2023 年 12 月 7 日 第 35 回 コンピューター・シンポジウム (ComSys 2023)

高い性能と可搬性を目指した TCP/IP スタックの設計と実装

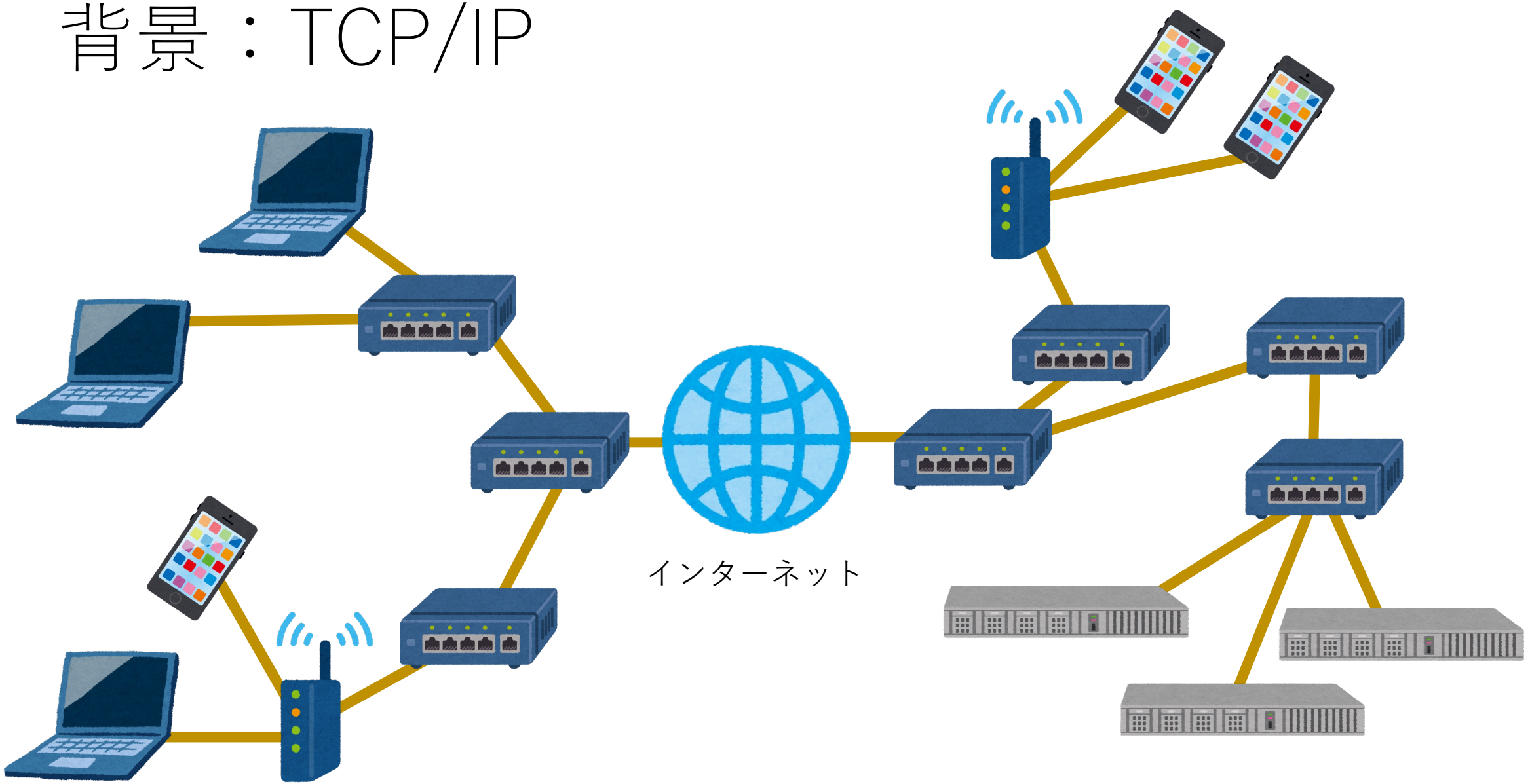
IIJ 技術研究所

安形 憲一

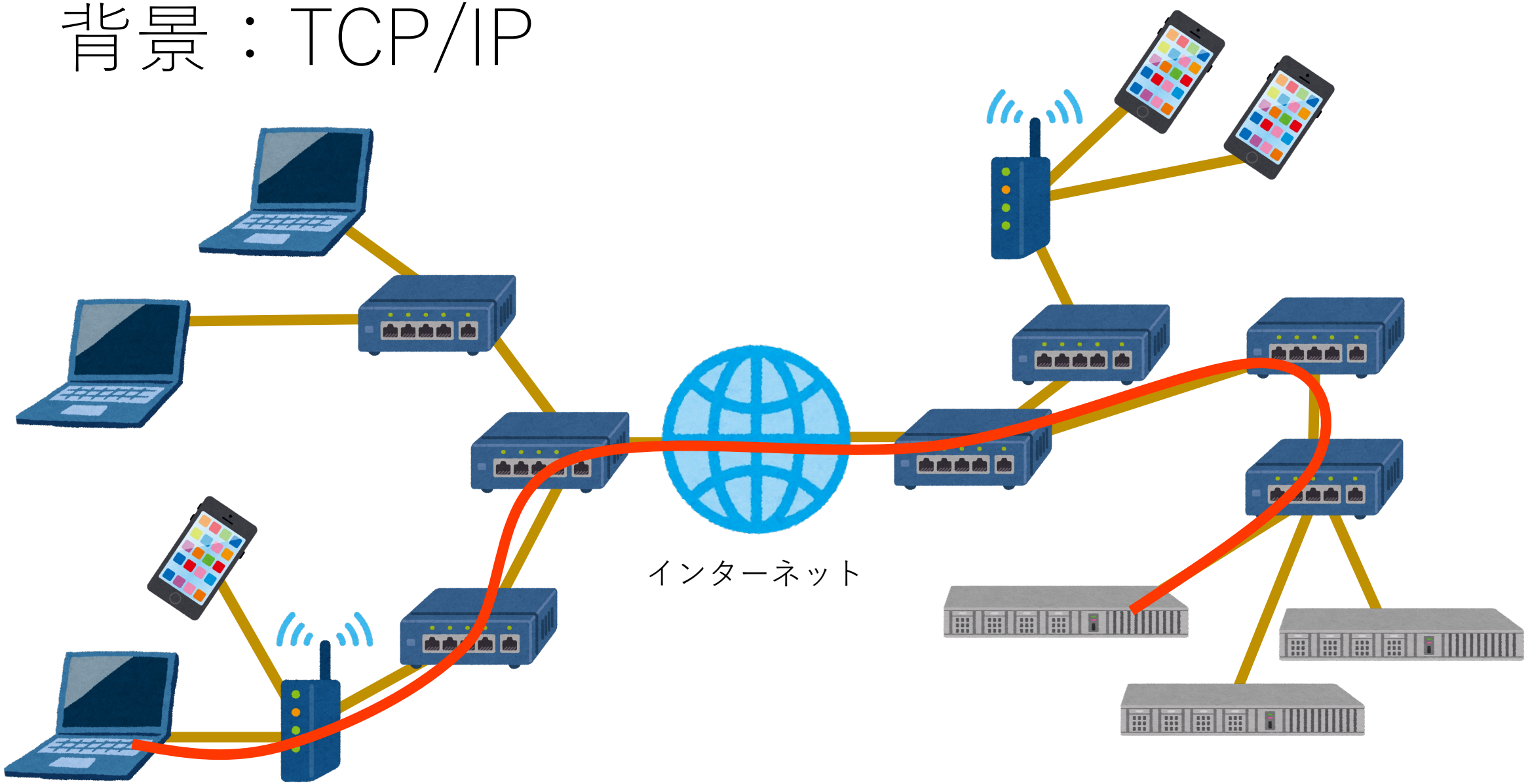
概要

- 様々なシステムに組み込みやすく、かつ、高い性能を発揮できる TCP/IP スタック実装の模索

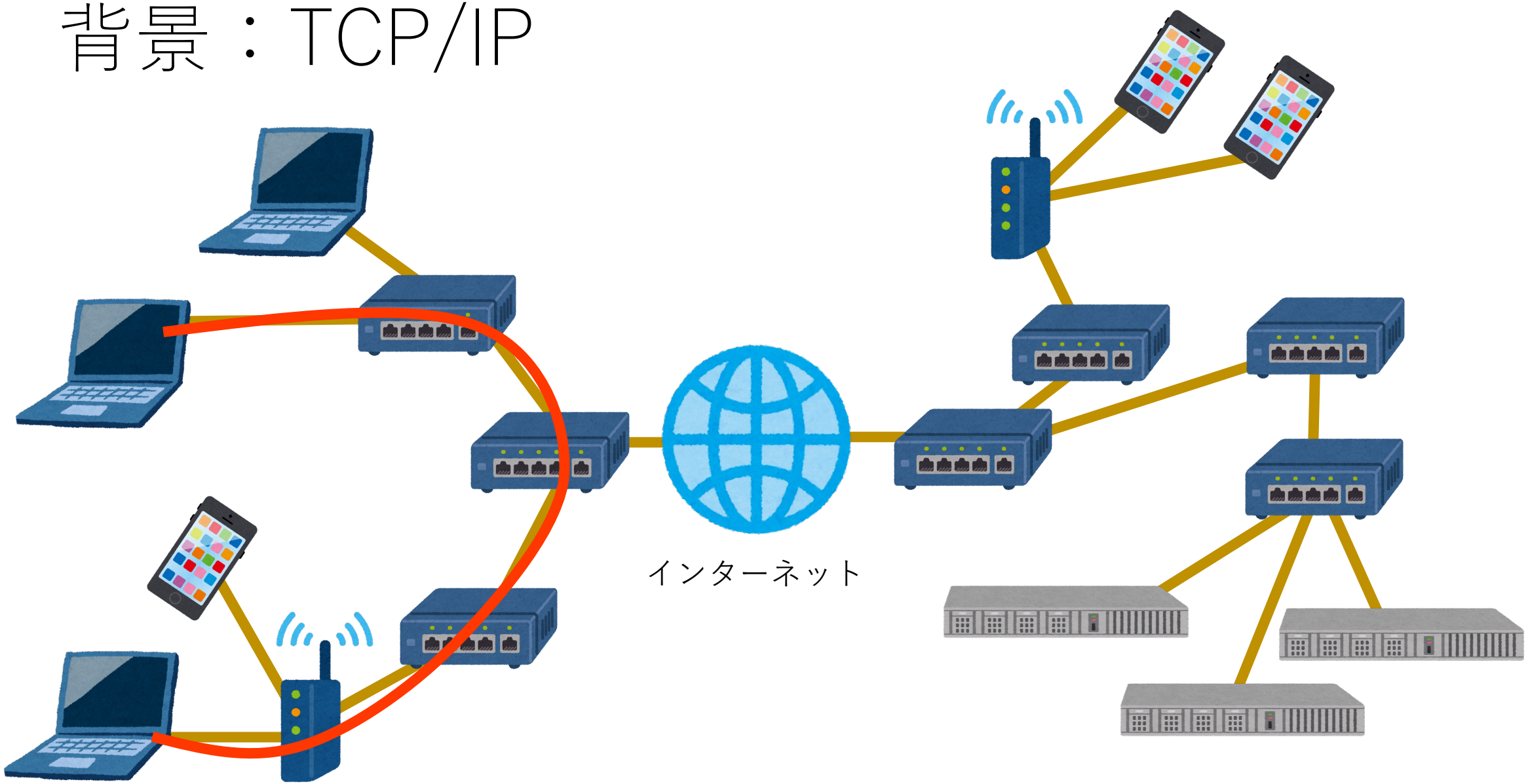
背景：TCP/IP



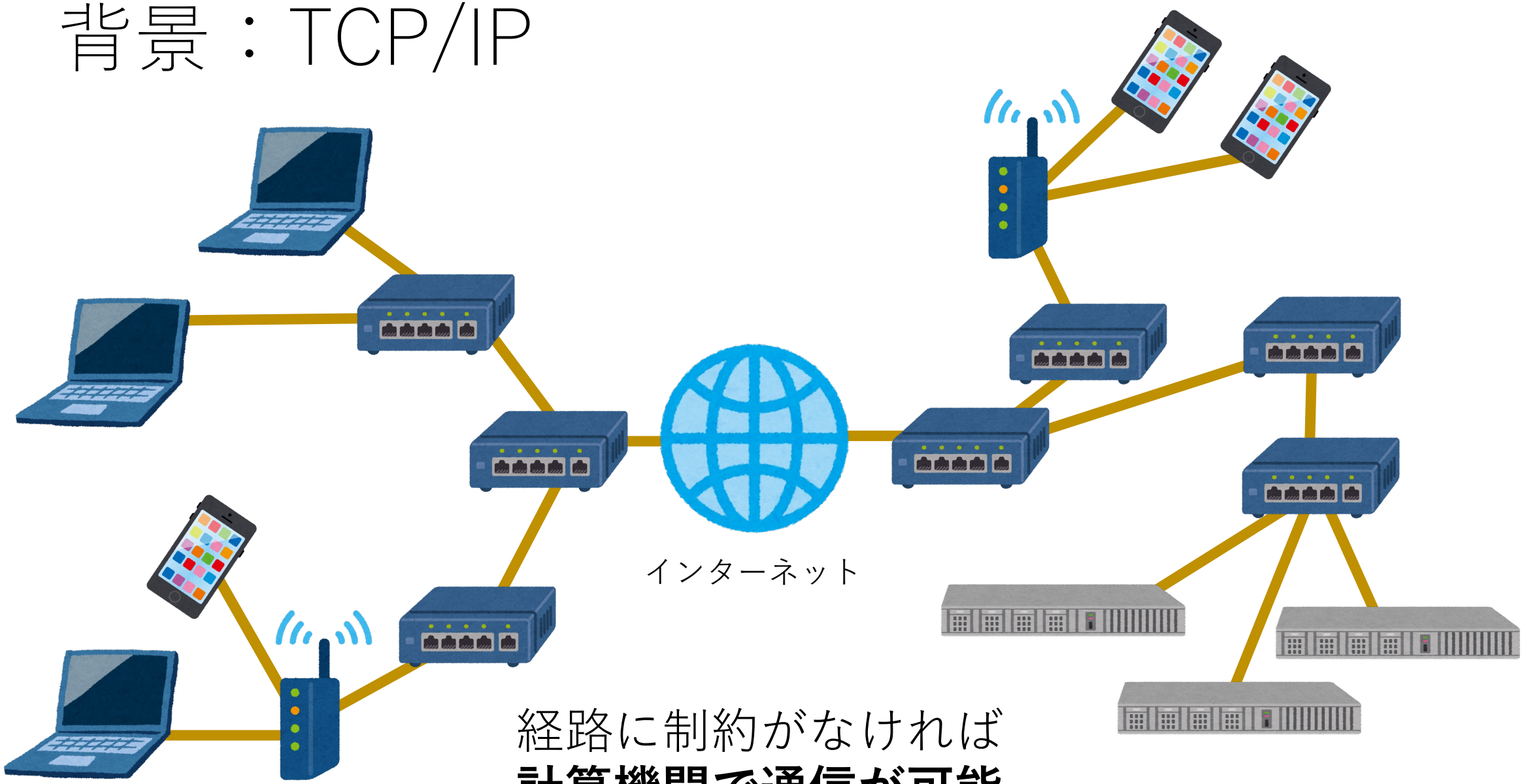
背景：TCP/IP



背景：TCP/IP

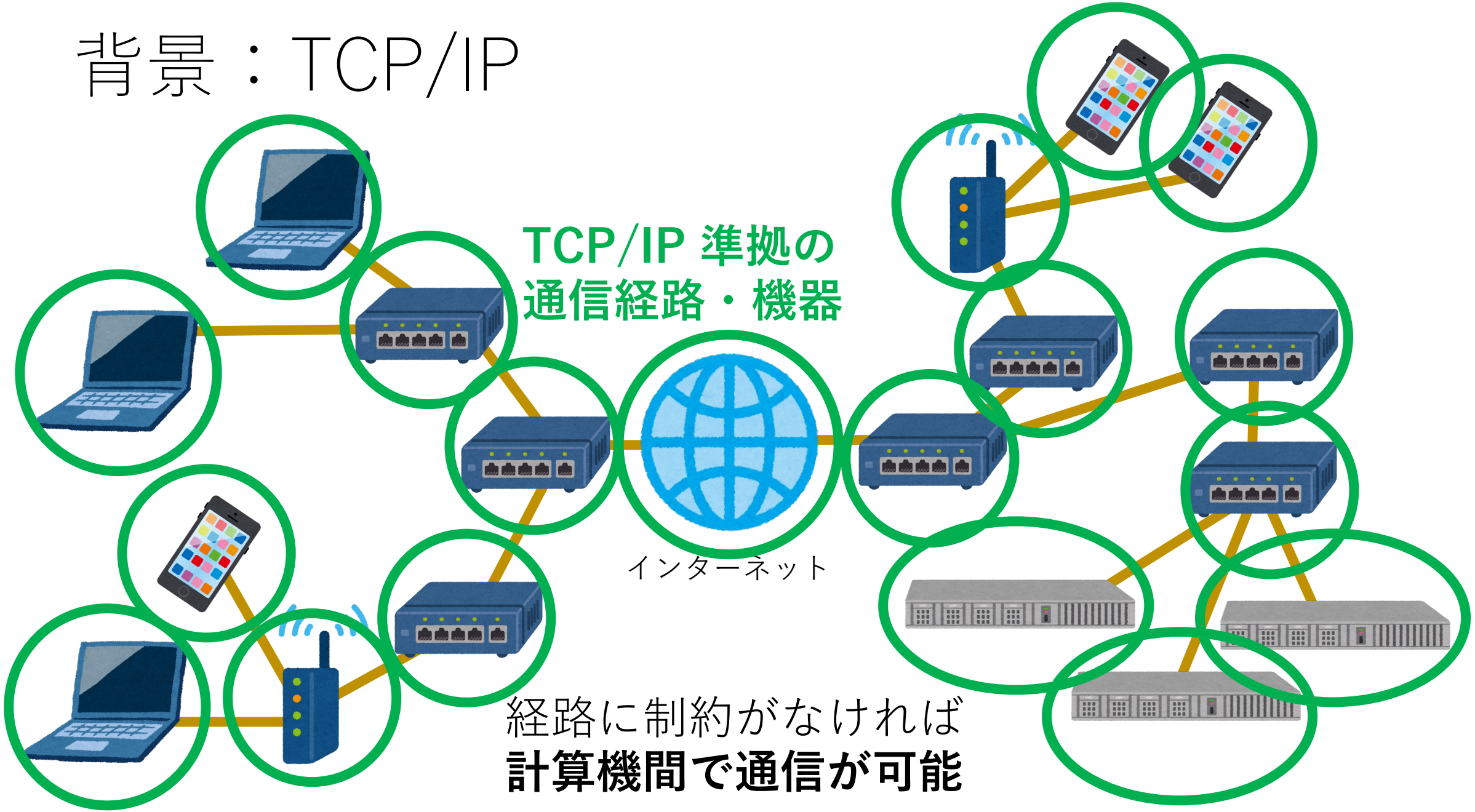


背景：TCP/IP



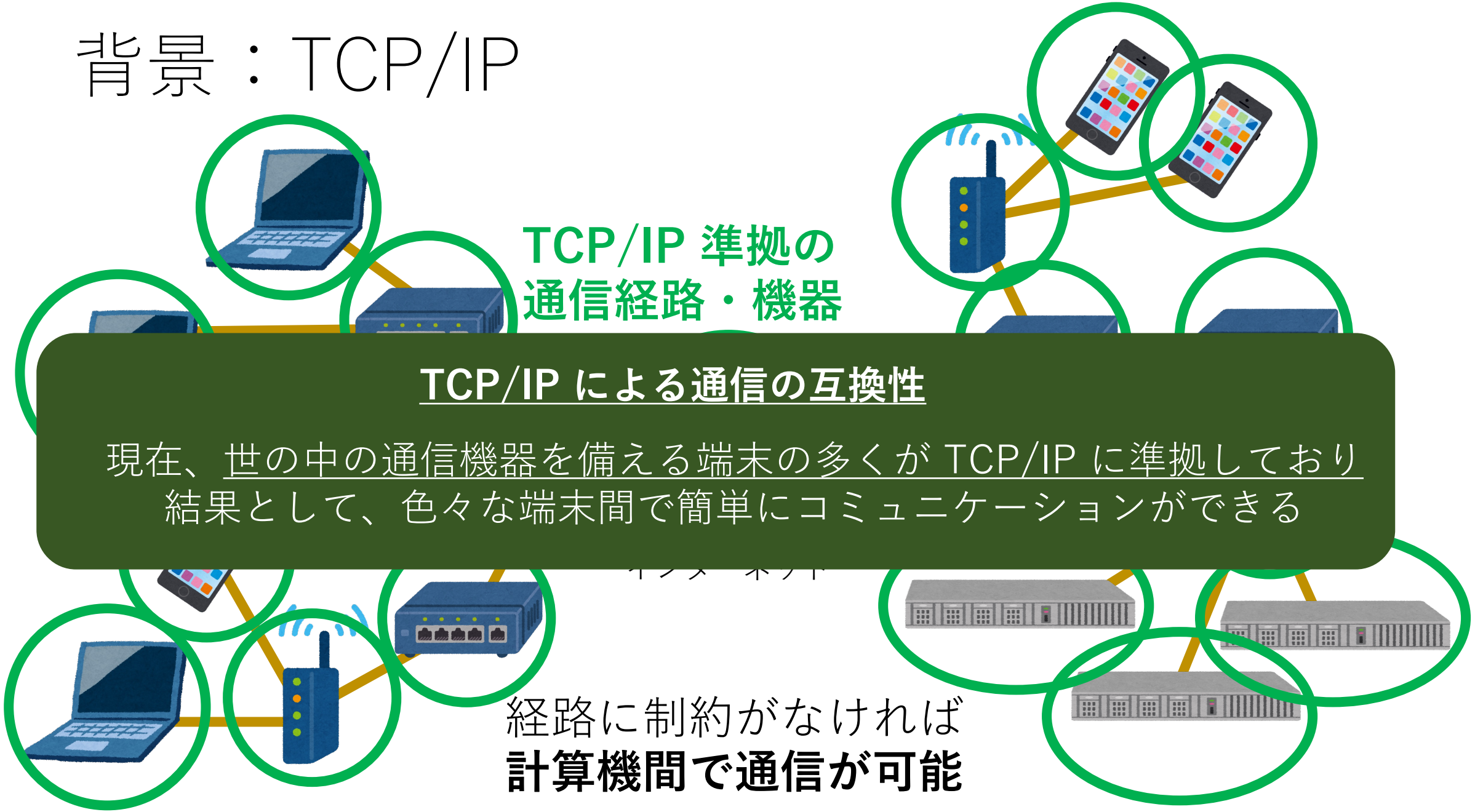
経路に制約がなければ
計算機間で通信が可能

