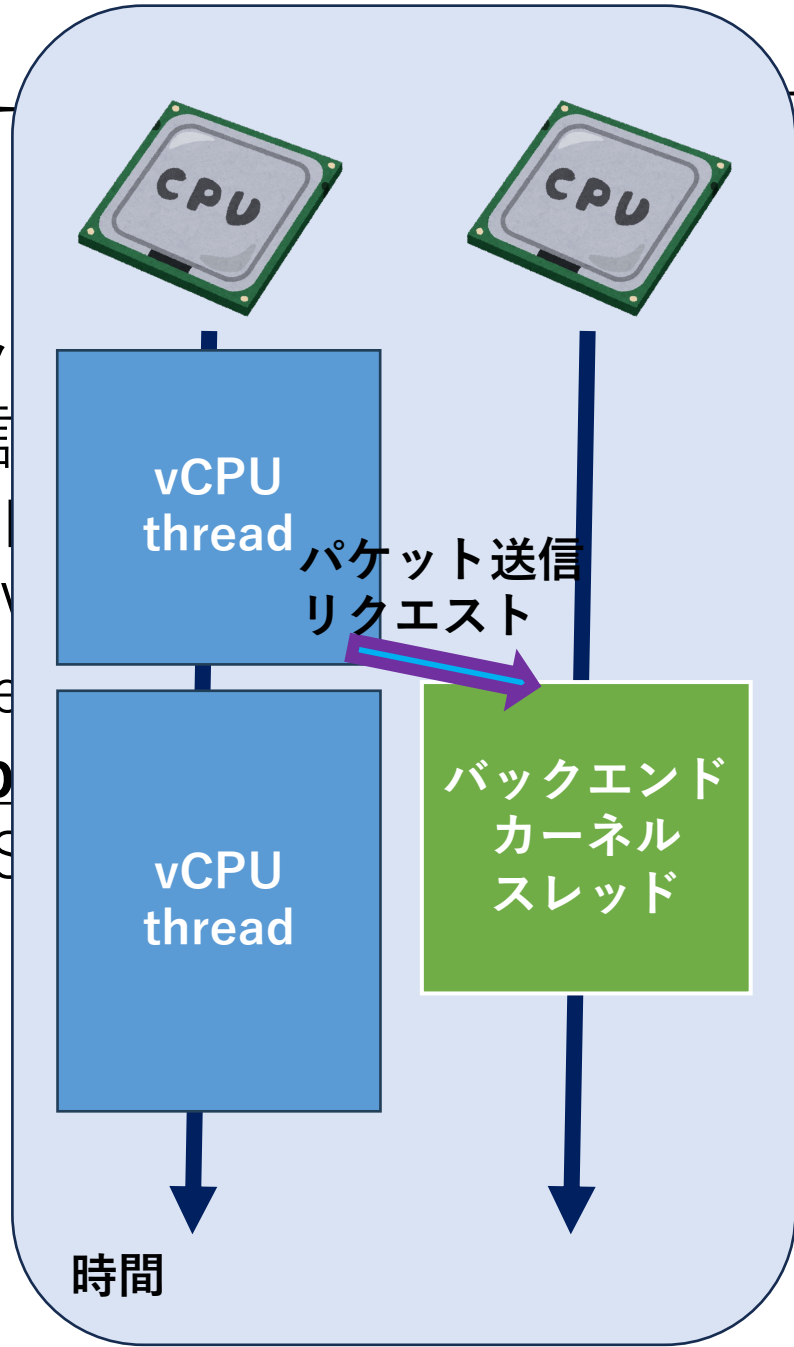
Xen に netmap/VALE を適用

実行のモデルを改善

仮想

速化

- パケット通信
 - Click
 - NetV
 - ptne
 - **Hyp**
 - ELIS



を使った仮想スイッチを仮想マシン

Xen に netmap/VALE を適用

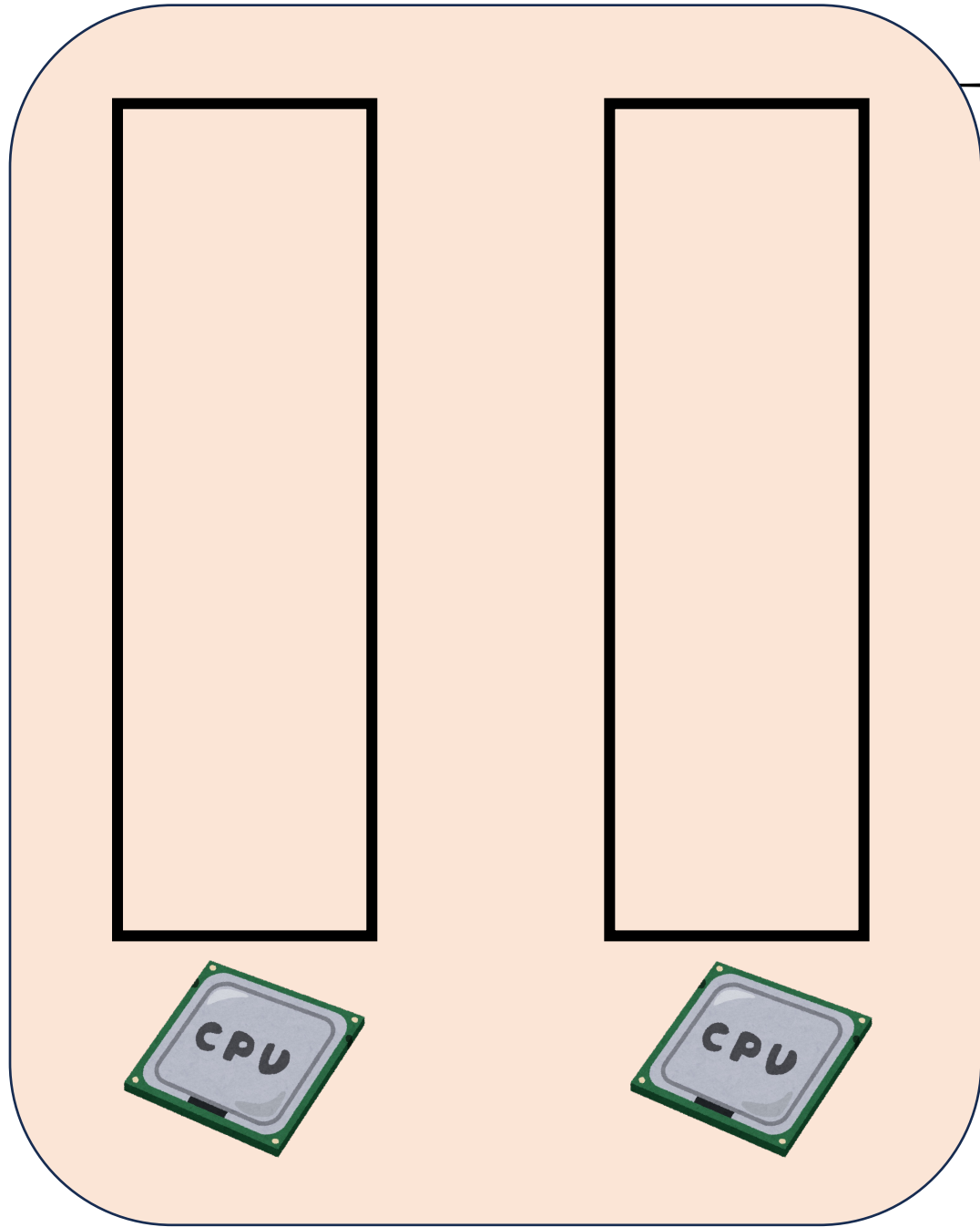
QEMU/KVM に DPDK を適用

(AN 2016)

Xen に netmap/VALE を適用

実行のモデルを改善

主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い



高速化

を使った仮想スイッチを仮想マシ

— Xen に netmap/VALE を適用

— QEMU/KVM に DPDK を適用

(AN 2016)

— Xen に netmap/VALE を適用

実行のモデルを改善

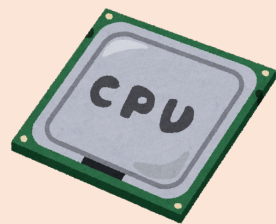
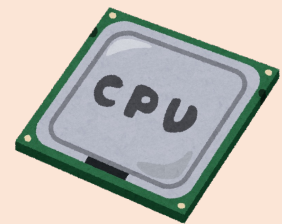
**主張：vCPU スレッドと仮想スイッチを実行する
バックエンドのスレッドを分けない方が良い**

高速化

を使った仮想スイッチを仮想マシン

CPU 時間
100 %

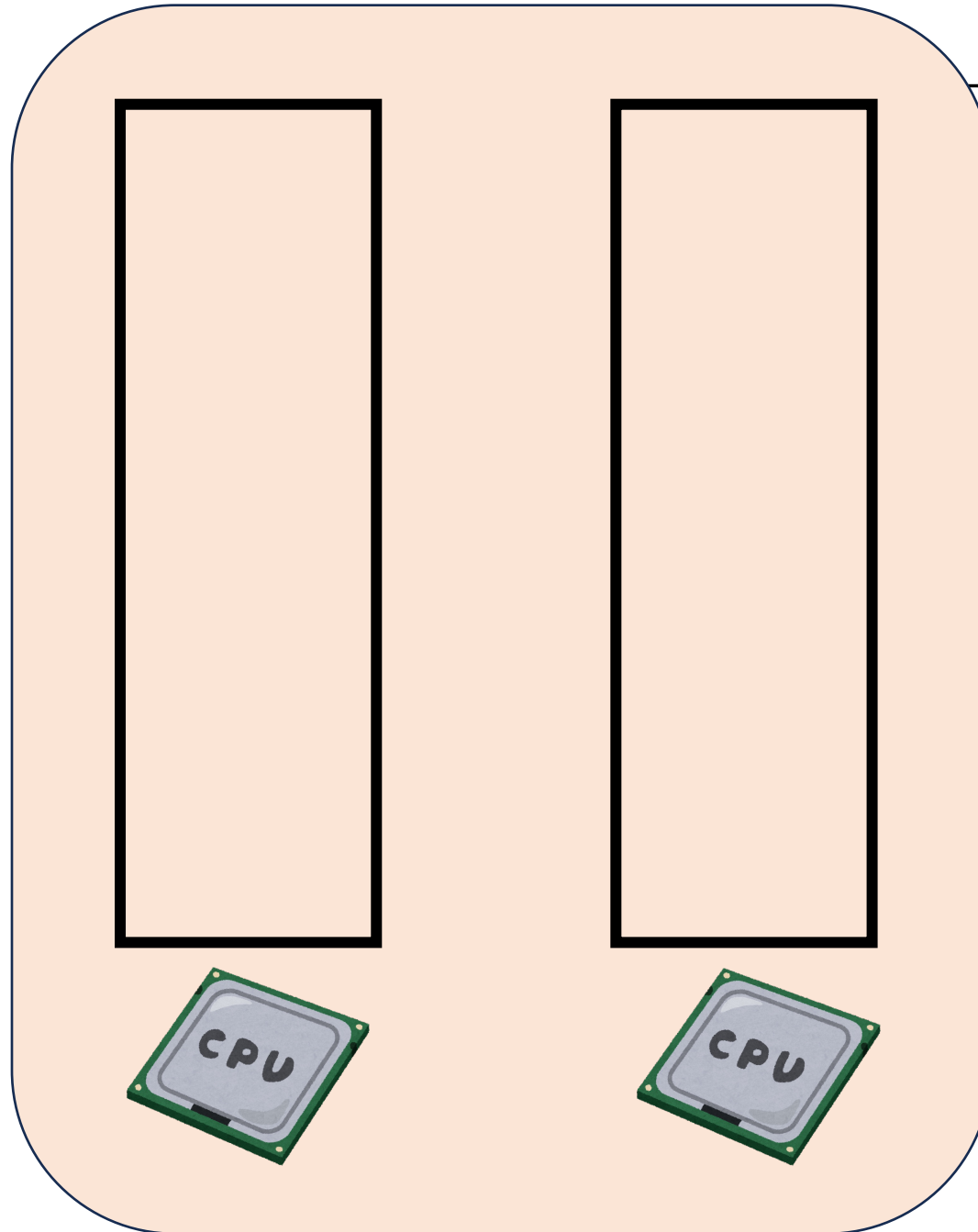
CPU 時間
100 %



- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

**主張：vCPU スレッドと仮想スイッチを実行する
バックエンドのスレッドを分けない方が良い**



高速化

を使った仮想スイッチを仮想マシ

— Xen に netmap/VALE を適用

— QEMU/KVM に DPDK を適用

(AN 2016)

— Xen に netmap/VALE を適用

実行のモデルを改善

**主張：vCPU スレッドと仮想スイッチを実行する
バックエンドのスレッドを分けない方が良い**

理想的には性能のために CPU は最大限利用できるべき

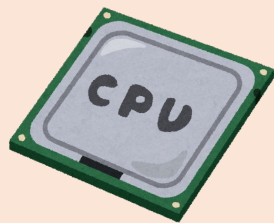
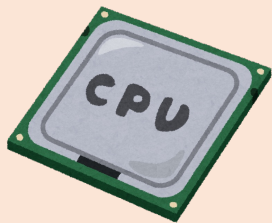
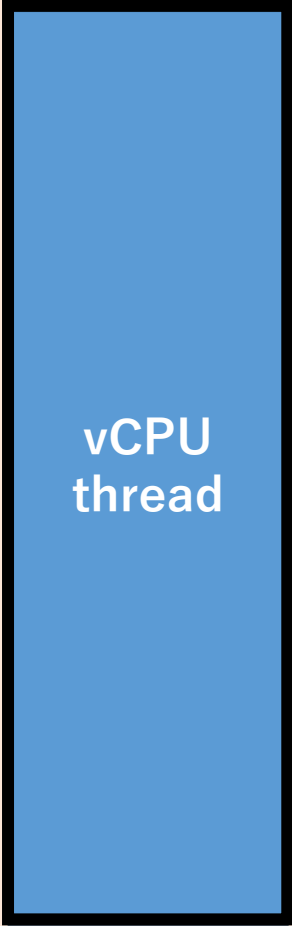
高速化

を使った仮想スイッチを仮想マシン

理想

vCPU thread

カーネルスレッド



- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

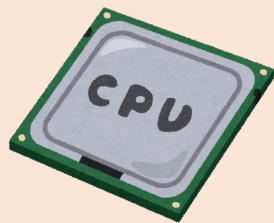
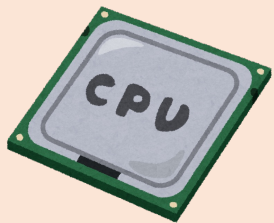
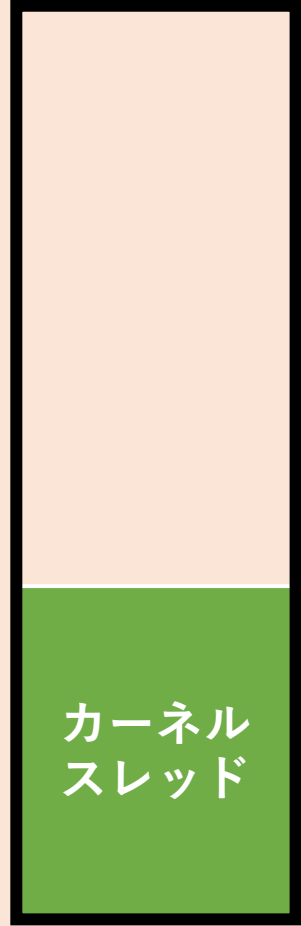
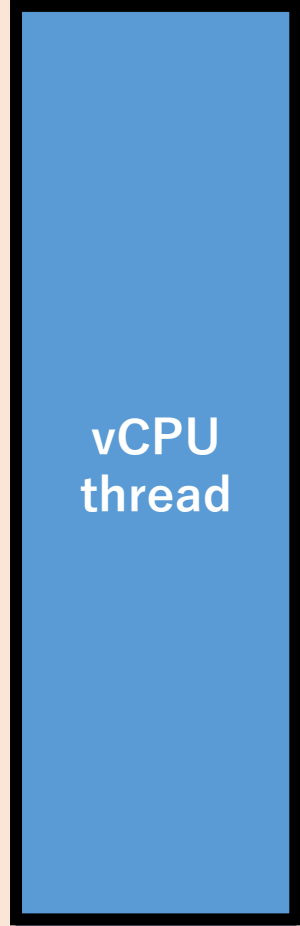
主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

実際は、vCPU かバックエンドのスレッドどちらかが常にボトルネックになる
(ワークロード依存)

高速化

を使った仮想スイッチを仮想マシ

実際



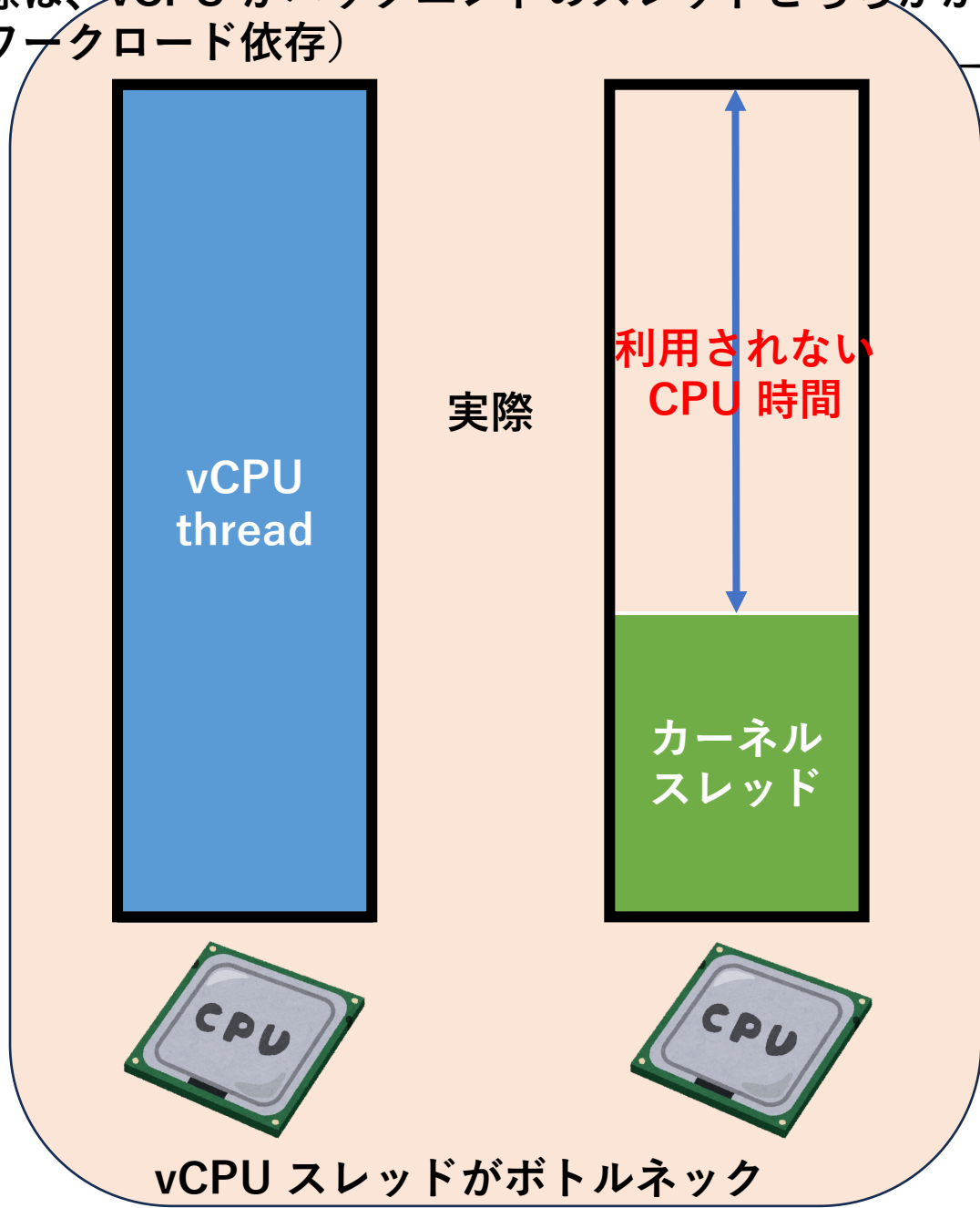
vCPU スレッドがボトルネック

- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

実際は、vCPU かバックエンドのスレッドどちらかが常にボトルネックになる
(ワークロード依存)



高速化

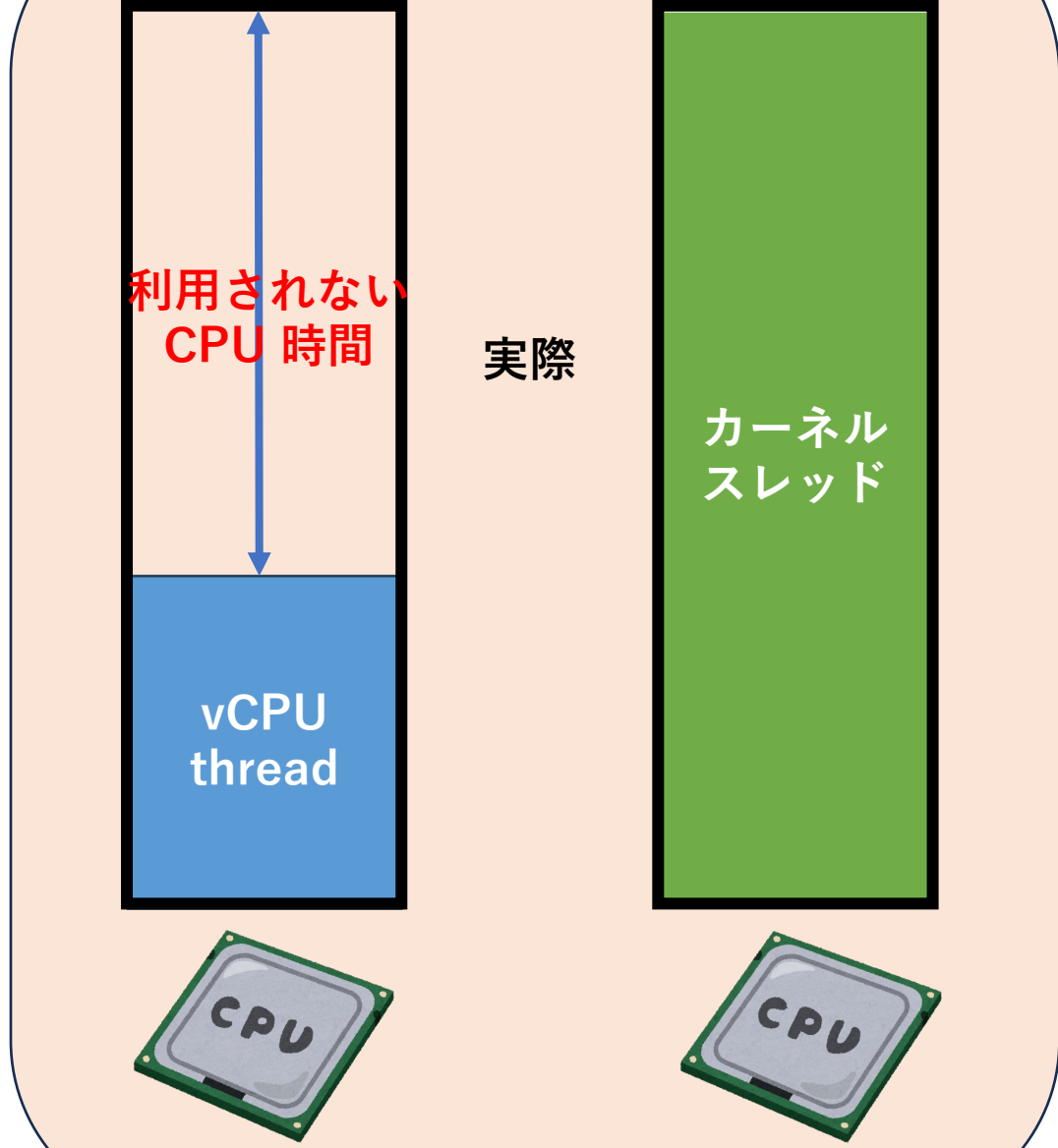
を使った仮想スイッチを仮想マシ

- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

主張：vCPU スレッドと仮想スイッチを実行する
バックエンドのスレッドを分けない方が良い

実際は、vCPU かバックエンドのスレッドどちらかが常にボトルネックになる
(ワークロード依存)



高速化

を使った仮想スイッチを仮想マシ

- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

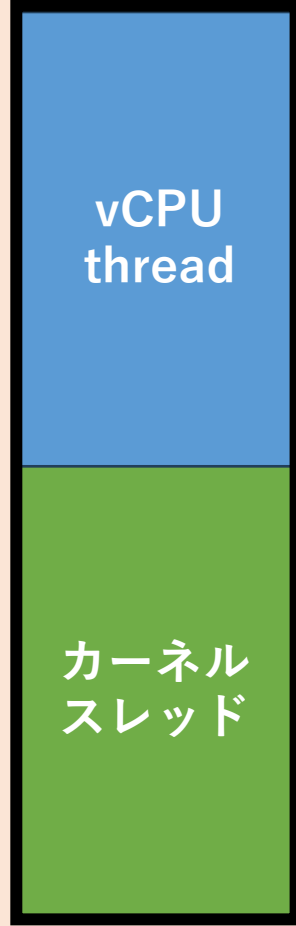
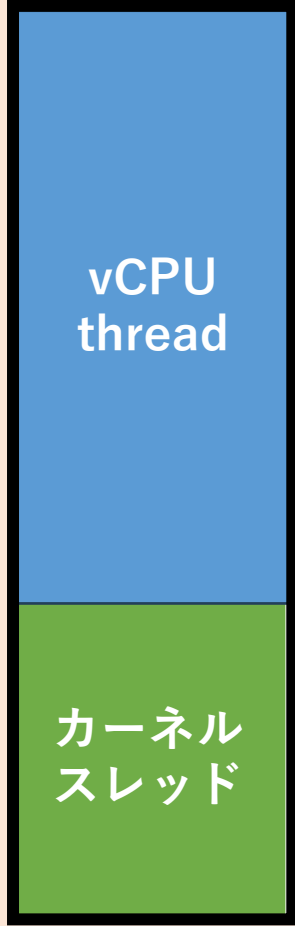
主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

バックエンドのカーネルスレッドがボトルネック

解決策：vCPUとバックエンドのカーネルスレッドは同じCPUコアの上で動かす

高速化

を使った仮想スイッチを仮想マシ



Xen に netmap/VALE を適用

QEMU/KVM に DPDK を適用

(AN 2016)

Xen に netmap/VALE を適用

実行のモデルを改善

主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

解決策：vCPUとバックエンドのカーネルスレッドは同じCPUコアの上で動かす

利点：ワークロードが変化しても、CPUのvCPUとバックエンドへの割り当てが常に理想的

高速化

を使った仮想スイッチを仮想マシン

Xenにnetmap/VALEを適用

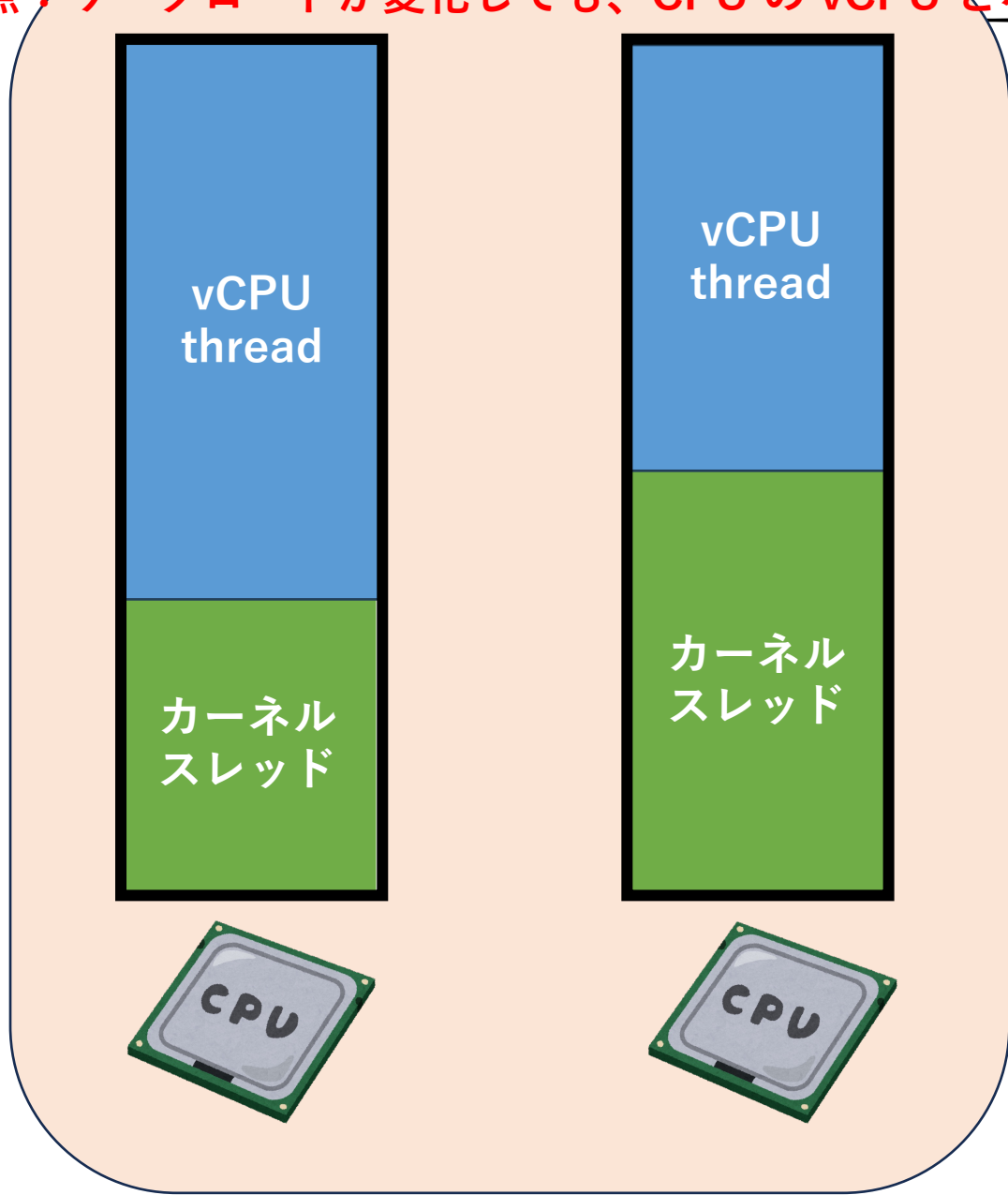
QEMU/KVMにDPDKを適用

(AN 2016)