背景：TCP/IP



経路に制約がなければ
計算機間で通信が可能

背景：TCP/IP



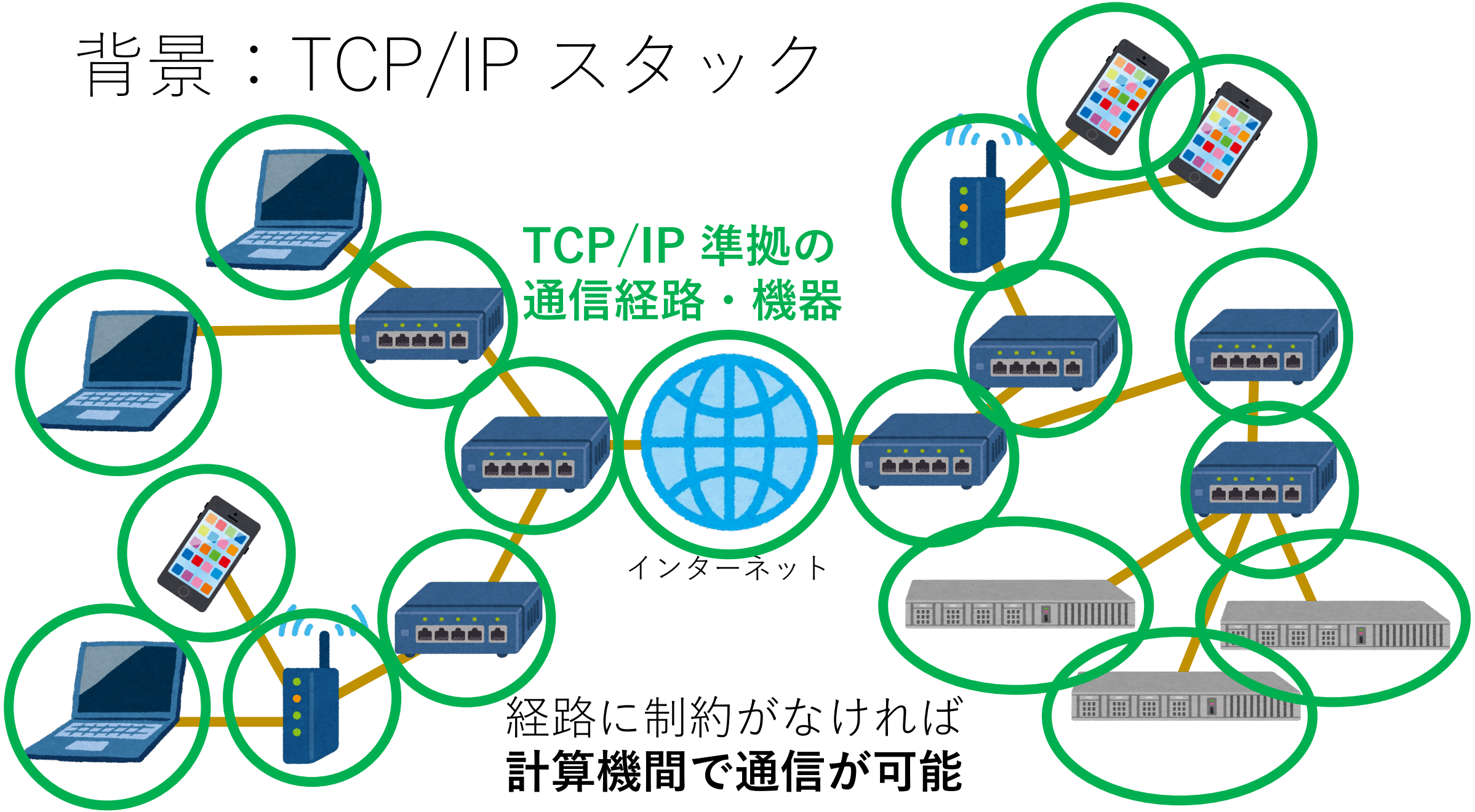
TCP/IP 準拠の
通信経路・機器

TCP/IP による通信の互換性

現在、世の中の通信機器を備える端末の多くが TCP/IP に準拠しており
結果として、色々な端末間で簡単にコミュニケーションができる

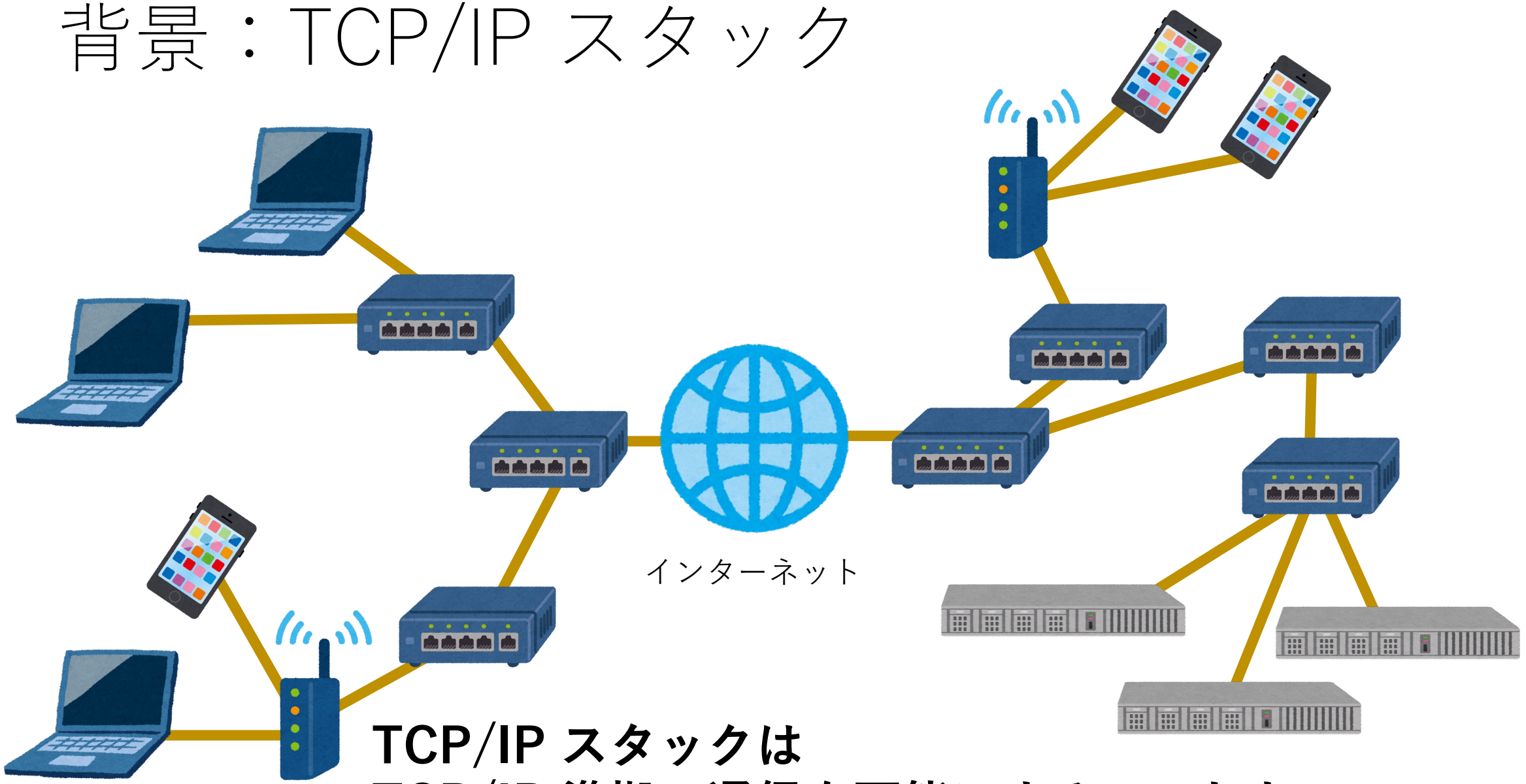
経路に制約がなければ
計算機間で通信が可能

背景：TCP/IP スタック



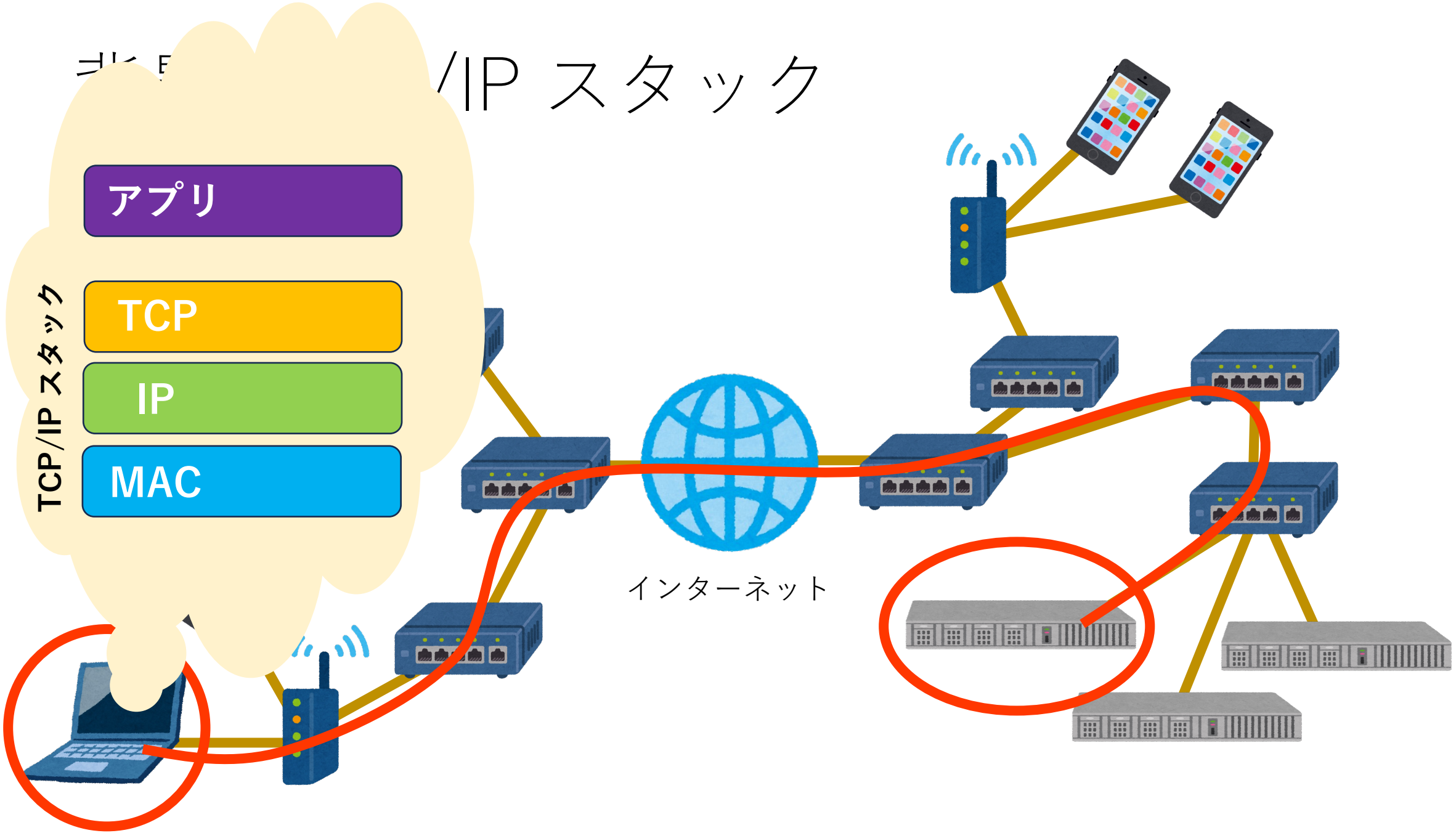
経路に制約がなければ
計算機間で通信が可能

背景：TCP/IP スタック

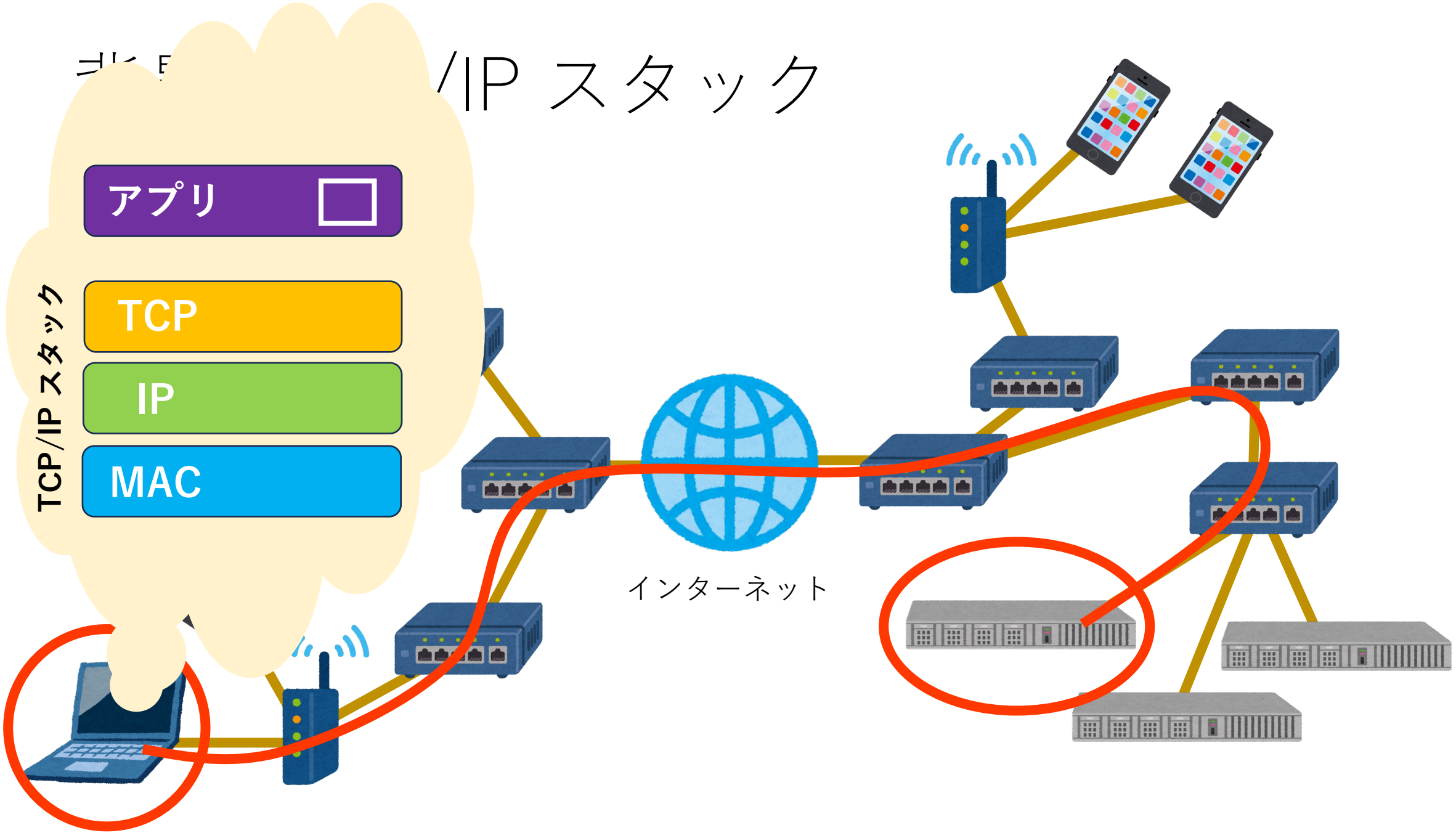


TCP/IP スタックは
TCP/IP 準拠の通信を可能にするソフトウェア

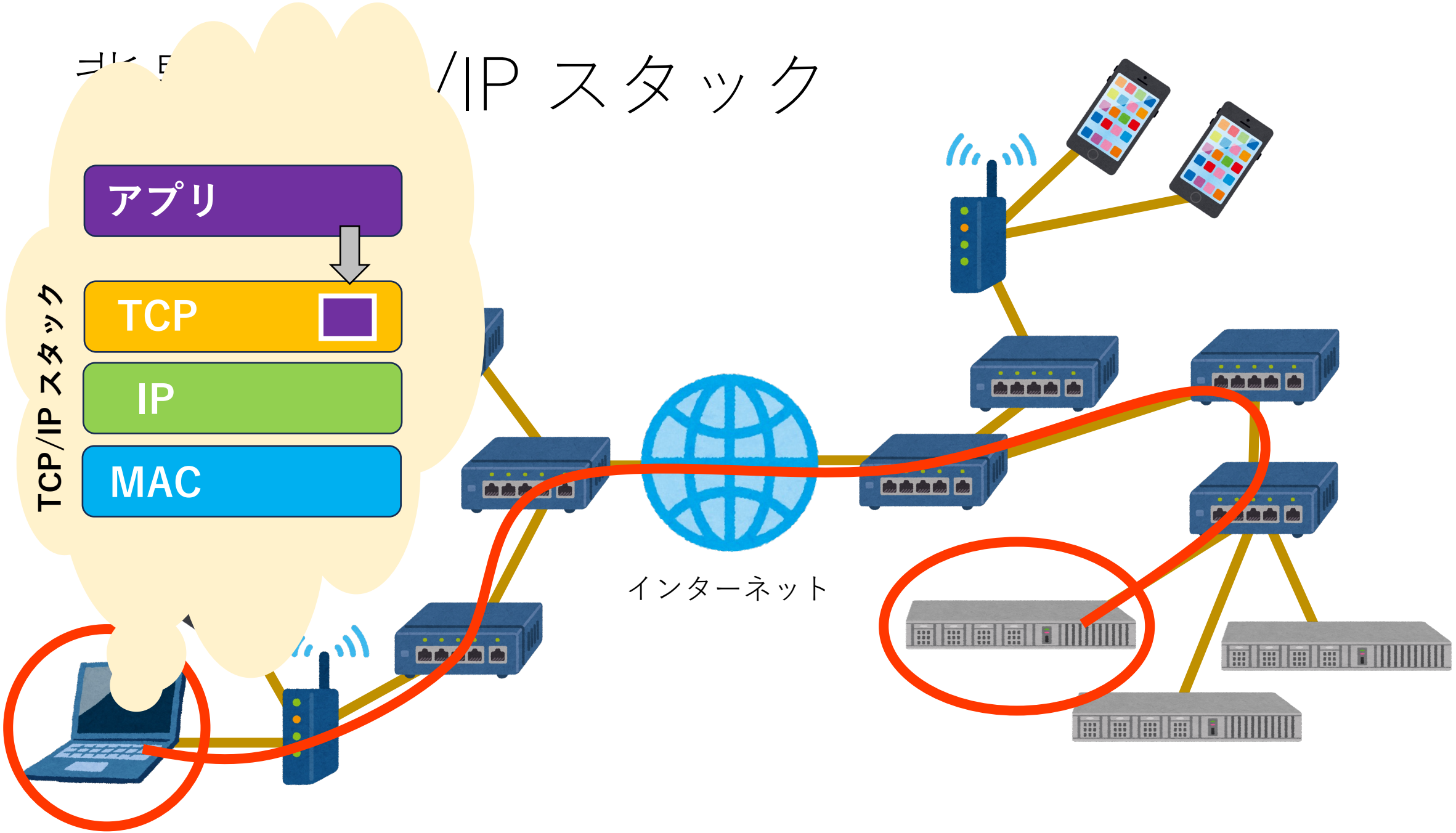
/IP スタック



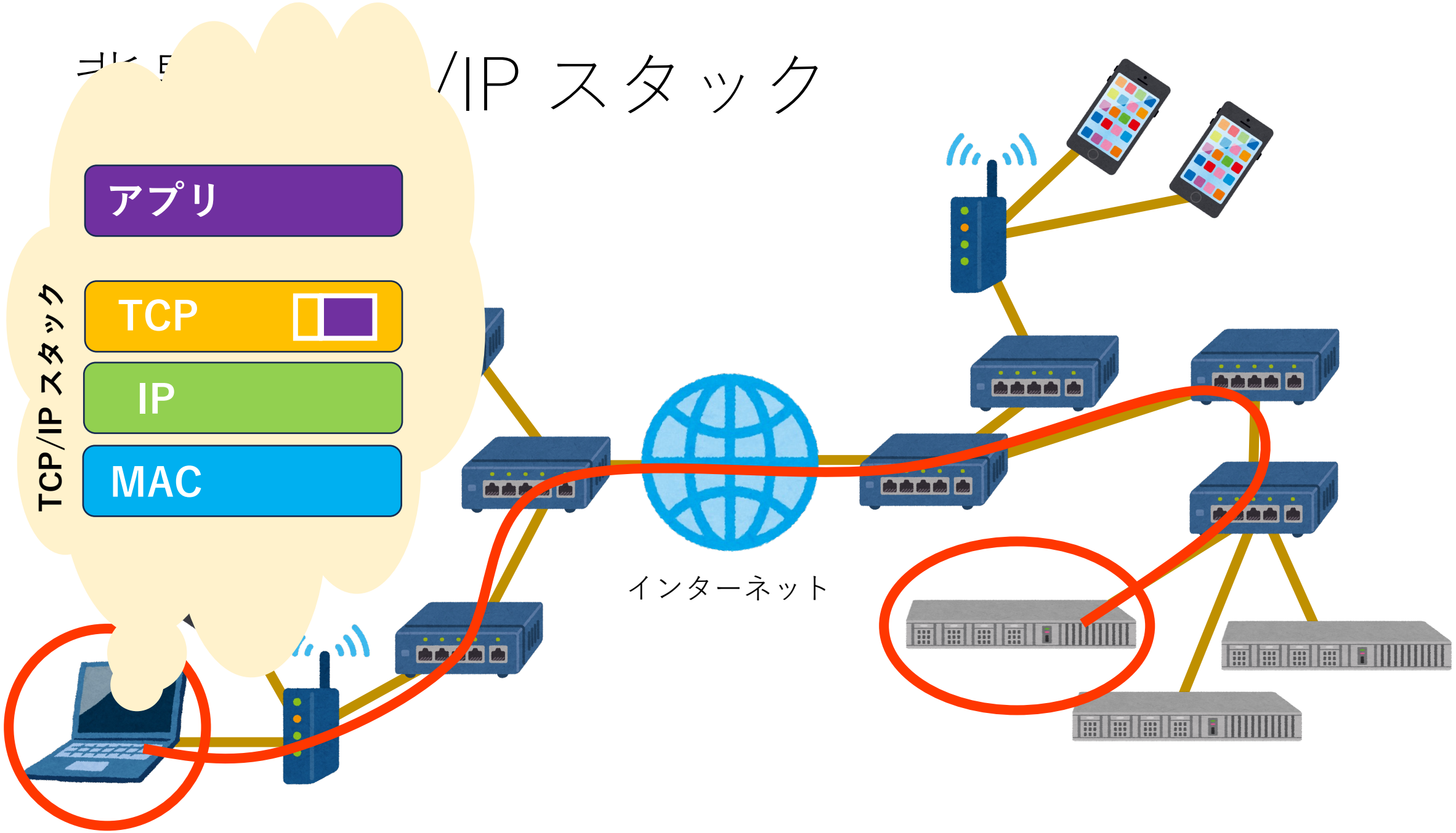
/IP スタック



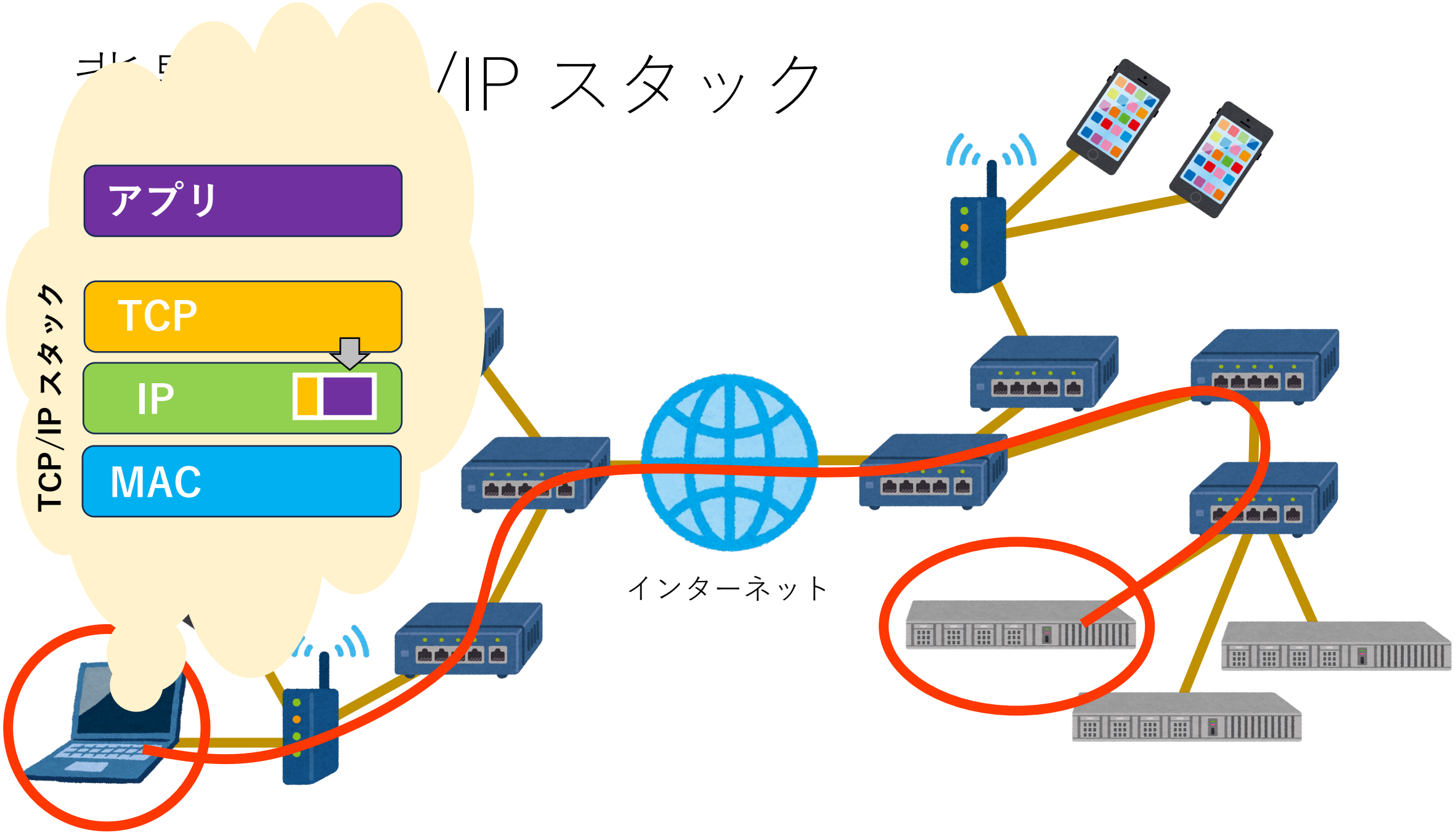
/IP スタック



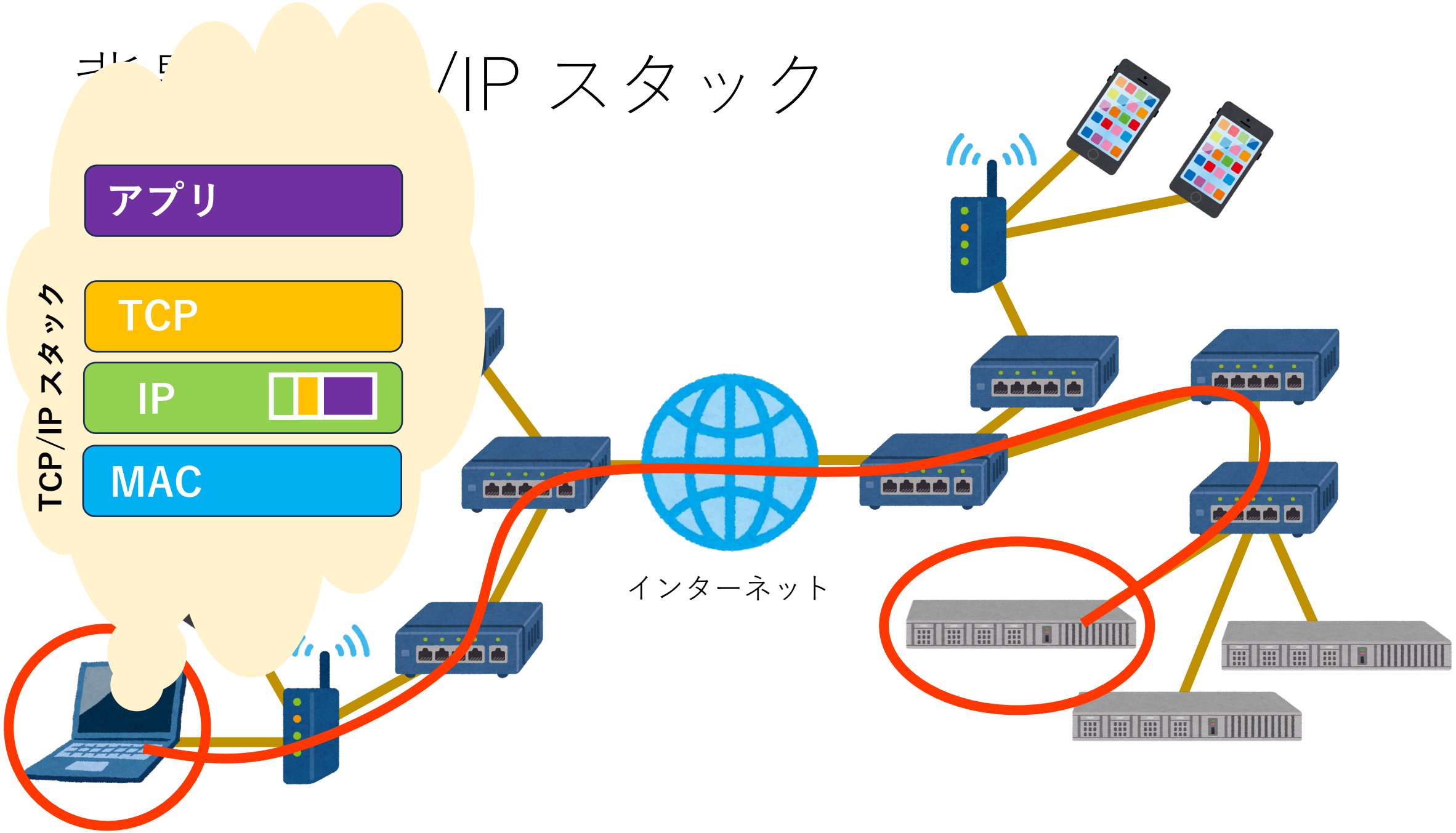
/IP スタック



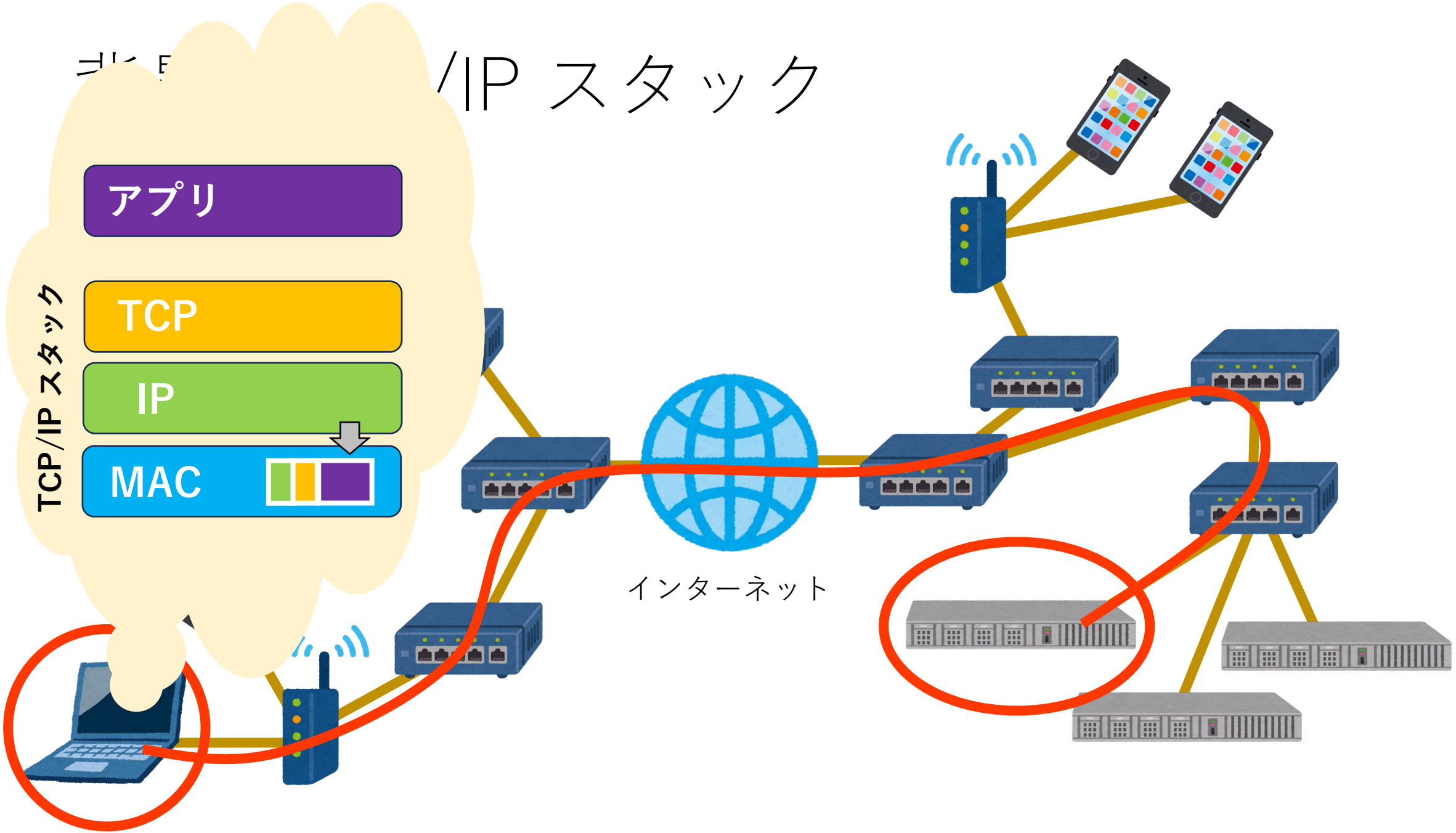
/IP スタック



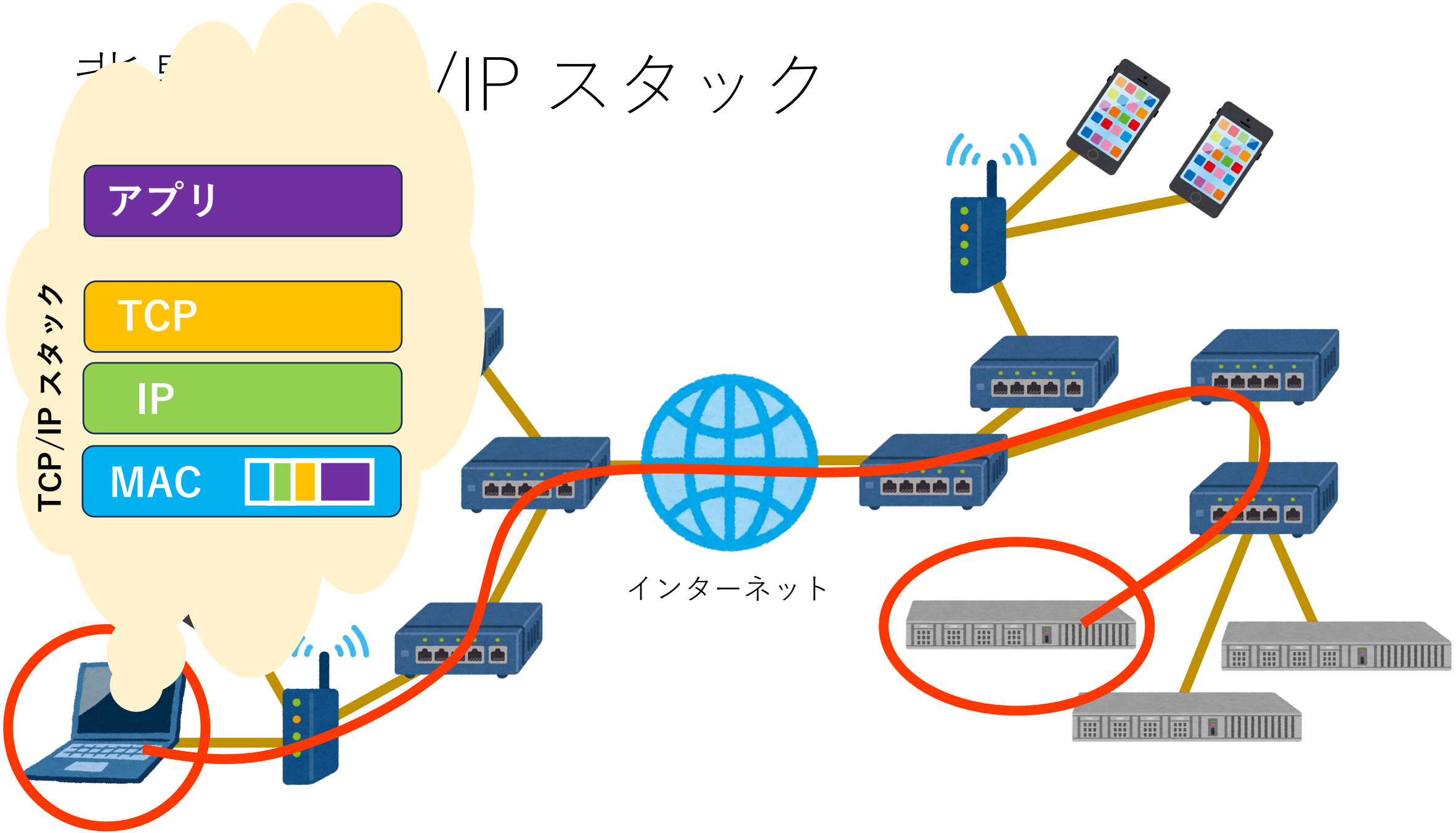
/IP スタック



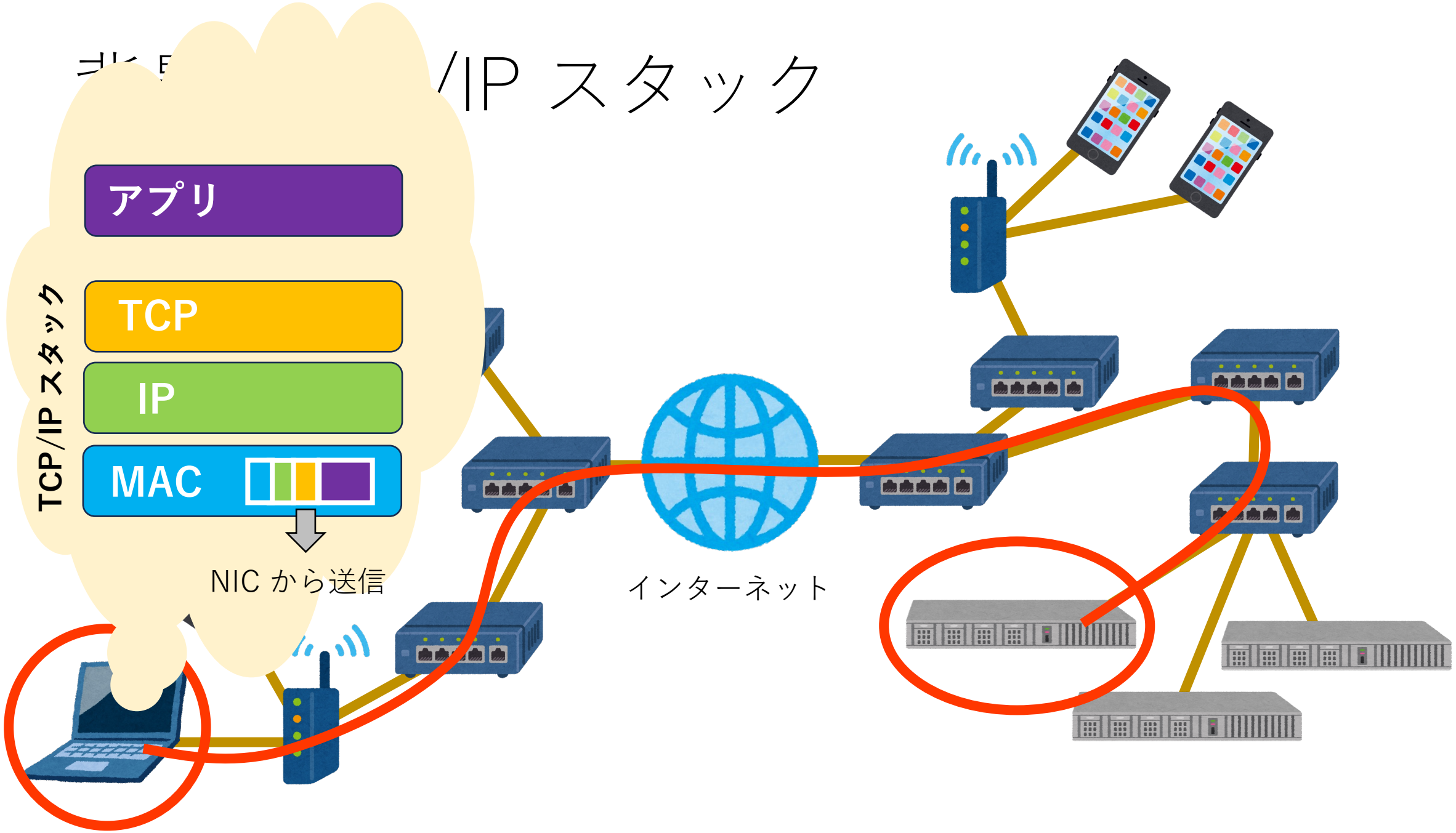
/IP スタック



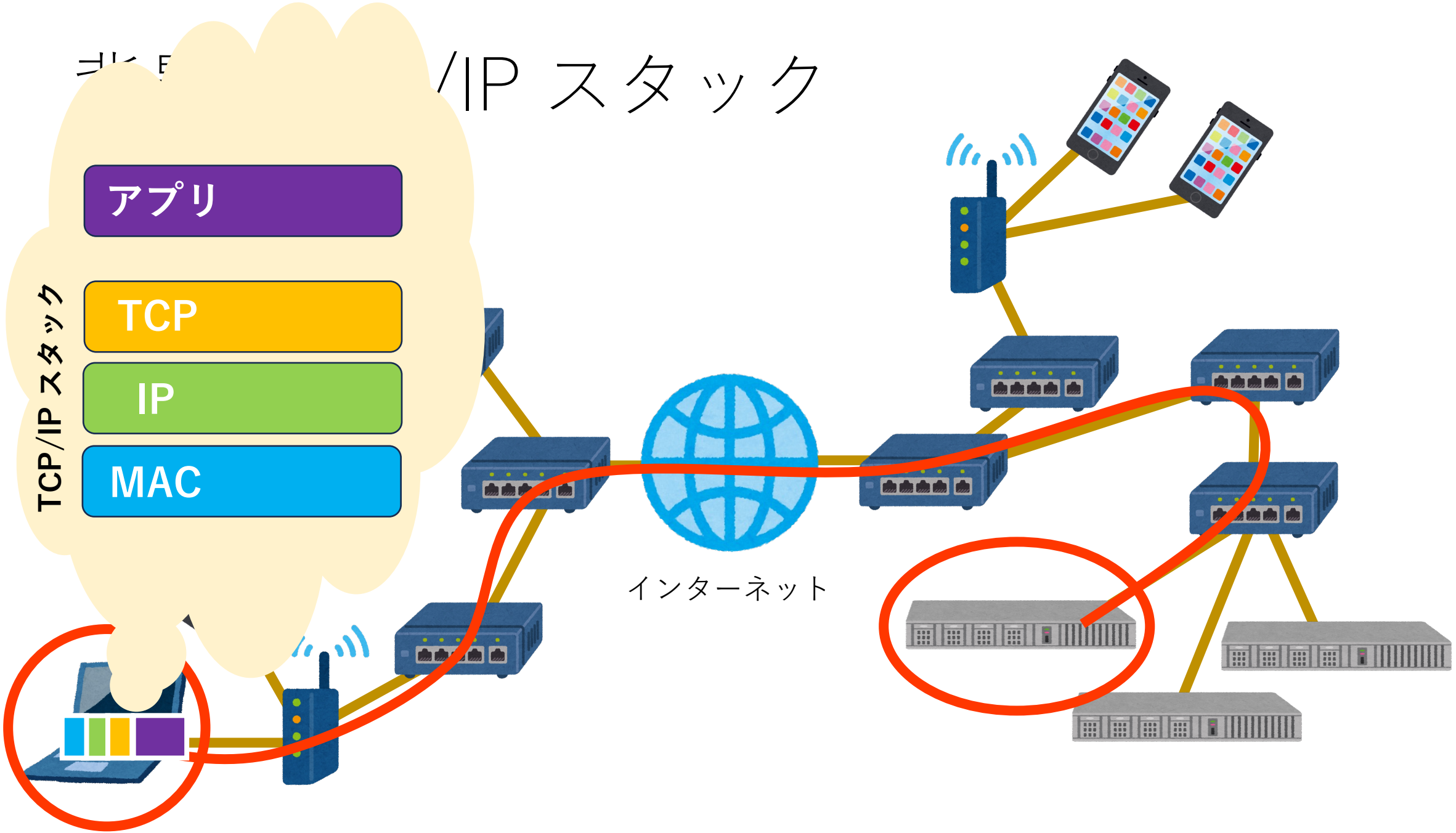
/IP スタック



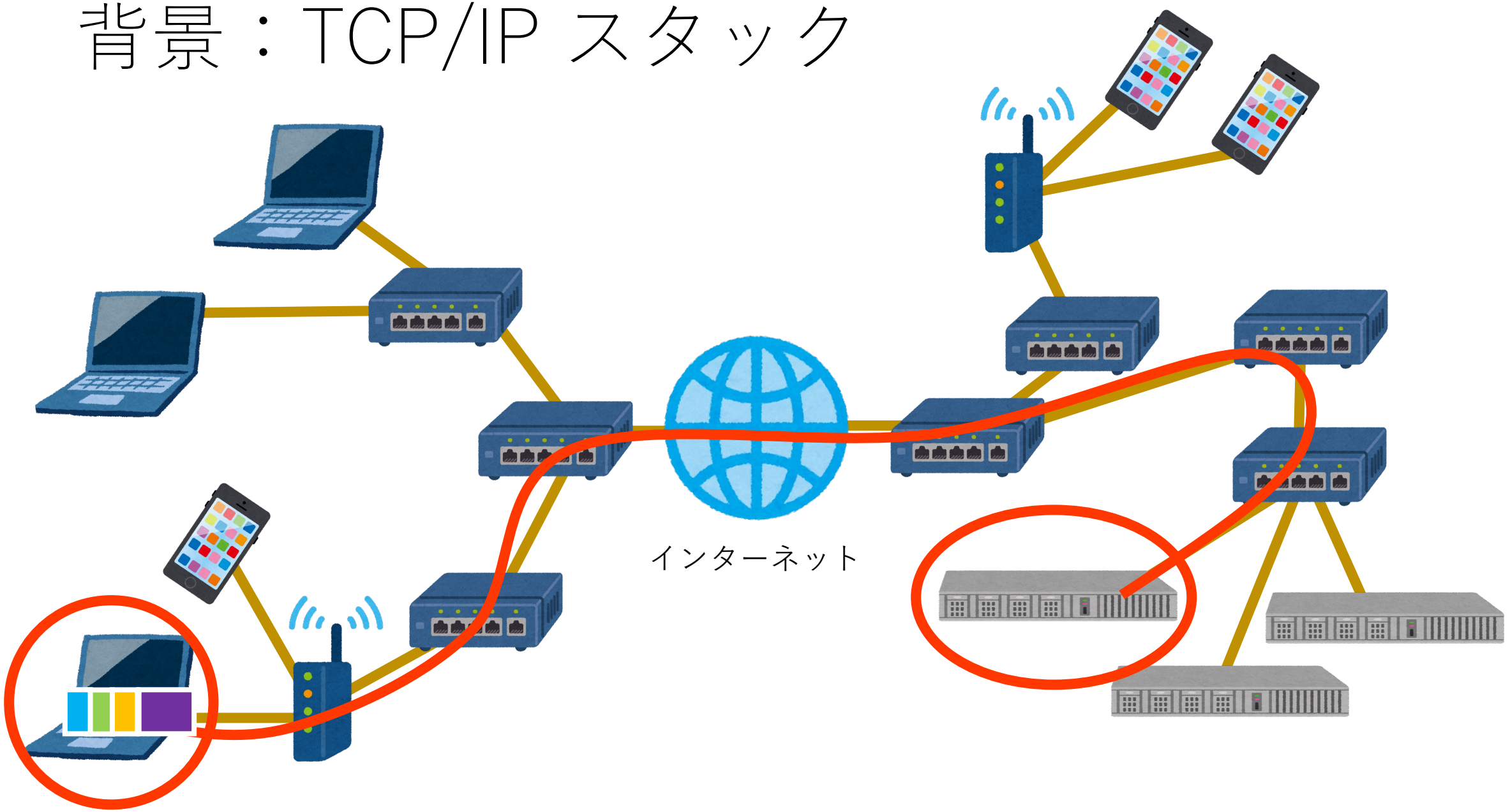
/IP スタック



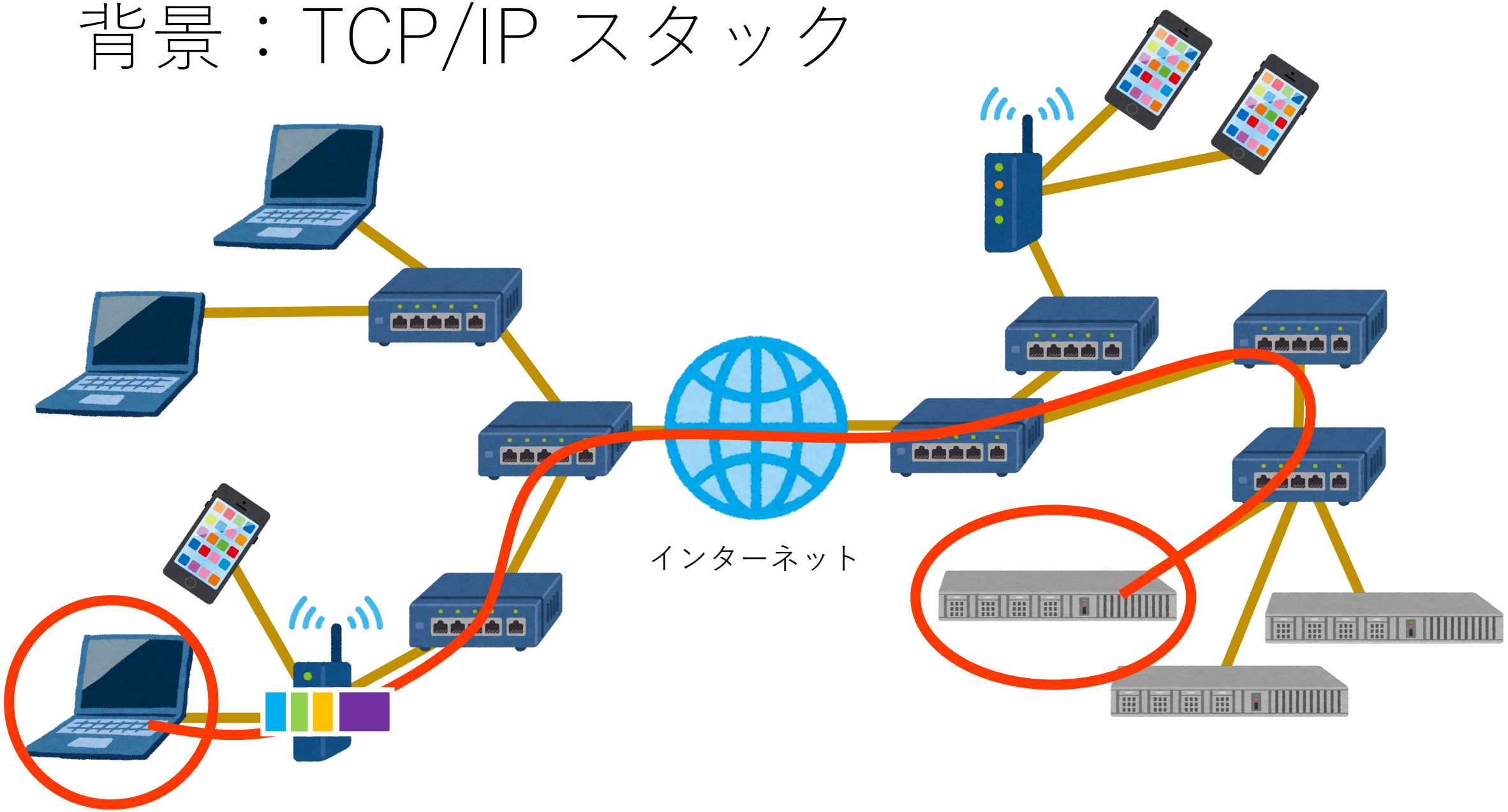