Xenにnetmap/VALEを適用

実行のモデルを改善

主張：vCPUスレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い



解決策：vCPUとバックエンドのカーネルスレッドは同じCPUコアの上で動かす

利点：ワークロードが変化しても、CPUのvCPUとバックエンドへの割り当てが常に理想的

高速化

を使った仮想スイッチを仮想マシン

Xenにnetmap/VALEを適用

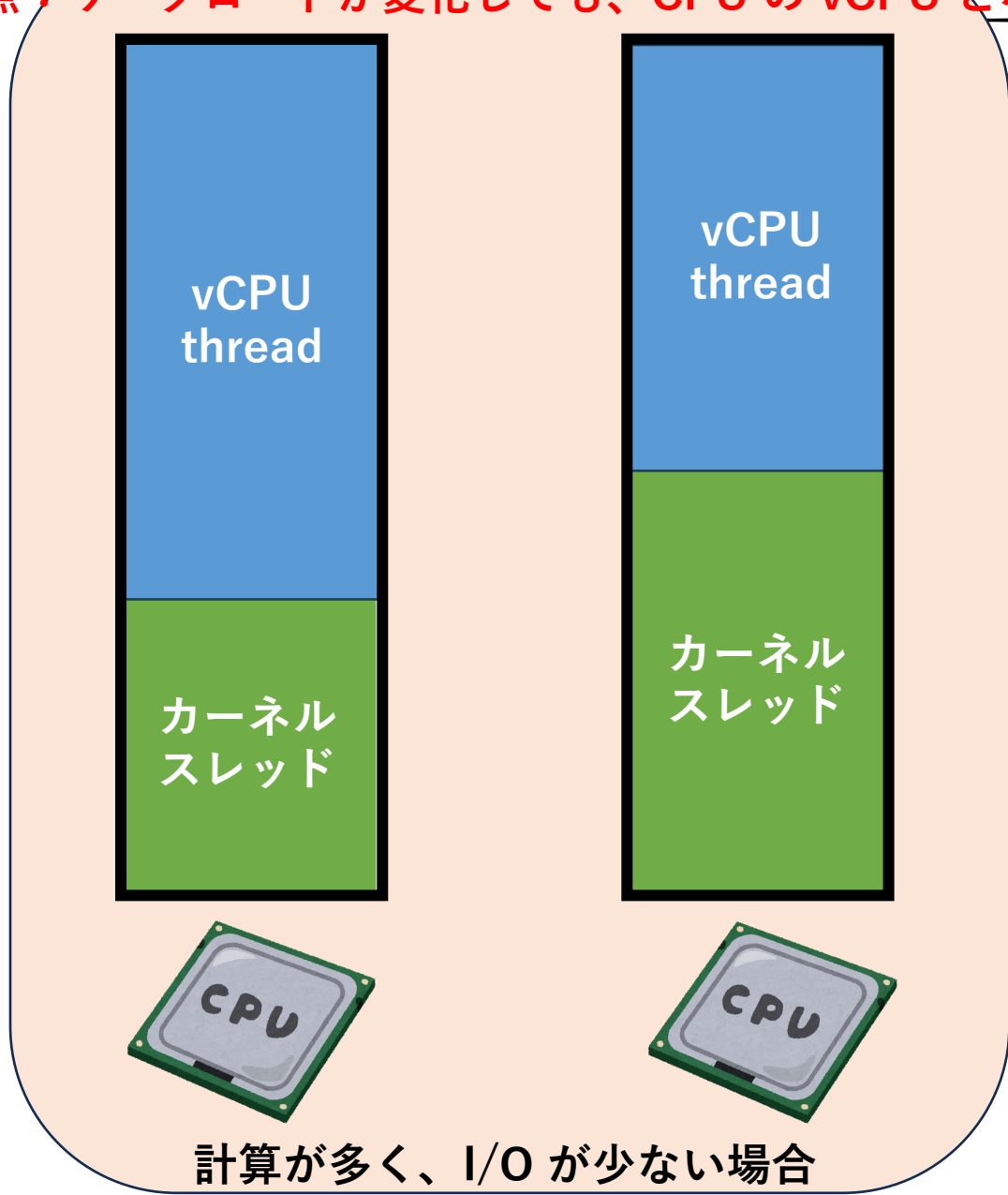
QEMU/KVMにDPDKを適用

(AN 2016)

Xenにnetmap/VALEを適用

実行のモデルを改善

主張：vCPUスレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い



計算が多く、I/Oが少ない場合

解決策：vCPUとバックエンドのカーネルスレッドは同じCPUコアの上で動かす

利点：ワークロードが変化しても、CPUのvCPUとバックエンドへの割り当てが常に理想的

高速化

を使った仮想スイッチを仮想マシン

Xenにnetmap/VALEを適用

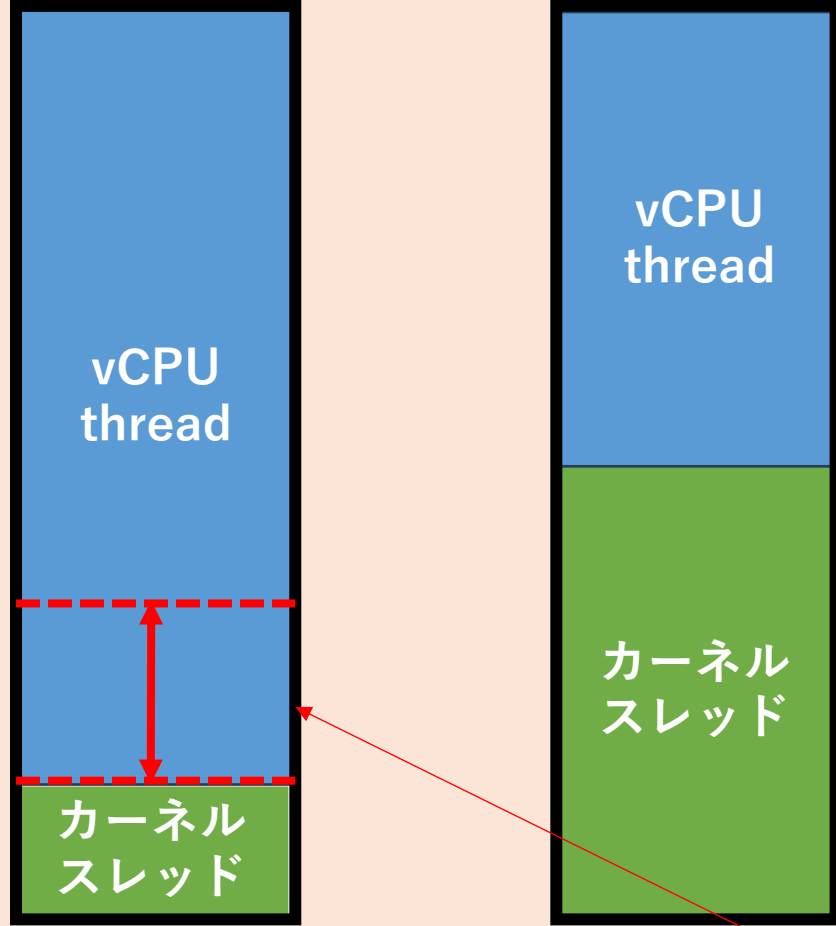
QEMU/KVMにDPDKを適用

(AN 2016)

Xenにnetmap/VALEを適用

実行のモデルを改善

主張：vCPUスレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い



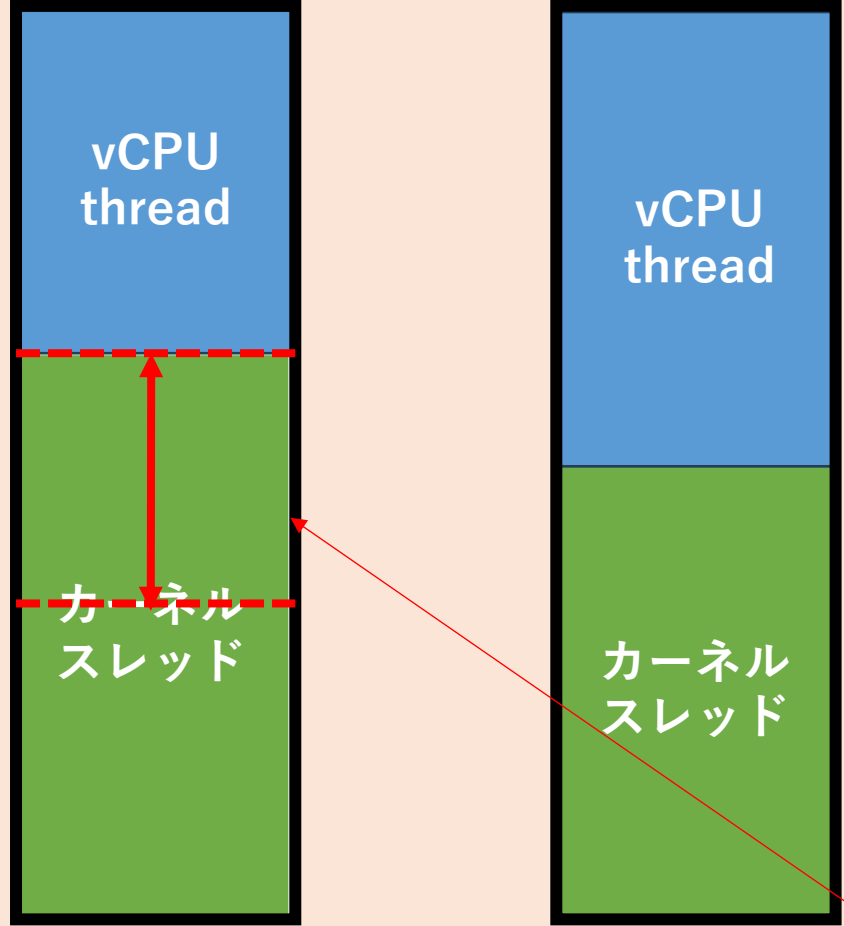
計算が少なく、I/Oが多い場合：バックエンドに利用されないCPU時間をvCPUが使える

解決策：vCPUとバックエンドのカーネルスレッドは同じCPUコアの上で動かす

利点：ワークロードが変化しても、CPUのvCPUとバックエンドへの割り当てが常に理想的

高速化

を使った仮想スイッチを仮想マシン



- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

実行のモデルを改善

主張：vCPUスレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

計算が多く、I/Oが少ない場合：vCPUに利用されないCPU時間をバックエンドが使える

解決策：vCPU とバックエンドのカーネルスレッドは同じ CPU コア
利点：ワークロードが変化しても、CPU の vCPU とバックエンド

高速化

を使った

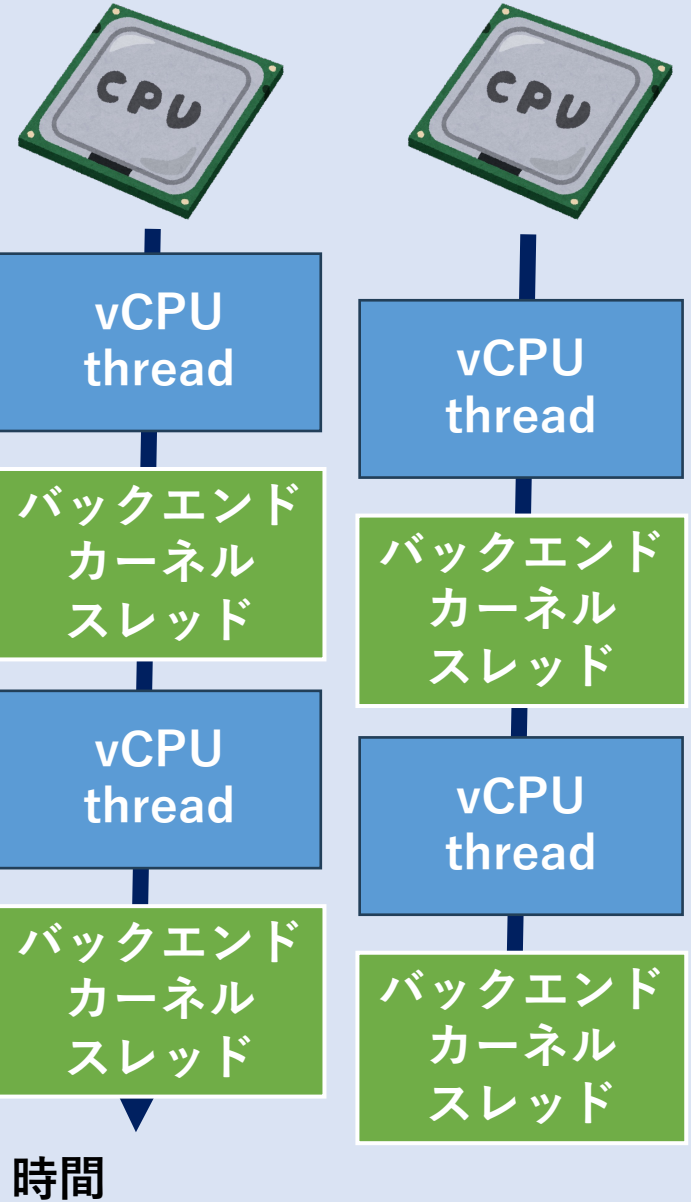
Xen

QEM

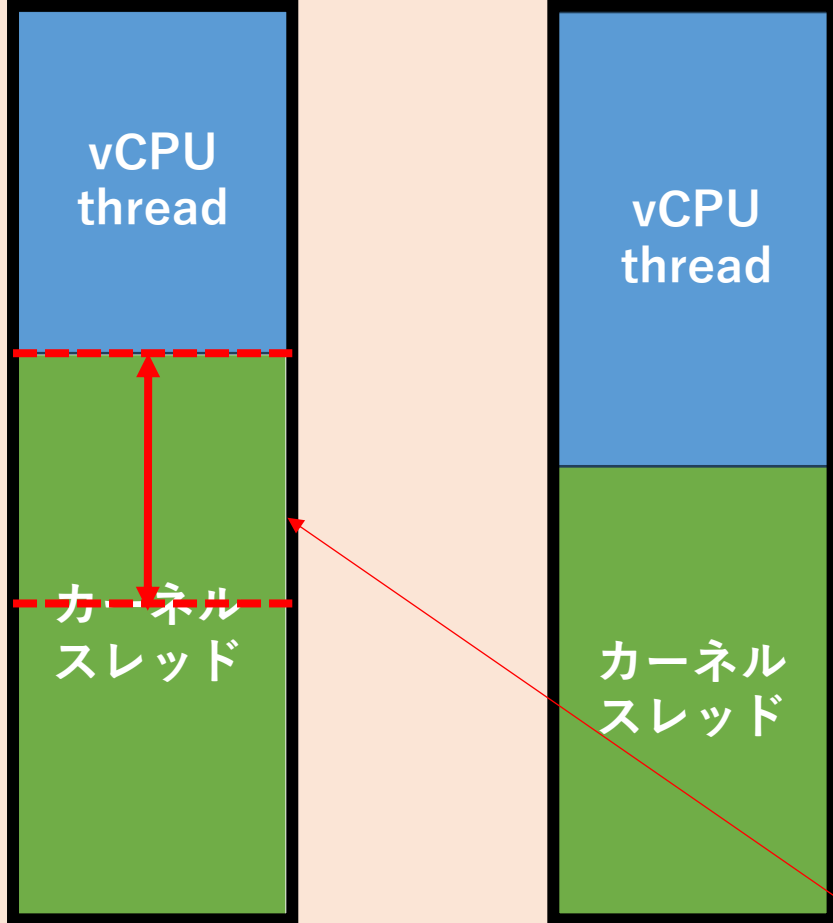
(AN 2016)

主張：バック

この解決策の課題



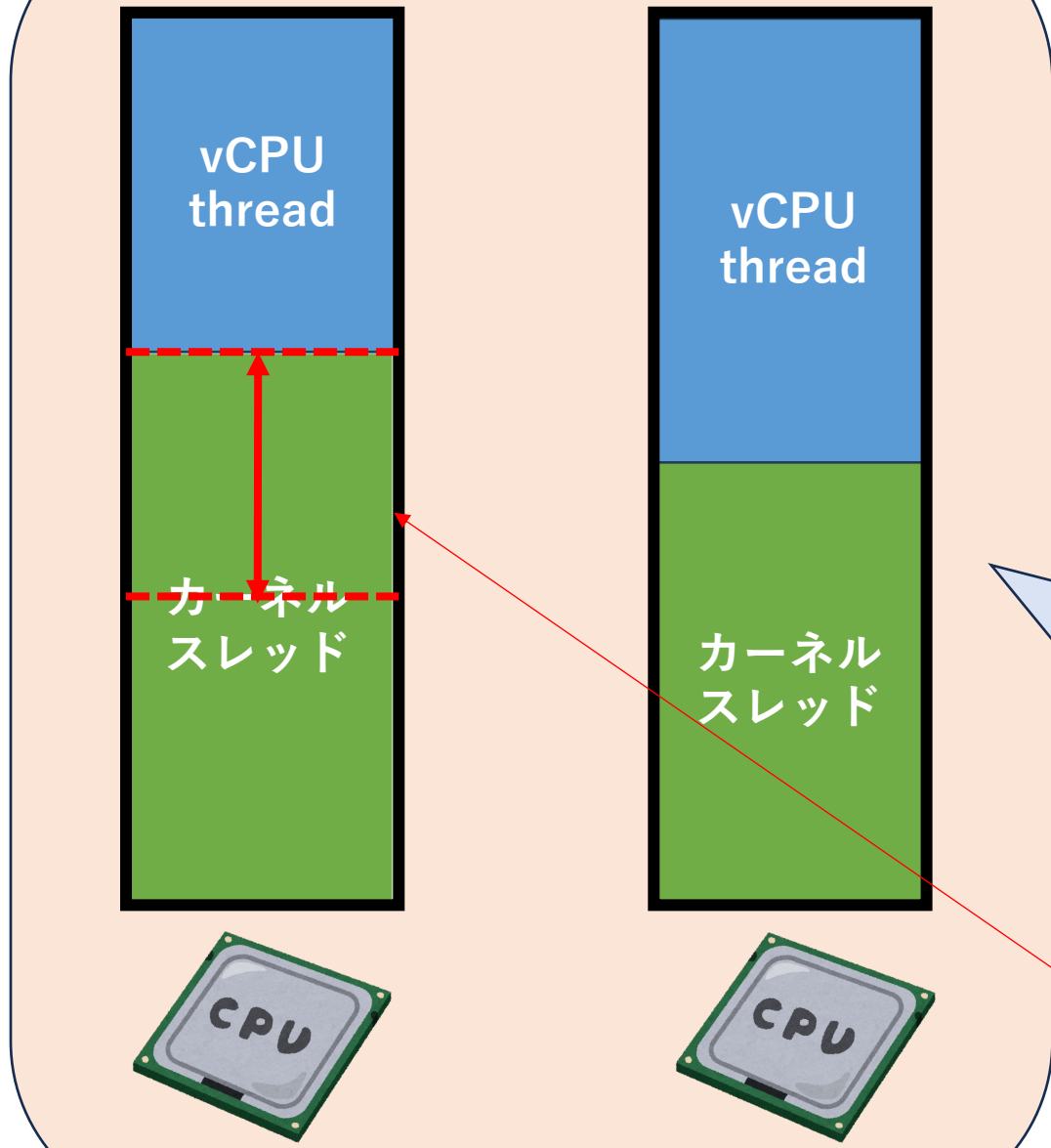
行する
い



計算が多く、I/O が少ない場合：vCPU に利用されない CPU コアが増える

解決策：vCPU とバックエンドのカーネルスレッドは同じ CPU コア
利点：ワークロードが変化しても、CPU の vCPU とバックエンド

高速化



計算が多く、I/O が少ない場合：vCPU に利用されない

を使った

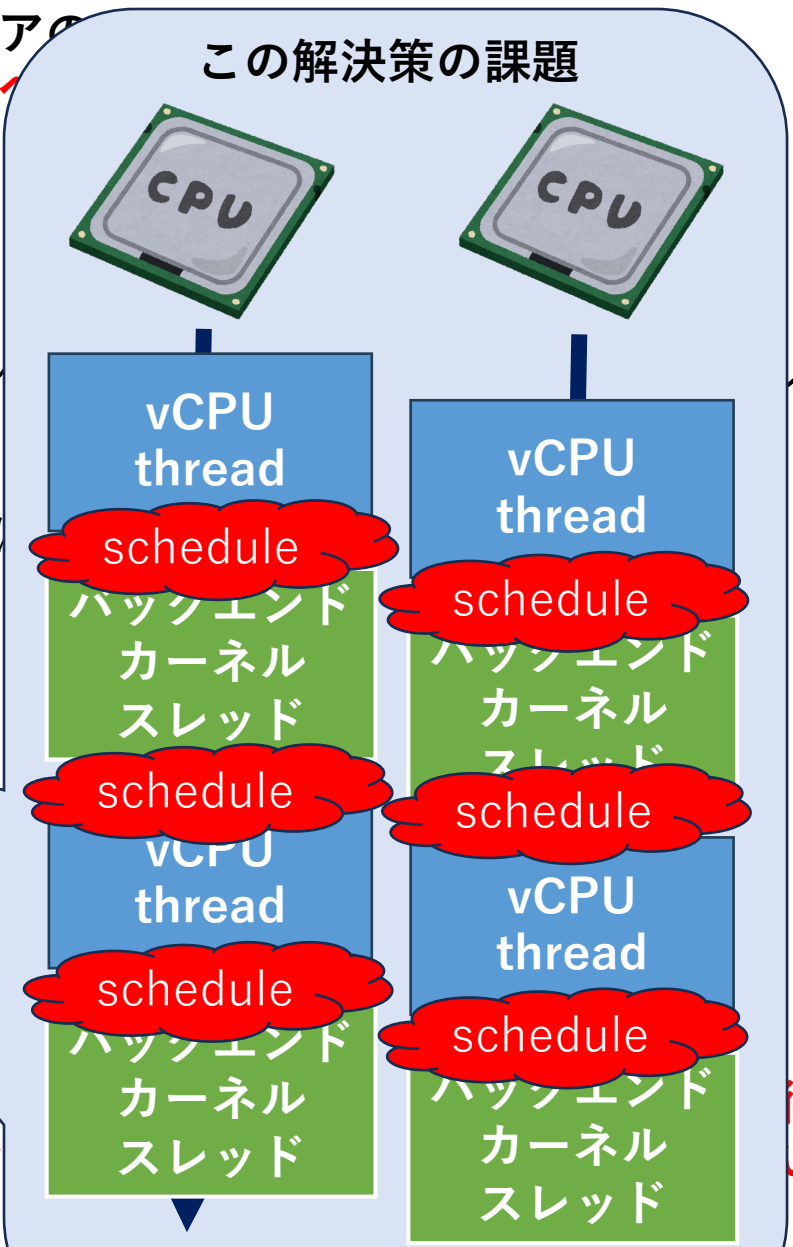
Xen

QEM

(AN 2016)

主張：バック

この解決策の課題



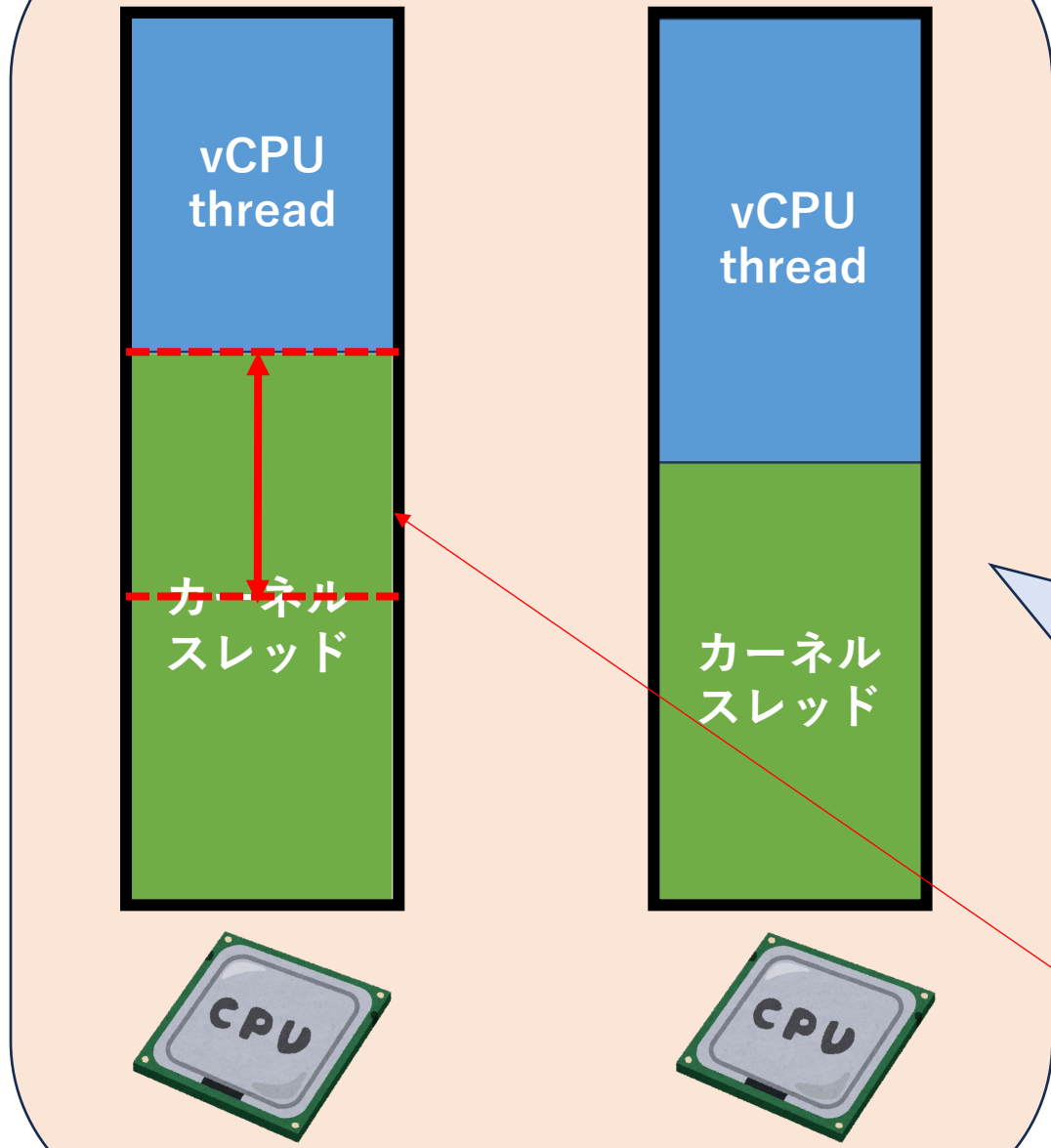
時間 スレッド切り替えのためのスケジューリングコスト

行する

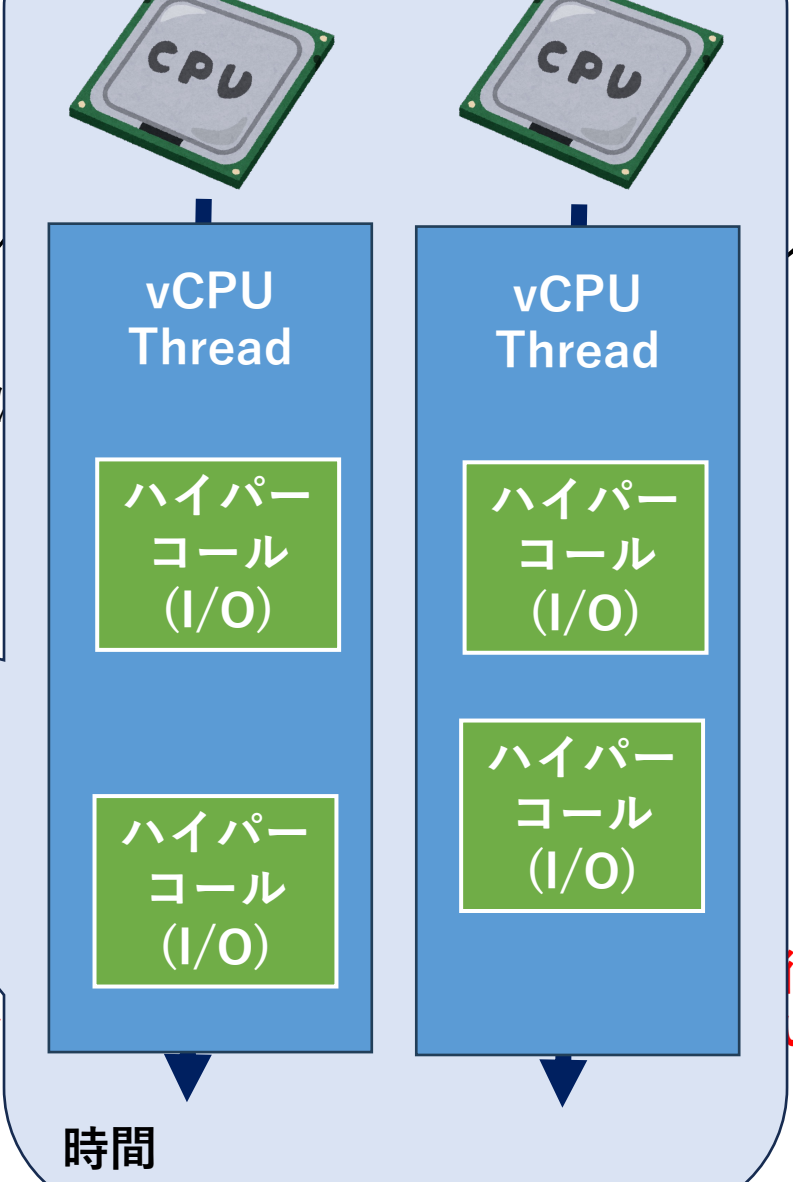
解決策：vCPU とバックエンドのカーネルスレッドは同じ CPU コア

利点：ワークロードが変化しても、CPU の vCPU とバックエンド

高速化



提案手法
I/O をハイパーコール内で実行



を使った

Xen

QEM

(AN 2016)

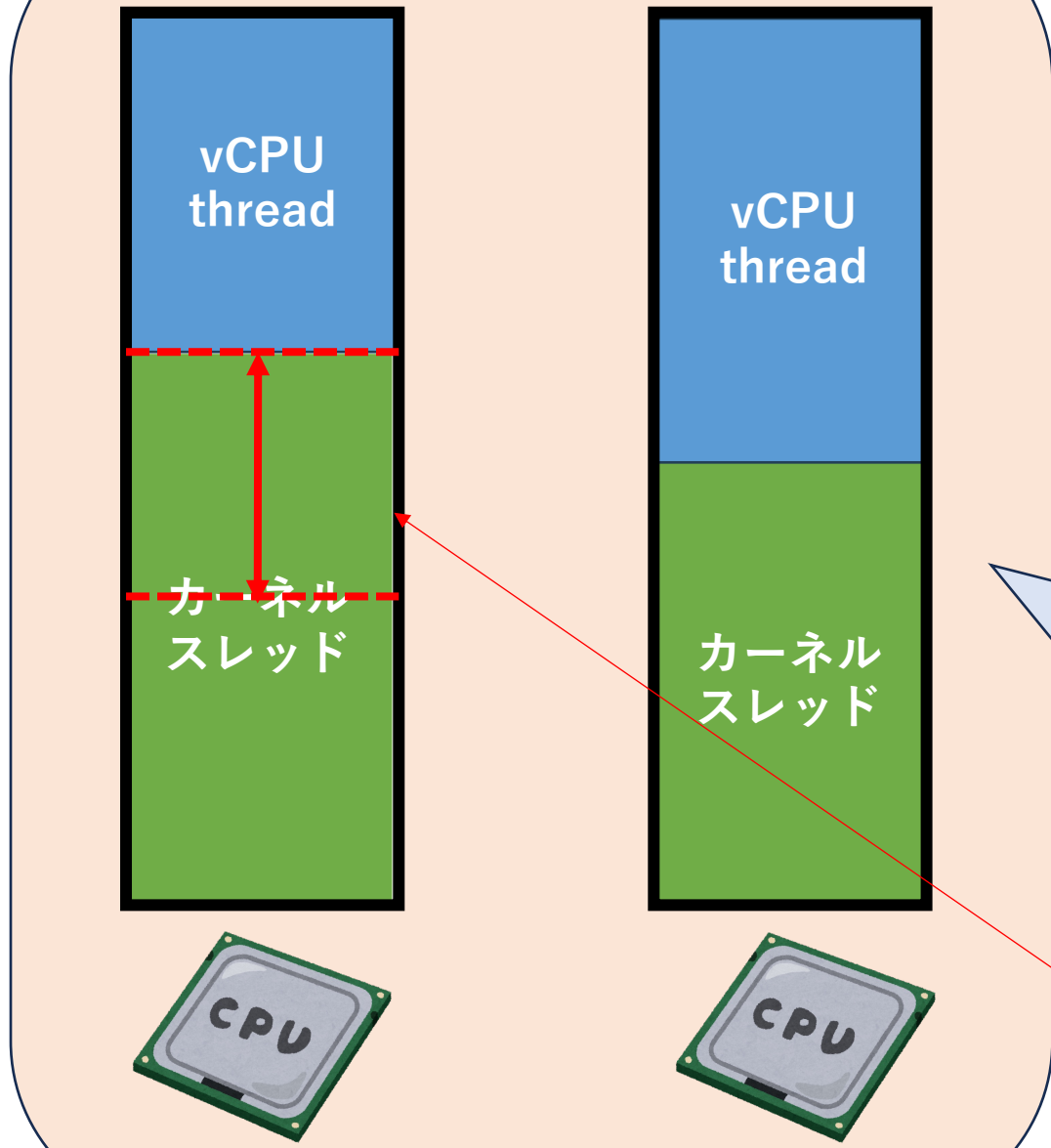
主張
バック

行する
い

計算が多く、I/O が少ない場合：vCPU に利用されない CPU コアが増える

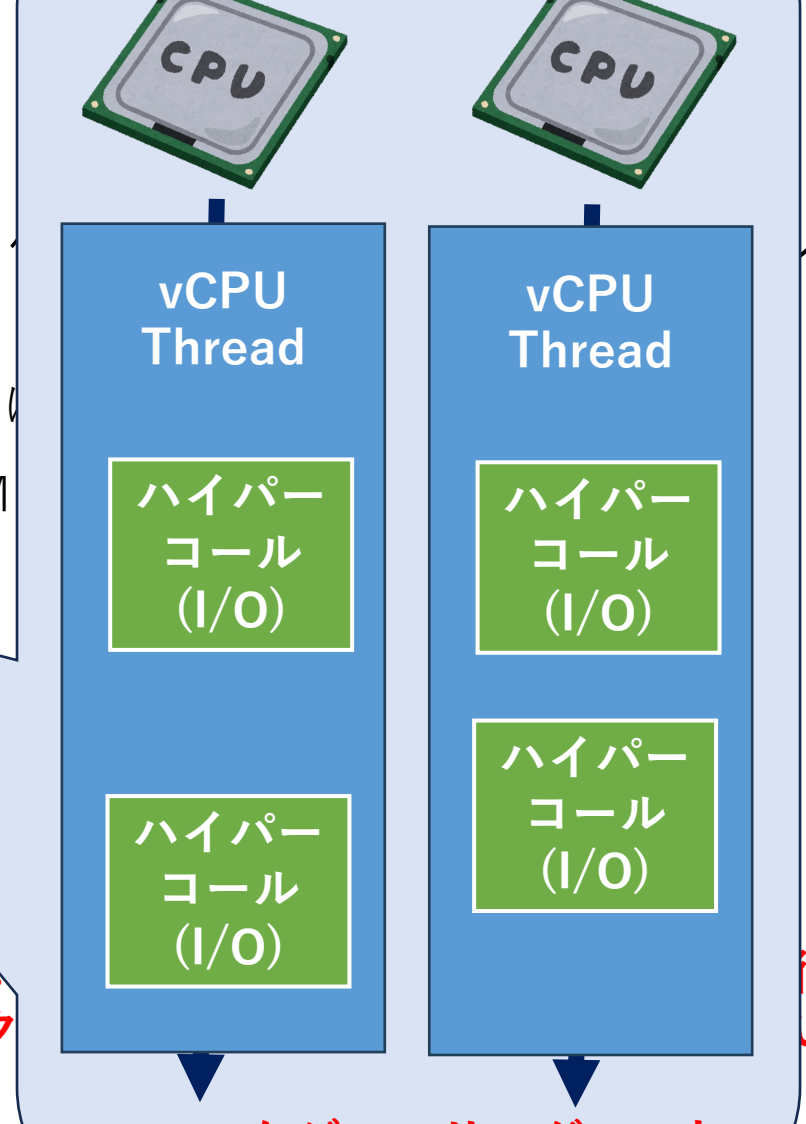
解決策：vCPU とバックエンドのカーネルスレッドは同じ CPU コア
利点：ワークロードが変化しても、CPU の vCPU とバックエンド

高速化



計算が多く、I/O が少ない場合：vCPU に利用されない

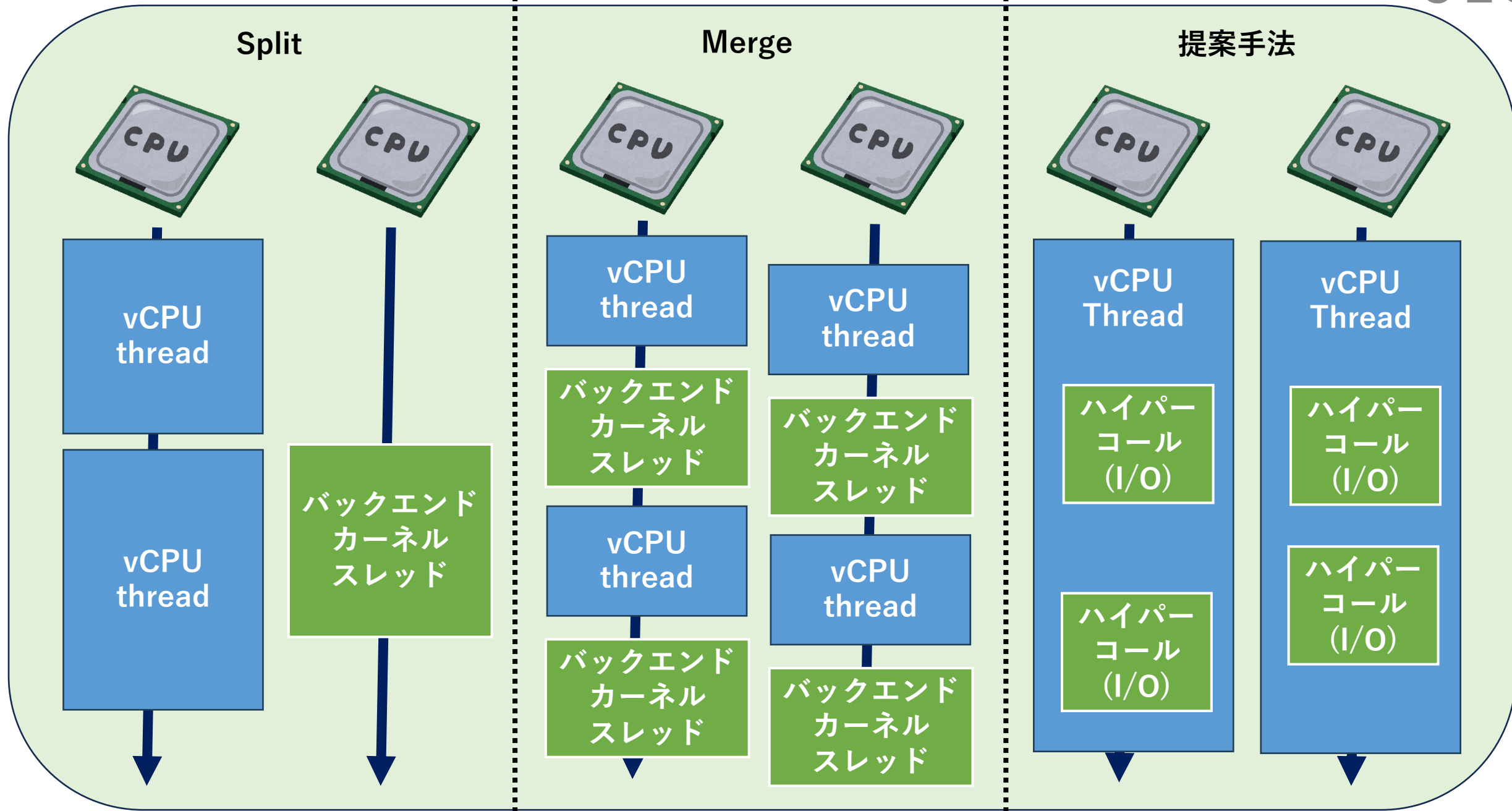
提案手法
I/O をハイパーコール内で実行



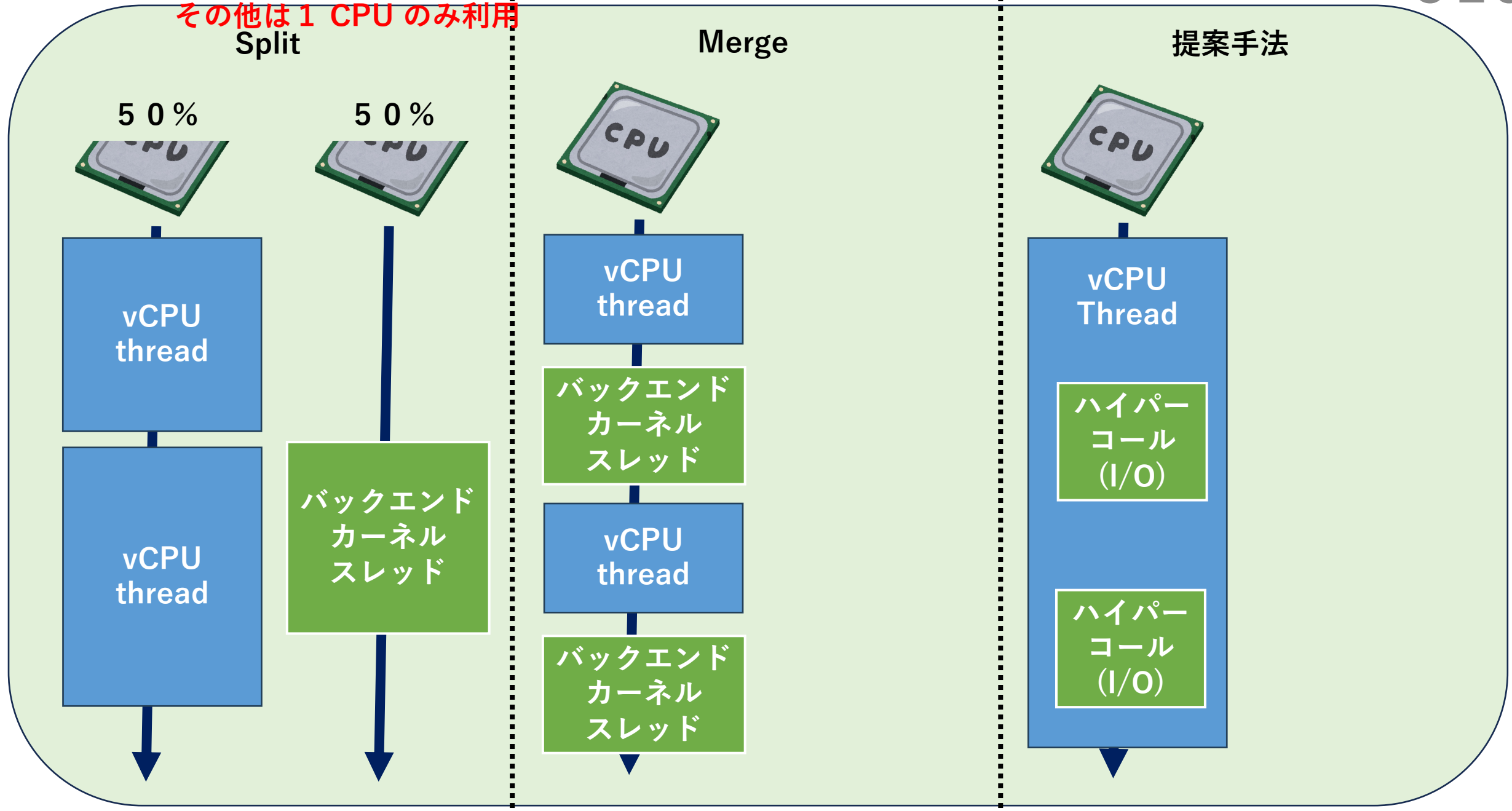
主張：バック

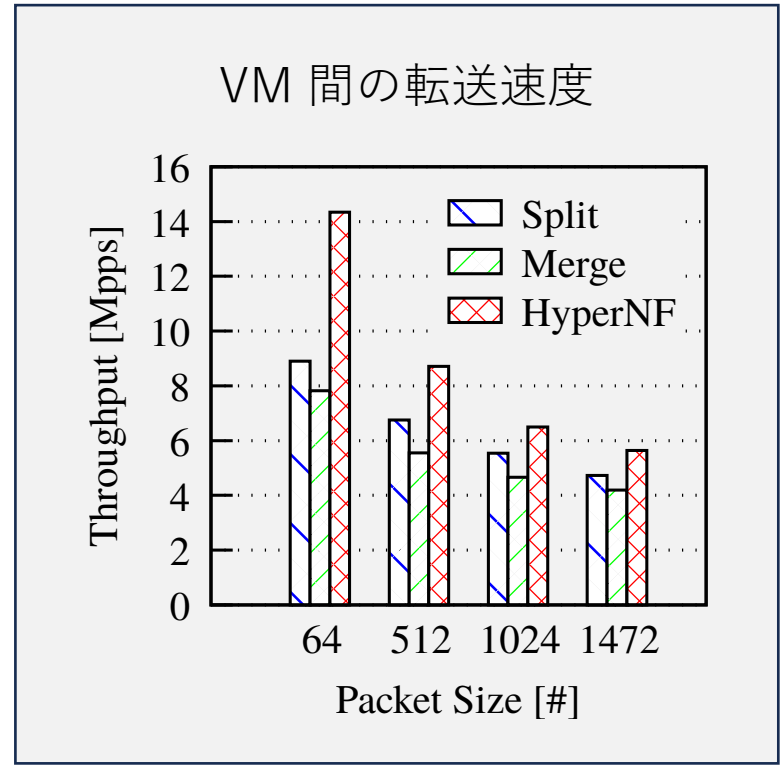
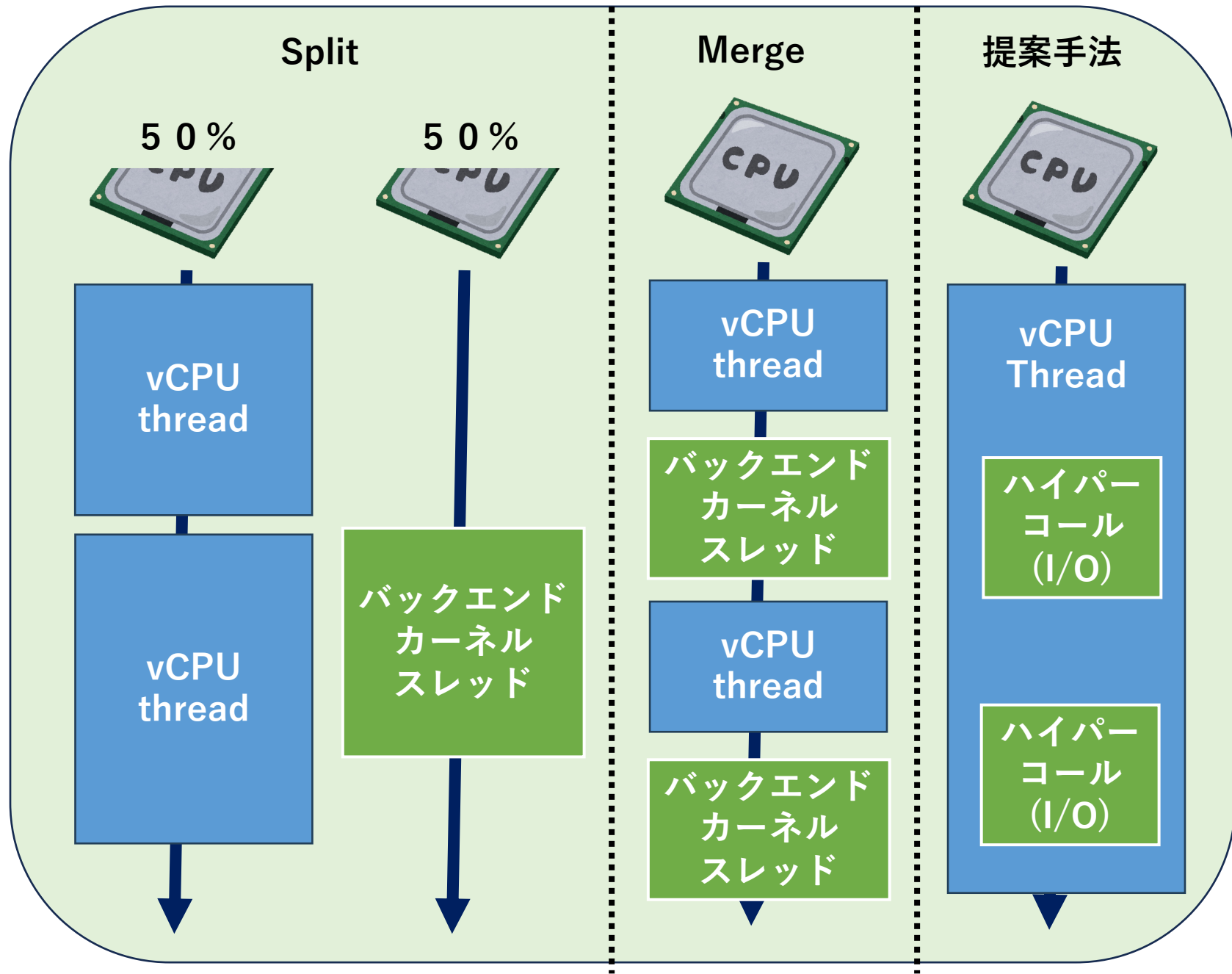
時間 スケジューリングコストをなくすることができる

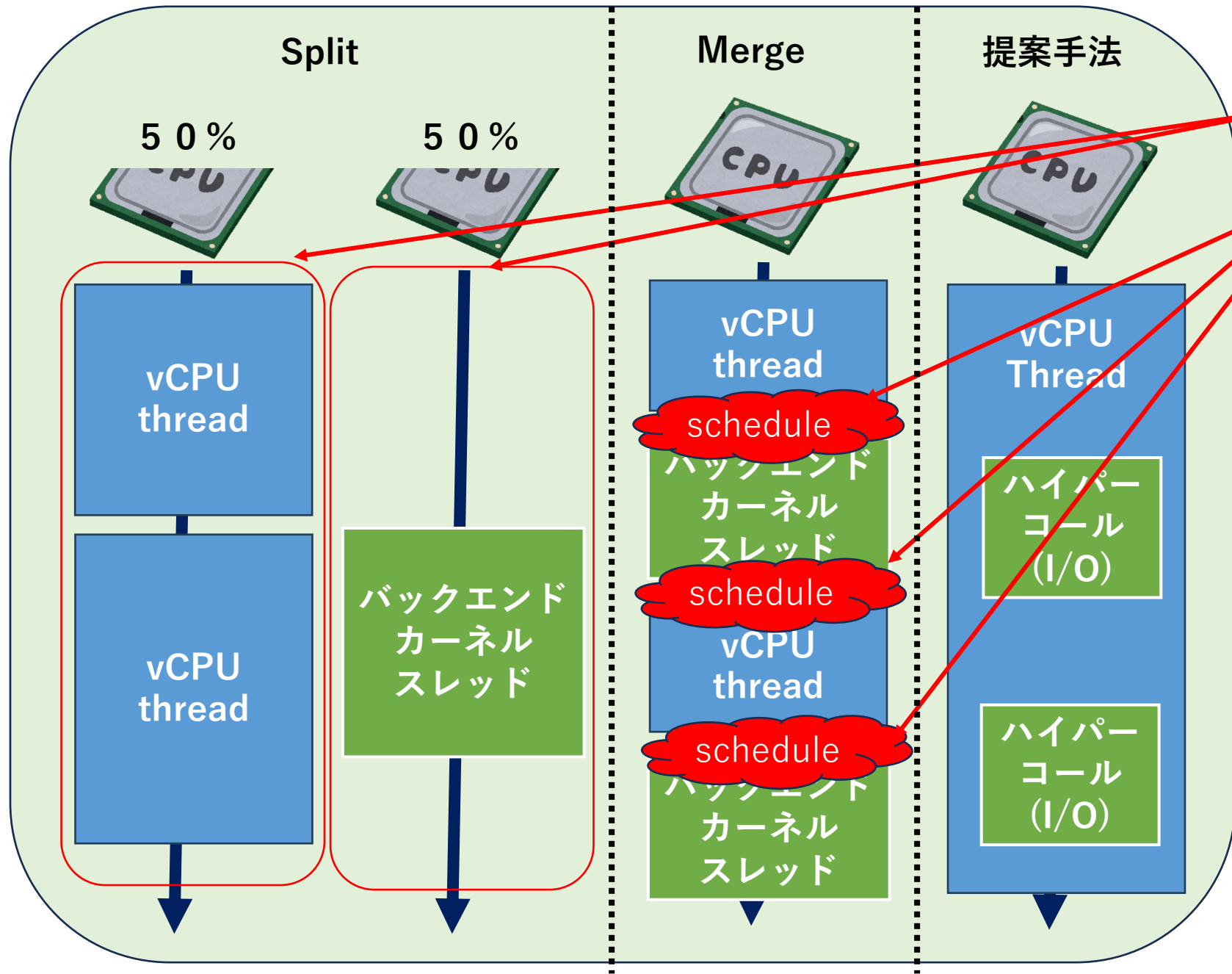
実行する



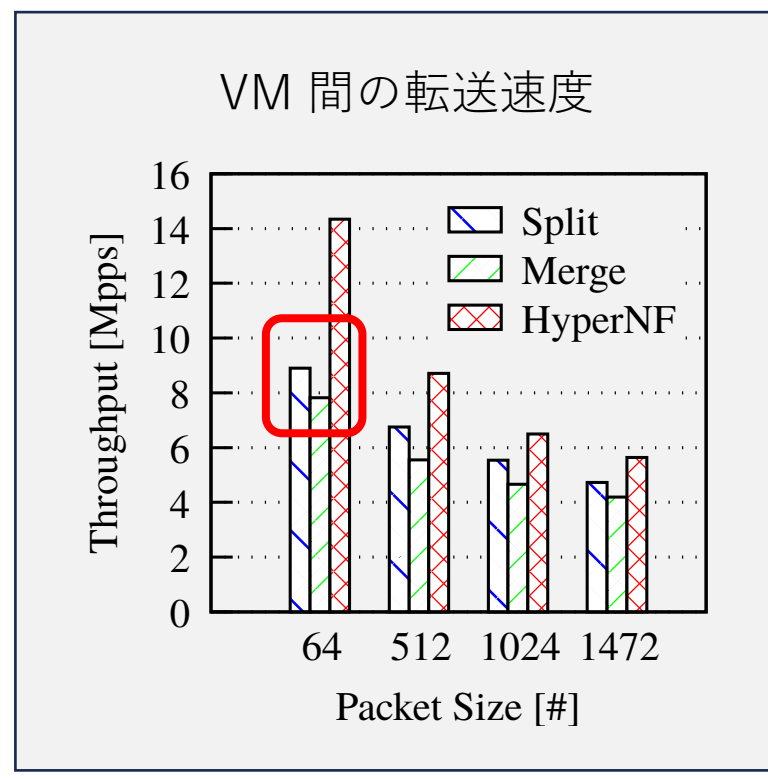
その他は 1 CPU のみ利用

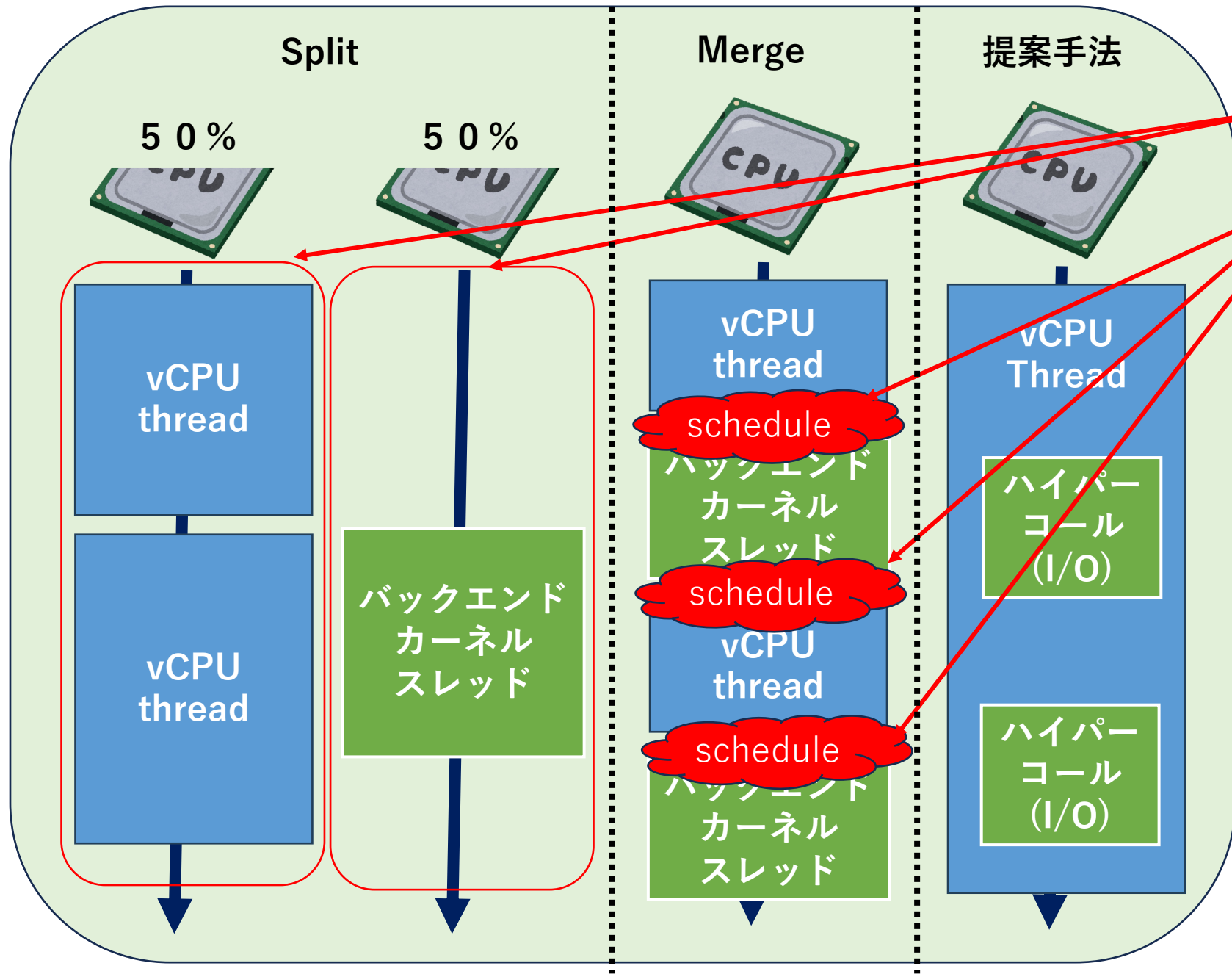






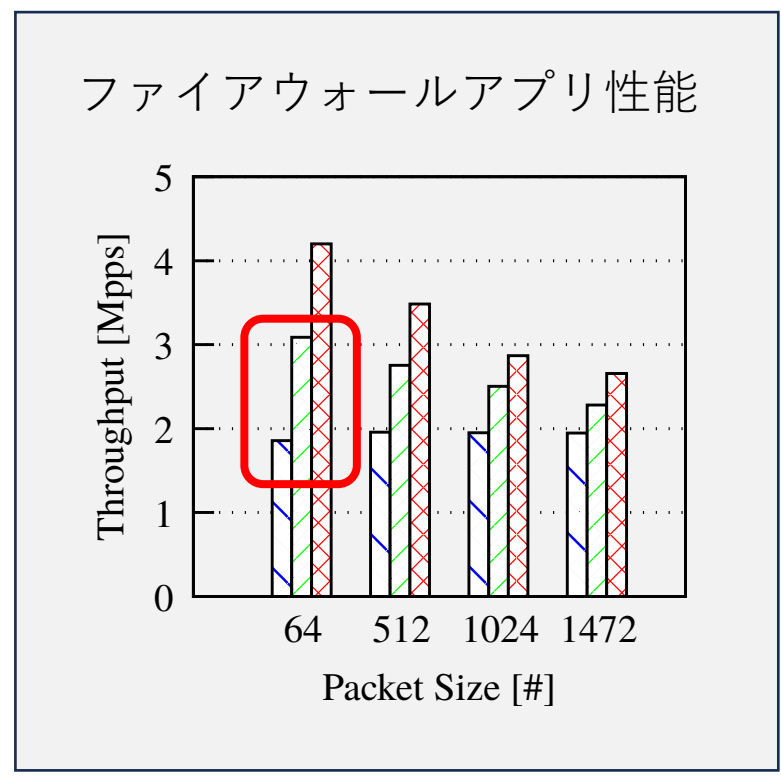
vCPU とバックエンドに必要な CPU 時間が同じくらいだと
 スケジュールのコストが小さい
 Split の方が Merge より速い