/IP スタック



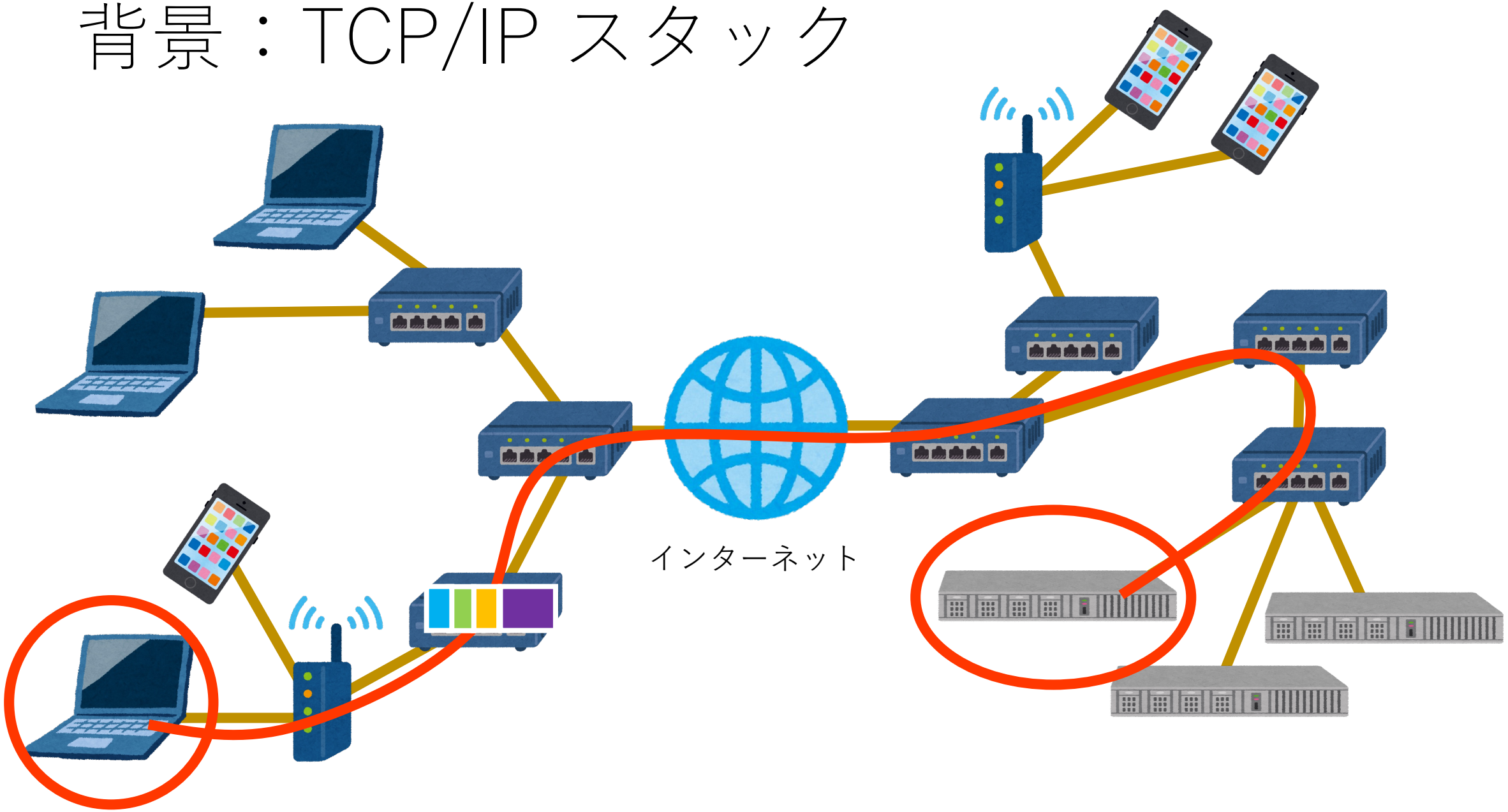
背景：TCP/IP スタック



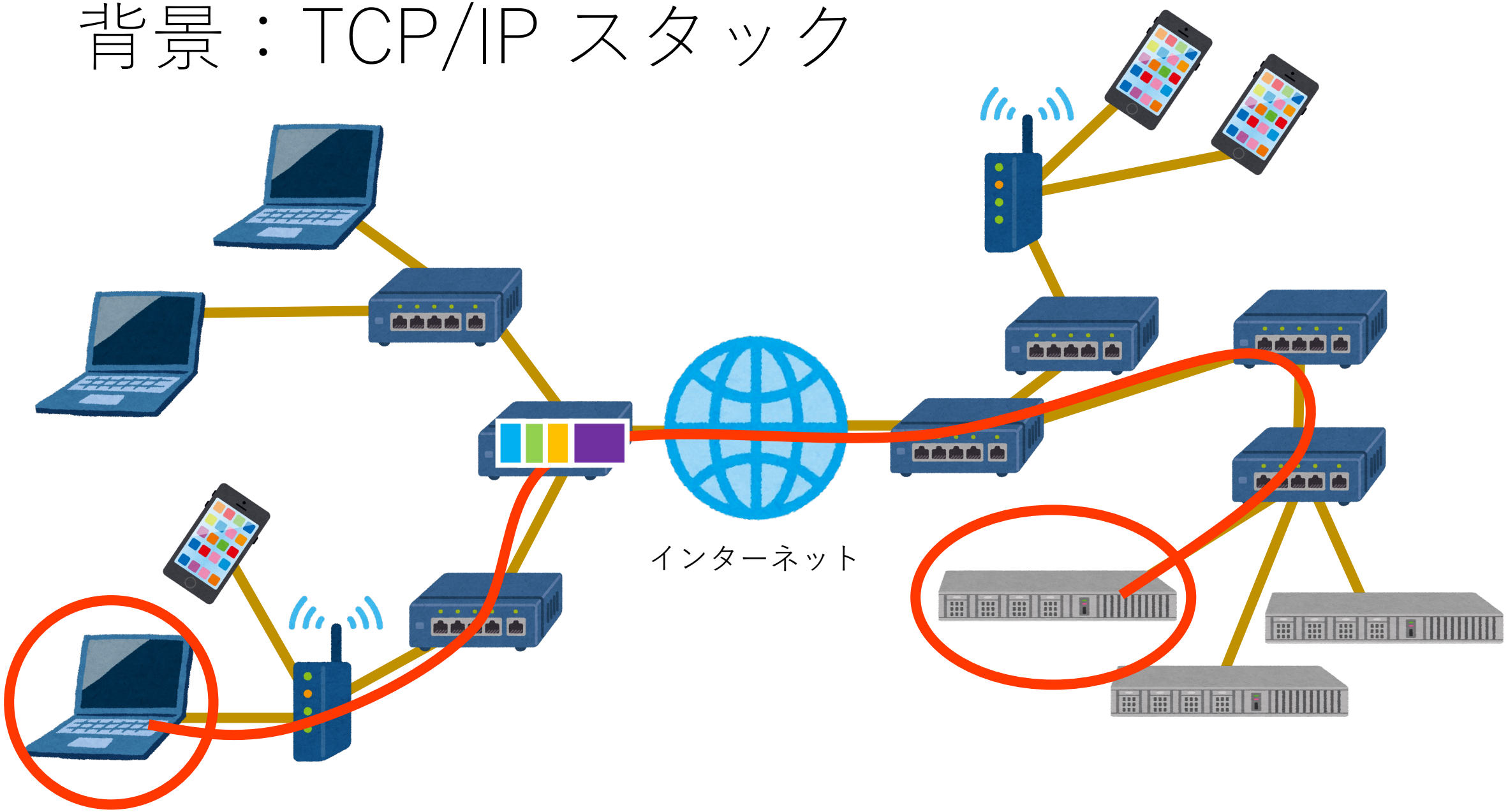
背景：TCP/IP スタック



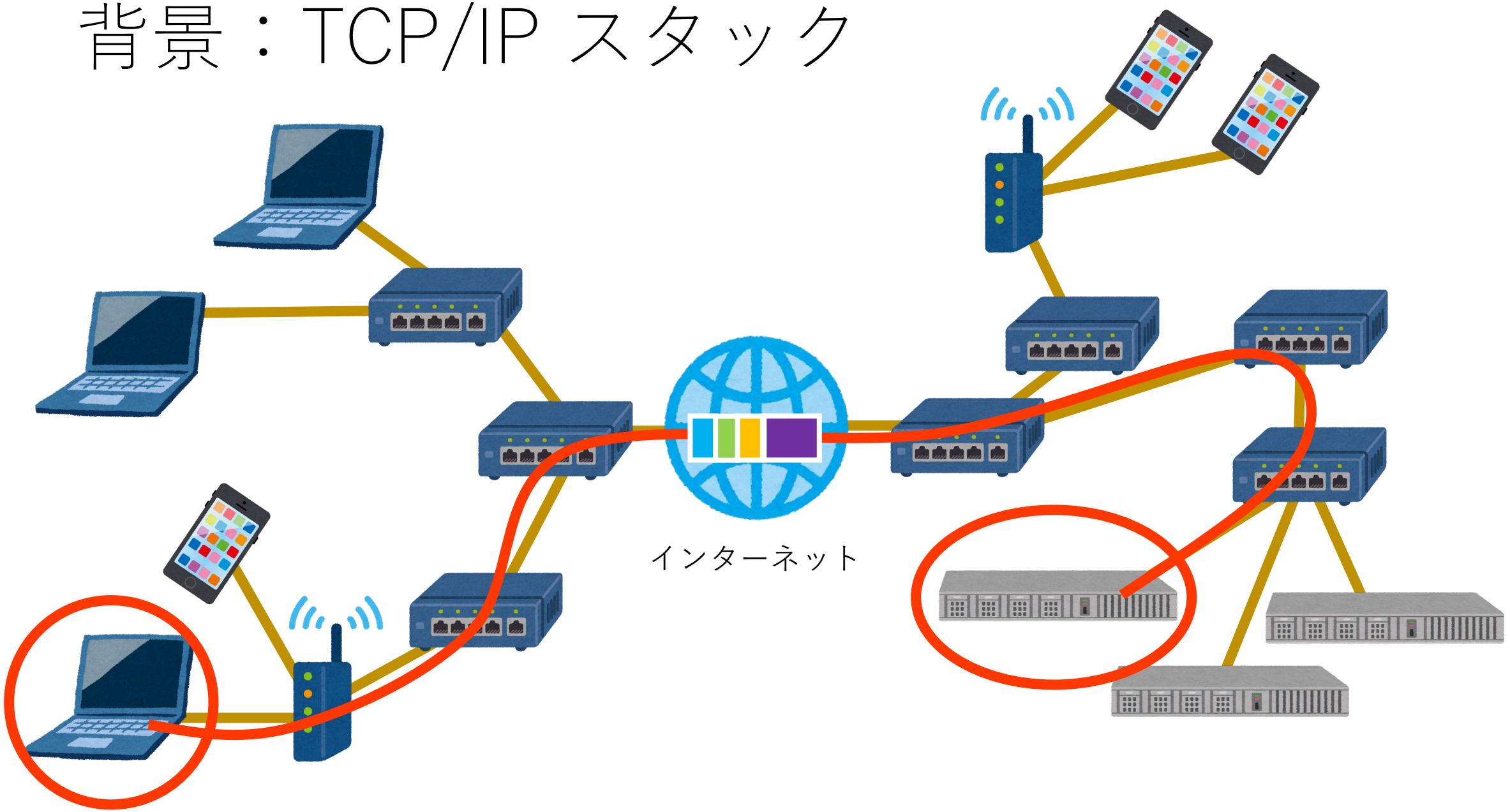
背景：TCP/IP スタック



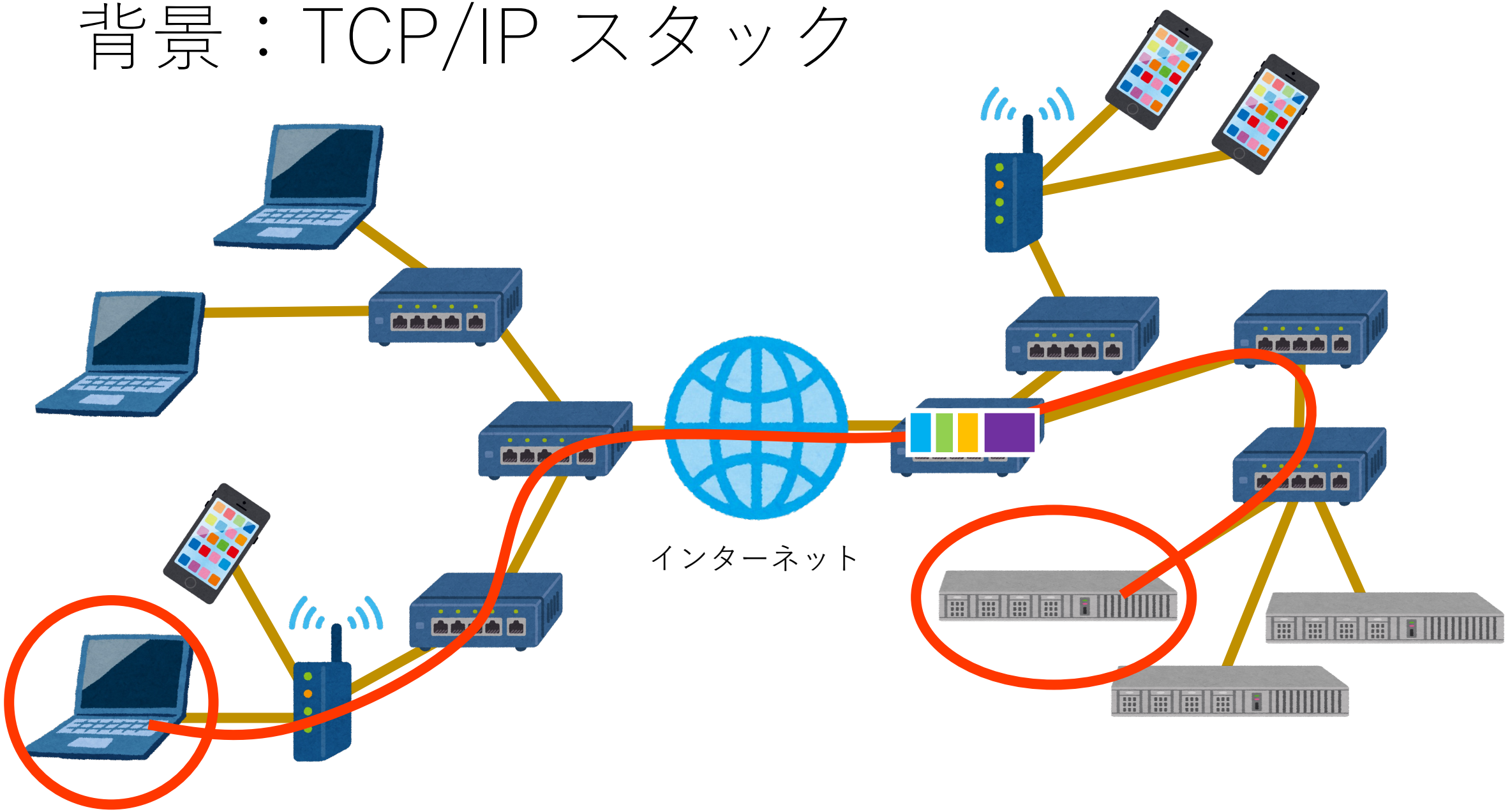
背景：TCP/IP スタック



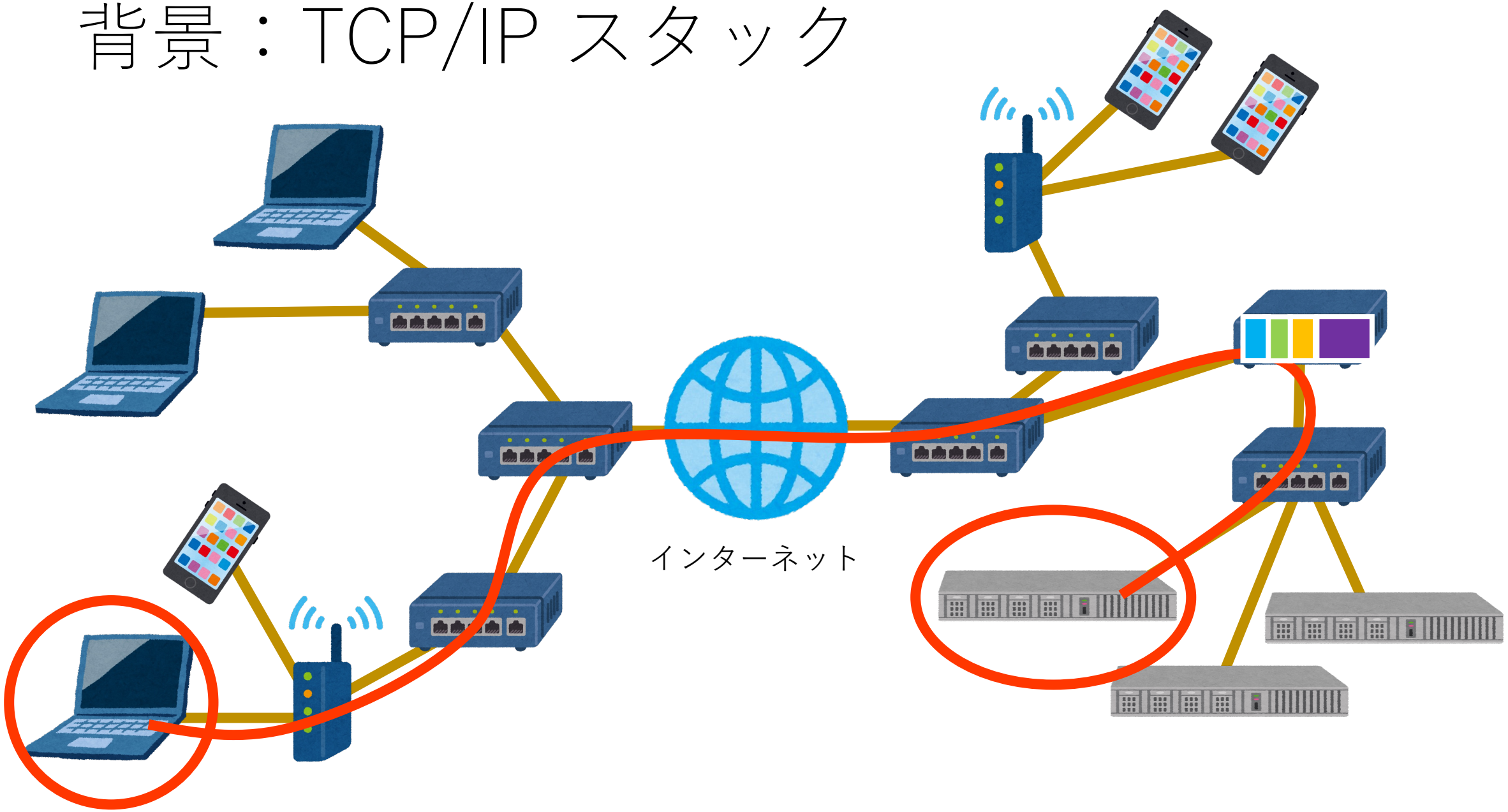
背景：TCP/IP スタック



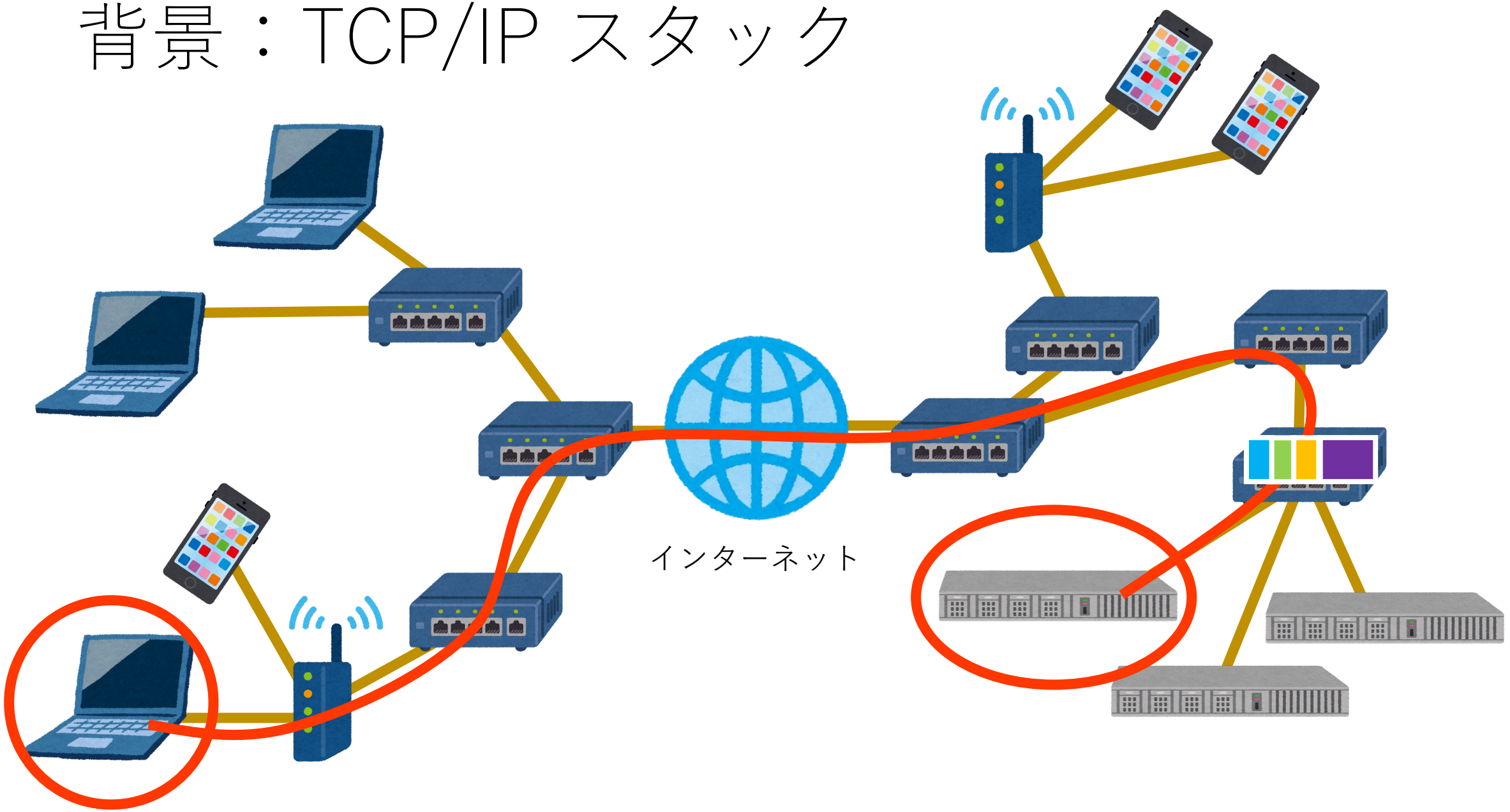
背景：TCP/IP スタック



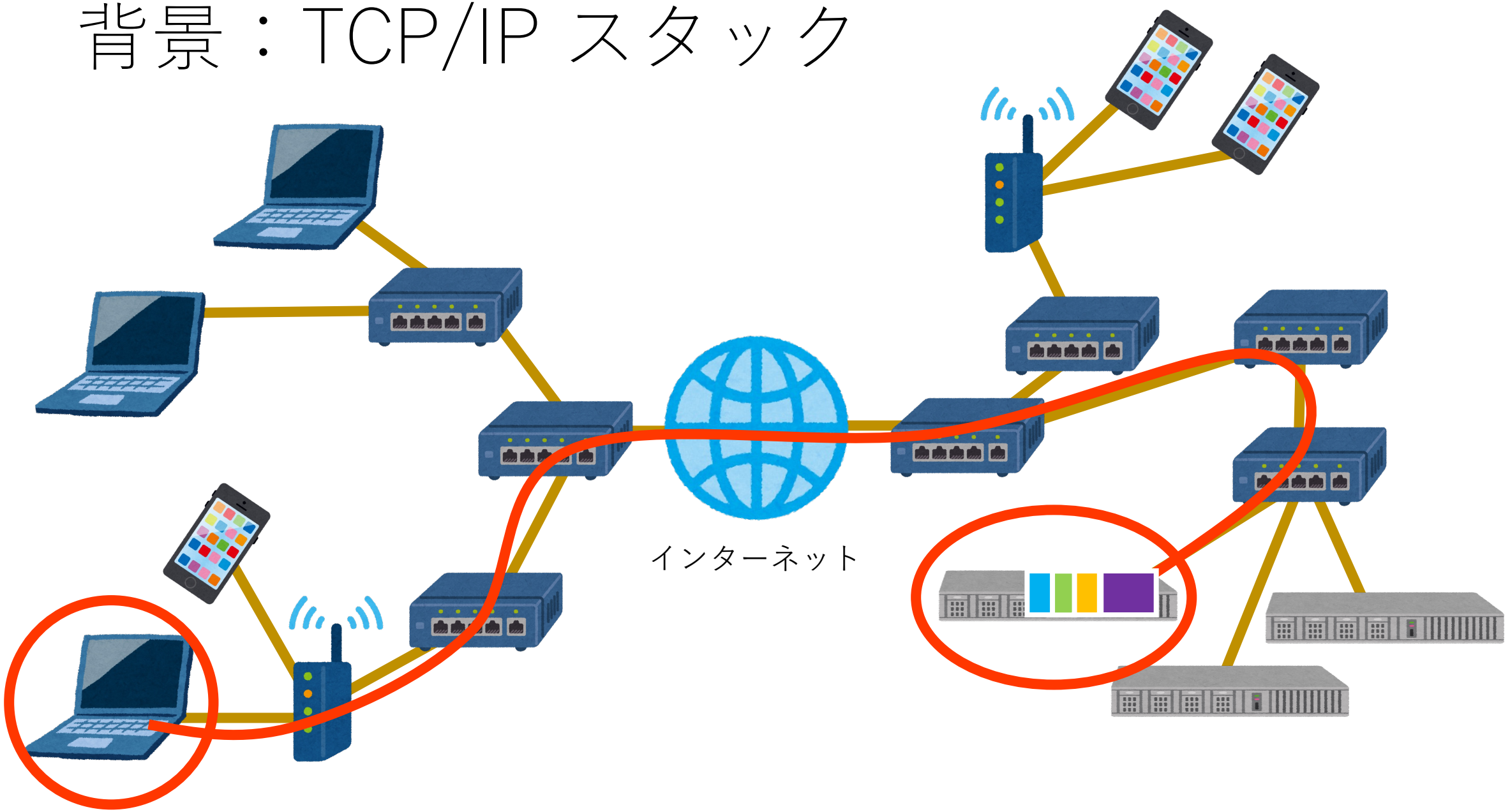
背景：TCP/IP スタック



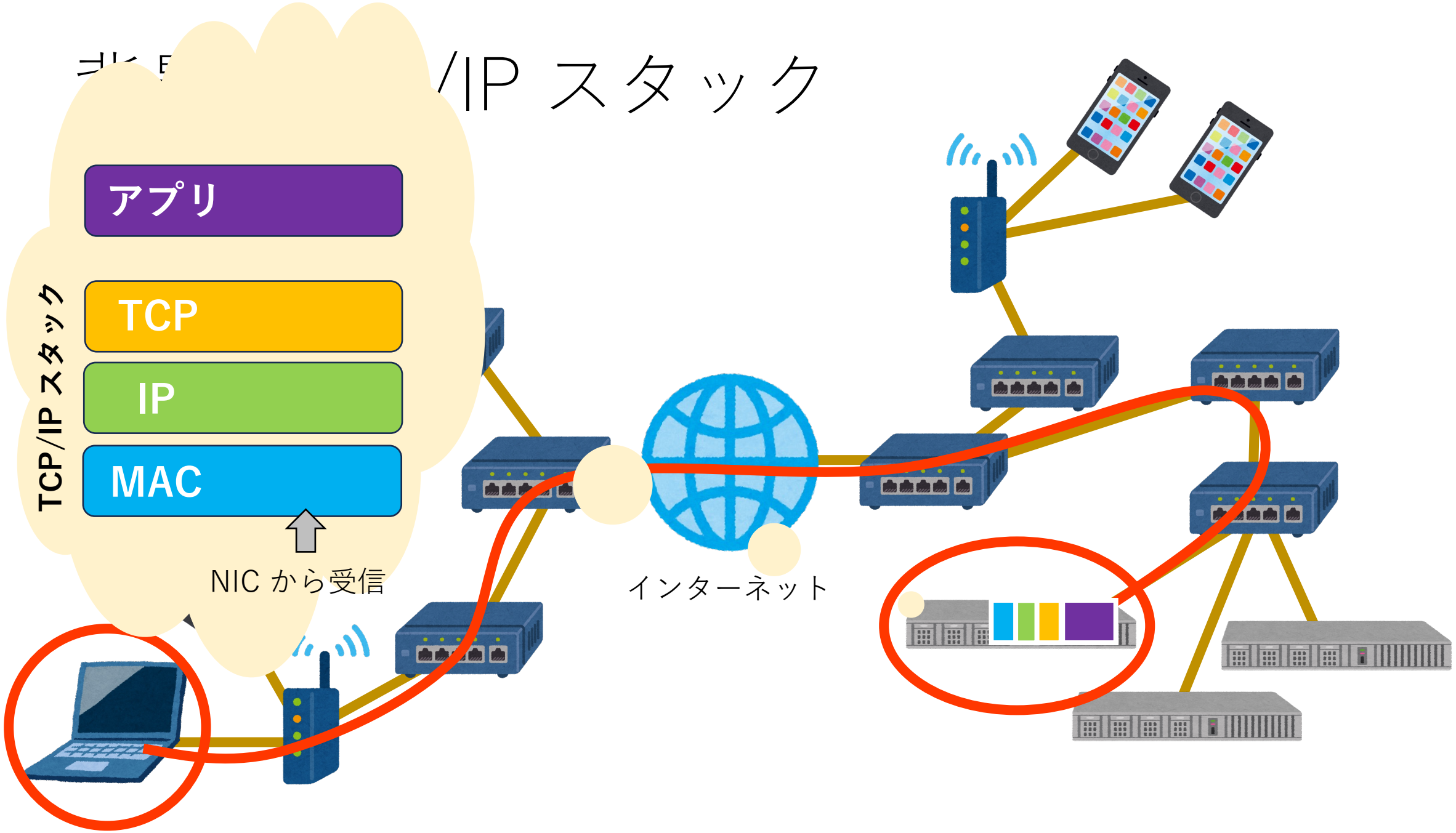