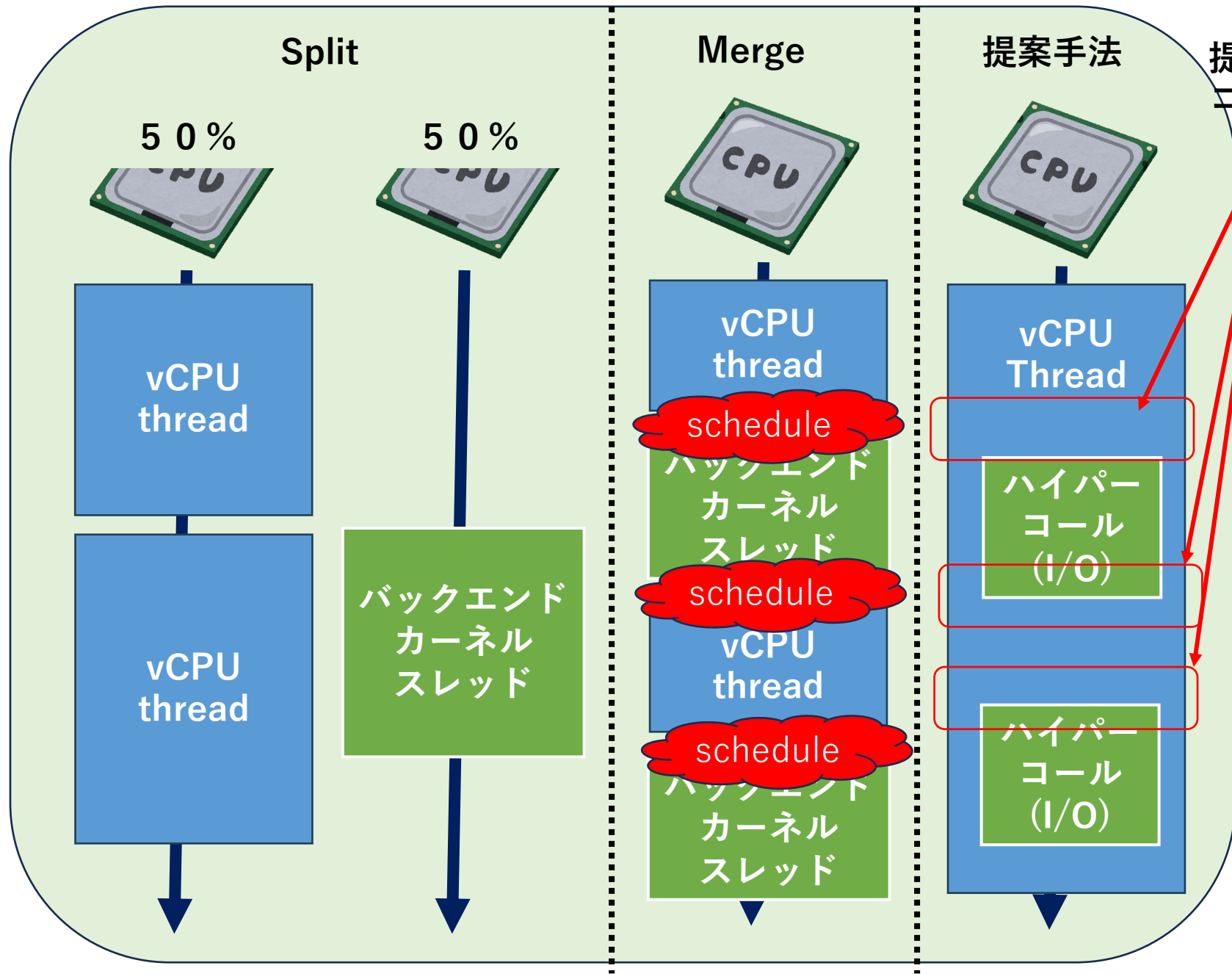




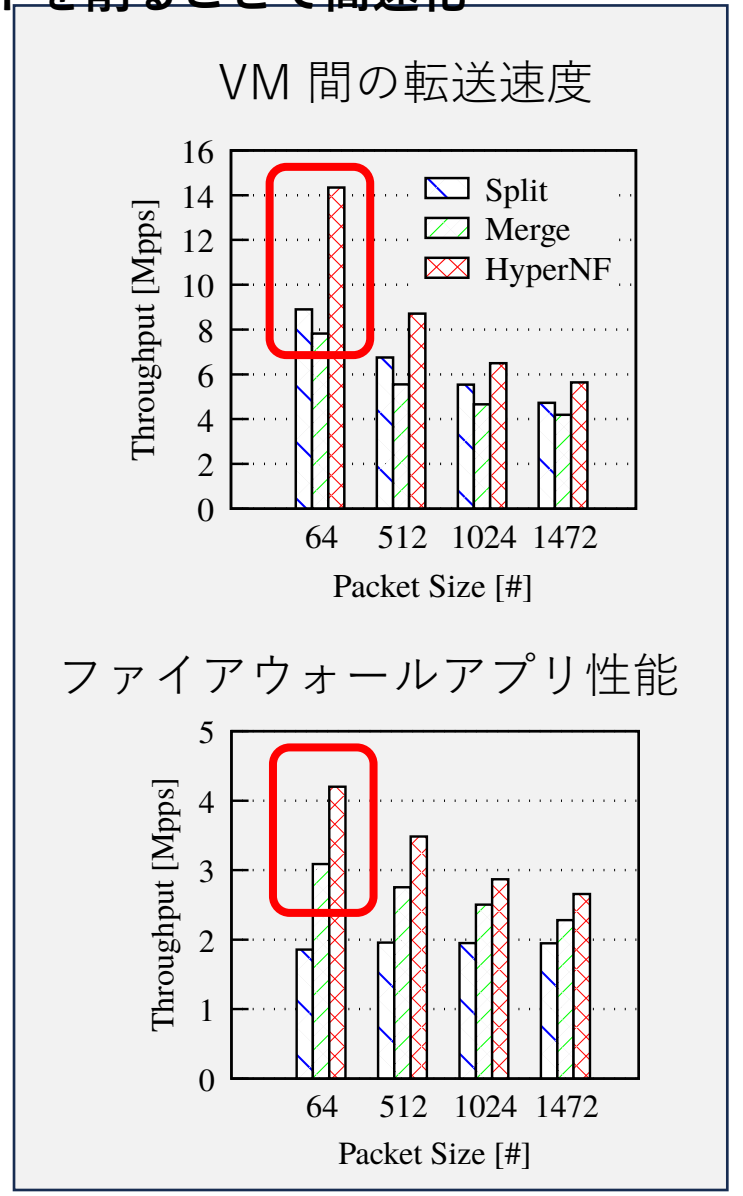
vCPU とバックエンドに必要な CPU 時間に偏りがあると

スケジュールのコストがあっても Merge の方が Split より速い





提案手法は Merge からスケジュールのコストを削ることで高速化



仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014) ← Xen に netmap/VALE を適用
 - NetVM (NSDI 2014) ← QEMU/KVM に DPDK を適用
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - **HyperNF (SoCC 2017)** ← Xen に netmap/VALE を適用
 - ELISA (ASPLOS 2023) **実行のモデルを改善**

仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014) ← Xen に netmap/VALE を適用
 - NetVM (NSDI 2014) ← QEMU/KVM に DPDK を適用
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - **HyperNF (SoCC 2017)** ← Xen に netmap/VALE を適用
 - ELISA (ASPLOS 2023) **実行のモデルを改善**


パイプラインのような処理の場合は、スレッドを分けない方が CPU 利用効率が良い

研究紹介

仮想マシン通信について

仮想 I/O リクエストに伴う vCPU コンテキストからの exit 削減

仮想マシン通信の高速化

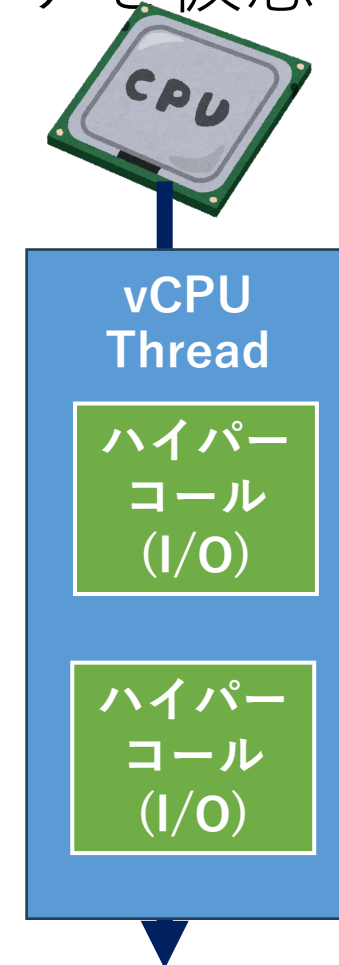
- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)**  改善

仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)**



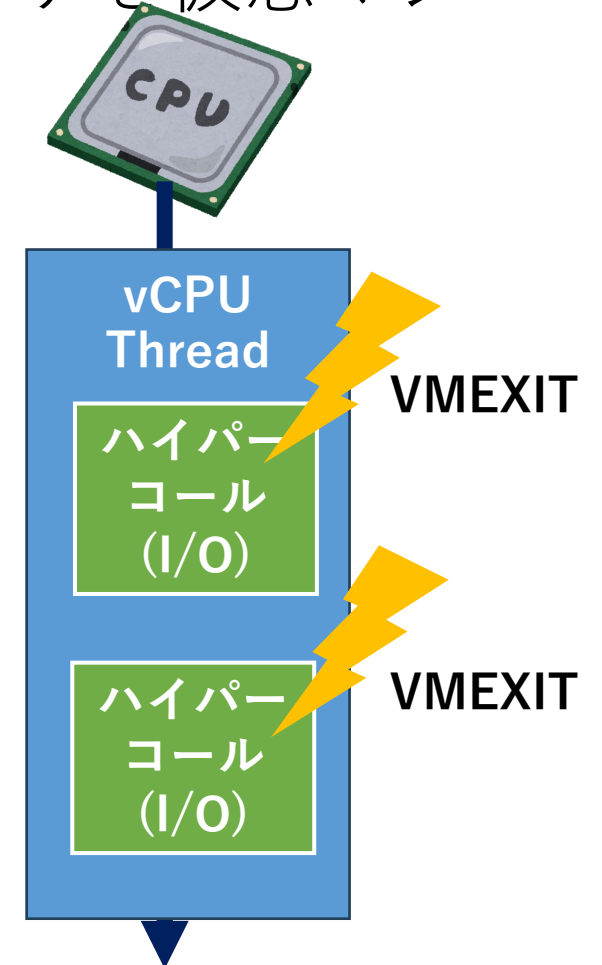
改善



仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)** ← 改善

問題：ハイパーコールの呼び出しのたびに
vCPU からハイパーバイザーのコンテキストへ exit する必要がある

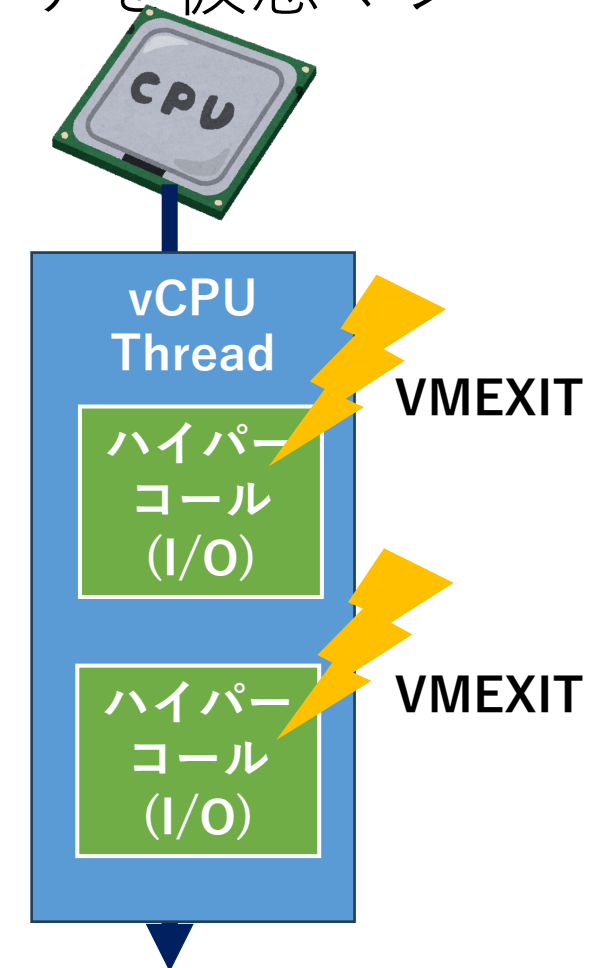


仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)** ← 改善

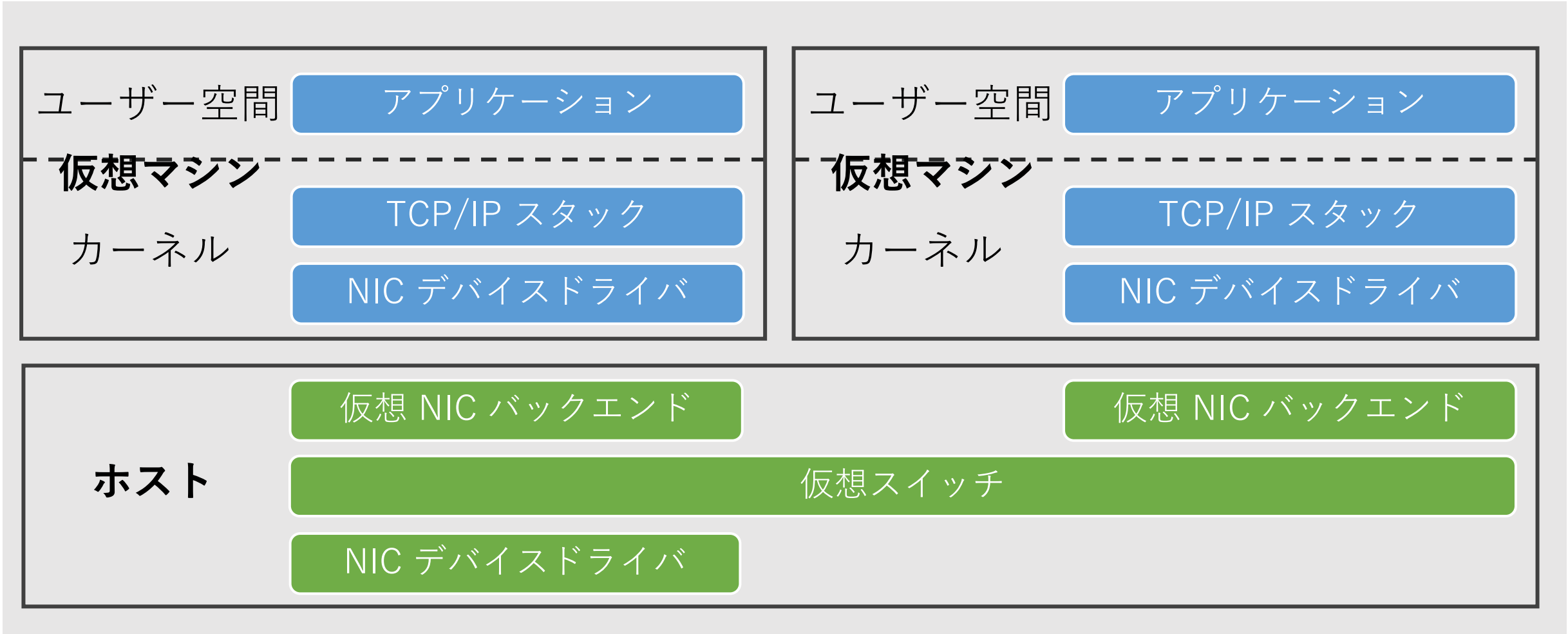
問題：ハイパーコールの呼び出しのたびに
vCPU からハイパーバイザーのコンテキストへ exit する必要がある

モチベーション：vCPU から exit しないで I/O を実行できるようにしたい



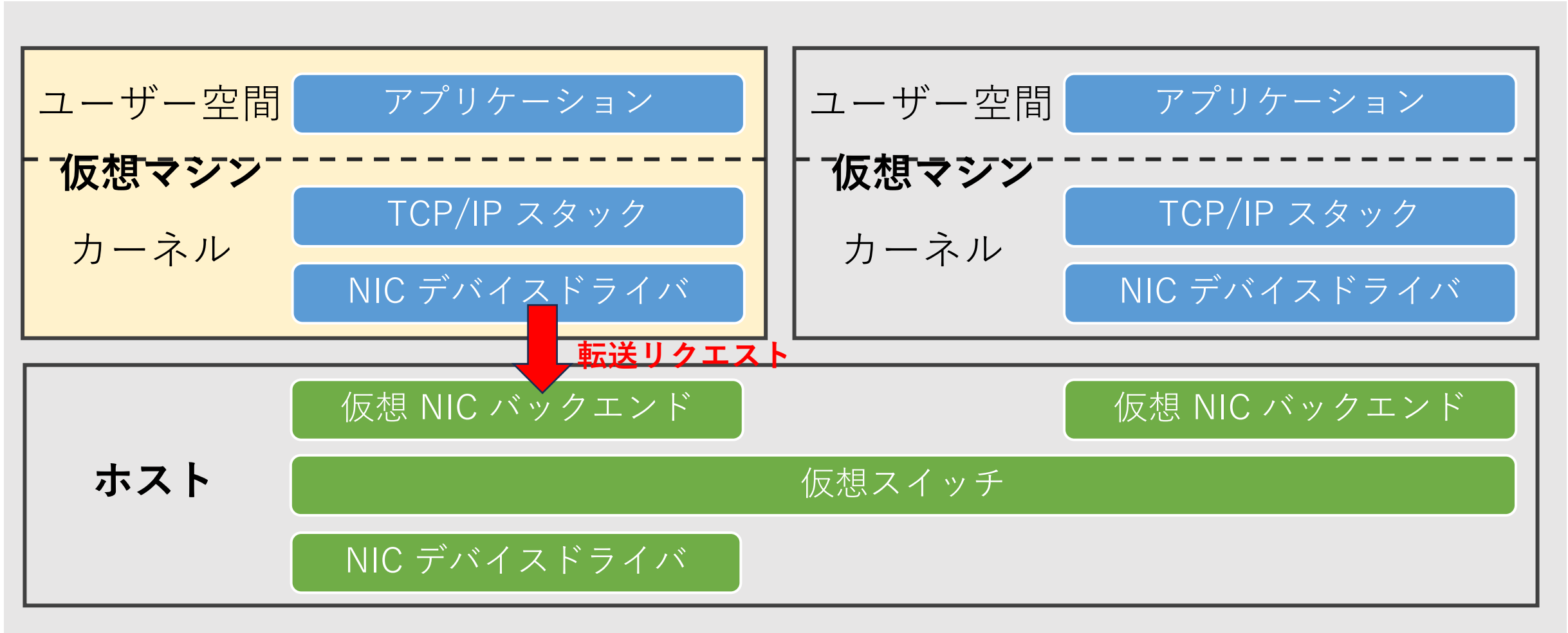
仮想マシン通信

比較的一般的な構成



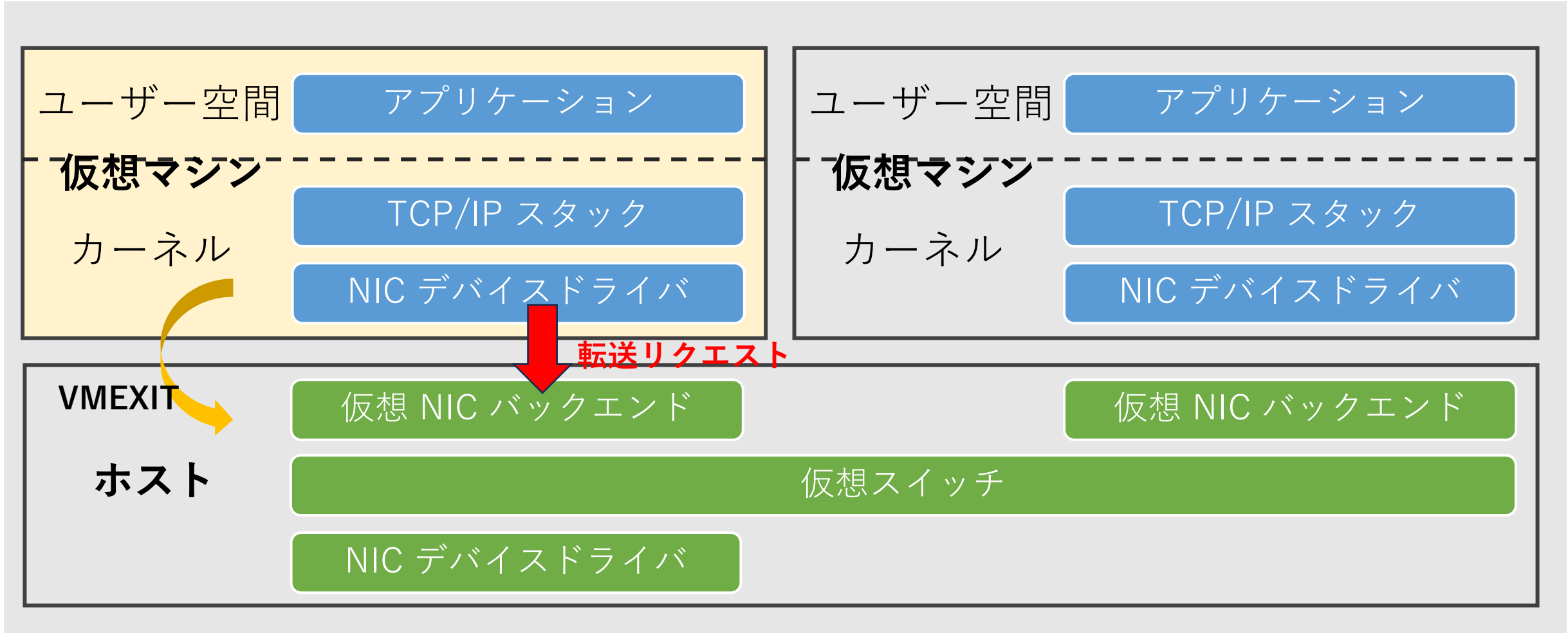
仮想マシン通信

比較的一般的な構成



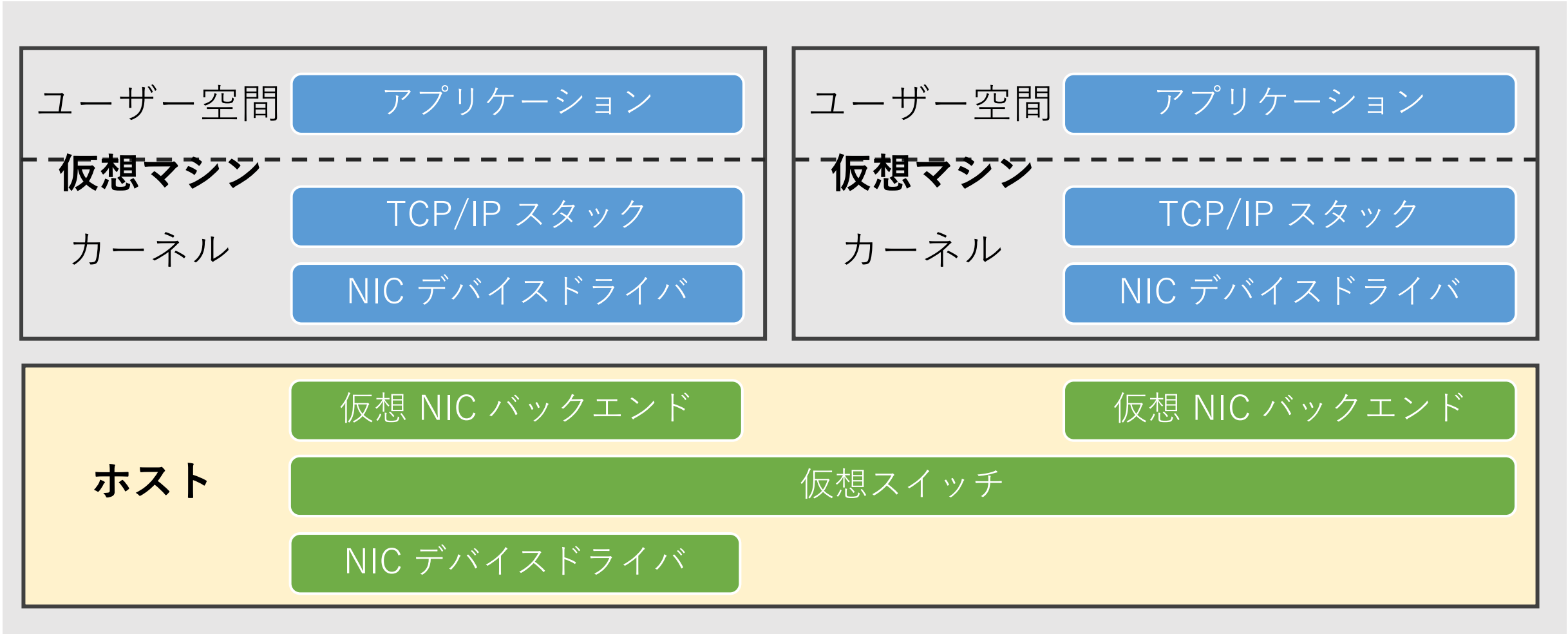
仮想マシン通信

比較的一般的な構成



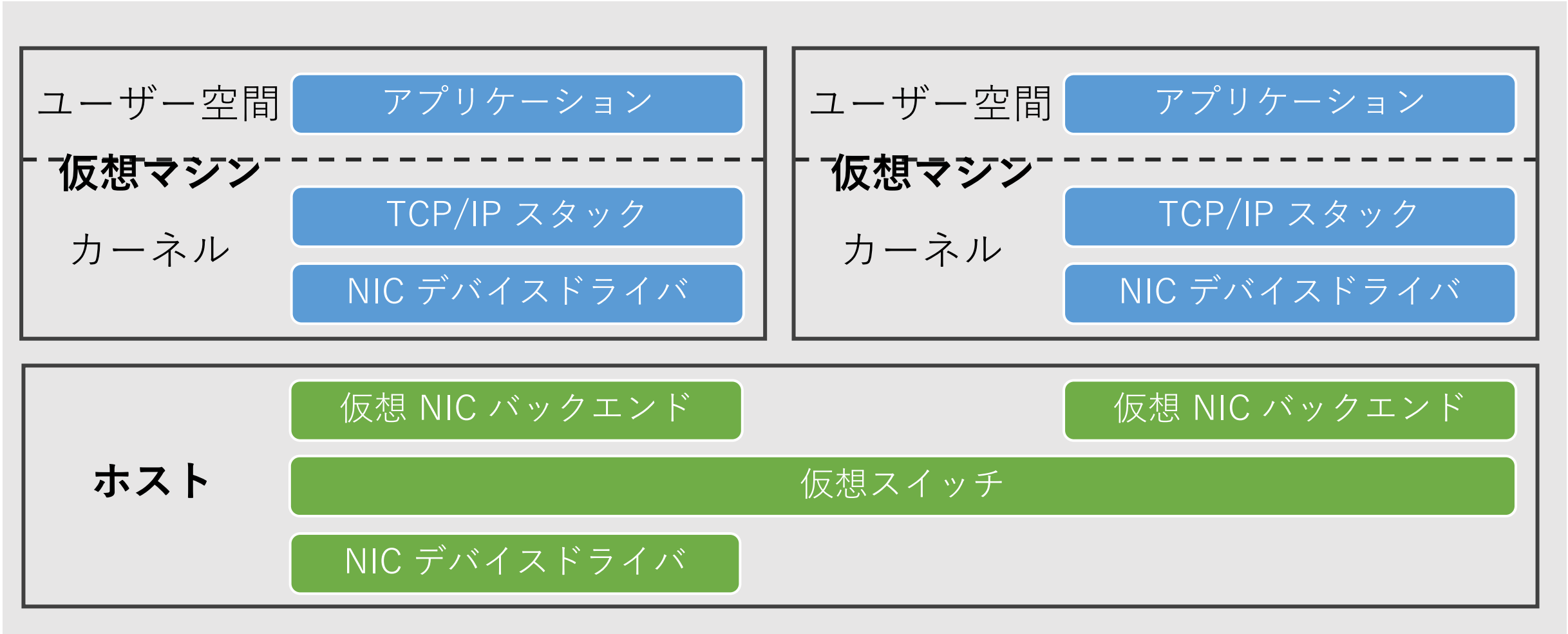
仮想マシン通信

比較的一般的な構成



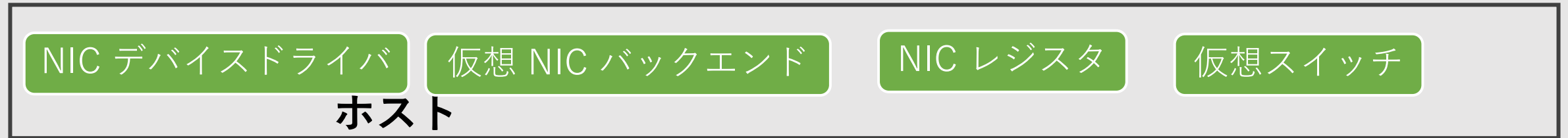
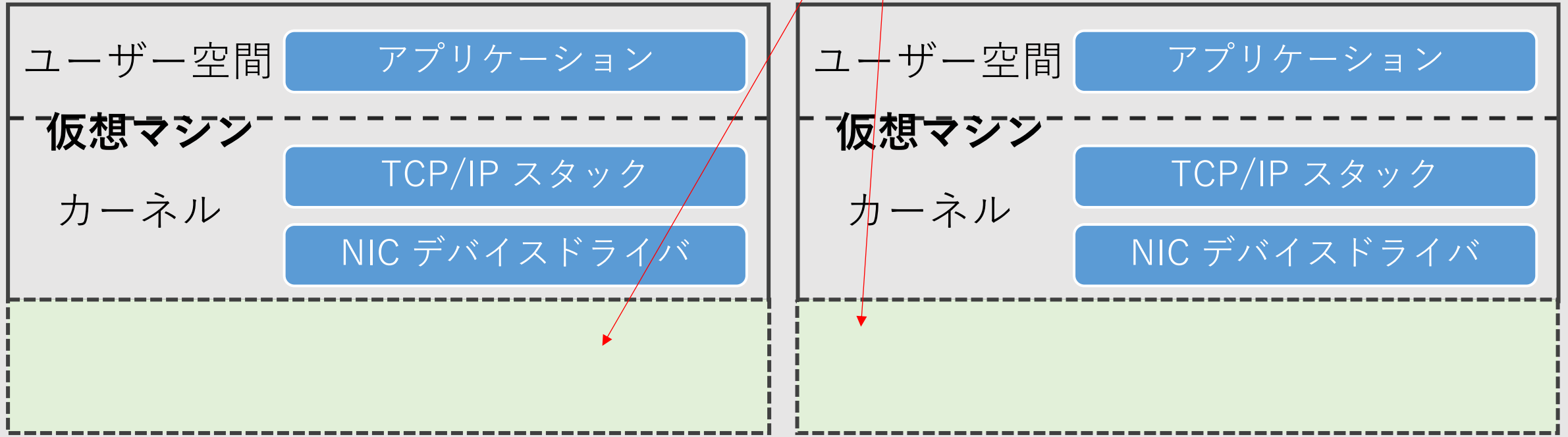
仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加