背景：TCP/IP スタック



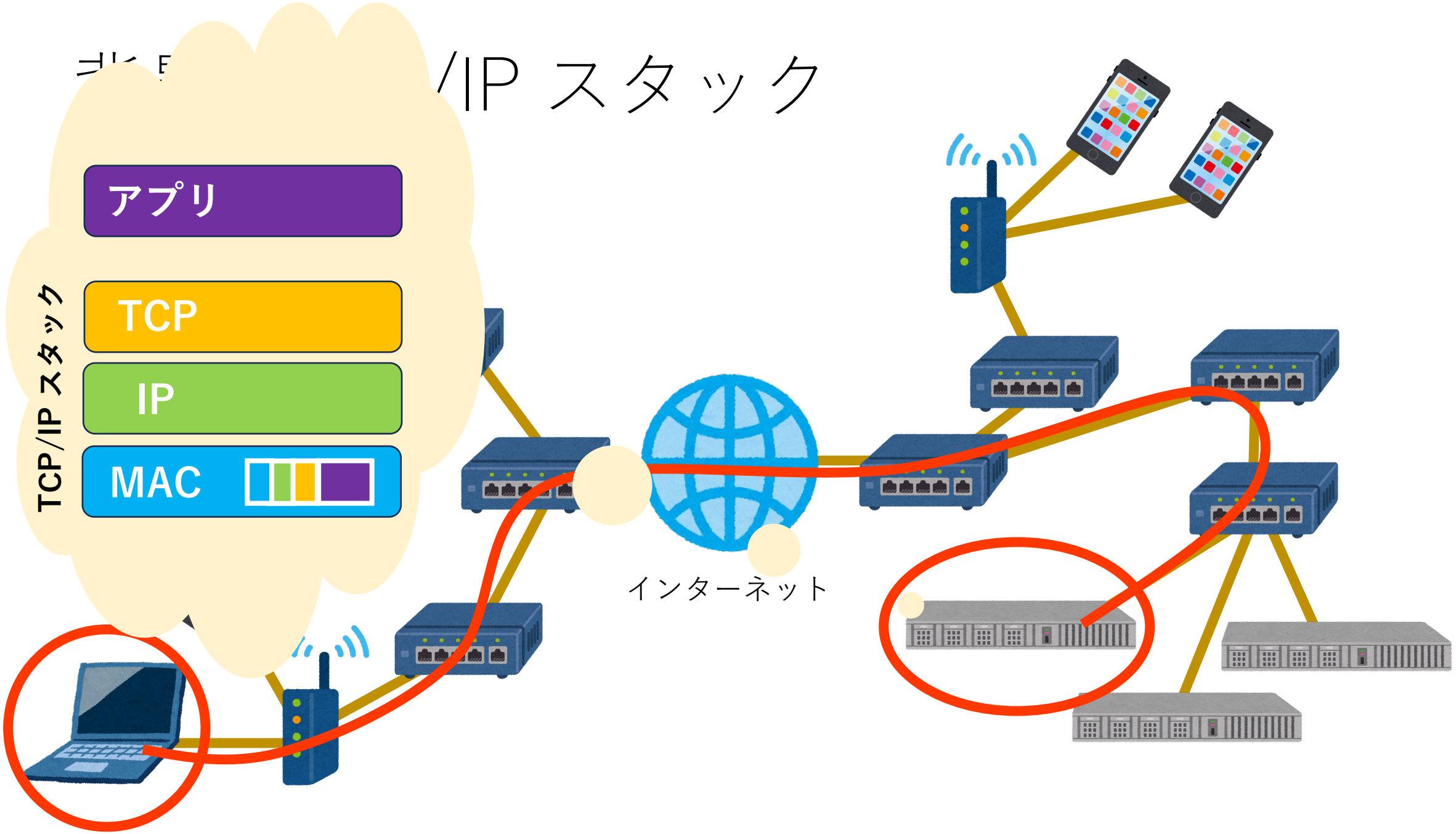
背景：TCP/IP スタック



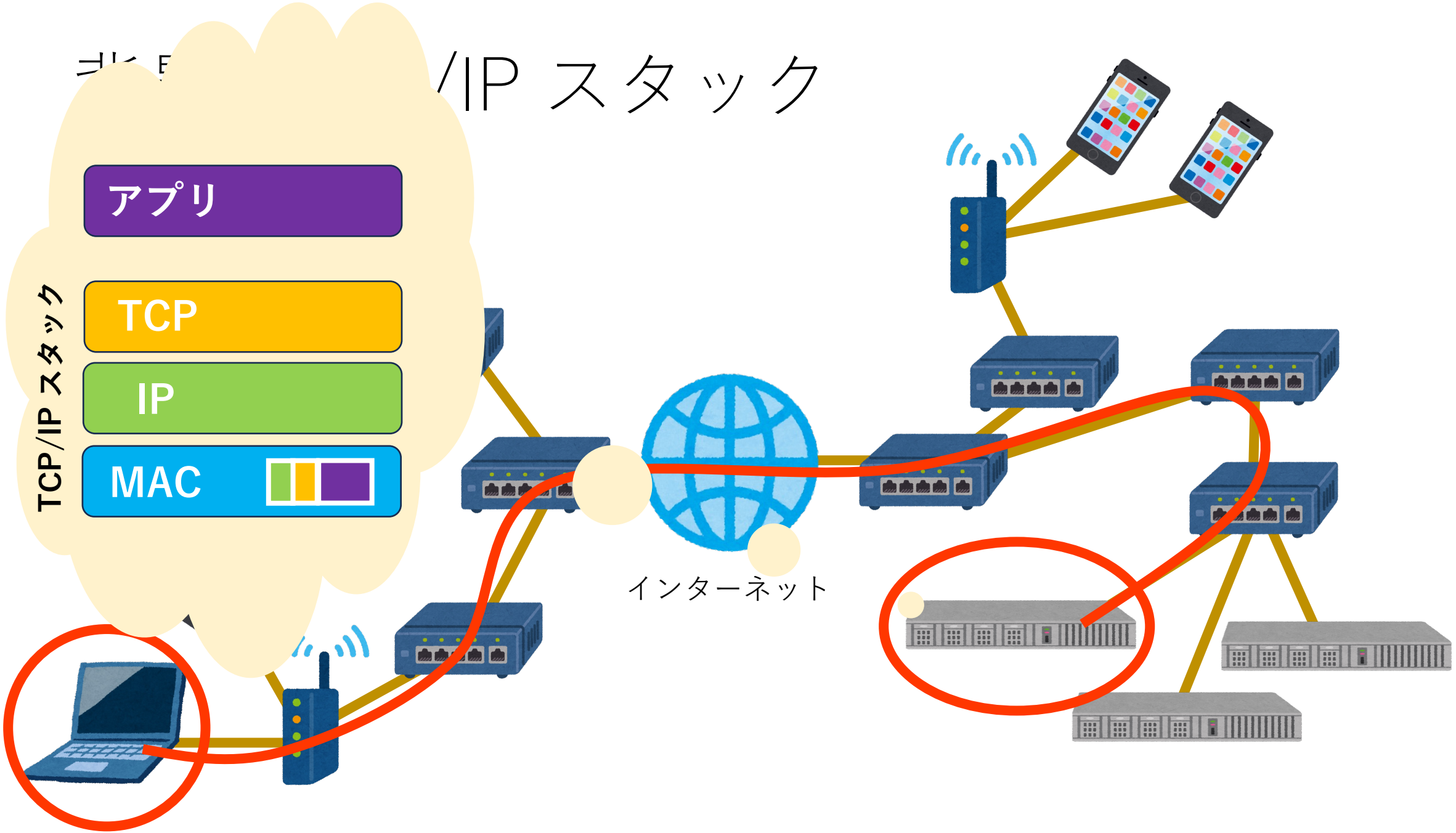
/IP スタック



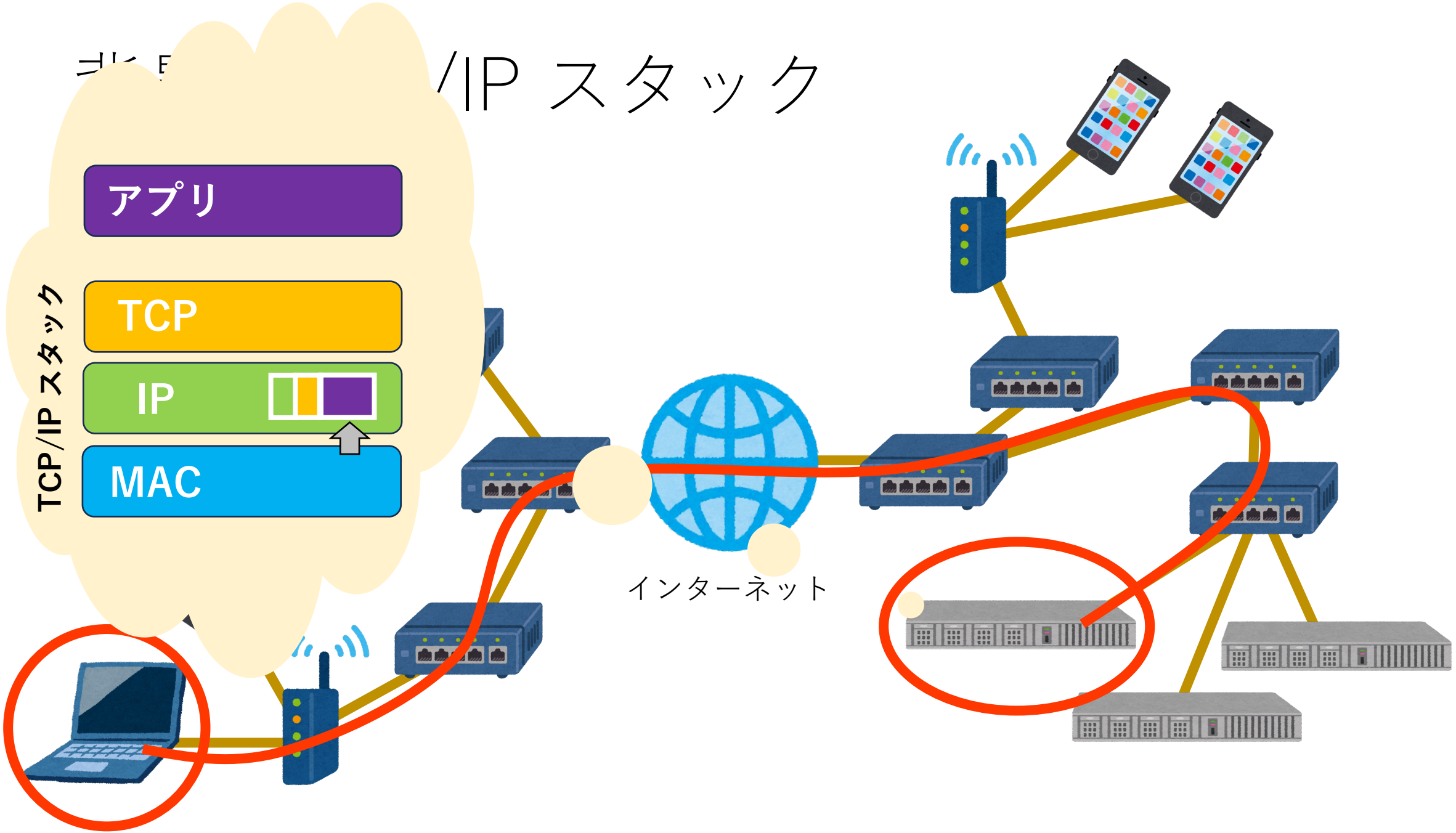
/IP スタック



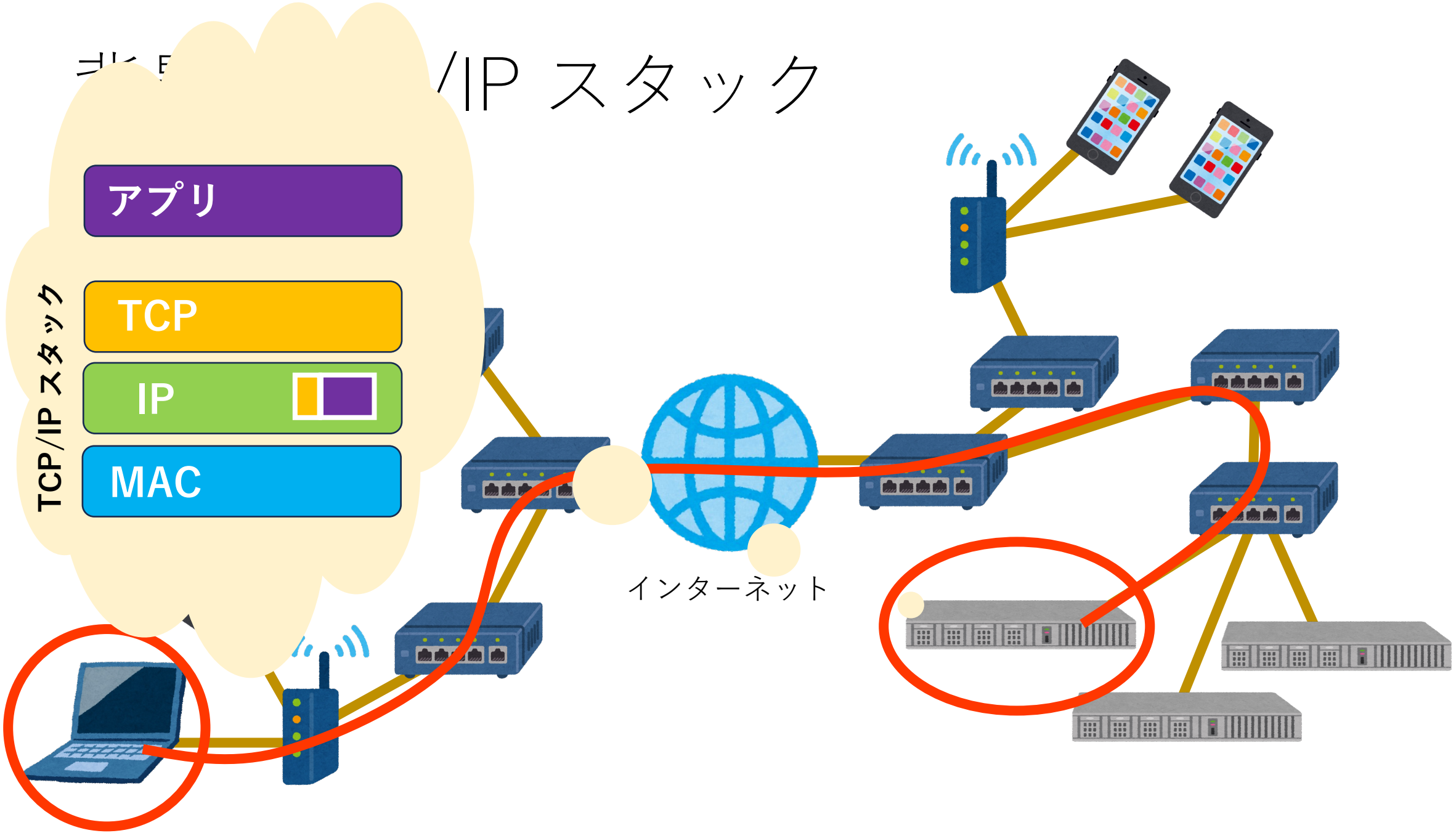
/IP スタック



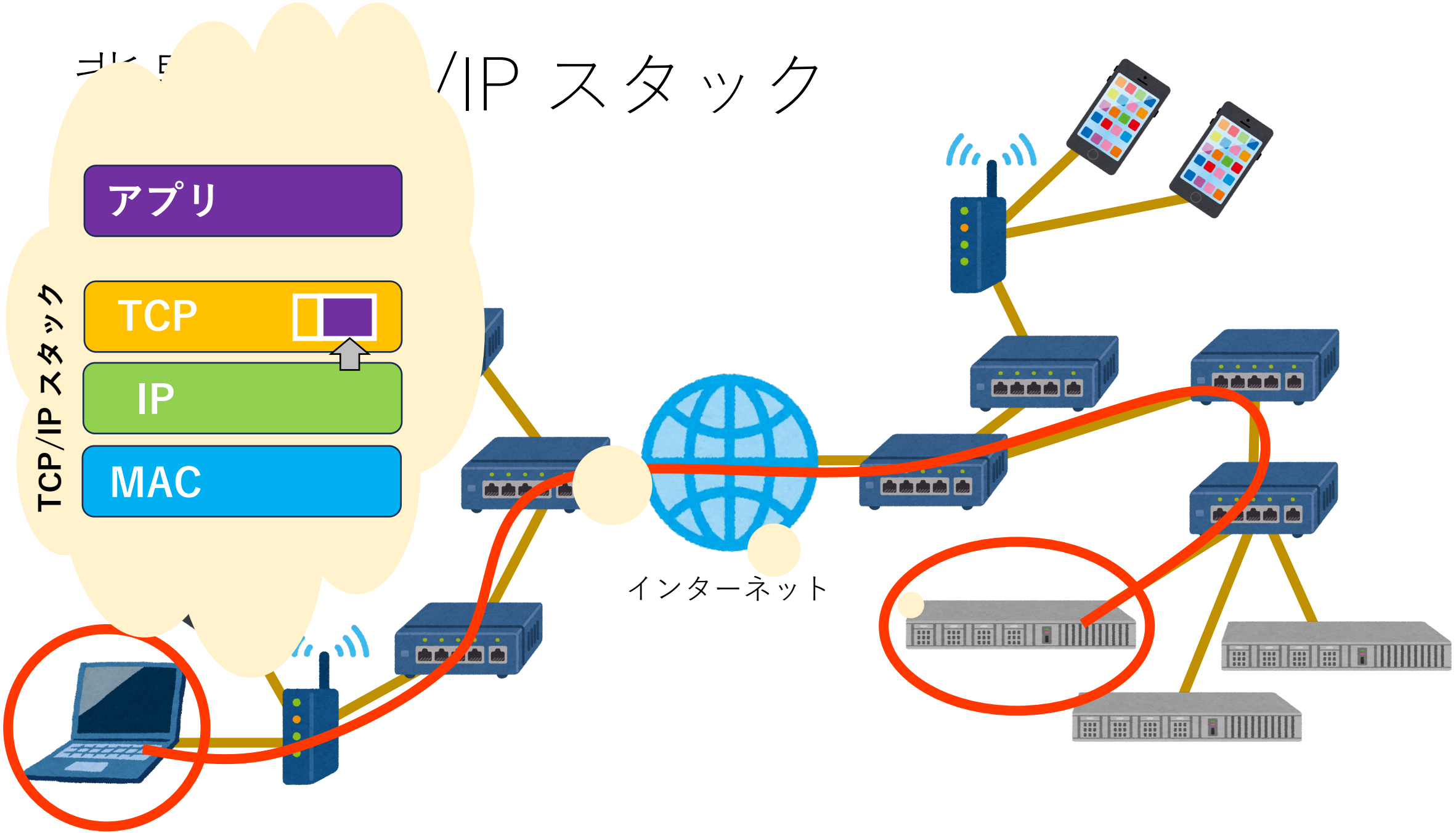
/IP スタック



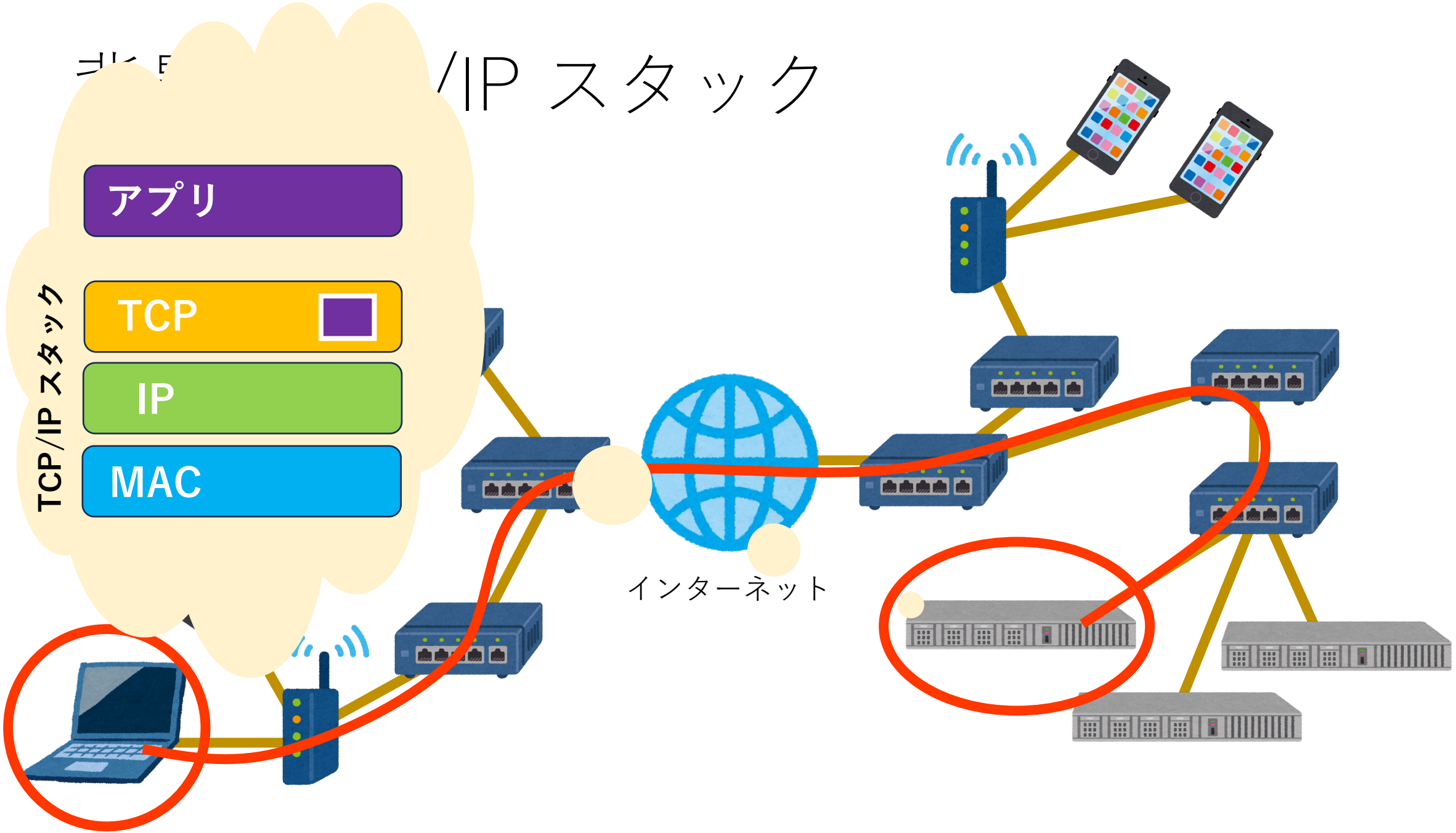
/IP スタック



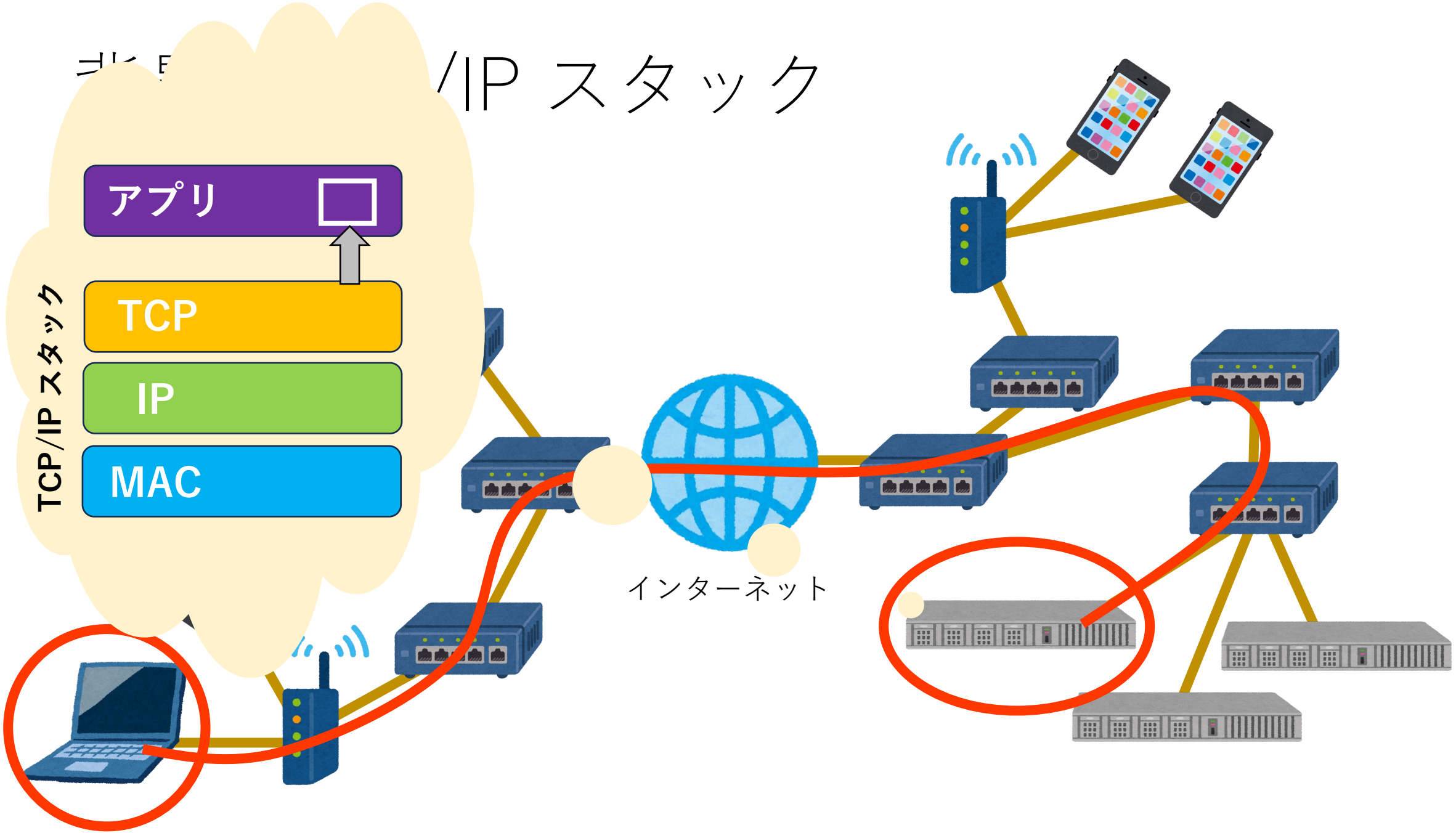
/IP スタック



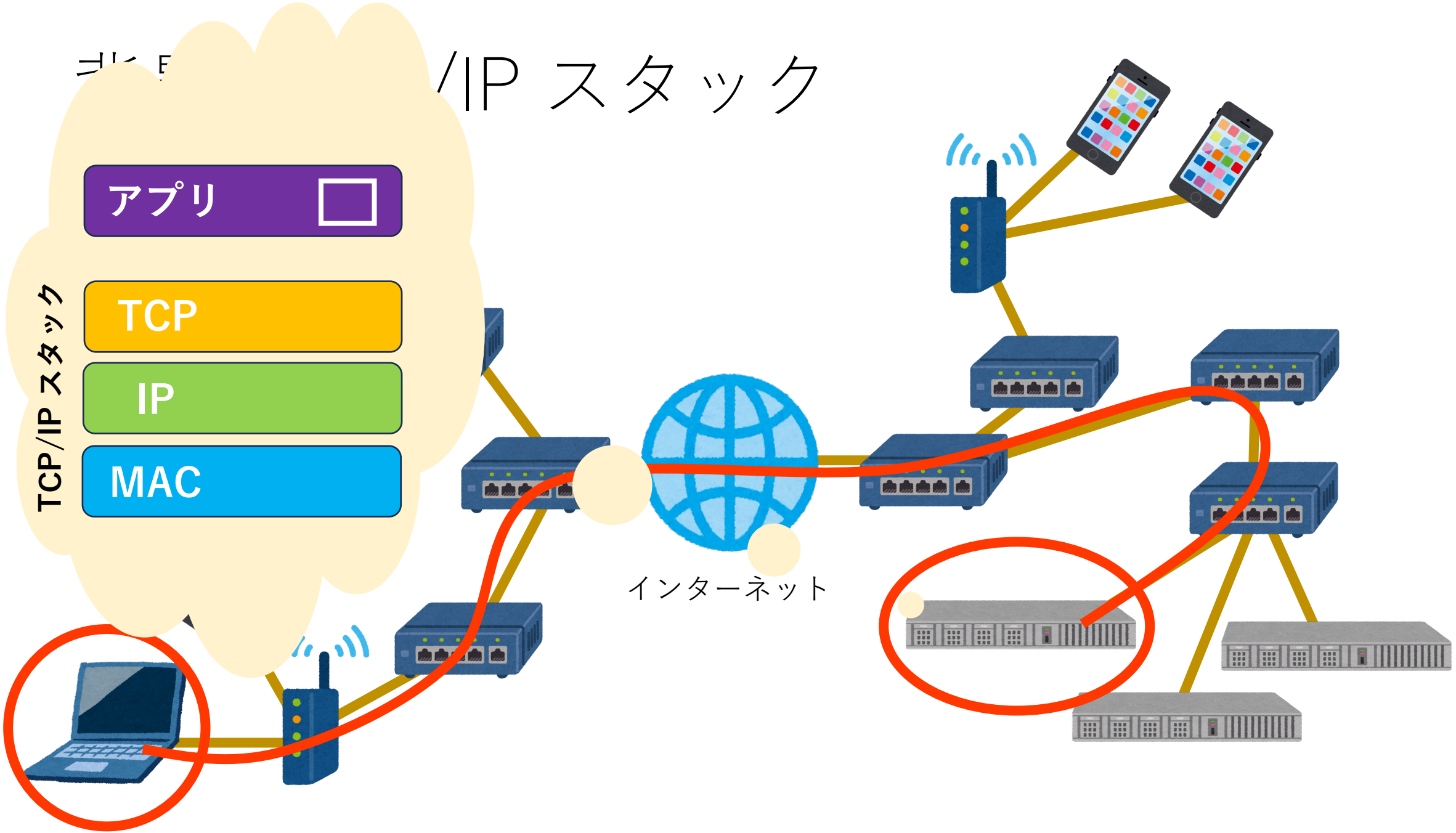
/IP スタック