仮想マシン通信

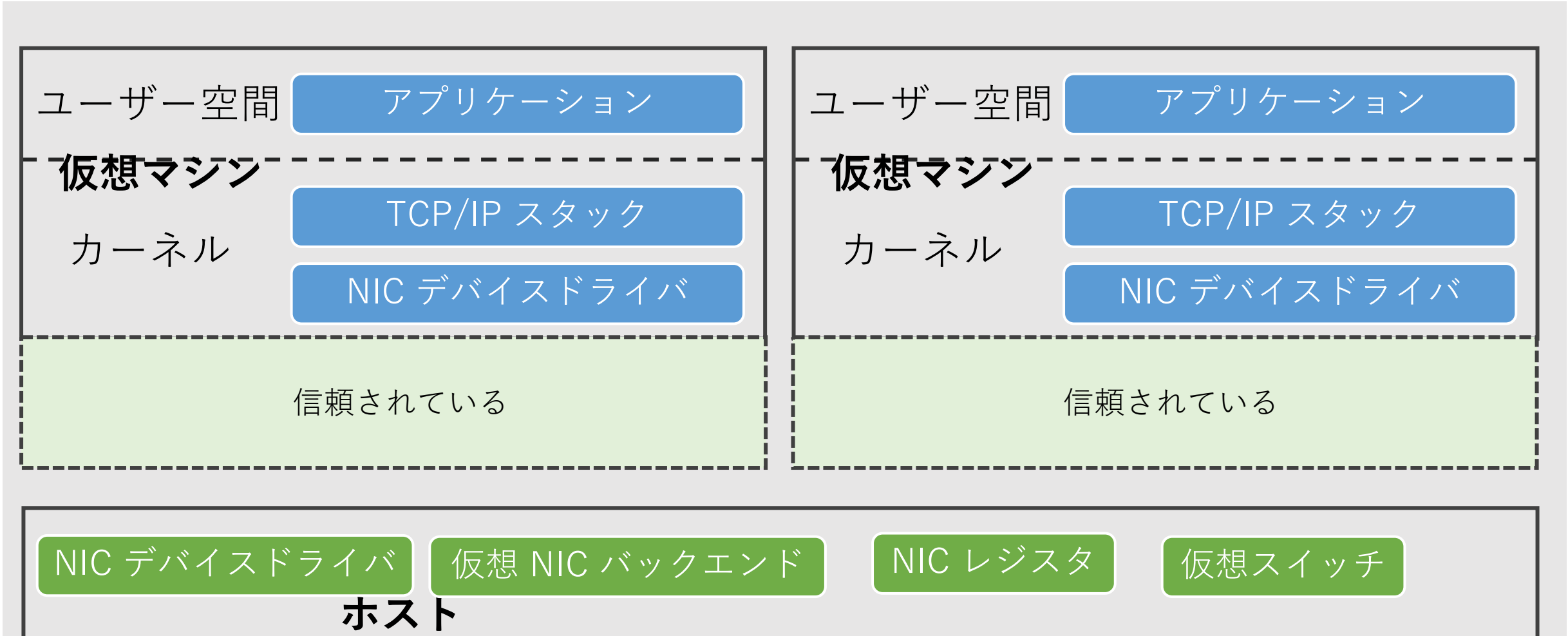
提案手法：仮想マシンに新しいコンテキストを追加



仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

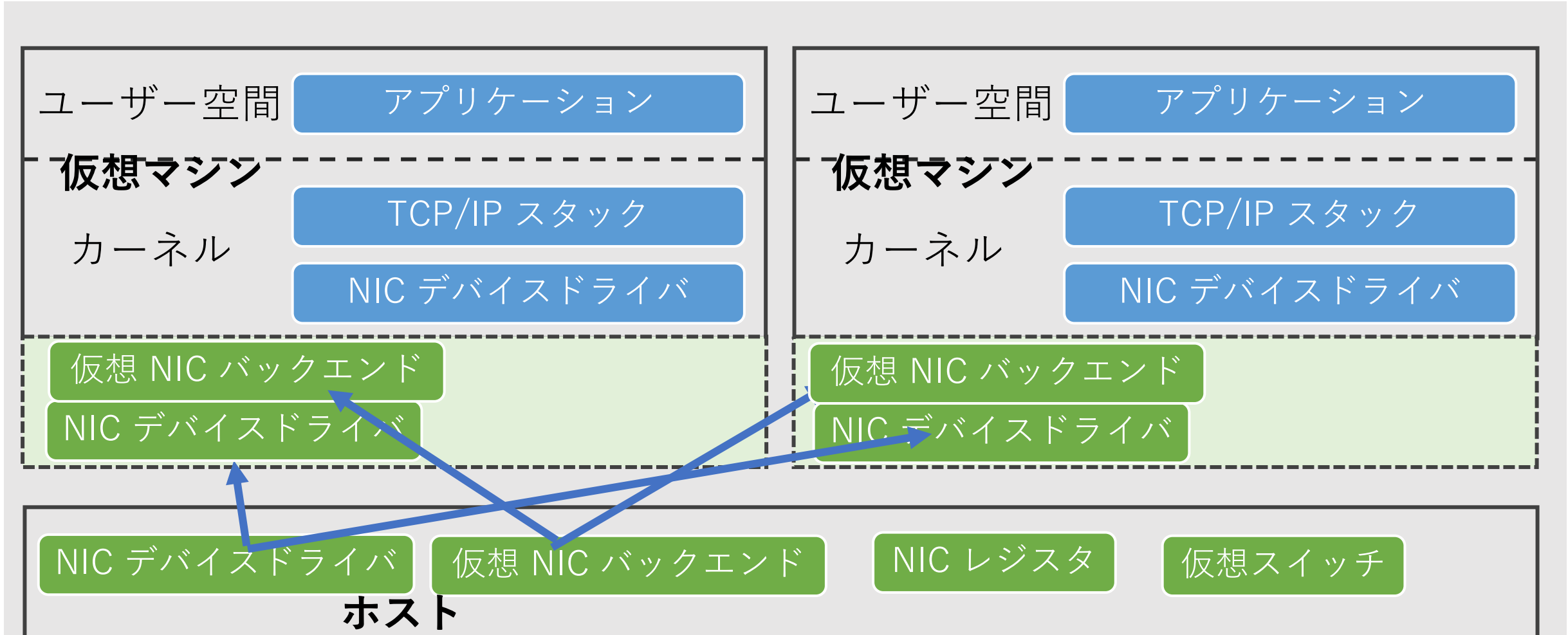
- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定



仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

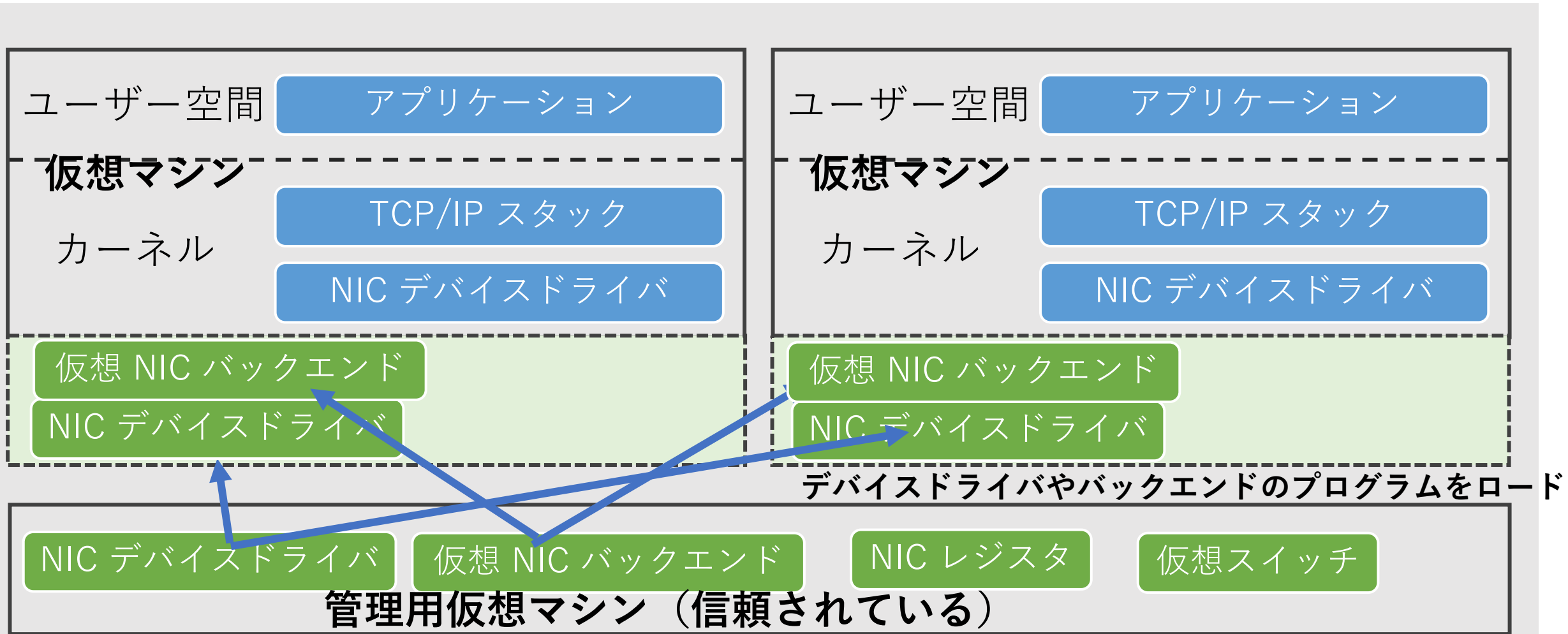
- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定



仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定



デバイスドライバやバックエンドのプログラムをロード

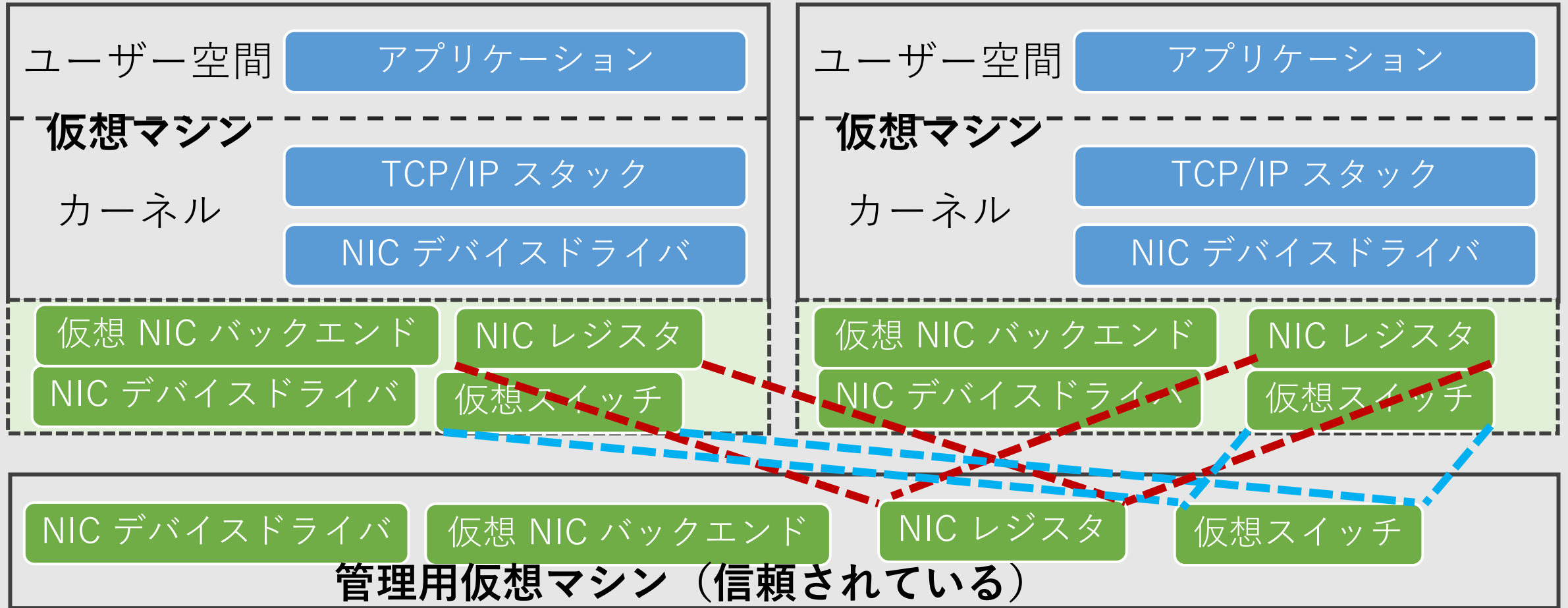
管理用仮想マシン (信頼されている)

仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定

仮想マシン間で共有される NIC レジスタや仮想スイッチ関連オブジェクトをマップ



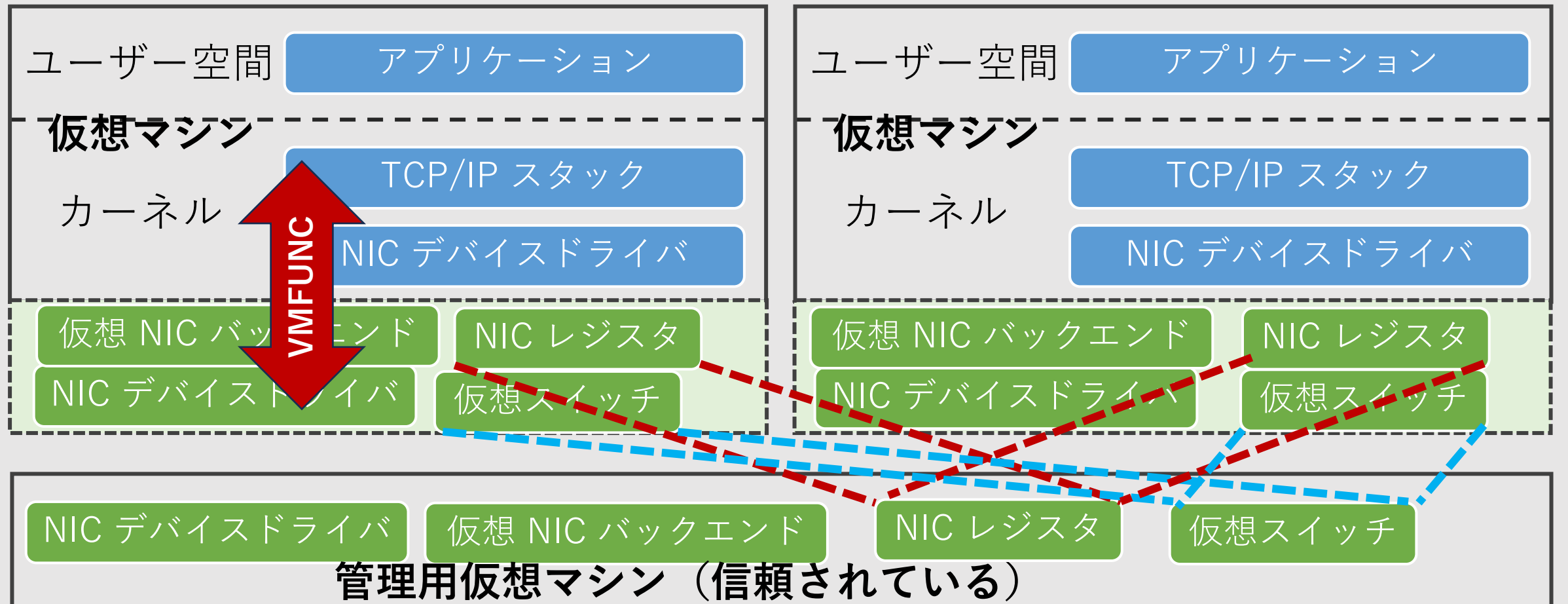
管理用仮想マシン (信頼されている)

仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定

ポイント：コンテキストの移行には VMFUNC という CPU 命令を利用



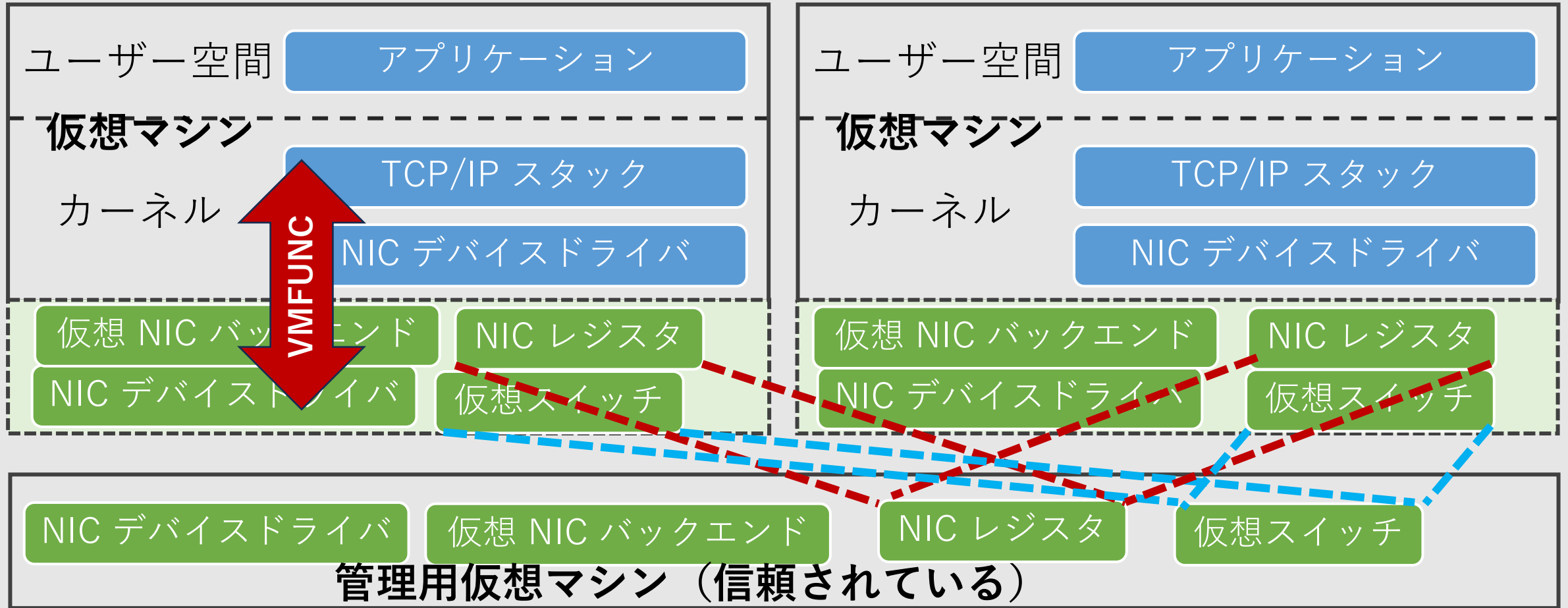
仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加

- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定

ポイント：コンテキストの移行には VMFUNC という CPU 命令を利用

VMFUNC は vCPU からの exit を発生させない：結果、切り替えが速い



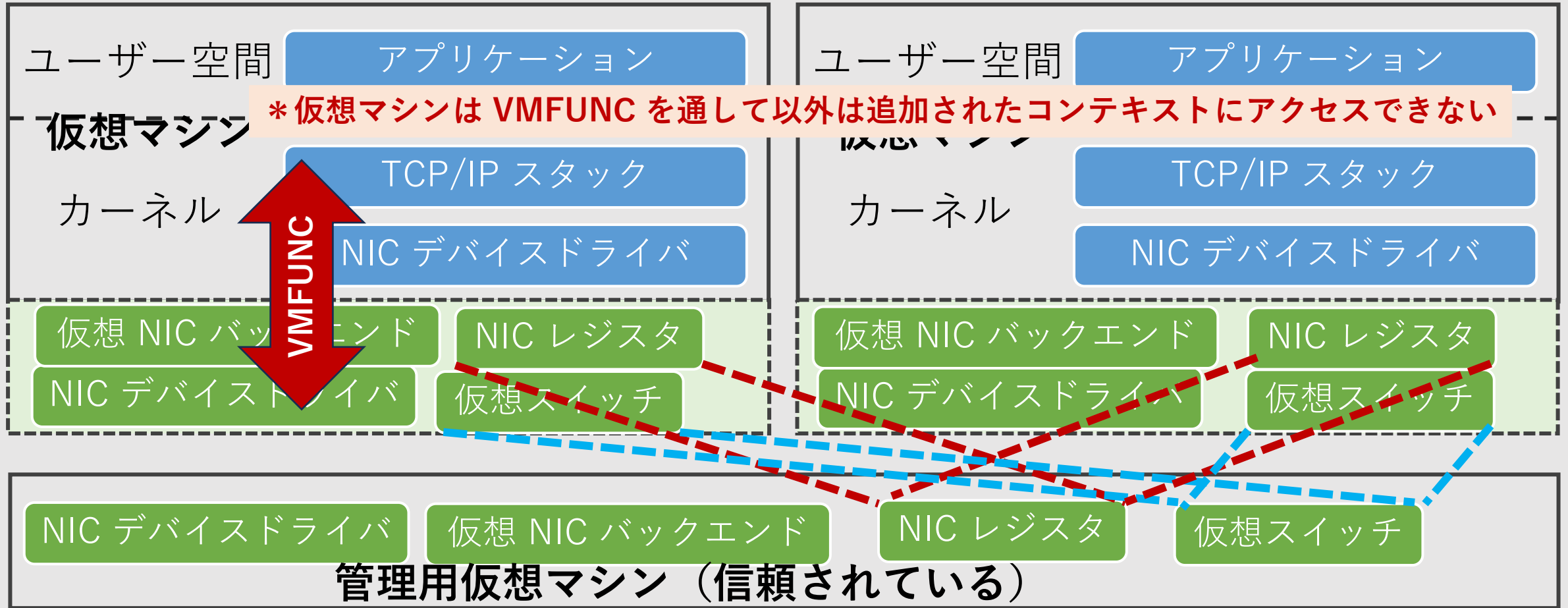
仮想マシン通信

提案手法：仮想マシンに新しいコンテキストを追加


- これらは仮想マシンコンテキストの一部
- ホストと同じく信頼されているドメインとして想定

ポイント：コンテキストの移行には VMFUNC という CPU 命令を利用

VMFUNC は vCPU からの exit を発生させない：結果、切り替えが速い

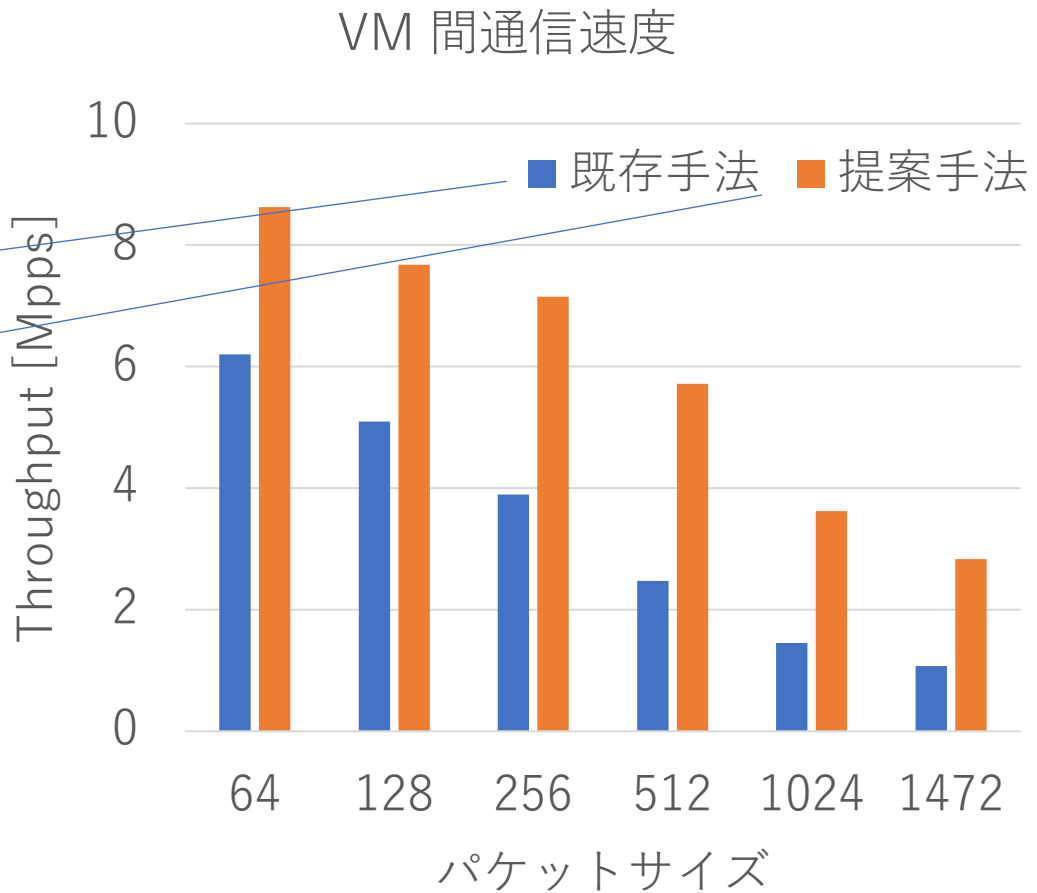


仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)**  改善

仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)**



仮想マシン通信の高速化

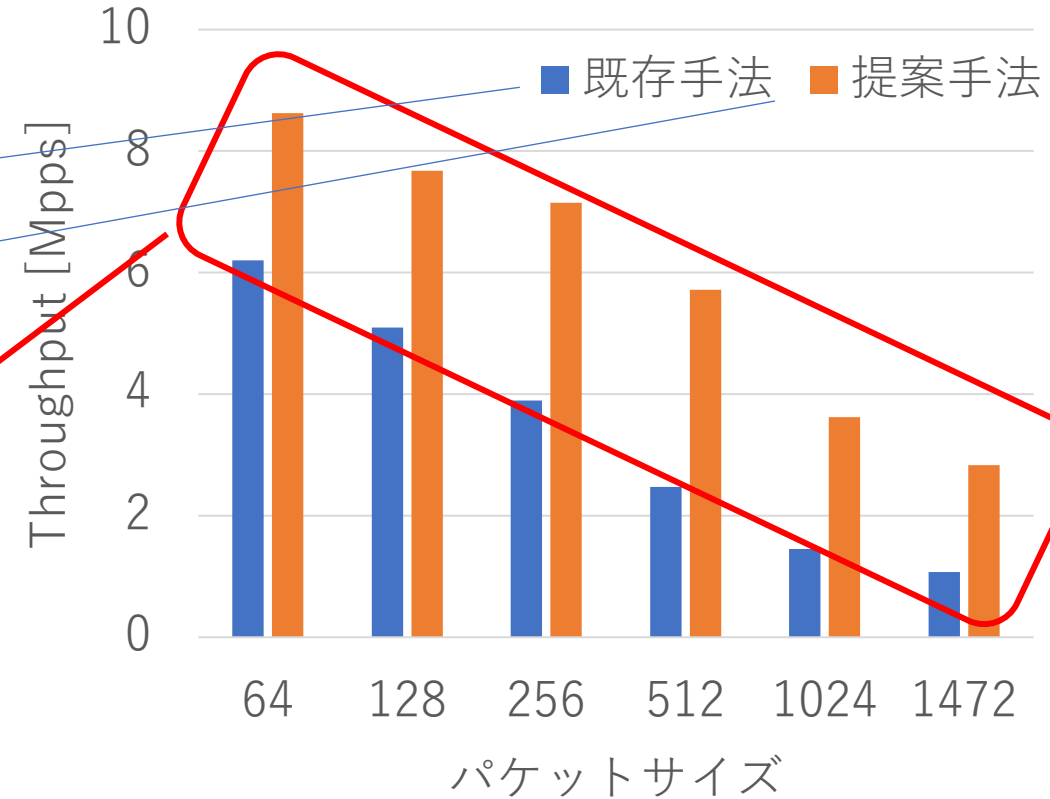
• パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用

- ClickOS (NSDI 2014)
- NetVM (NSDI 2014)
- ptnetmap (ANCS 2015, LANMAN 2016)
- HyperNF (SoCC 2017)
- **ELISA (ASPLOS 2023)**



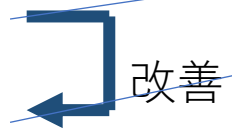
VMEXIT をなくすことによる改善

VM 間通信速度

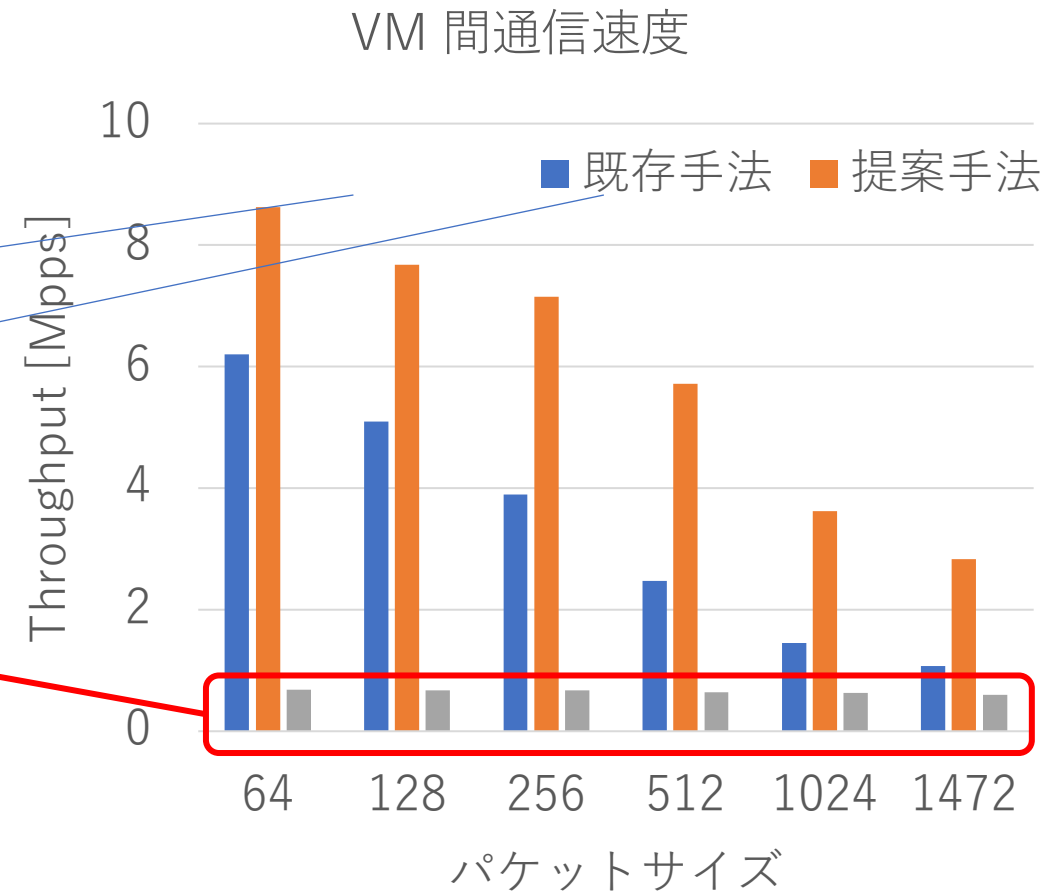


仮想マシン通信の高速化

- パケット I/O フレームワークを使った仮想スイッチを仮想マシン通信基盤へ適用
 - ClickOS (NSDI 2014)
 - NetVM (NSDI 2014)
 - ptnetmap (ANCS 2015, LANMAN 2016)
 - HyperNF (SoCC 2017)
 - **ELISA (ASPLOS 2023)**



Linux vhost-net はこれくらい



最近の取り組み

最近の取り組み：TCP/IP スタック自作

- モチベーション
 - 他のシステムと統合しやすく
 - マルチコア環境で利用できる実装がほしい
 - + 性能のボトルネックがどこから来るかに興味がある
- まだ実装途中ですがよろしければお試しく下さい
 - ソースコード：<https://github.com/yasukata/iip>

モチベーション

- 既存の多くの TCP/IP 実装は込み入ったことをしようと思うと取り回しが良くない場合がある

モチベーション

- 具体的に、既存の多くの TCP/IP スタック実装は
 1. 特定の OS、ライブラリやネットワーク I/O 機能に依存
 2. それら機能が TCP/IP スタック外部から隠蔽されている
 3. TCP/IP スタック自体にプロトコル処理を行うスレッドが含まれる
- 結果として、
 1. **他のシステムとの統合・コンパイル自体が難しい場合がある**
 2. **機能の隠蔽によって、最適化がしにくくなる場合がある**
 3. **プロトコル処理を行うスレッドの実行形式が限定される**

1. 統合・コンパイルが難しい

- 例えば、Shenango や Caladan のように独自のユーザー空間スレッドでプロトコル処理を実行しようとする、既存の pthread や pthread を想定したロックに依存した TCP/IP スタック実装は組み合わせるのが難しい
- 新しく設計・実装された OS 等の既存の標準ライブラリ等との互換が十分でないシステムに適用するのが難しい

2. 機能隠蔽により最適化しにくくなる

- 例えば、sendfile システムコールのようにディスクと NIC 間のデータの受け渡しのメモリコピーを削減したいと思った時

既存の TCP/IP スタックが
想定する利用法

ディスクから読み出したデータ



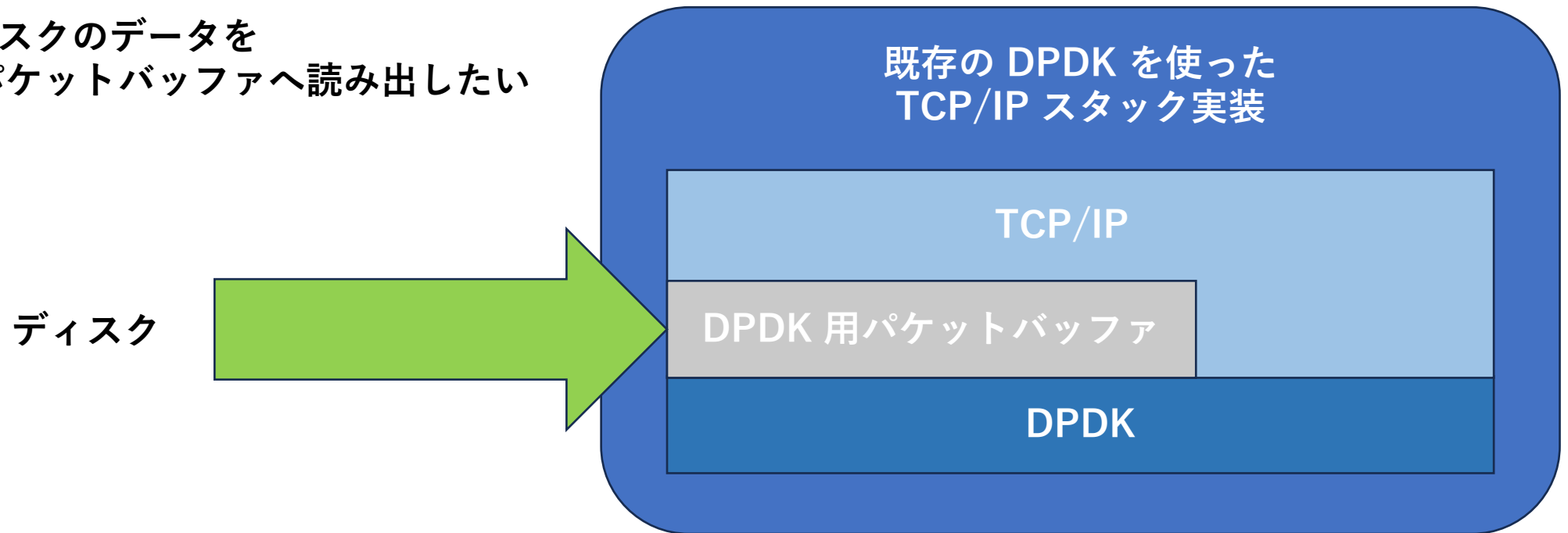
ディスク



2. 機能隠蔽により最適化しにくくなる

- 例えば、sendfile システムコールのようにディスクと NIC 間のデータの受け渡しのメモリコピーを削減したいと思った時

本当はディスクのデータを
DPDK 用パケットバッファへ読み出したい

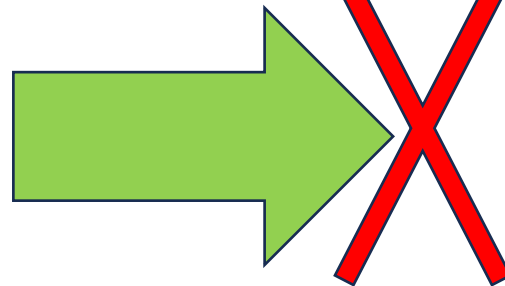


2. 機能隠蔽により最適化しにくくなる

- 例えば、sendfile システムコールのようにディスクと NIC 間のデータの受け渡しのメモリコピーを削減したいと思った時

本当はディスクのデータを
DPDK 用パケットバッファへ読み出したい

ディスク



既存の DPDK を使った
TCP/IP スタック実装

TCP/IP

DPDK 用パケットバッファ

DPDK

多くの実装で、DPDK とそのパケットバッファは
TCP/IP スタック実装内部に隠蔽され
ディスクの直接的なデータ読み込み先として指定できない

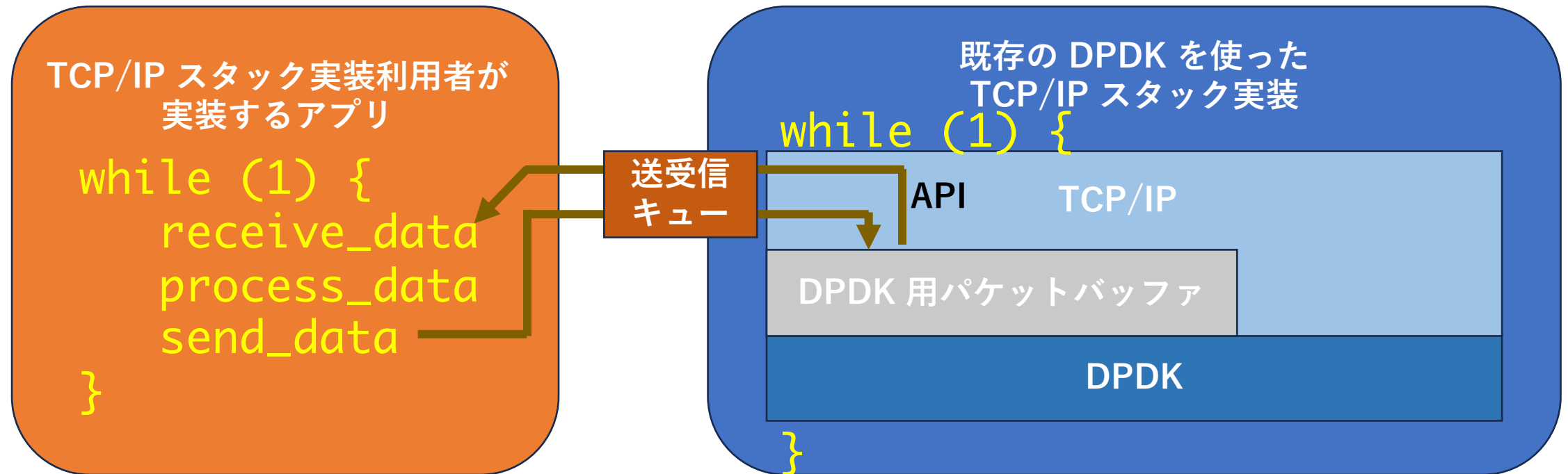
3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる



3. スレッドの実行形式が制限される

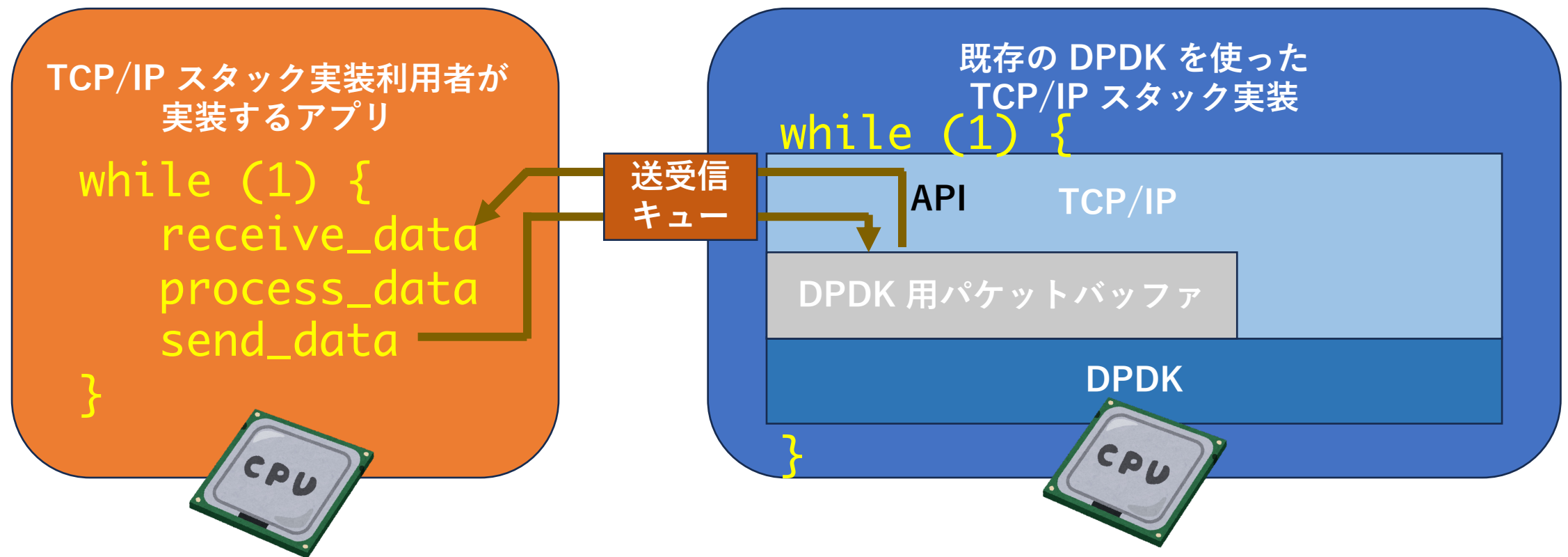
- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する



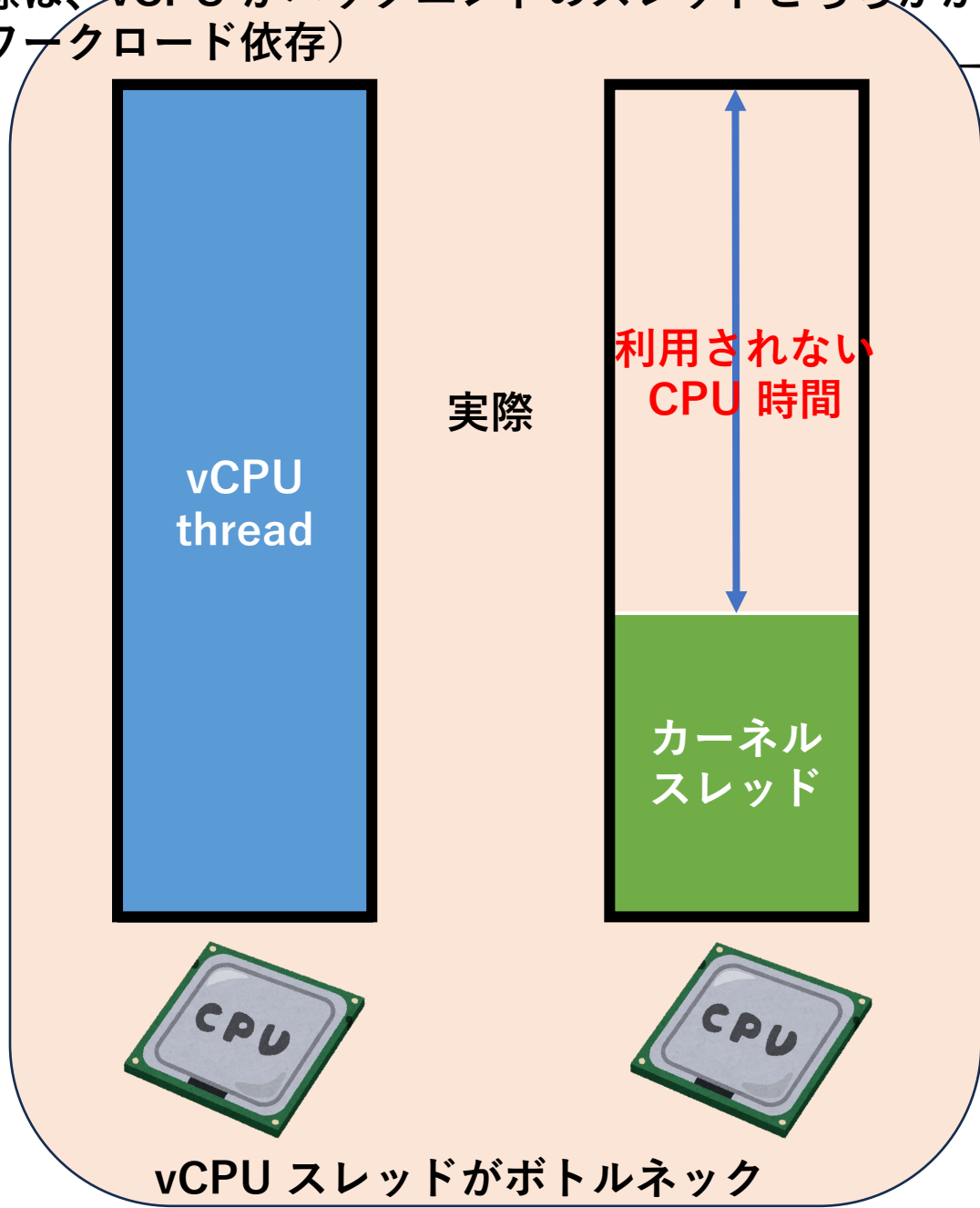
3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する

典型的な設定ではアプリと TCP/IP スタック実装のスレッドは別の CPU コアで実行する



実際は、vCPU かバックエンドのスレッドどちらかが常にボトルネックになる
(ワークロード依存)



高速化

を使った仮想スイッチを仮想マシ

- Xen に netmap/VALE を適用
- QEMU/KVM に DPDK を適用
- (AN 2016)
- Xen に netmap/VALE を適用

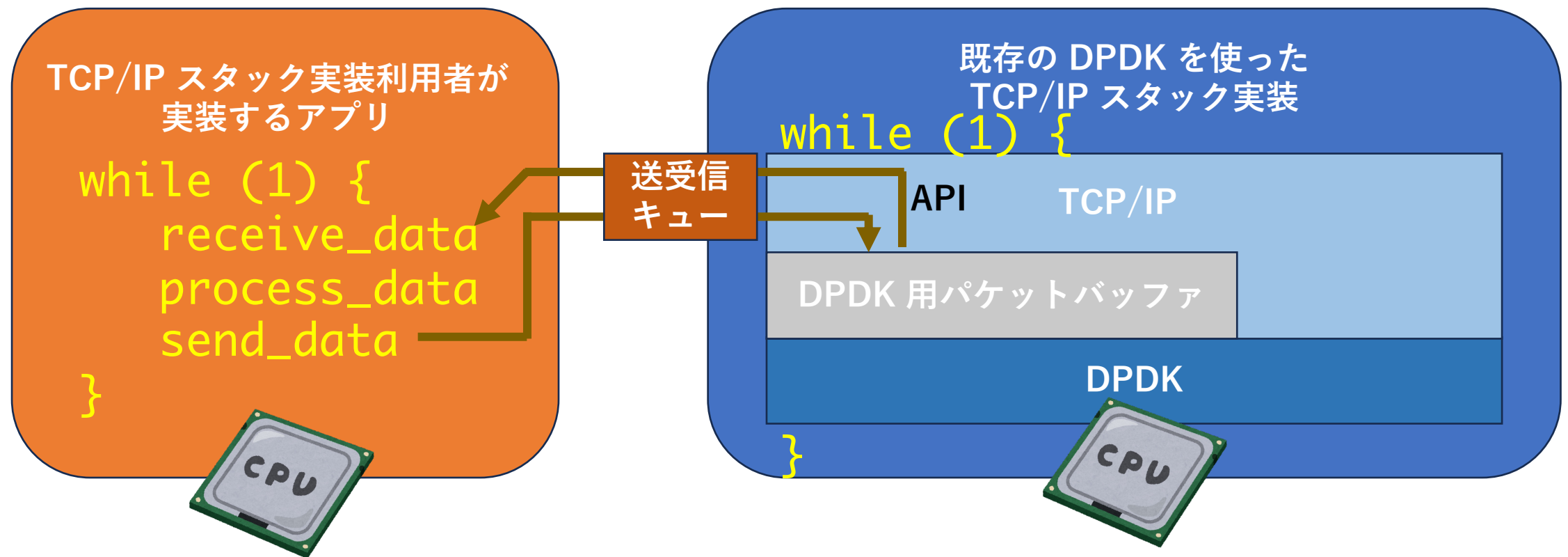
実行のモデルを改善

主張：vCPU スレッドと仮想スイッチを実行するバックエンドのスレッドを分けない方が良い

3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する

典型的な設定ではアプリと TCP/IP スタック実装のスレッドは別の CPU コアで実行する



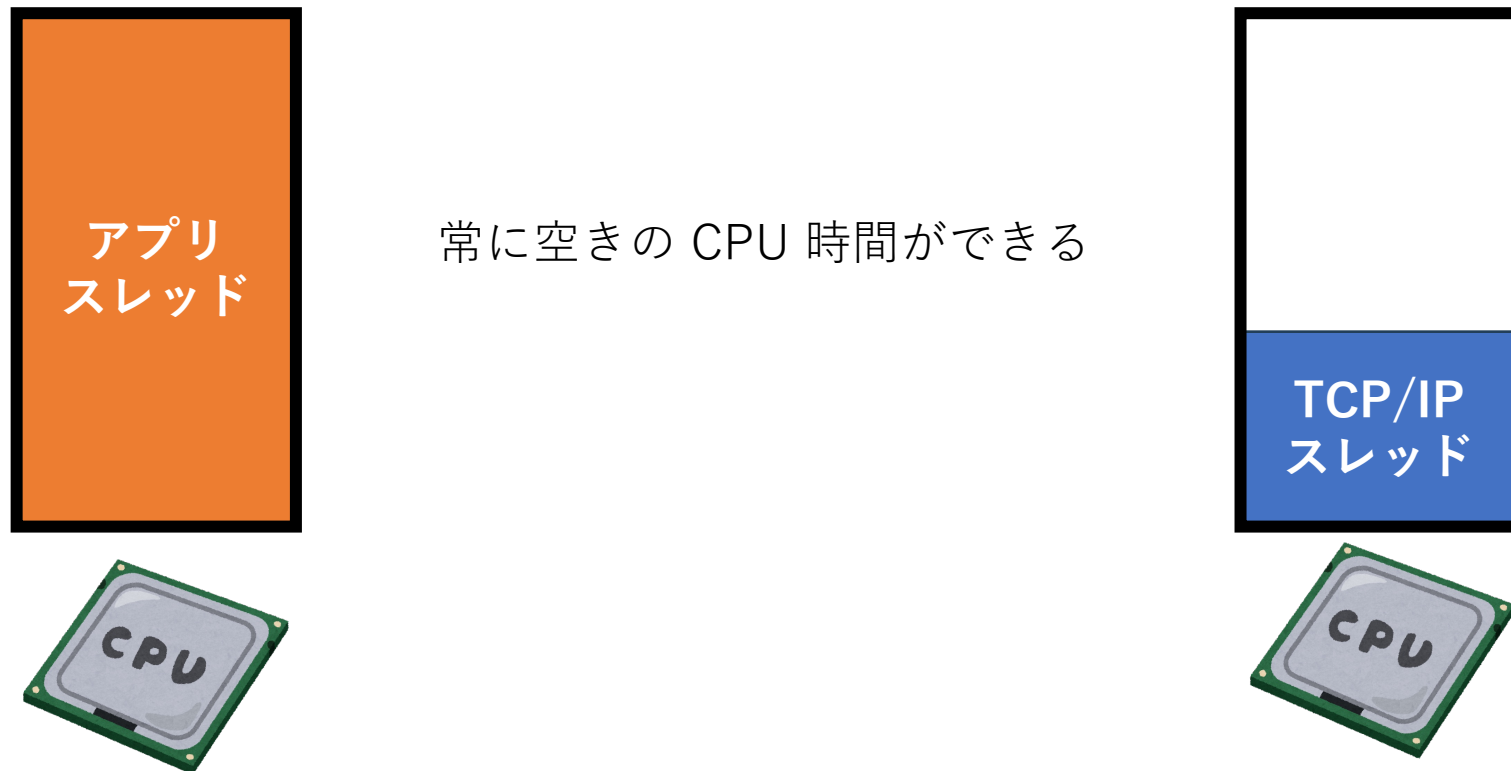
3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する
典型的な設定ではアプリと TCP/IP スタック実装のスレッドは別の CPU コアで実行する



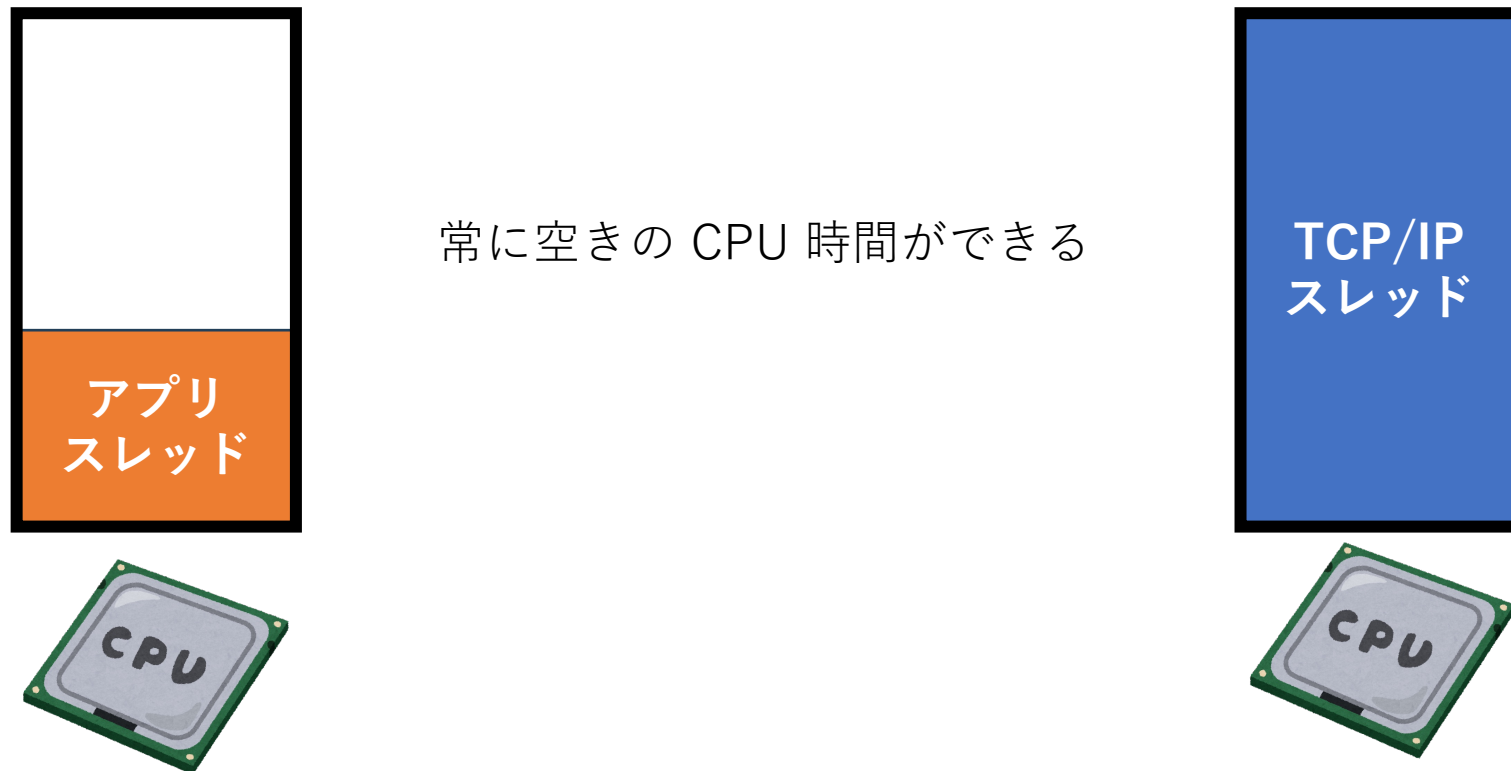
3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する
典型的な設定ではアプリと TCP/IP スタック実装のスレッドは別の CPU コアで実行する



3. スレッドの実行形式が制限される

- 既存の実装の多くは自前でプロトコル処理を行うスレッドを含んでいる
TCP/IP スタック実装利用者は以下のような感じでアプリを実装する
典型的な設定ではアプリと TCP/IP スタック実装のスレッドは別の CPU コアで実行する



3. スレッドの実行形式が制限される

- 理想的には
 - NIC からデータを受け取る
 - TCP/IP スタック受信処理
 - アプリ固有処理
 - TCP/IP スタック送信処理
 - NIC からデータを送信する
- 上記を一つのスレッドで実行できた方が嬉しい

lwIP

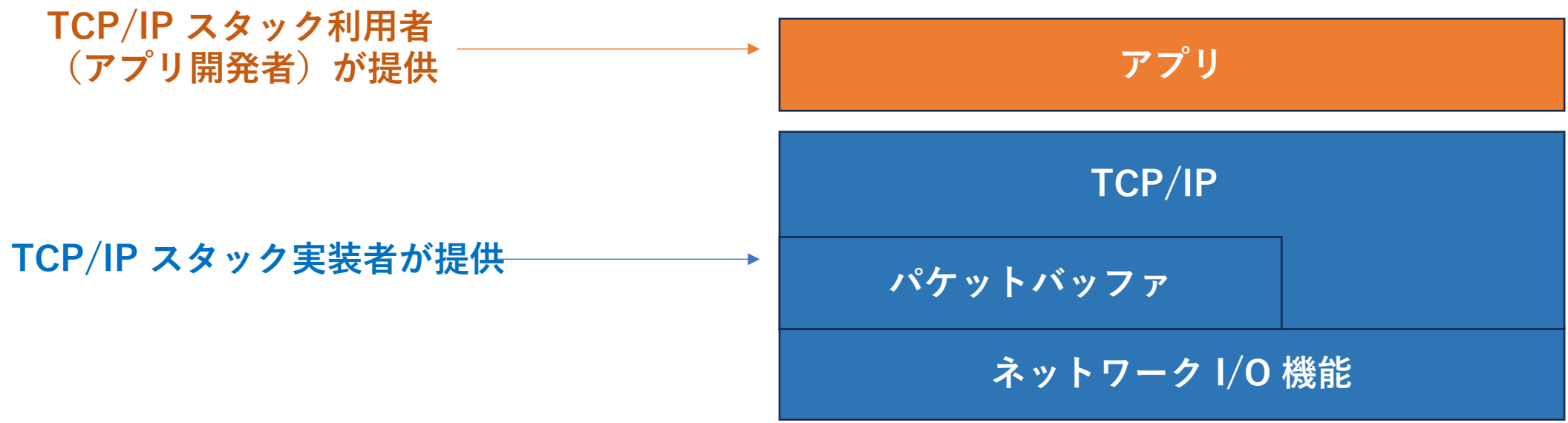
- 小さい組み込みデバイスを想定したポータブルな TCP/IP 実装
 - (個人的に) 非常に利用しやすい上に性能も高い
- 一方、
 - NIC のオフローディング機能に対応していない
 - NIC とアプリの間でコピーを削除しきれない
 - 複数スレッドで同時に lwIP を実行できるように作られていない

モチベーション

- 以下のような特性を持つ TCP/IP 実装が欲しい
 1. プロトコル処理の実装が特定の CPU、NIC、OS、ライブラリ、コンパイラ機能に依存しない
 2. 外部の実装に対して、隠蔽する機構が最小限
 3. TCP/IP スタックがプロトコル処理を実行するスレッドを持たない
 4. NIC のオフローディング機能を使える
 5. NIC とアプリの間でコピーをなくすことができる
 6. 複数スレッドで実行可能でマルチコア環境で性能がスケールする