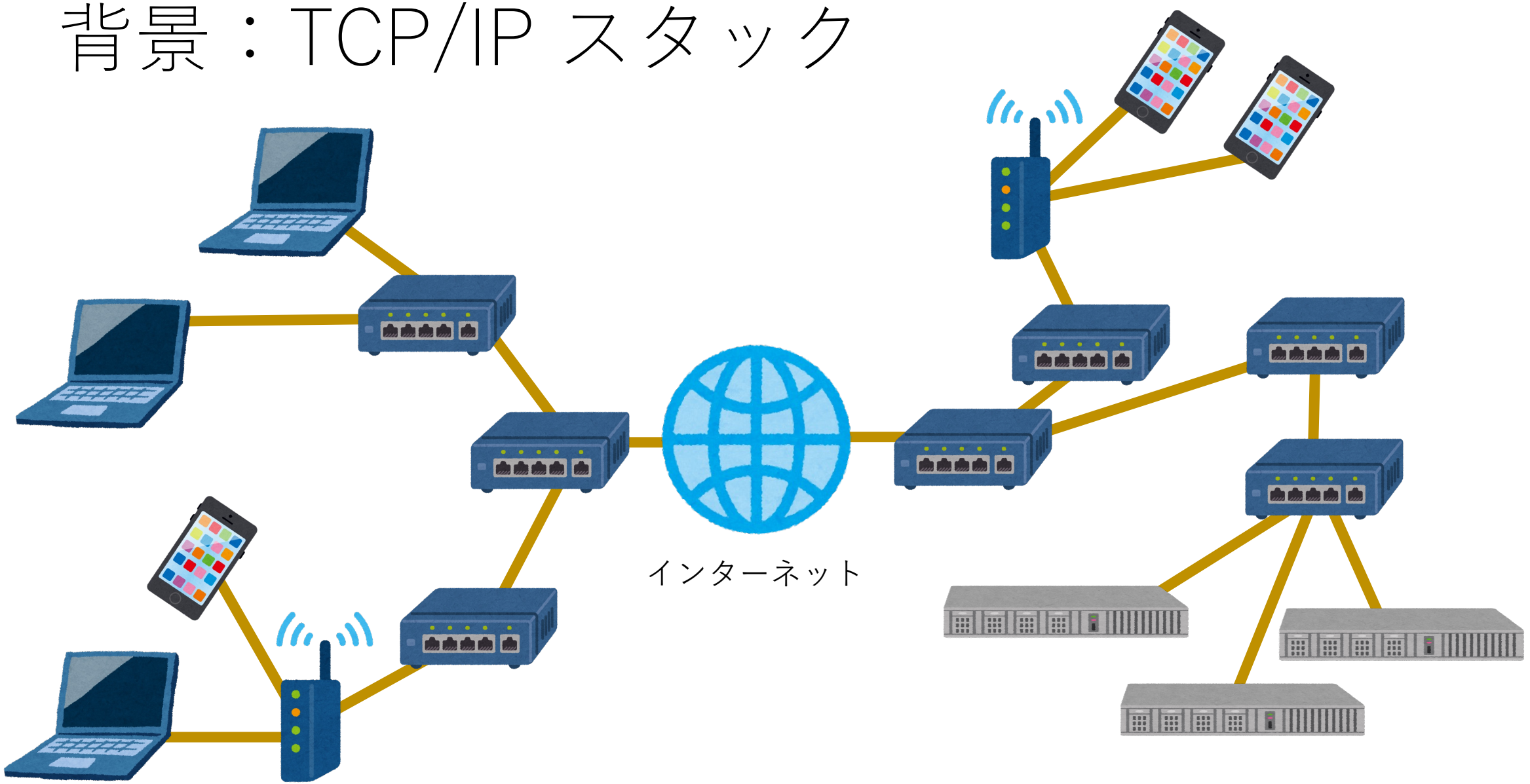
/IP スタック



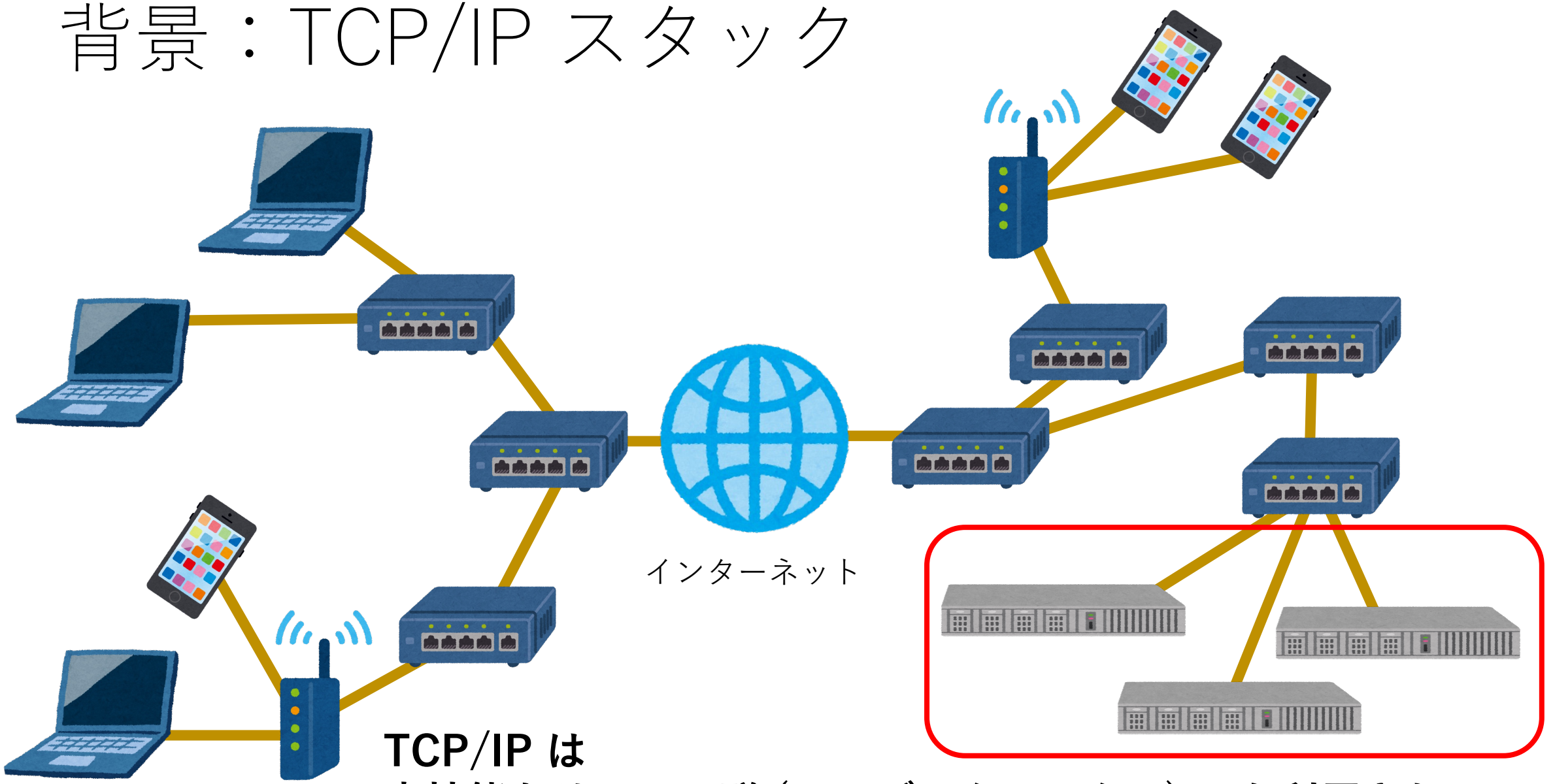
/IP スタック



背景：TCP/IP スタック

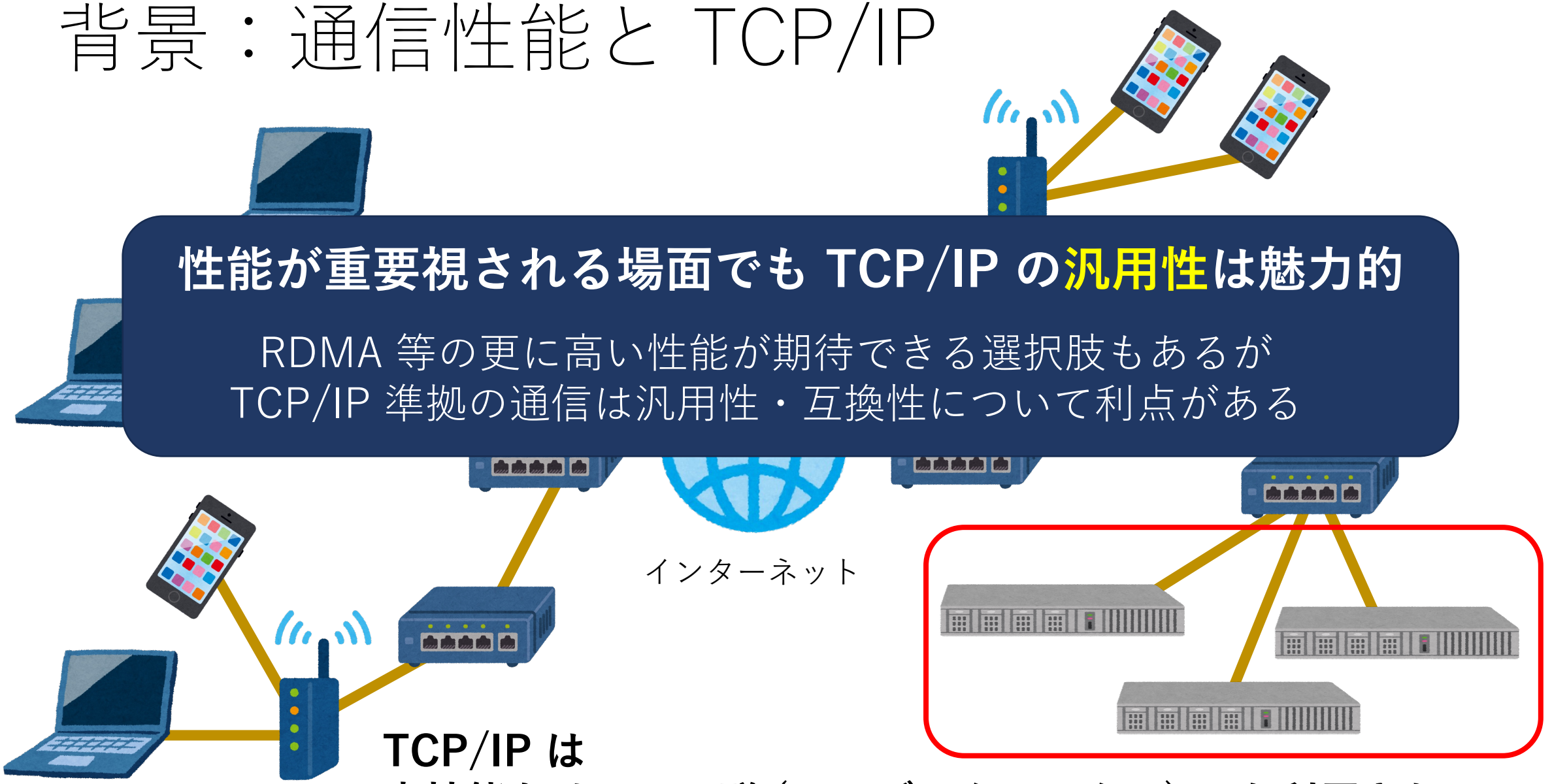


背景：TCP/IP スタック



TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

背景：通信性能と TCP/IP

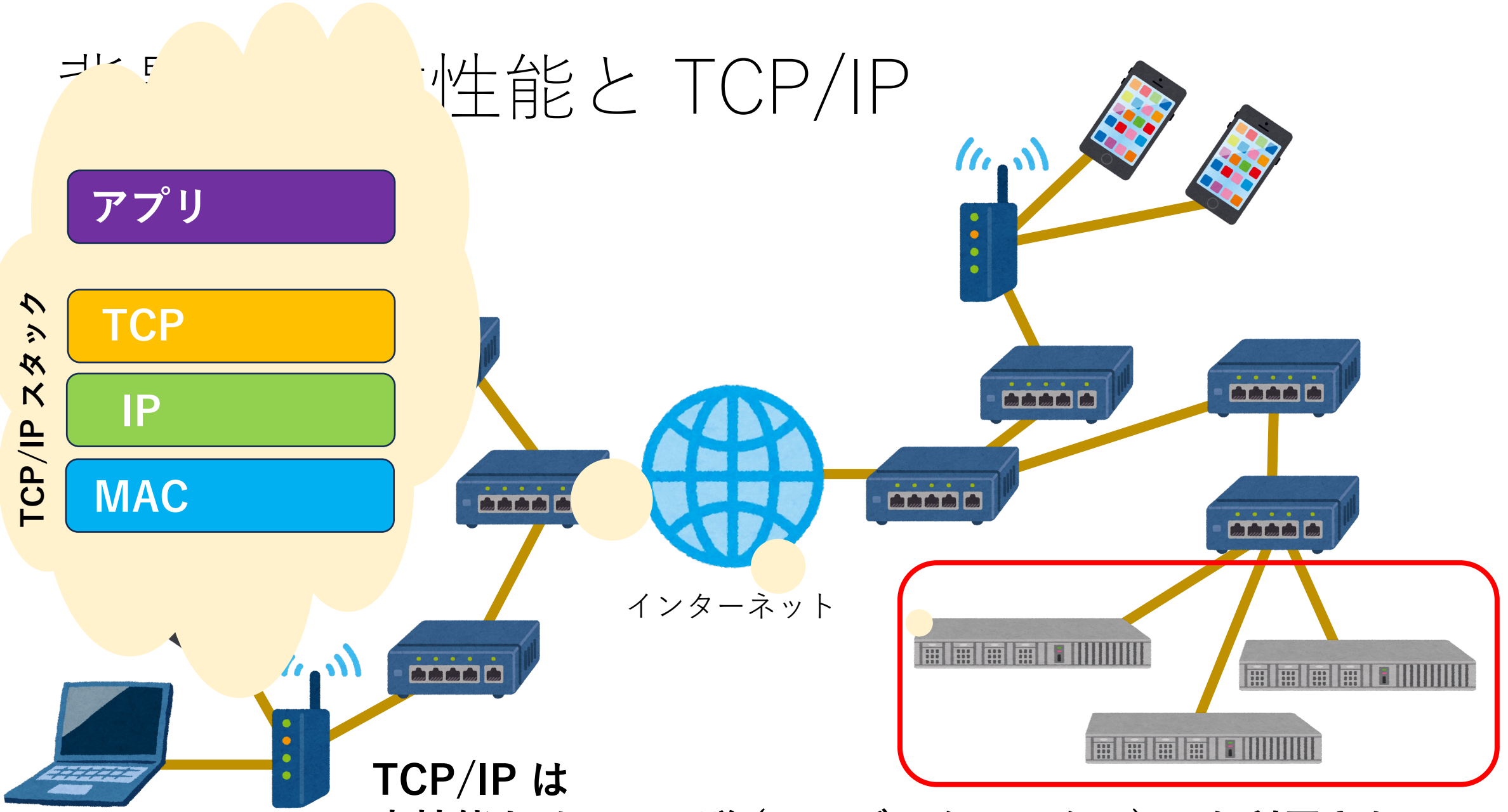


性能が重要視される場面でも TCP/IP の汎用性は魅力的

RDMA 等の更に高い性能が期待できる選択肢もあるが
TCP/IP 準拠の通信は汎用性・互換性について利点がある

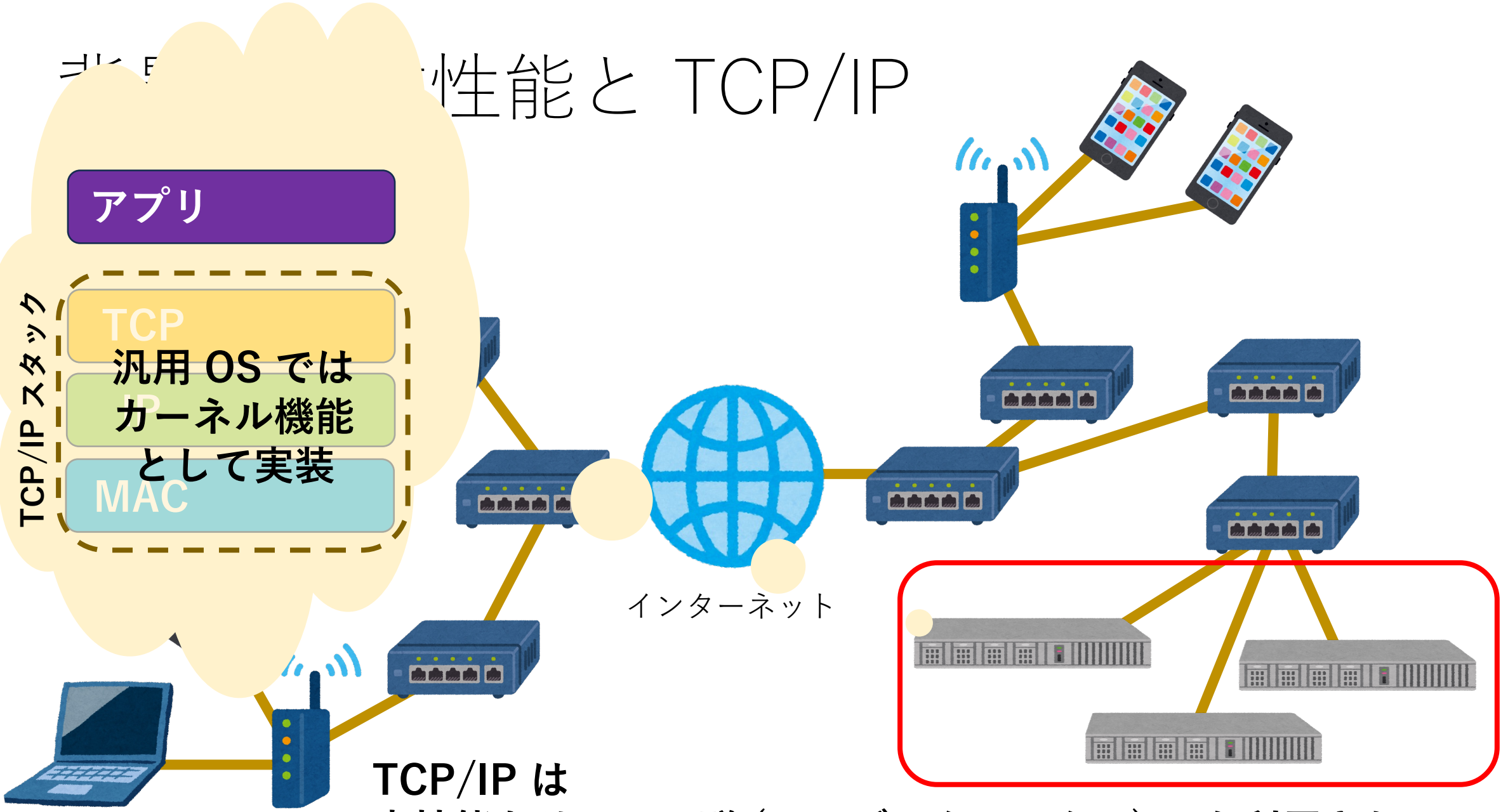
TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP



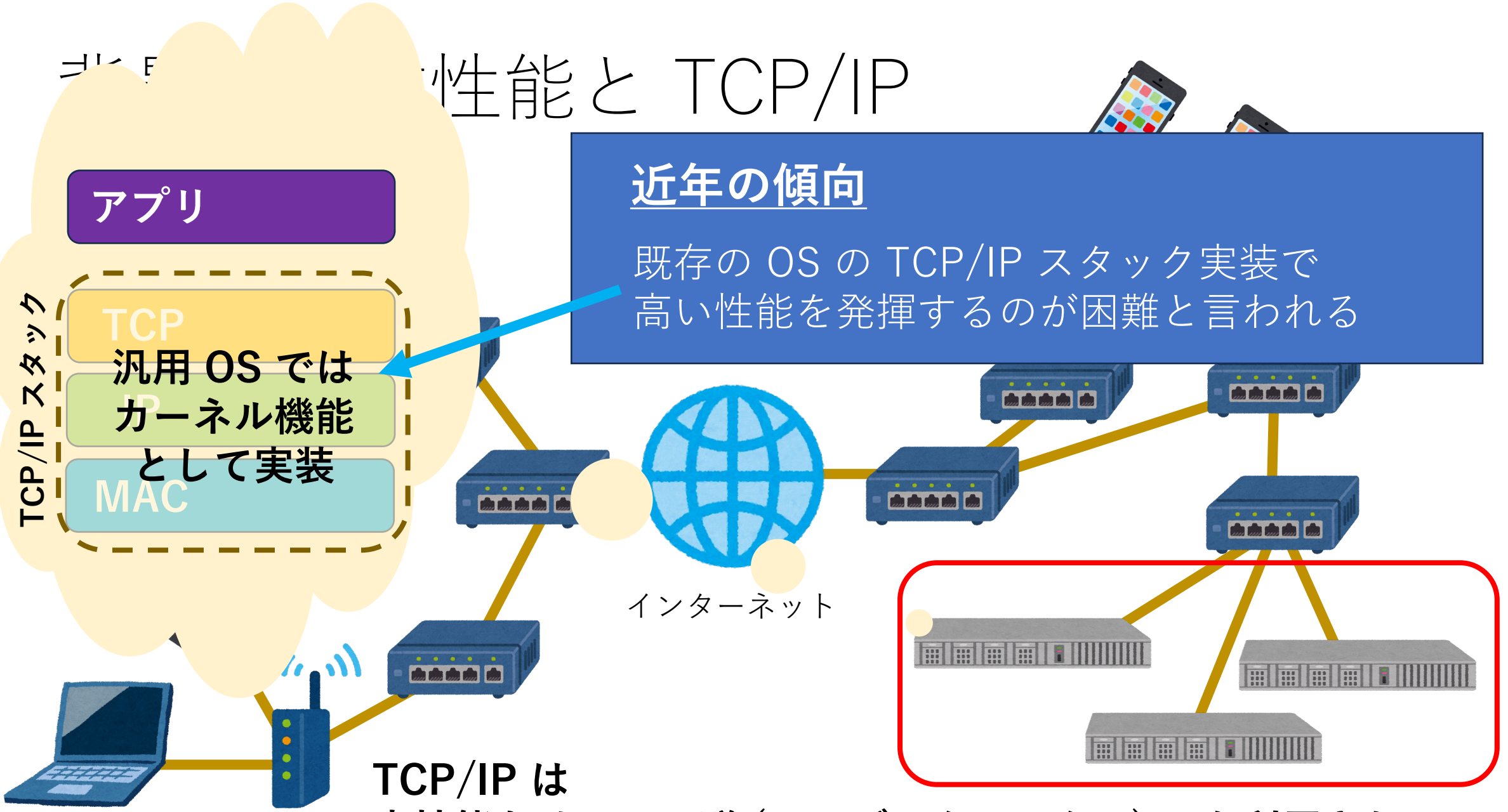
TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP



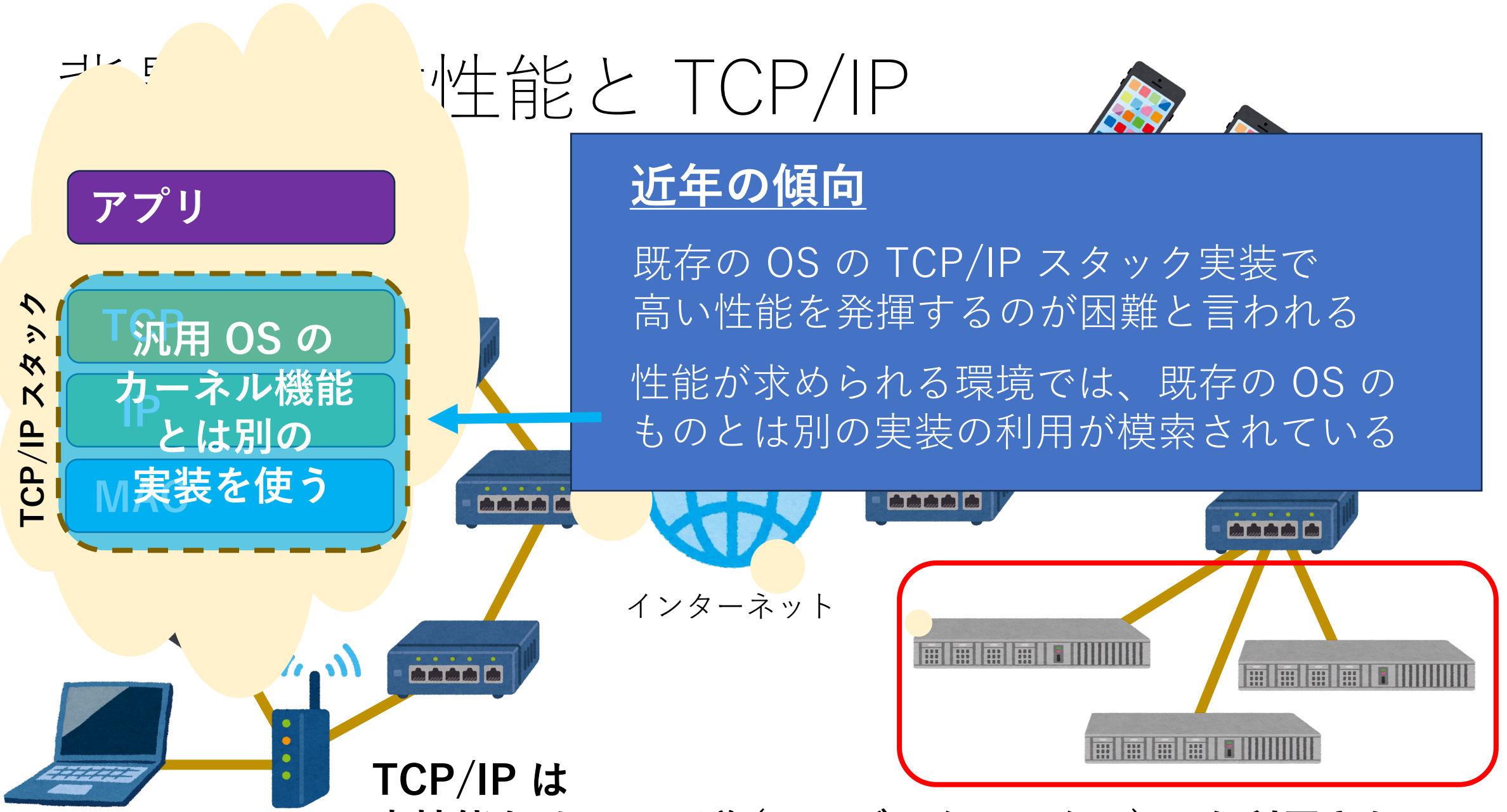
TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP



TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP



近年の傾向

既存の OS の TCP/IP スタック実装で
高い性能を発揮するのが困難と言われる

性能が求められる環境では、既存の OS の
ものとは別の実装の利用が模索されている

TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP

近年の傾向

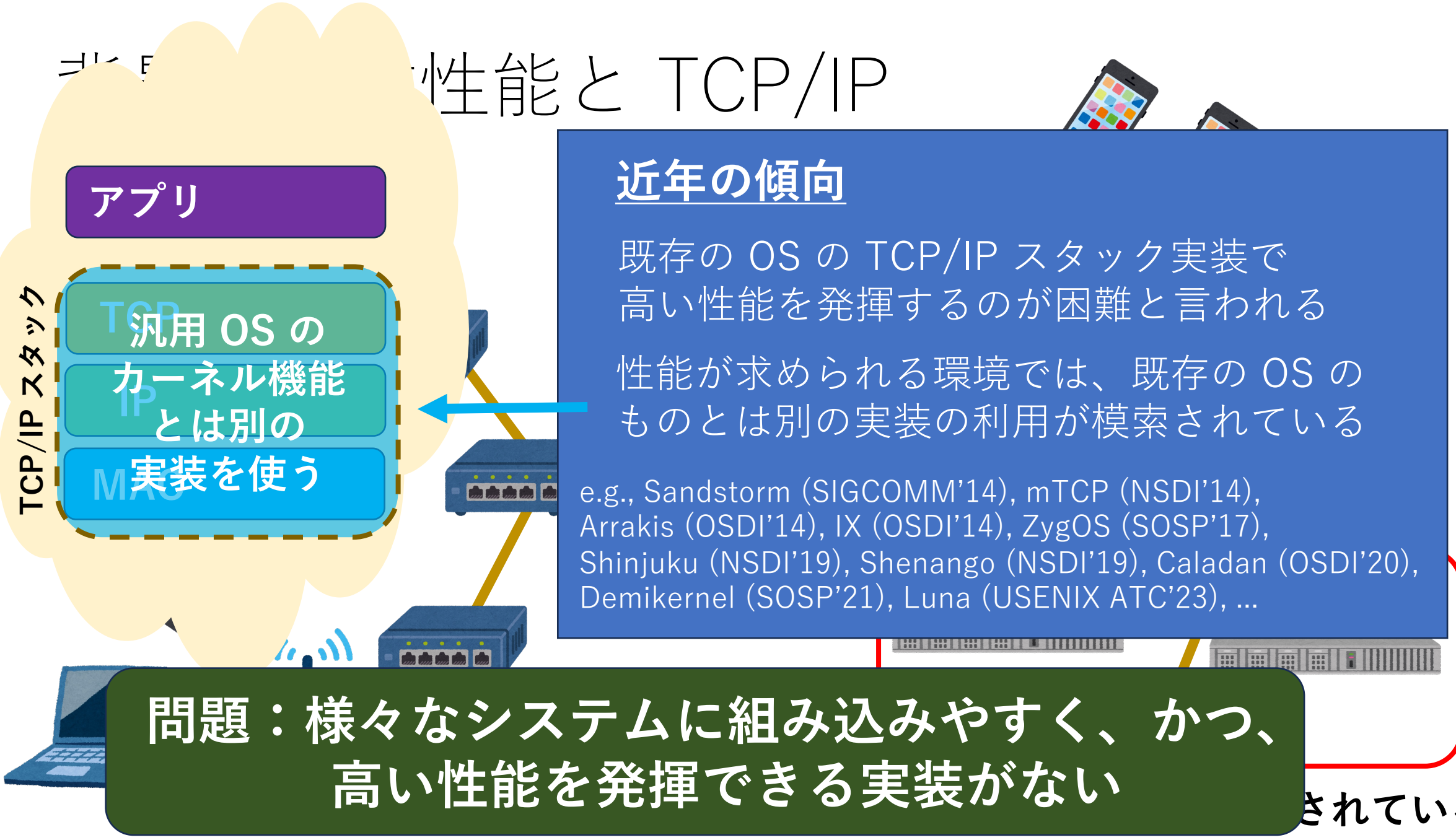
既存の OS の TCP/IP スタック実装で
高い性能を発揮するのが困難と言われる

性能が求められる環境では、既存の OS の
ものとは別の実装の利用が模索されている

e.g., Sandstorm (SIGCOMM'14), mTCP (NSDI'14),
Arrakis (OSDI'14), IX (OSDI'14), ZygOS (SOSP'17),
Shinjuku (NSDI'19), Shenango (NSDI'19), Caladan (OSDI'20),
Demikernel (SOSP'21), Luna (USENIX ATC'23), ...

TCP/IP は
高性能なサーバー群 (e.g, データセンター) でも利用されている

性能と TCP/IP



近年の傾向

既存の OS の TCP/IP スタック実装で
高い性能を発揮するのが困難と言われる

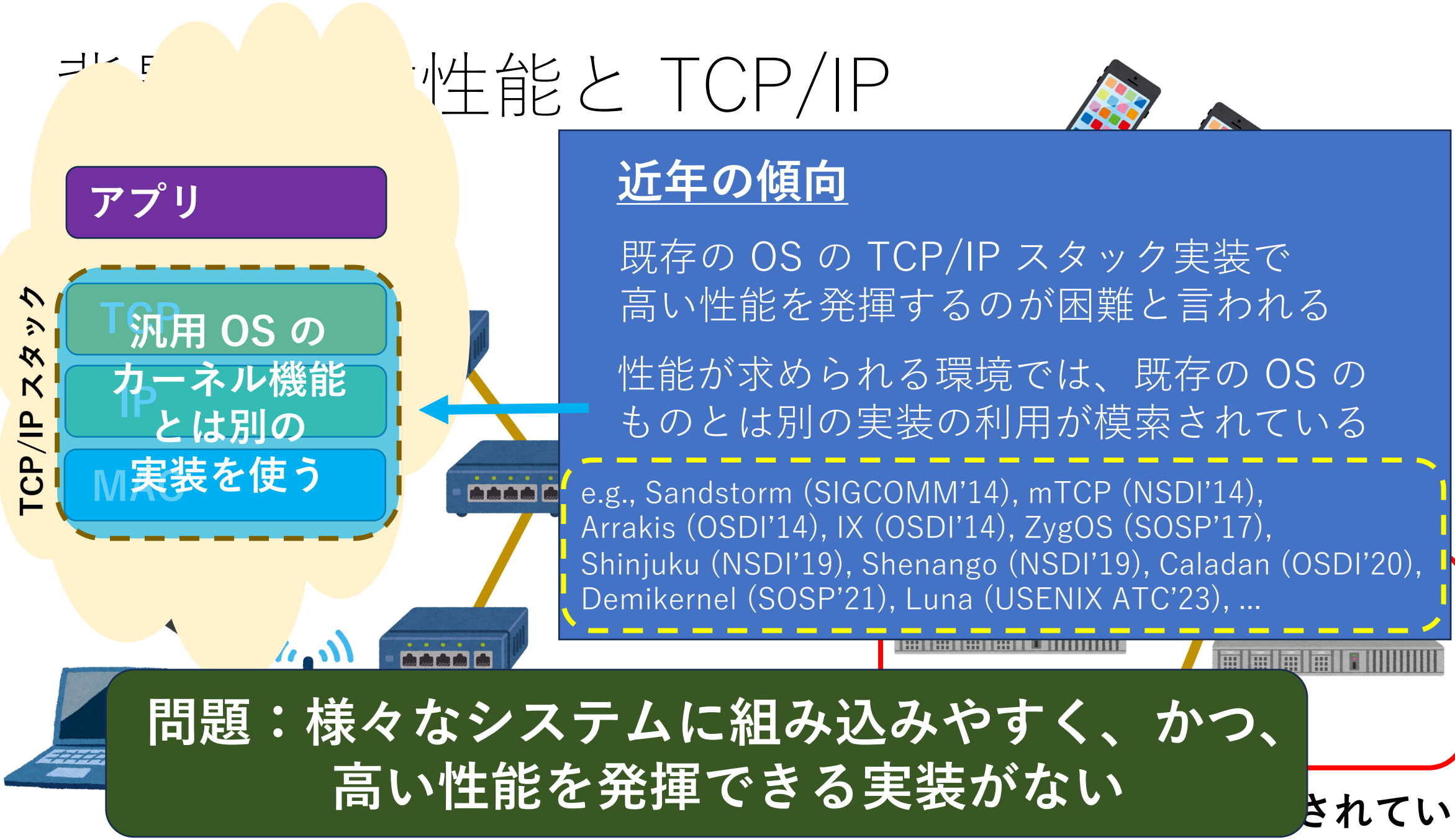
性能が求められる環境では、既存の OS の
ものとは別の実装の利用が模索されている

e.g., Sandstorm (SIGCOMM'14), mTCP (NSDI'14), Arrakis (OSDI'14), IX (OSDI'14), ZygOS (SOSP'17), Shinjuku (NSDI'19), Shenango (NSDI'19), Caladan (OSDI'20), Demikernel (SOSP'21), Luna (USENIX ATC'23), ...

問題：様々なシステムに組み込みやすく、かつ、高い性能を発揮できる実装がない

されている

性能と TCP/IP


アプリ
TCP/IP スタック
TCP
汎用 OS の
カーネル機能
とは別の
実装を使う
IP
MAC
近年の傾向
既存の OS の TCP/IP スタック実装で
高い性能を発揮するのが困難と言われる
性能が求められる環境では、既存の OS の
ものとは別の実装の利用が模索されている
e.g., Sandstorm (SIGCOMM'14), mTCP (NSDI'14),
Arrakis (OSDI'14), IX (OSDI'14), ZygOS (SOSP'17),
Shinjuku (NSDI'19), Shenango (NSDI'19), Caladan (OSDI'20),
Demikernel (SOSP'21), Luna (USENIX ATC'23), ...
問題：様々なシステムに組み込みやすく、かつ、
高い性能を発揮できる実装がない
されている

非…性能と TCP/IP

なぜ既にある実装を使わないのか？

既存の実装の多くが互換性や性能面で課題がある

例えば、
- VPP は Mellanox の NIC の機能をうまく使えない
- IX は特定の Linux のバージョンに依存する上、
  Intel 製 NIC しか利用できない

結局、新しく書き直すのが一番早いと判断した

Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu, “Deploying User-space TCP at Cloud Scale with LUNA”, USENIX ATC 2023
論文 3.4 章より

…向

…S の TCP/IP スタック実装で
…を発揮するのが困難と言われる

…められる環境では、既存の OS の
…の実装の利用が模索されている

…OMM’14), mTCP (NSDI’14),
…SDI’14), ZygOS (SOSP’17),
…SDI’19), Shen…ngo (NSDI’19), Caladan (OSDI’20),
Demikernel (SOSP’21), Luna (USENIX ATC’23), …

問題：様々なシステムに組み込みやすく、かつ、高い性能を発揮できる実装がない

…されている

近年は、**性能に特化した独自の実装**を行うアプローチが
広く模索されるようになり、高速な TCP/IP スタックは
そのような実装に必須のパーツであるため
利用しやすく、高速な TCP/IP スタック実装がないのは
多くの人にとっての不利益となっていると考える

TCP/IP スタック

とは別の
実装を使う

MAC

ものとは別の実装の利用が模索されている

e.g., Sandstorm (SIGCOMM'14), mTCP (NSDI'14),
Arrakis (OSDI'14), IX (OSDI'14), ZygOS (SOSP'17),
Shinjuku (NSDI'19), Shenango (NSDI'19), Caladan (OSDI'20),
Demikernel (SOSP'21), Luna (USENIX ATC'23), ...

**問題：様々なシステムに組み込みやすく、かつ、
高い性能を発揮できる実装がない**

されている

本研究

- 様々なシステムに組み込みやすく、かつ、高い性能を発揮できる TCP/IP スタック実装を模索する

組み込みやすさと性能についての要件

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

既存の性能に最適化された実装

- 😞 CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
- 😞 色々なスレッド実行形式に対応可能
- 😀 NIC のオフロード機能を利用できる
- 😀 メモリコピー回数を抑えられる
- 😀 マルチコア環境でのスケーラビリティ

e.g., Sandstorm (SIGCOMM'14), mTCP (NSDI'14), Arrakis (OSDI'14), IX (OSDI'14), ZygOS (SOSP'17), Shinjuku (NSDI'19), Shenango (NSDI'19), Caladan (OSDI'20), Demikernel (SOSP'21), Luna (USENIX ATC'23), ...

既存の可搬性に最適化された実装

- 😀 CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
- 😀 色々なスレッド実行形式に対応可能
- 😞 NIC のオフロード機能を利用できる
- 😞 メモリコピー回数を抑えられる
- 😞 マルチコア環境でのスケーラビリティ

可搬性を考慮した TCP/IP スタック実装は
小さな組み込みデバイスが主な用途である
場合が多く、性能への配慮が不十分になりがち

e.g, lwIP, uIP, picoTCP, FNET

提案する TCP/IP 実装

- CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
- 色々なスレッド実行形式に対応可能
- NIC のオフロード機能を利用できる
- メモリコピー回数を抑えられる
- マルチコア環境でのスケーラビリティ

組み込みやすさと性能についての要件

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

組み込みやすさと性能についての要件

1. **CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い**
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

TCP/IP 実装が持ち込む依存関係の弊害

- 依存する要素は利用可能な環境を大きく狭める

非…性能と TCP/IP

なぜ既にある実装を使わないのか？

既存の実装の多くが互換性や性能面で課題がある

例えば、
- VPP は Mellanox の NIC の機能をうまく使えない
- IX は特定の Linux のバージョンに依存する上、
  Intel 製 NIC しか利用できない

結局、新しく書き直すのが一番早いと判断した

Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu, “Deploying User-space TCP at Cloud Scale with LUNA”, USENIX ATC 2023
論文 3.4 章より

…向

…S の TCP/IP スタック実装で
…を発揮するのが困難と言われる

…められる環境では、既存の OS の
…の実装の利用が模索されている

…OMM’14), mTCP (NSDI’14),
…SDI’14), ZygOS (SOSP’17),
…SDI’19), Shen…ngo (NSDI’19), Caladan (OSDI’20),
Demikernel (SOSP’21), Luna (USENIX ATC’23), …

問題：様々なシステムに組み込みやすく、かつ、高い性能を発揮できる実装がない

…されている

非…性能と TCP/IP

なぜ既にある実装を使わないのか？

既存の実装の多くが互換性や性能面で課題がある

例えば、
- VPP は Mellanox の NIC の機能をうまく使えない
- IX は特定の Linux のバージョンに依存する上、
  Intel 製 NIC しか利用できない

結局、新しく書き直すのが一番早いと判断した

Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu, “Deploying User-space TCP at Cloud Scale with LUNA”, USENIX ATC 2023
論文 3.4 章より

傾向

S の TCP/IP スタック実装で
を発揮するのが困難と言われる

められる環境では、既存の OS の
の実装の利用が模索されている

OMM’14), mTCP (NSDI’14),
SDI’14), ZygOS (SOSP’17),
SDI’19), Shenango (NSDI’19), Caladan (OSDI’20),
Demikernel (SOSP’21), Luna (USENIX ATC’23), …

問題：様々なシステムに組み込みやすく、かつ、高い性能を発揮できる実装がない

されている

TCP/IP 実装が持ち込む依存関係の弊害

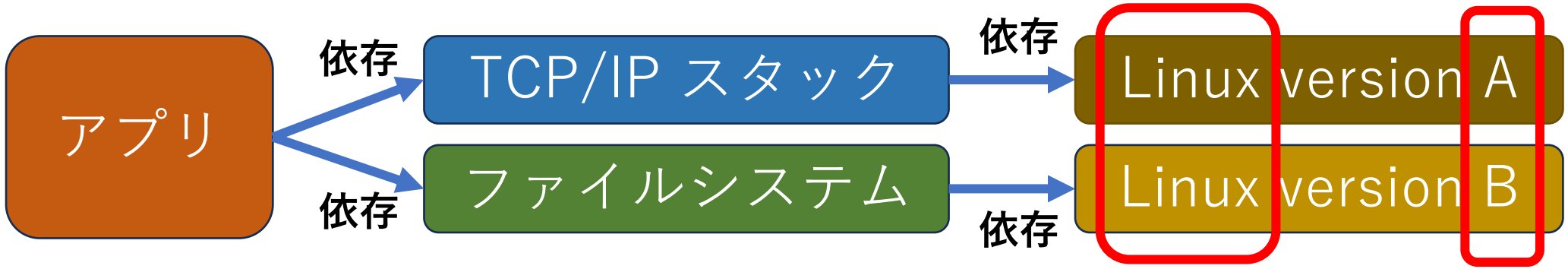
- 依存する要素は利用可能な環境を大きく狭める

TCP/IP 実装が持ち込む依存関係の弊害

- 依存する要素は利用可能な環境を大きく狭める
- 依存されている側にも多くの場合に依存する要素がある
  - 一つ依存要素を増やすと間接的に依存要素が大きく増えやすい
  - 依存する二つ以上の要素が共存できない要素に依存すると、その実装が使えなくなる