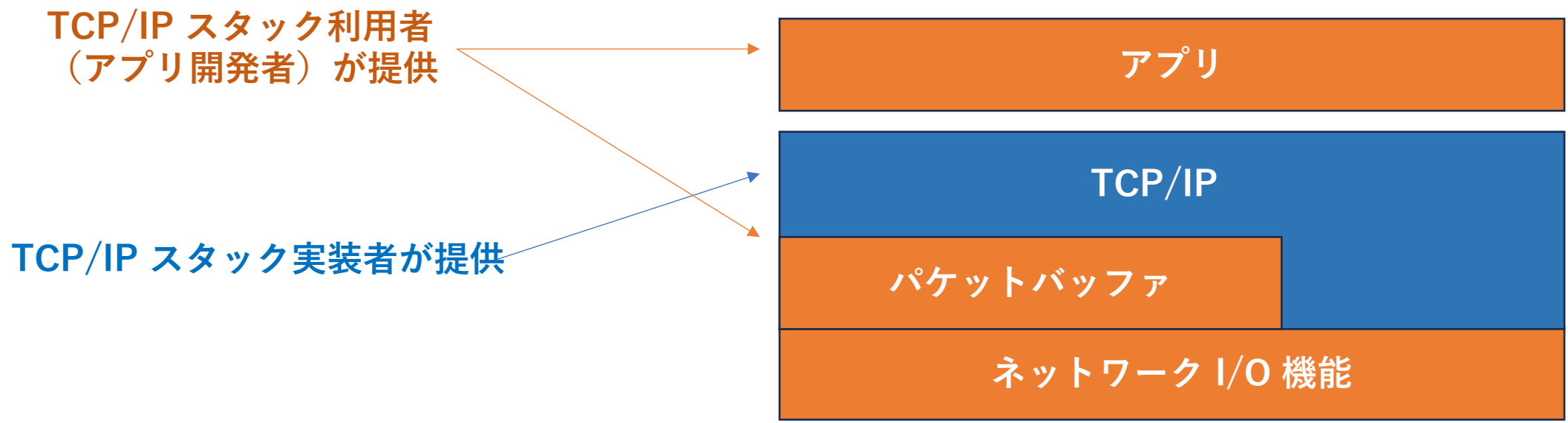
実装ポイント

多くの既存の実装の構成



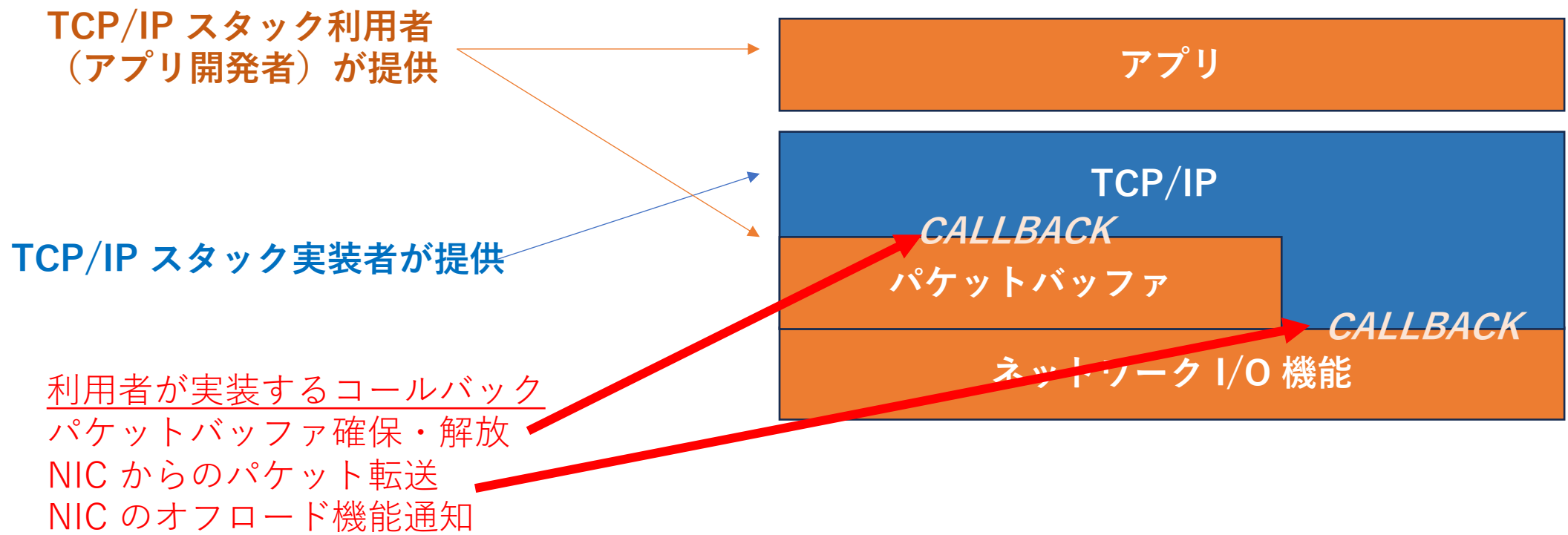
実装ポイント

今回の実装の構成



実装ポイント

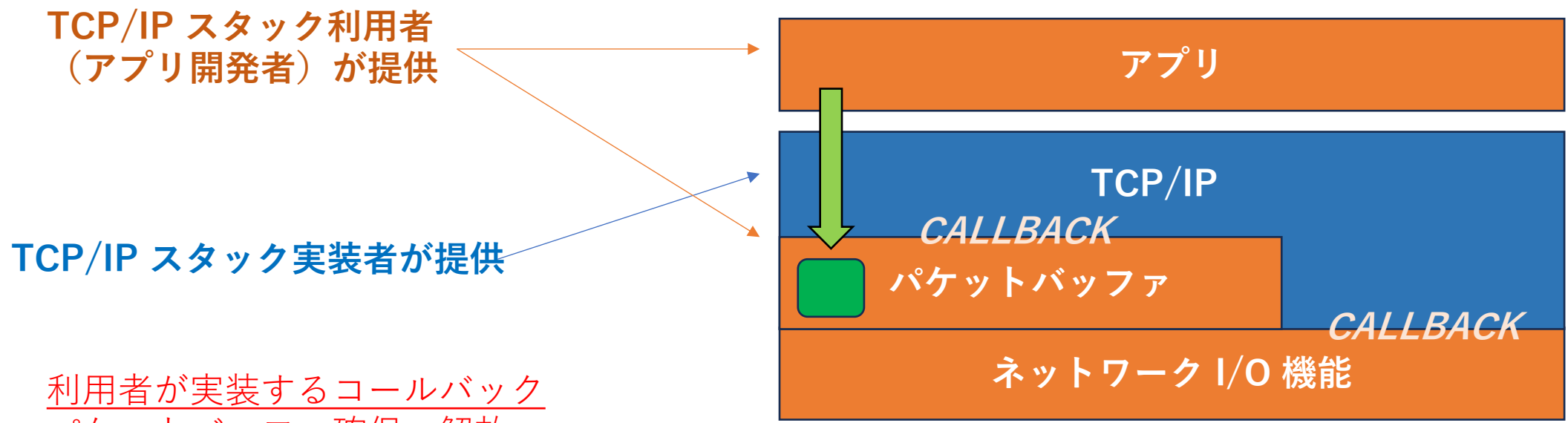
今回の実装の構成



実装ポイント

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める



- 利用者が実装するコールバック
- パケットバッファ確保・解放
- NIC からのパケット転送
- NIC のオフロード機能通知

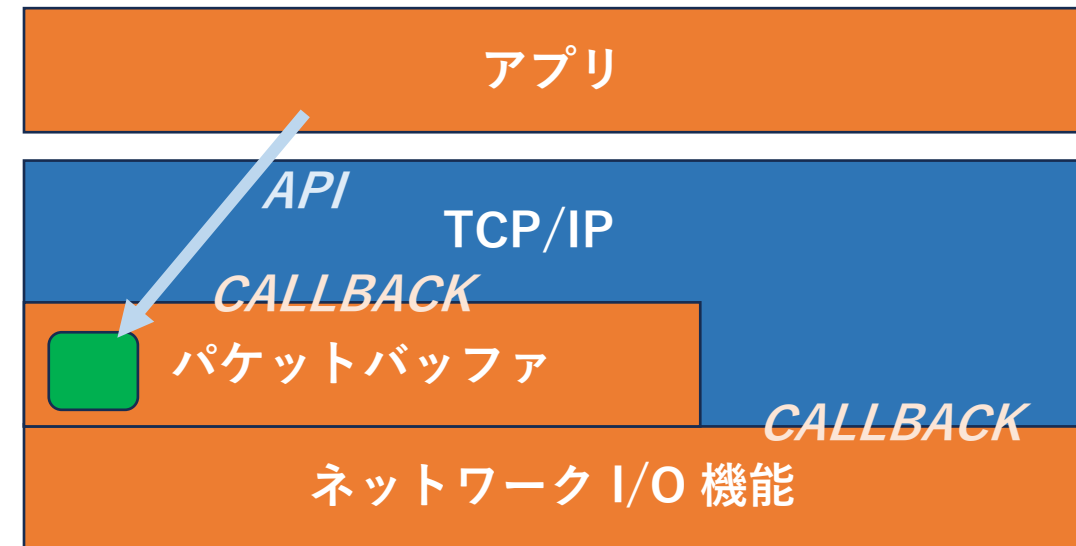
実装ポイント

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す

TCP/IP スタック利用者
(アプリ開発者) が提供

TCP/IP スタック実装者が提供



利用者が実装するコールバック
パケットバッファ確保・解放
NIC からのパケット転送
NIC のオフロード機能通知

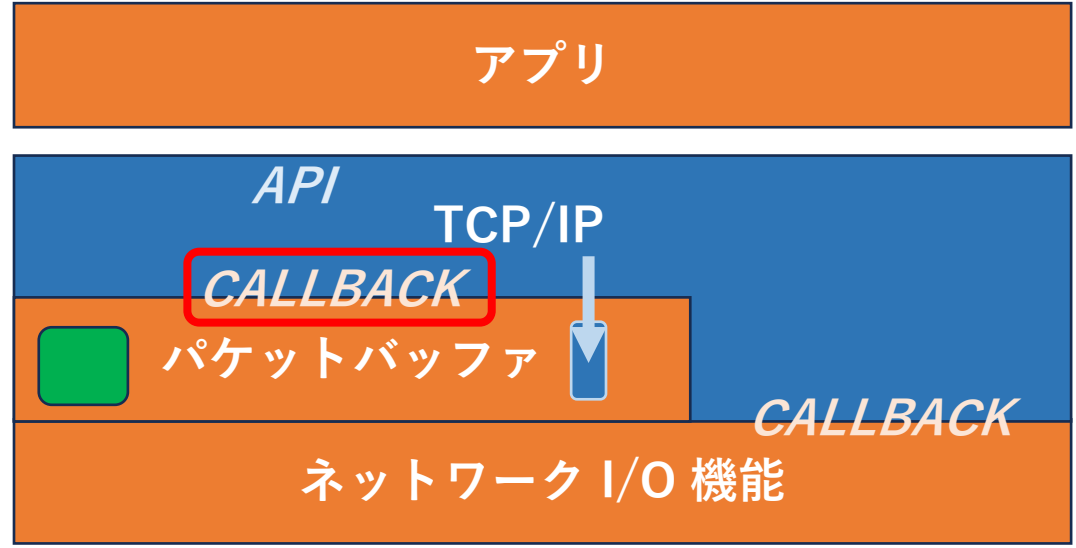
実装ポイント

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す

TCP/IP スタック利用者
(アプリ開発者) が提供

TCP/IP スタック実装者が提供



利用者が実装するコールバック
パケットバッファ確保・解放
NIC からのパケット転送
NIC のオフロード機能通知

TCP/IP スタックは利用者が実装したコールバックを使って
ヘッダを配置するためのパケットバッファを確保+ヘッダを用意

実装ポイント

今回の実装の構成

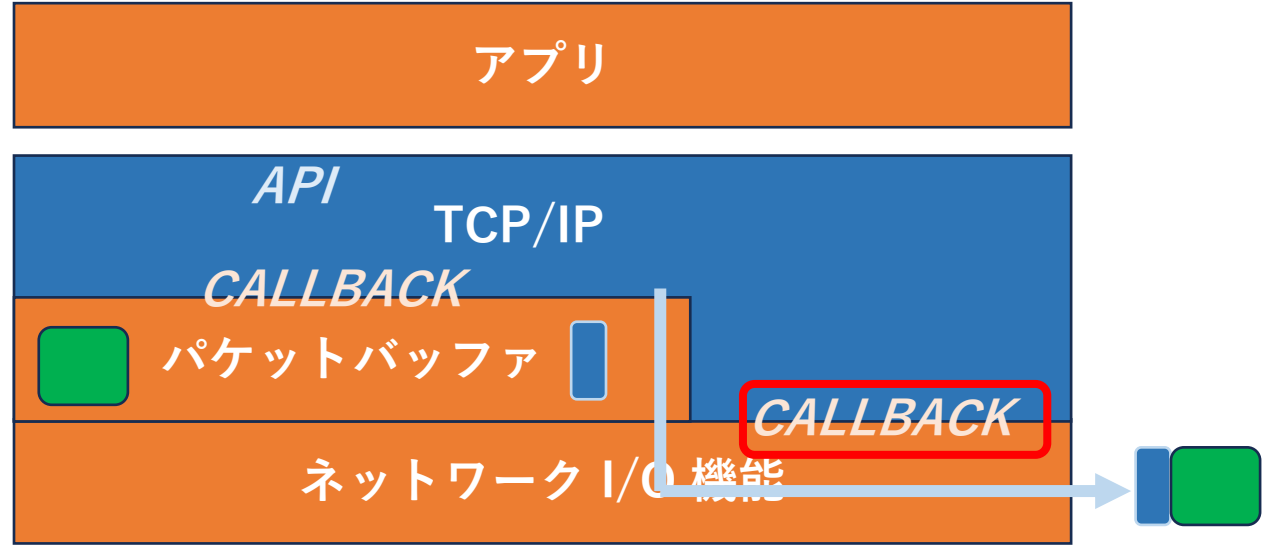
アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す

TCP/IP スタック利用者
(アプリ開発者) が提供

TCP/IP スタック実装者が提供

利用者が実装するコールバック
パケットバッファ確保・解放
NIC からのパケット転送
NIC のオフロード機能通知

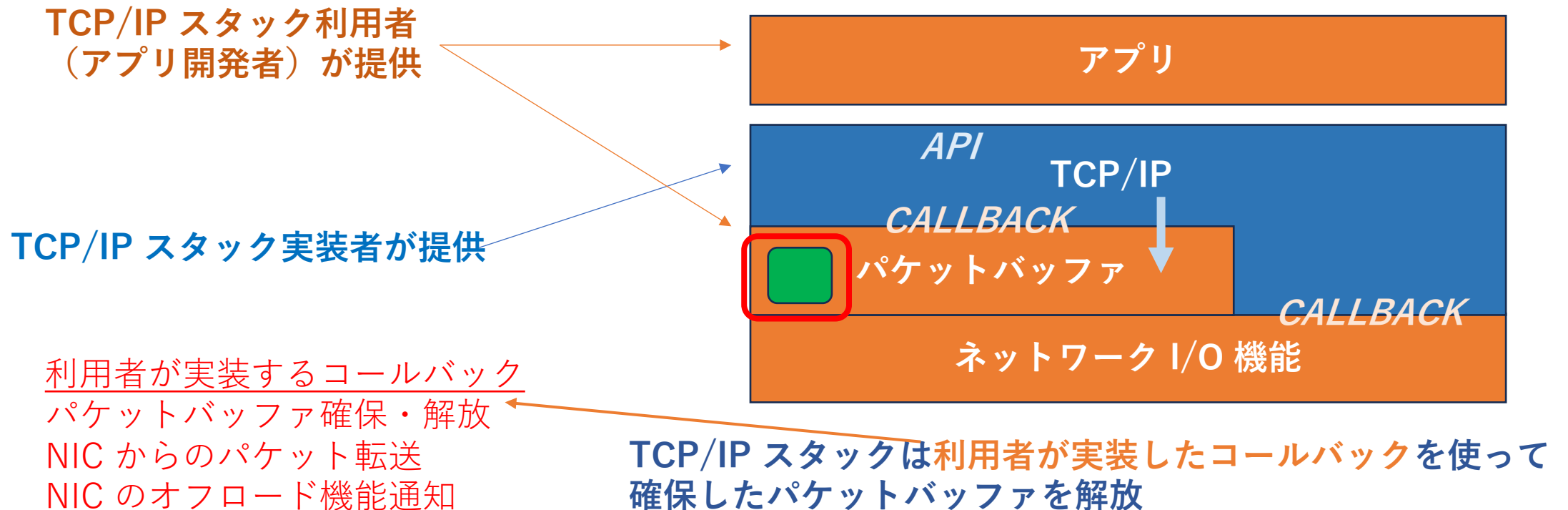
TCP/IP スタックは利用者が実装したコールバックを使って
NIC の Scatter Gather 機能でペイロードにヘッダを結合して
パケットを送信：ペイロードのメモリコピーはなし



実装ポイント

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す



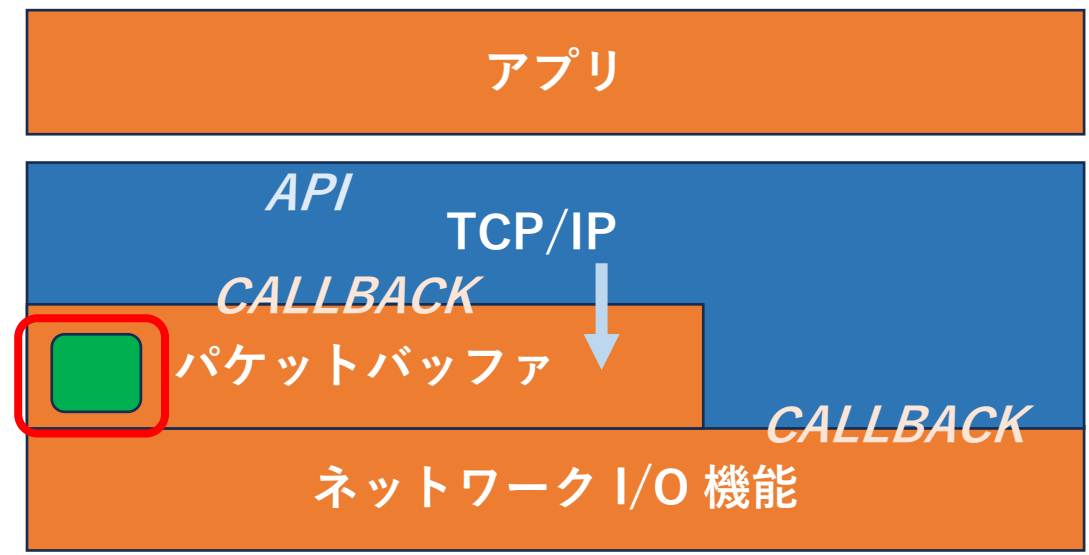
実装ポイント

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す

TCP/IP スタック利用者
(アプリ開発者) が提供

TCP/IP スタック実装者が提供



利用者が実装するコールバック
パケットバッファ確保・解放
NIC からのパケット転送
NIC のオフロード機能通知

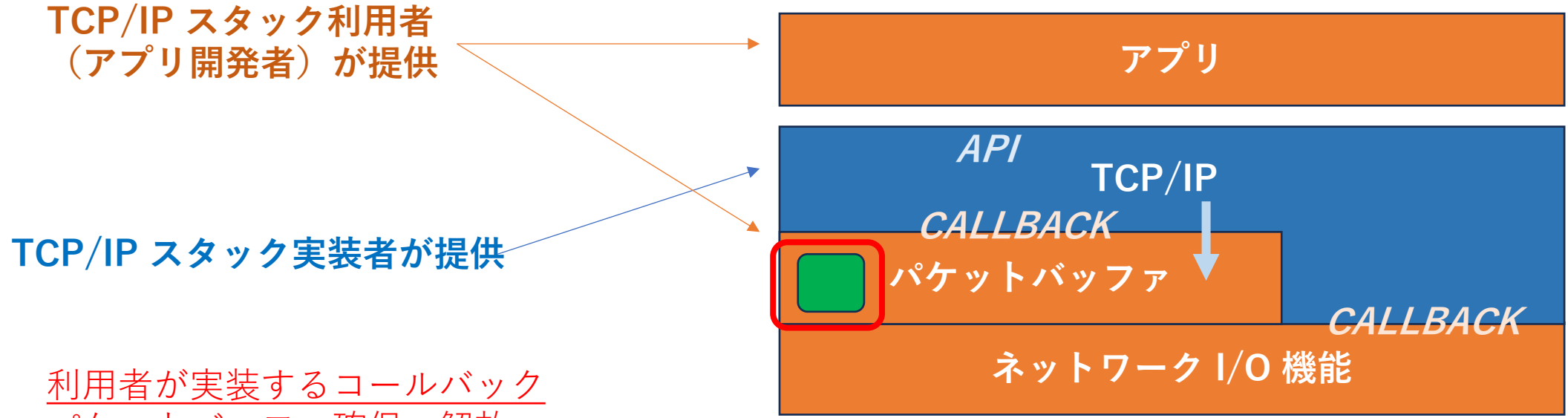
ここで利用者がアプリで書き込んだパケットバッファを解放しなければ、同じパケットバッファ上のデータを別の宛先へ送ることもできます

実装ポイント

利用者にこのような自由度を残せるところが TCP/IP スタック実装が機能の隠蔽を行わない利点

今回の実装の構成

アプリはパケットバッファへ直接送信したいデータを書き込める
送信用 API には、パケットバッファへのポインタを渡す



利用者が実装するコールバック
パケットバッファ確保・解放
NIC からのパケット転送
NIC のオフロード機能通知

ここで利用者がアプリで書き込んだパケットバッファを解放しなければ、同じパケットバッファ上のデータを別の宛先へ送ることもできます

実験環境 (同じ設定のマシン 2 台)

CPU: 2 x 16-core Intel Xeon Gold 6326 CPU @ 2.90GHz (合計 32 コア)

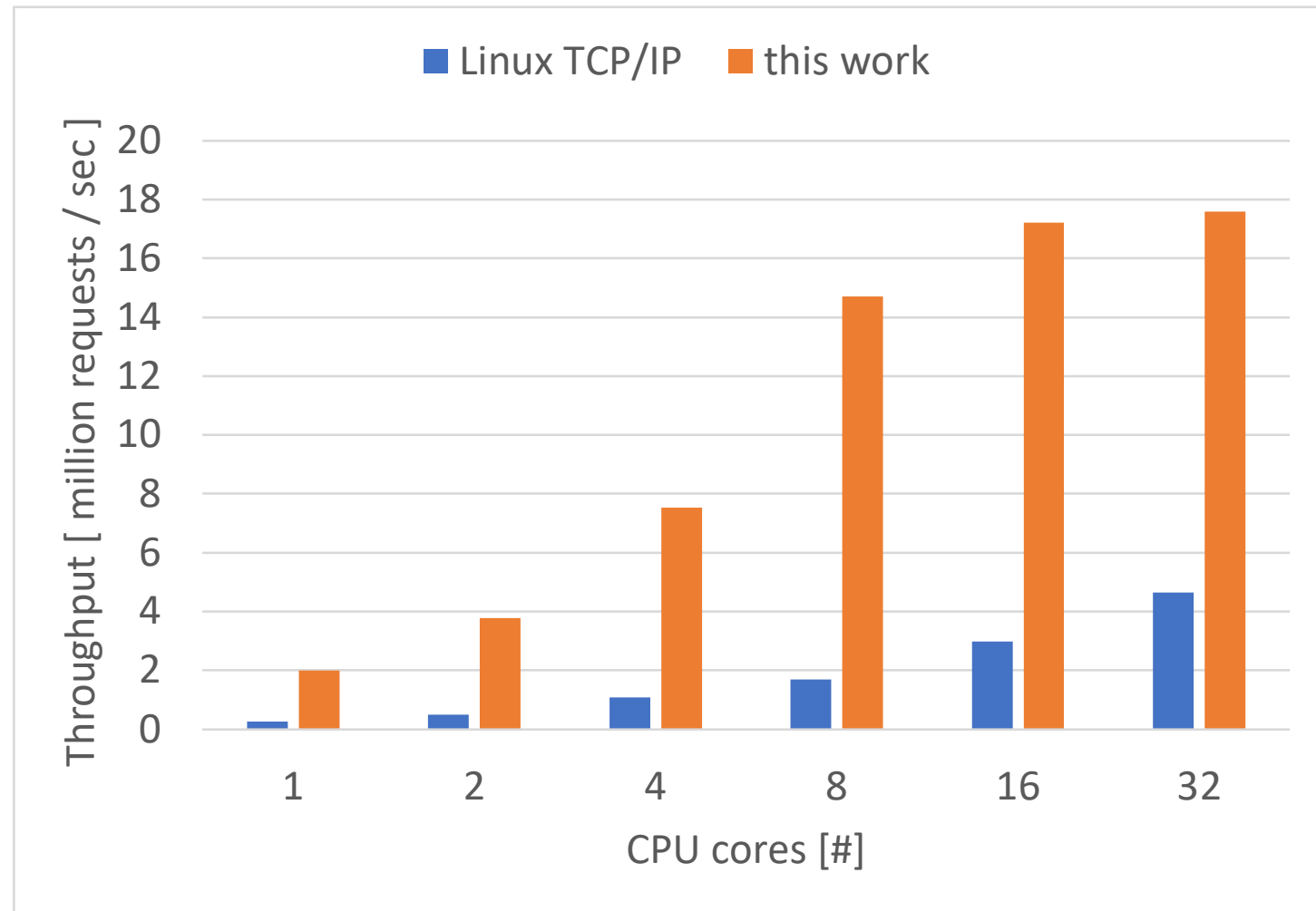
NIC: Mellanox ConnectX-5 100 Gbps NIC (マシン間はケーブルを直繋ぎして接続)

OS: Linux 6.2

性能

• ベンチマーク

- TCP ペイロードは 4 ~ 64 バイト
- サーバーが各 CPU コアが 16 TCP 接続へ応答するようクライアントは接続数を調整
- TCP 接続は確立後切断しない
- なるべく高速にメッセージの交換を行う



性能

実験環境

CPU: 2 x 16-core Intel Xeon Gold 6326 CPU @ 2.90GHz (合計 32 コア)

NIC: Mellanox ConnectX-5 100 Gbps NIC (マシン間はケーブルを直繋ぎして接続)

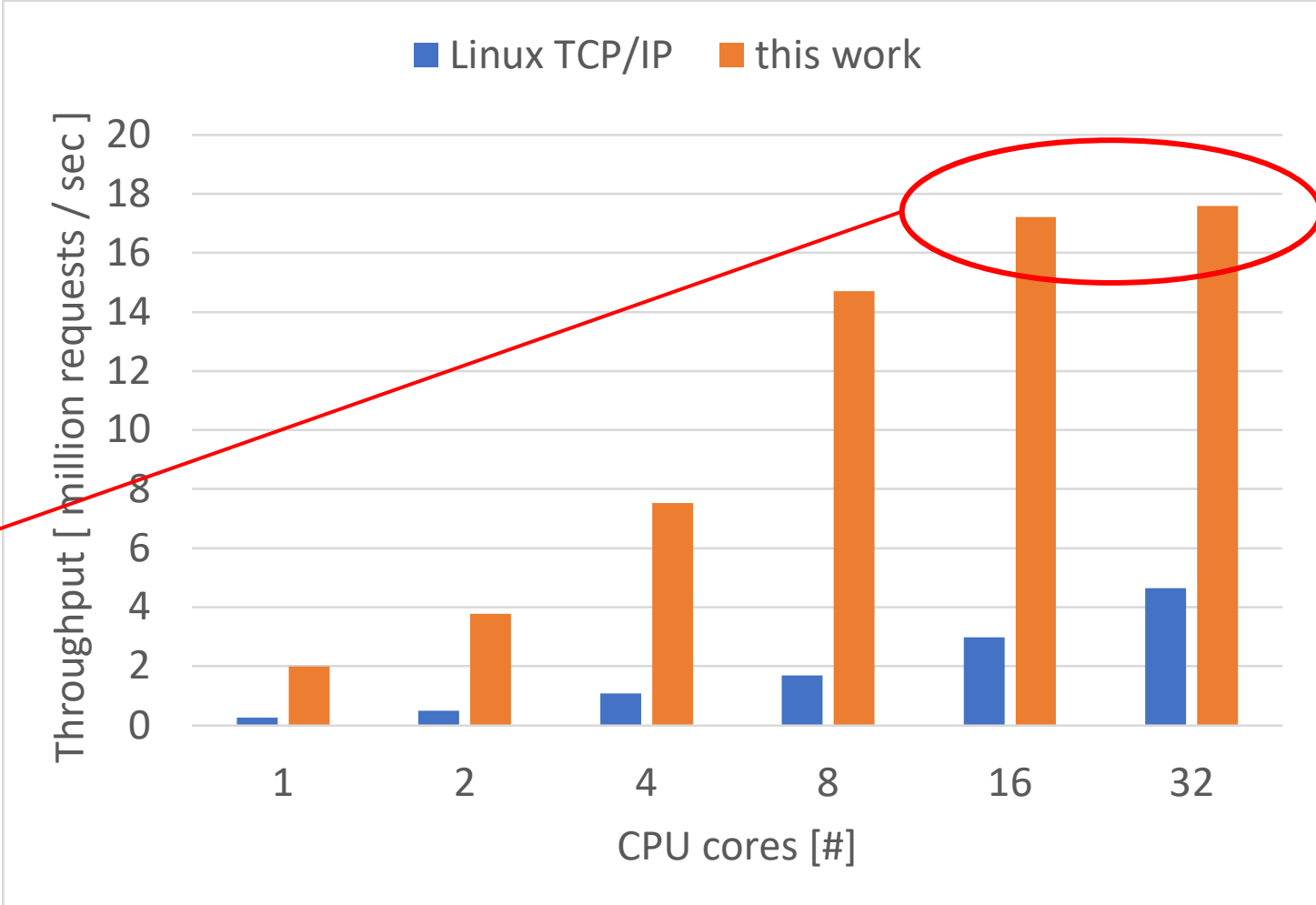
OS: Linux 6.2

• ベンチマーク

- TCP ペイロードは 4 ~ 64 バイト
- サーバーが各 CPU コアが 16 TCP 接続へ 応答するようクライアントは接続数を調整
- TCP 接続は確立後切断しない
- なるべく高速にメッセージの交換を行う

16 CPU コア以降、コア数を増やしてもあまり速くならなかった

何故？



簡単な調査

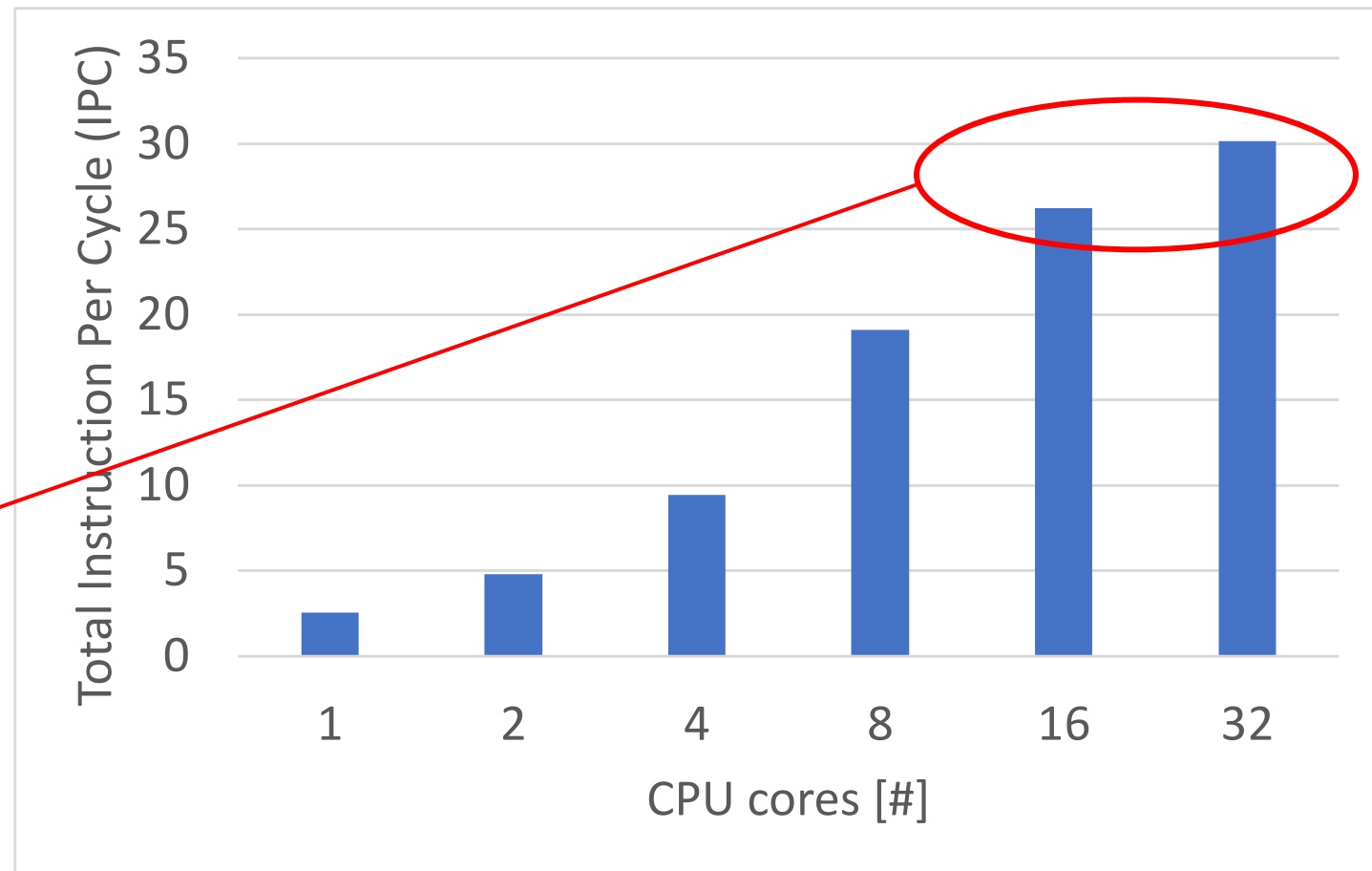
- pqos コマンドでメモリに関する情報を取得
 - Instruction Per Cycle (IPC)
 - Cache Miss
 - Last-Level Cache occupancy
 - Memory Bandwidth

<https://github.com/intel/intel-cmt-cat/wiki/PQoS-monitoring-metrics-definition>

Instruction Per Cycle (IPC)

- 利用している CPU コア
全ての IPC の合計

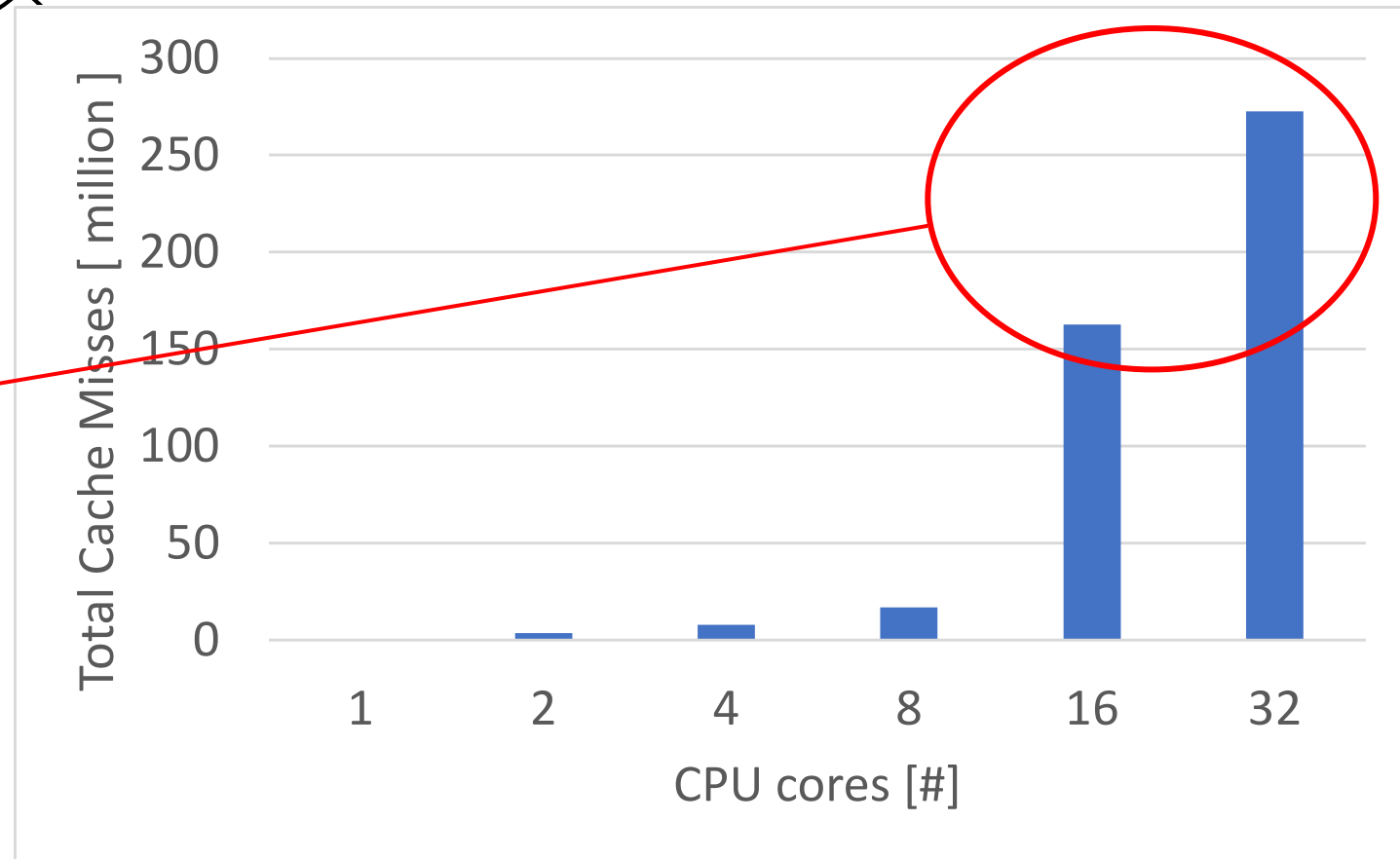
16 CPU コア以降、コア数を増やしても
IPC があまり増えていない



キャッシュミス回数

- 利用している CPU コアで観測されたキャッシュミス回数の合計

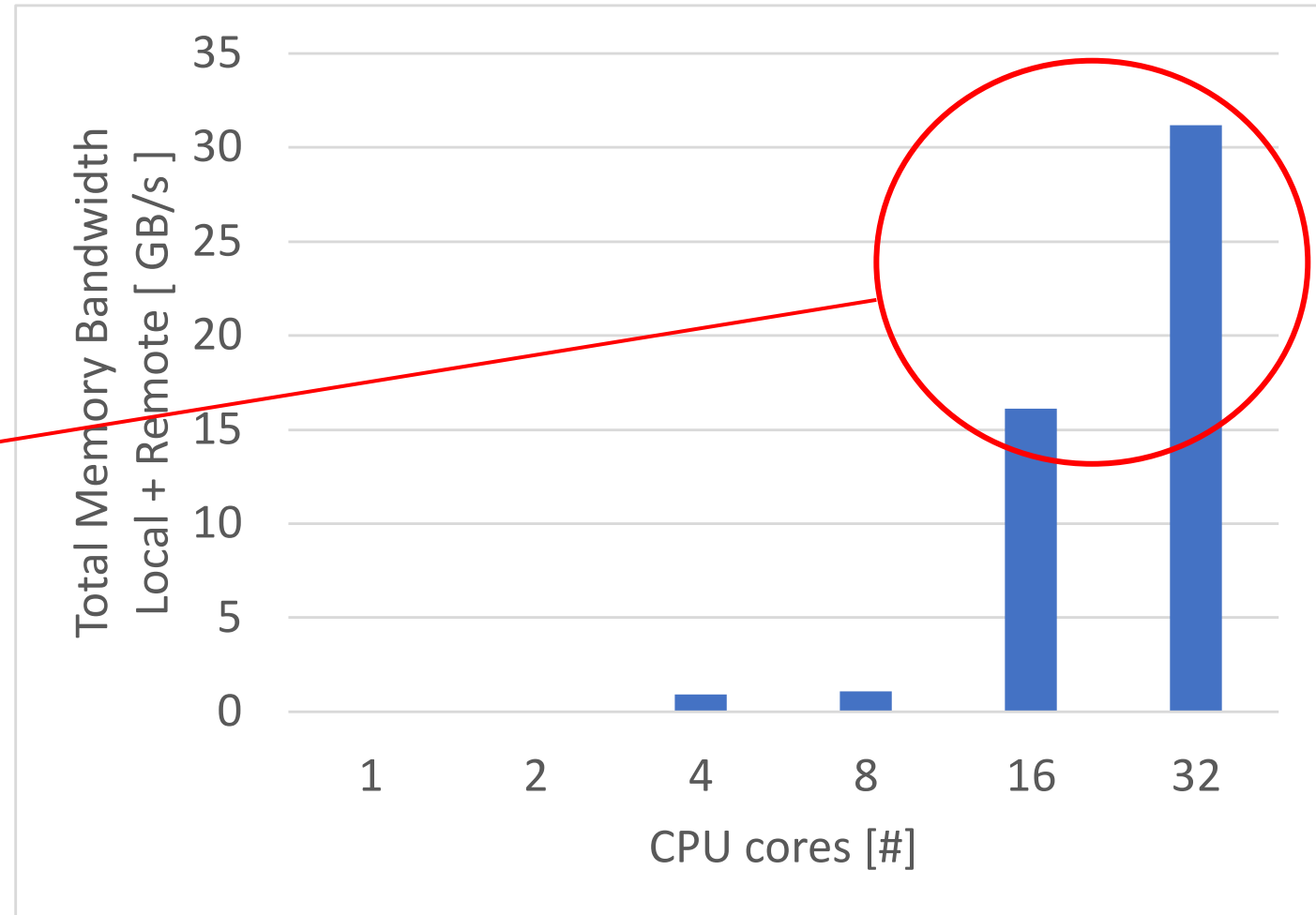
16 CPU コア以降、8 コアまでの時と比べてキャッシュミス回数が大幅に増加



メモリ帯域使用状況

- 利用している CPU コア
で観測されたメモリ帯域
使用の合計

キャッシュミス増加に合わせて
メモリ帯域の利用が増加

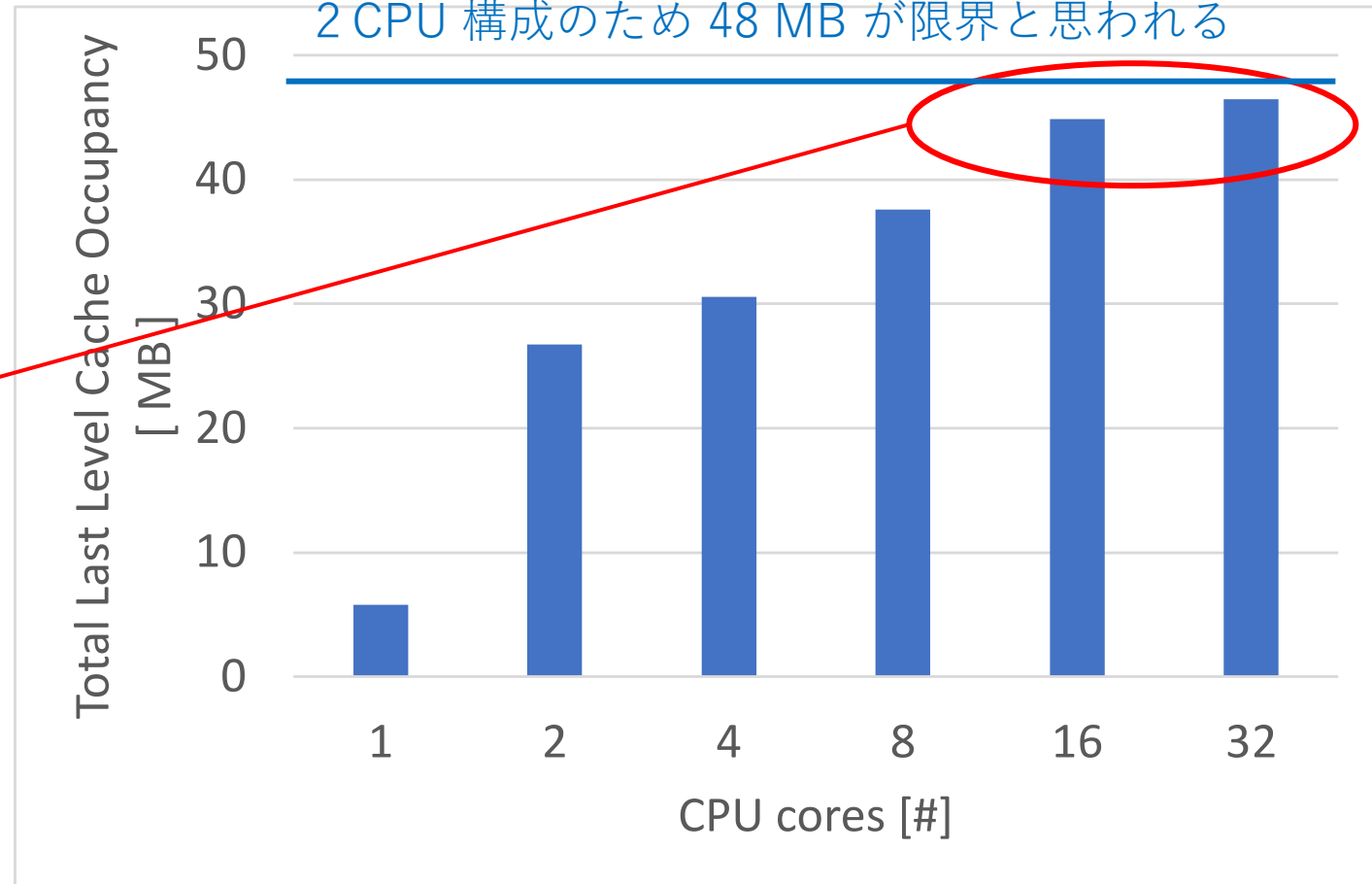


キャッシュ占有状況

- 利用している CPU コアの占有しているキャッシュサイズの合計

コアを増やしてもデータがキャッシュに乗らなくなって性能が制限されているかも？

今回のマシンの CPU は 1 つあたり 24 MB のキャッシュを持っており、2 CPU 構成のため 48 MB が限界と思われる

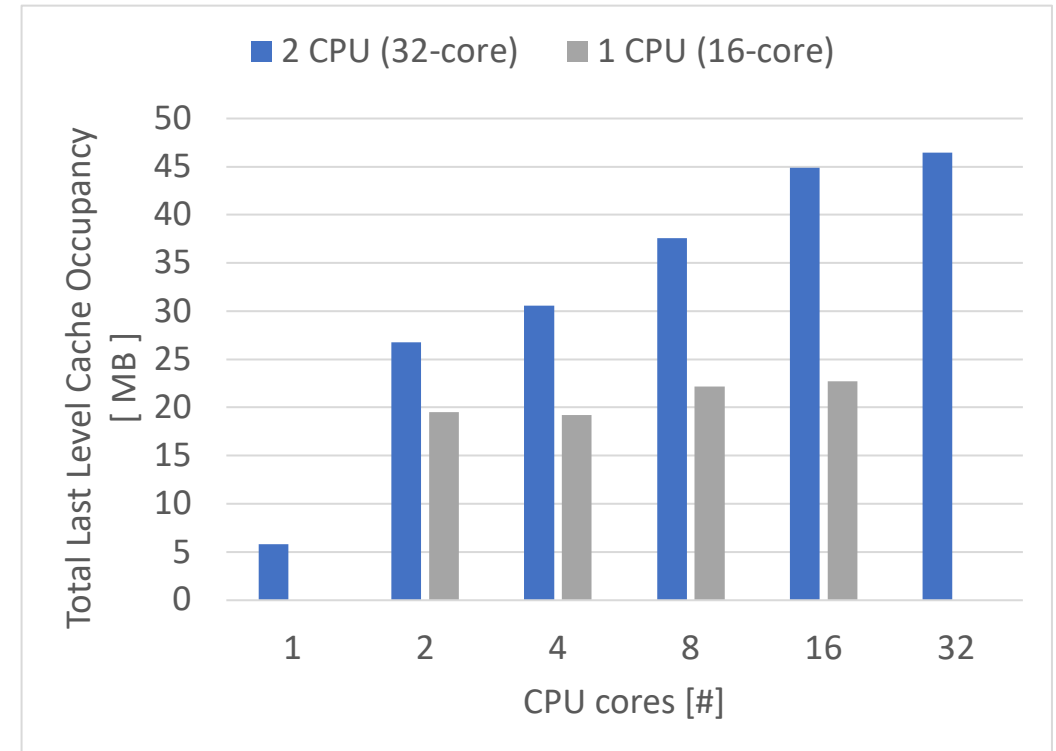
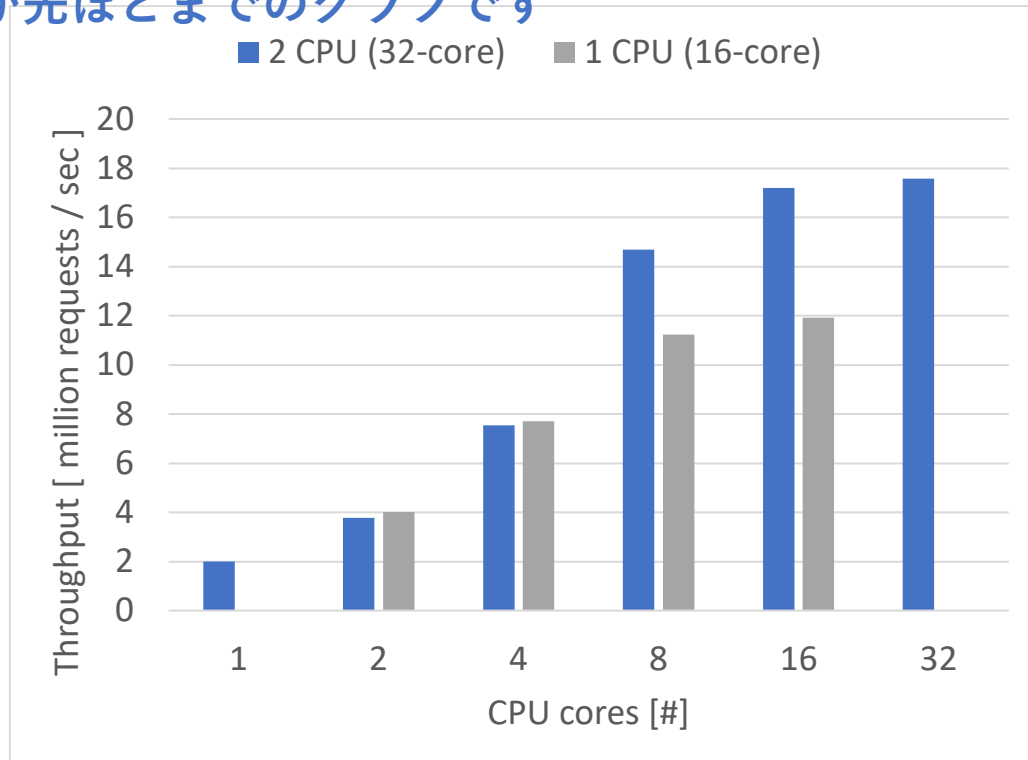


利用する CPU の数を 1 つにしてみる

- 先ほどまでは 2 つの CPU のコアを同数利用していたので、今度は全てのスレッドを同じ CPU で動かしてみる

青が先ほどまでのグラフです

使えるキャッシュサイズが前の実験と比べて半分になる

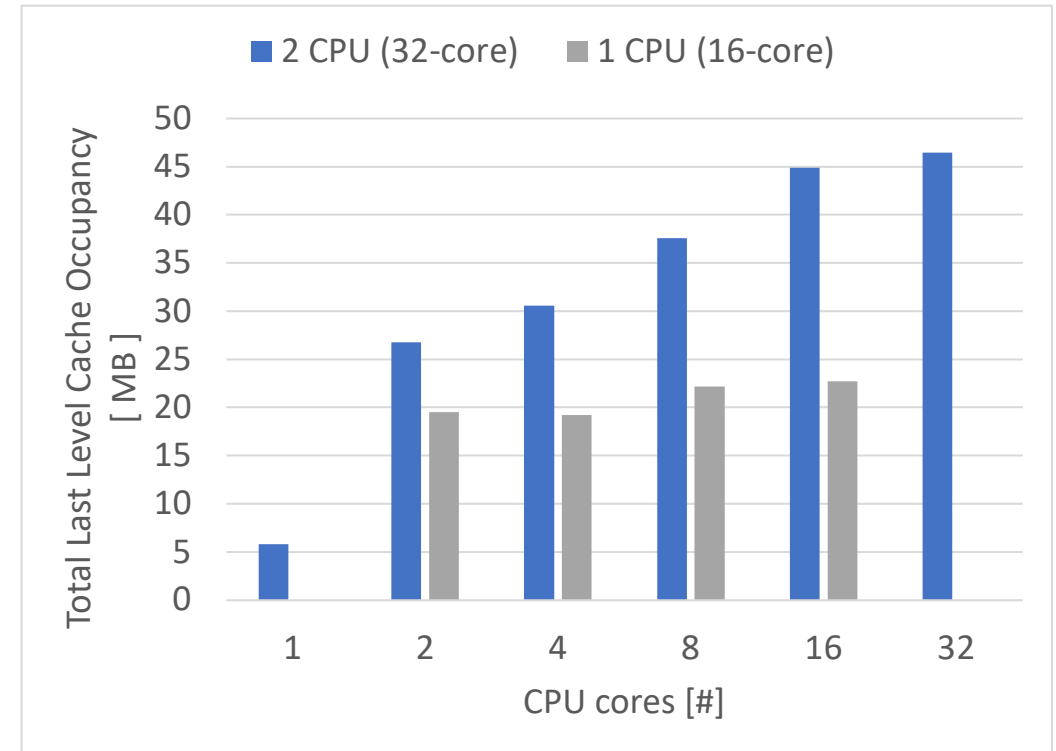
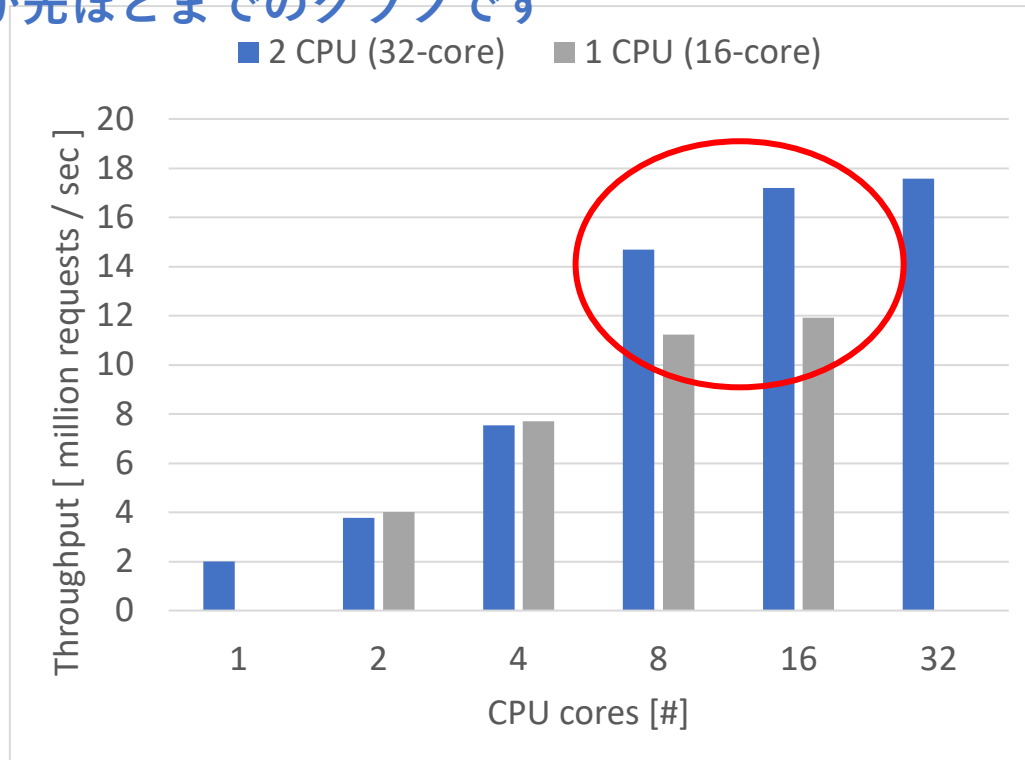


利用する CPU の数を 1 つにしてみる

- 先ほどまでは 2 つの CPU のコアを同数利用していたので、今度は全てのスレッドを同じ CPU で動かしてみる

青が先ほどまでのグラフです

使えるキャッシュサイズが前の実験と比べて半分になる



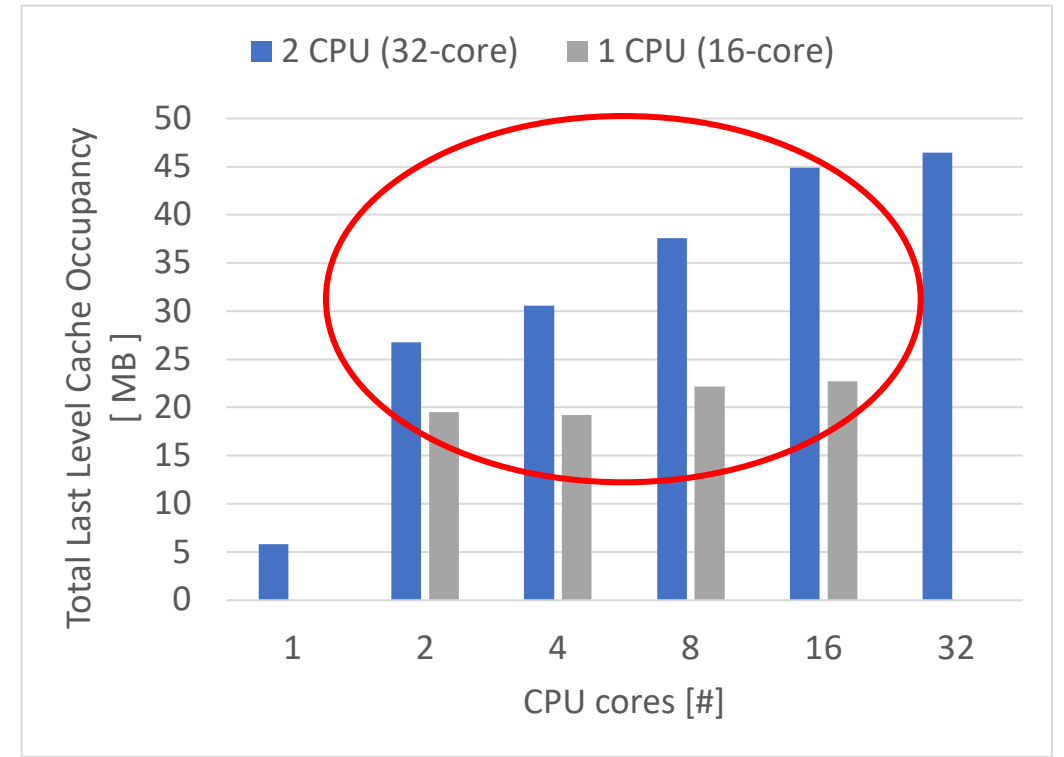
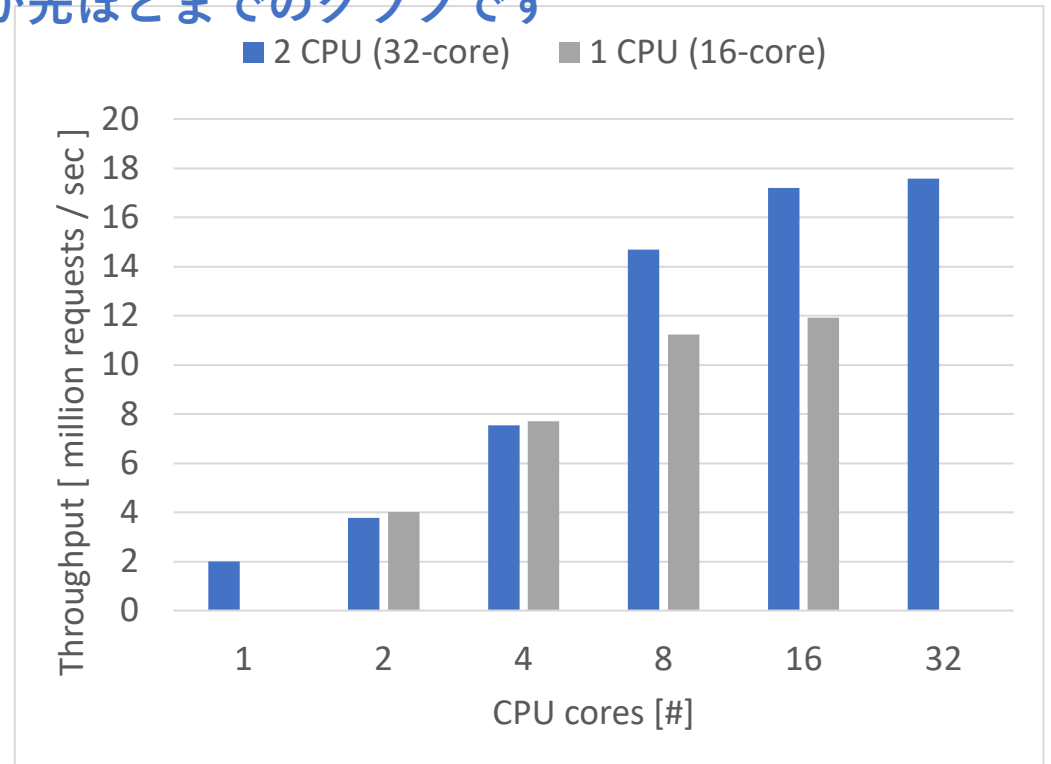
1 CPUの時の方が同じ CPU コア数でも性能が下がった

利用する CPU の数を 1 つにしてみる

- 先ほどまでは 2 つの CPU のコアを同数利用していたので、今度は全てのスレッドを同じ CPU で動かしてみる

青が先ほどまでのグラフです

使えるキャッシュサイズが前の実験と比べて半分になる



1 CPUの時の方が同じ CPU コア数でも性能が下がった

使えるキャッシュサイズは性能に影響がありそう

まとめ

まとめ

- NIC の高速化による性能の伸び代を活かす研究についてご紹介しました
 - 引用等は技術レポート「Internet Infrastructure Review (IIR) Vol. 60」をご参照ください
 - HTML / PDF 版：<https://www.ij.ad.jp/dev/report/iir/060.html>
- TCP/IP スタックを自作してみて、比較的最近のハードウェアでの性能の限界がどこから来るか簡単に調査してみました
 - よろしければお試しく下さい：<https://github.com/yasukata/iip>