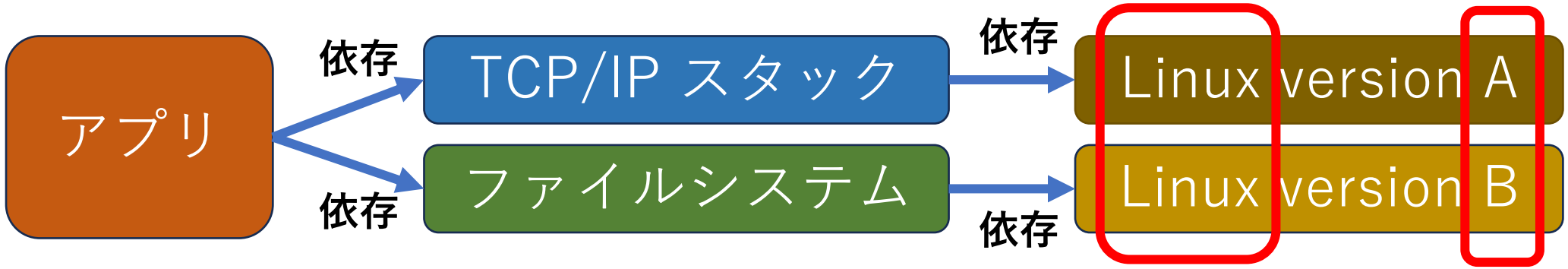
TCP/IP 実装が持ち込む依存関係の弊害

- 依存する要素は利用可能な環境を大きく狭める
- 依存されている側にも多くの場合に依存する要素がある
  - 一つ依存要素を増やすと間接的に依存要素が大きく増えやすい
  - 依存する二つ以上の要素が共存できない要素に依存すると、その実装が使えなくなる



TCP/IP 実装が持ち込む依存関係の弊害

- 依存する要素は利用可能な環境を大きく狭める
- 依存されている側にも多くの場合に依存する要素がある
  - 一つ依存要素を増やすと間接的に依存要素が大きく増えやすい
  - 依存する二つ以上の要素が共存できない要素に依存すると、その実装が使えなくなる



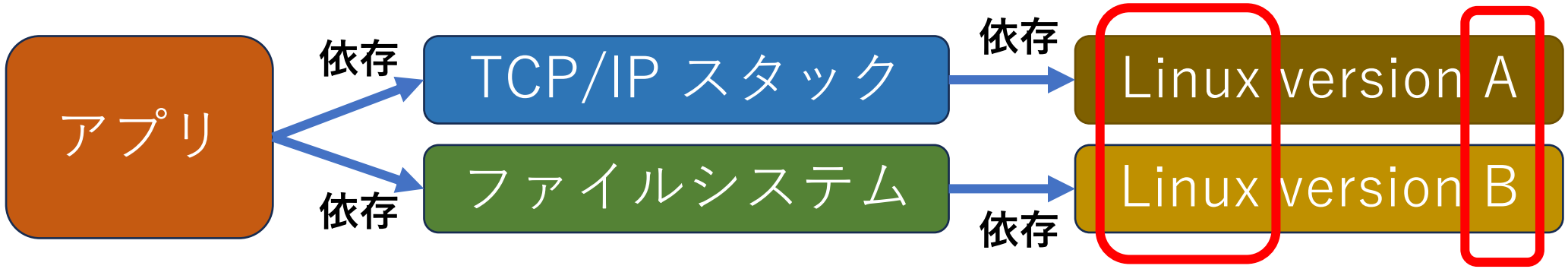
上の場合 TCP/IP スタックとファイルシステムを併用できない

複数のバージョンのカーネルが一つのシステムに存在できないため

TCP/IP 実装が持ち込む依存関係の弊害

- 依存する要素は利用可能な環境を大きく狭める
- 依存されている側にも多くの場合に依存する要素がある
- が使えなくなる

TCP/IP スタック実装の利用者への制約を最小限にするために
TCP/IP スタック実装が持ち込む依存要素は最低限であるべき



上の場合 TCP/IP スタックとファイルシステムを併用できない

複数のバージョンのカーネルが一つのシステムに存在できないため

標準に準拠するだけでは不十分

- POSIX 等の標準的な実行環境以外もよく利用される
  - Arrakis, Unikernels : 新しい OS
  - Shenango/Caladan : 独自のユーザー空間ランタイムで TCP/IP 処理

標準に準拠するだけでは不十分

- POSIX 等の標準的な実行環境以外もよく利用される
  - Arrakis, Unikernels : 新しい OS
  - Shenango/Caladan : 独自のユーザー空間ランタイムで TCP/IP 処理
- POSIX 標準準拠だからといって新しく実装されるシステムと組み合わせやすいとは限らない
  - Arrakis : ポータブルな TCP/IP スタック実装 (lwIP) を利用
  - Shenango/Caladan : 独自に新しく TCP/IP スタックを実装

標準に準拠するだけでは不十分

- POSIX 等の標準的な実行環境以外もよく利用される
  - Arrakis, Unikernels : 新しい OS
  - Shenango/Caladan : 独自のユーザー空間ランタイムで TCP/IP 処理
- POSIX 標準準拠だからといって新しく実装されるシステムと組み合わせやすいとは限らない
  - Arrakis : ポータブルな TCP/IP スタック実装 (lwIP) を利用
  - Shenango/Caladan : 独自に新しく TCP/IP スタックを実装

組み込みやすさの観点から、
OS、ライブラリ、コンパイラへの依存度は最低限であるべき

組み込みやすさと性能についての要件

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. **色々なスレッド実行形式に対応可能**
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = app_logic(rx_payload);
3.enqueue(tx_payload, app_tx_queue);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる
4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

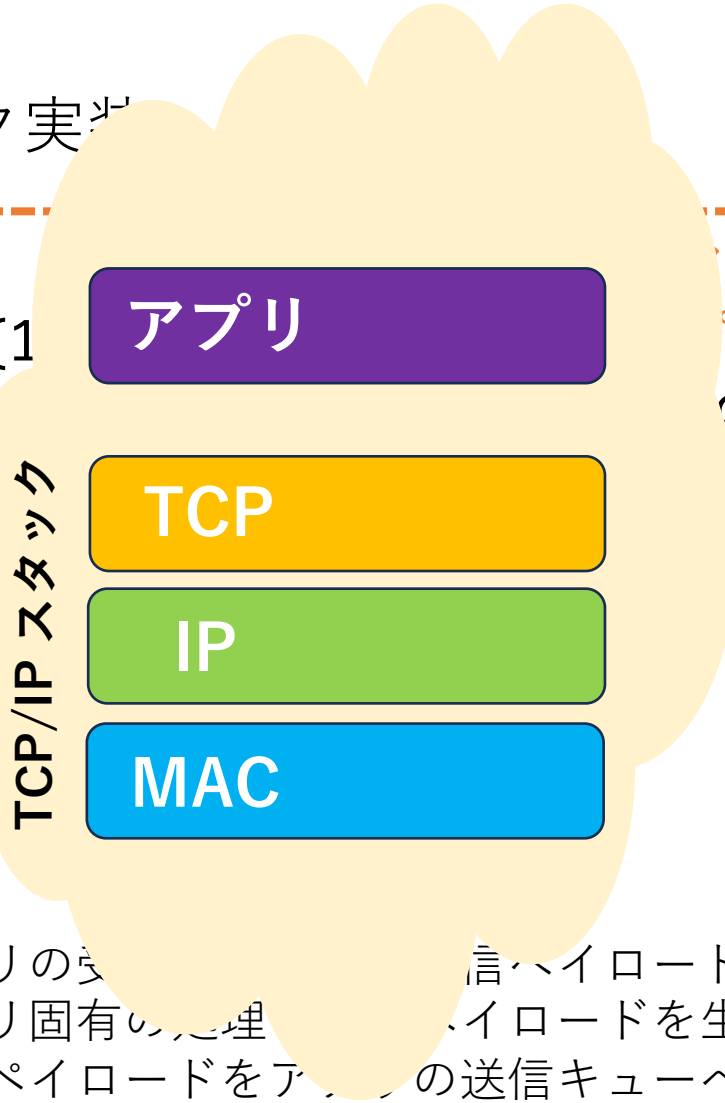
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

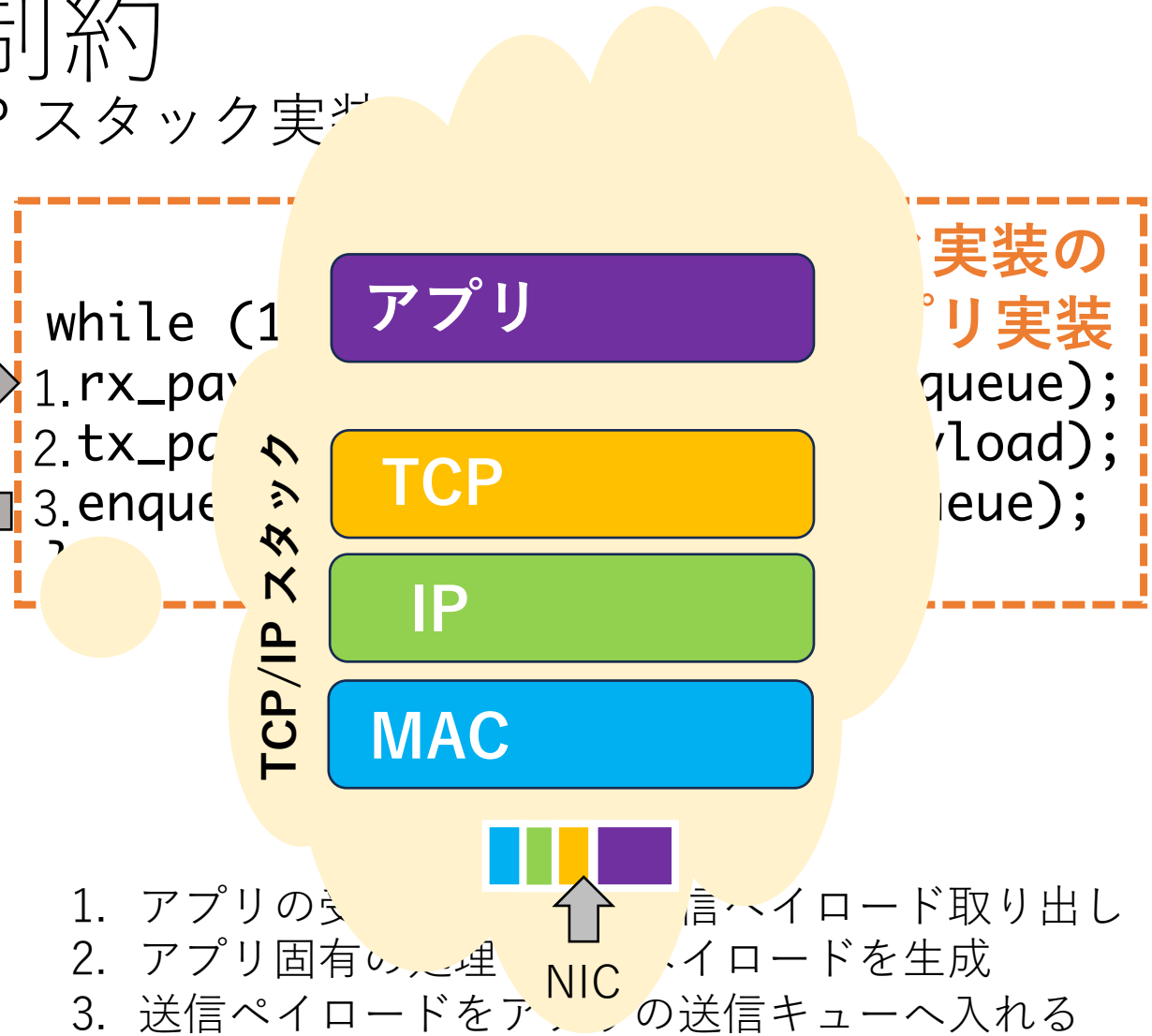
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 ペイロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

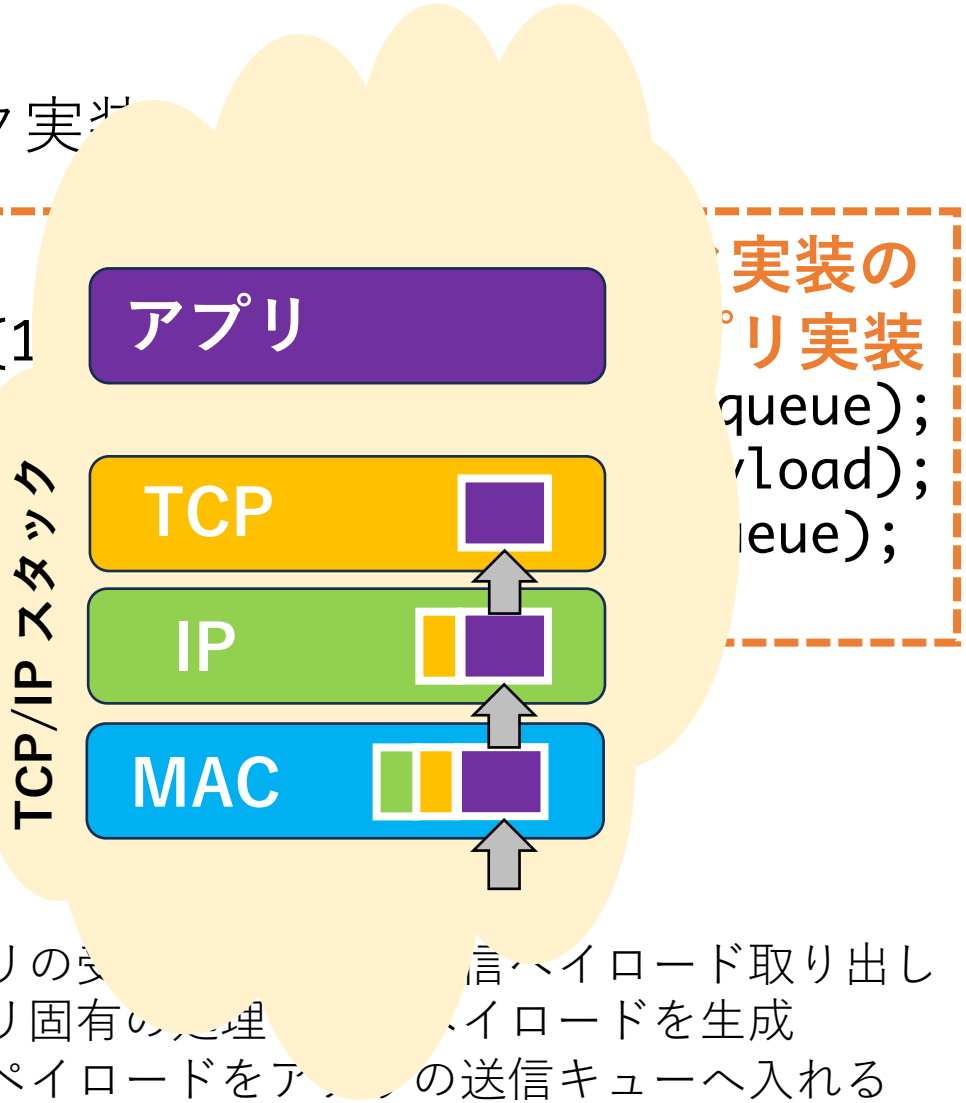
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

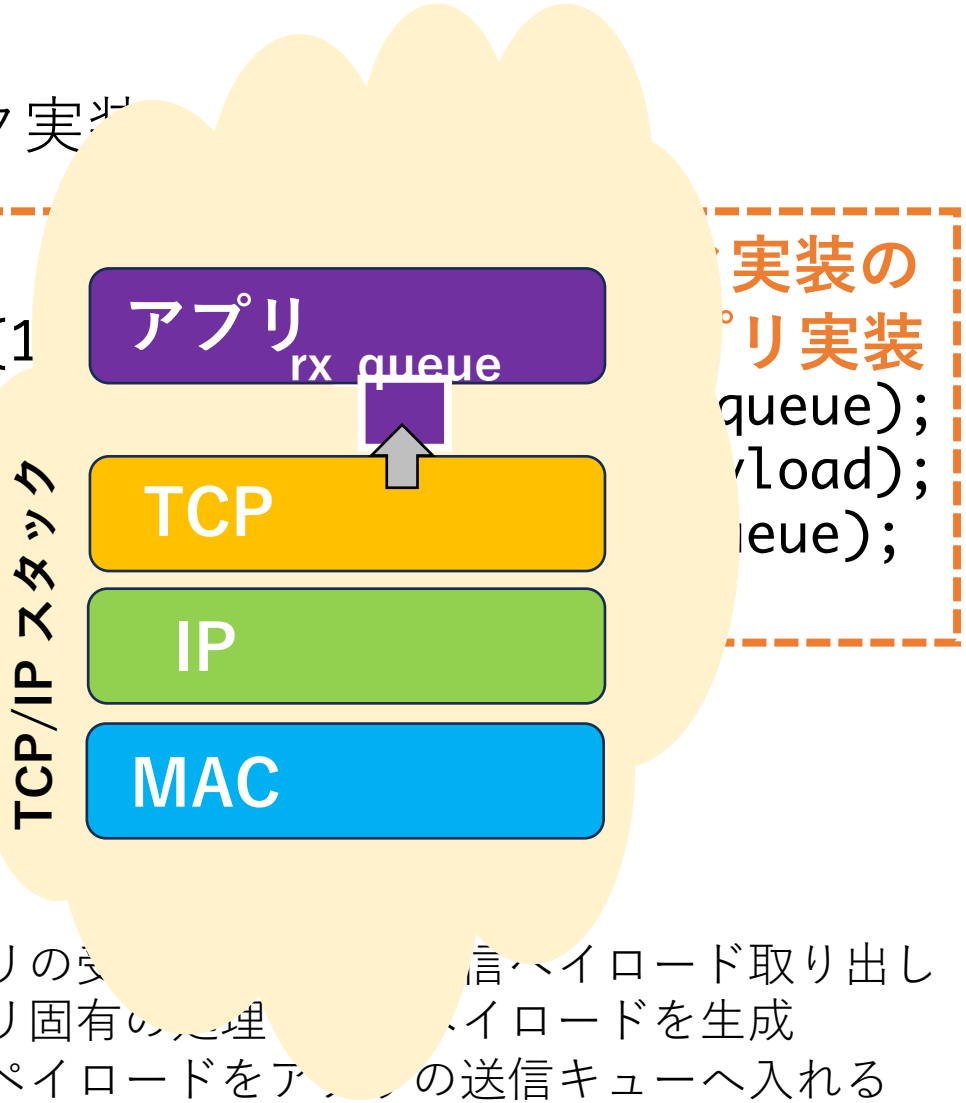
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

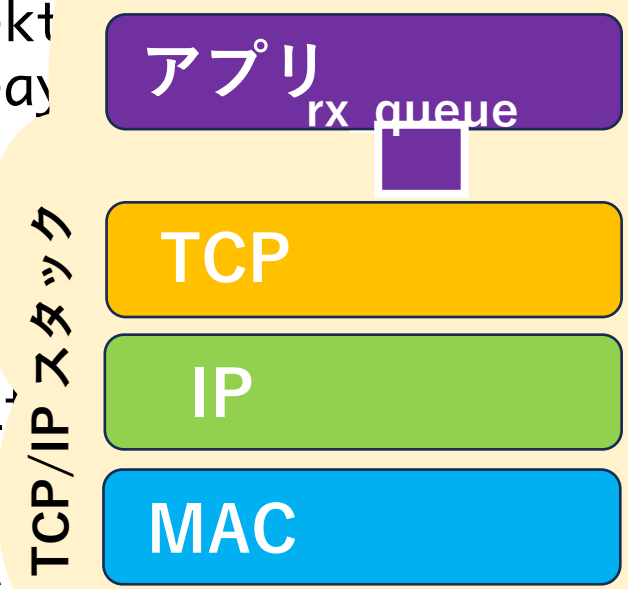
スレッド実行形式の制約

比較的一般的な[illegible]された TCP/IP スタック実装

```
while (
1. rx_pkt
2. rx_pay                    f);
3. enqu                      queue);

4. tx_p                      _qu  );
5. tx_p                      d);
6. nic_
}
```

TCP/IP スタック実装



1. NIC [illegible] 出し
2. TCP/IP [illegible]
3. 受信ペイ[illegible] 受信キューへ入れる

4. アプリの送信キュー[illegible] 送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = app_logic(rx_payload);
3.enqueue(tx_payload, app_tx_queue);
}
```

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

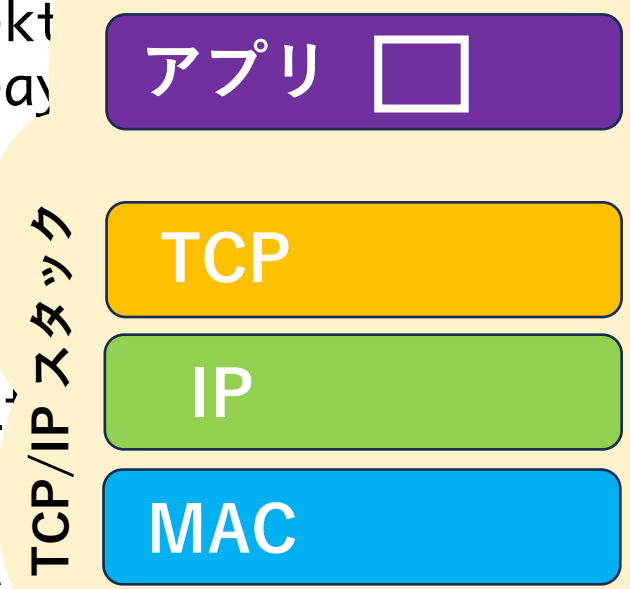
スレッド実行形式の制約

比較的一般的な[illegible]された TCP/IP スタック実装

```
while (
1. rx_pkt
2. rx_pay                    f);
3. enqu                      queue);

4. tx_p                      _qu   );
5. tx_p                      d);
6. nic_
}
```

TCP/IP スタック実装



1. NIC [illegible] 出し
2. TCP/IP [illegible]
3. 受信ペイ[illegible] 受信キューへ入れる

4. アプリの送信キュー[illegible] 送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = app_logic(rx_payload);
3.enqueue(tx_payload, app_tx_queue);
}
```

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

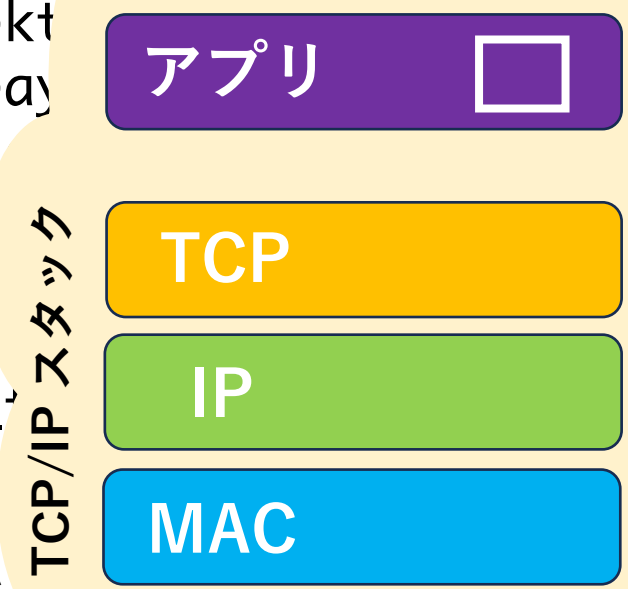
スレッド実行形式の制約

比較的一般的な[illegible]された TCP/IP スタック実装

```
while (
1. rx_pkt
2. rx_pay                    f);
3. enqu                      queue);

4. tx_p                      _qu  )
5. tx_p                      d);
6. nic_
}
```

TCP/IP スタック実装



1. NIC [illegible] 出し
2. TCP/IP [illegible]
3. 受信ペイ[illegible] 受信キューへ入れる

4. アプリの送信キュー[illegible] 送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = app_logic(rx_payload);
3.enqueue(tx_payload, app_tx_queue);
}
```

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

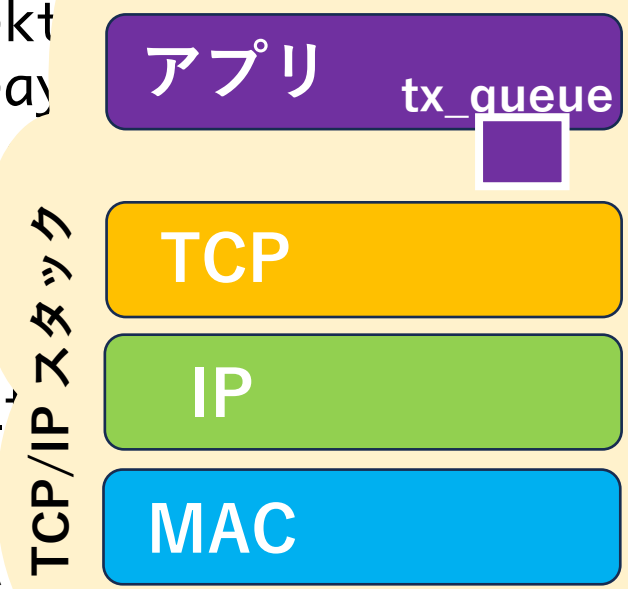
スレッド実行形式の制約

比較的一般的な[illegible]された TCP/IP スタック実装

```
while (
1. rx_pkt
2. rx_pay                    f);
3. enqu                      queue);

4. tx_p                      _qu  );
5. tx_p                      d);
6. nic_
}
```

TCP/IP スタック実装



1. NIC [illegible] 出し
2. TCP/IP [illegible]
3. 受信ペイ[illegible] 受信キューへ入れる

4. アプリの送信キュー[illegible] 送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = app_logic(rx_payload);
3.enqueue(tx_payload, app_tx_queue);
}
```

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

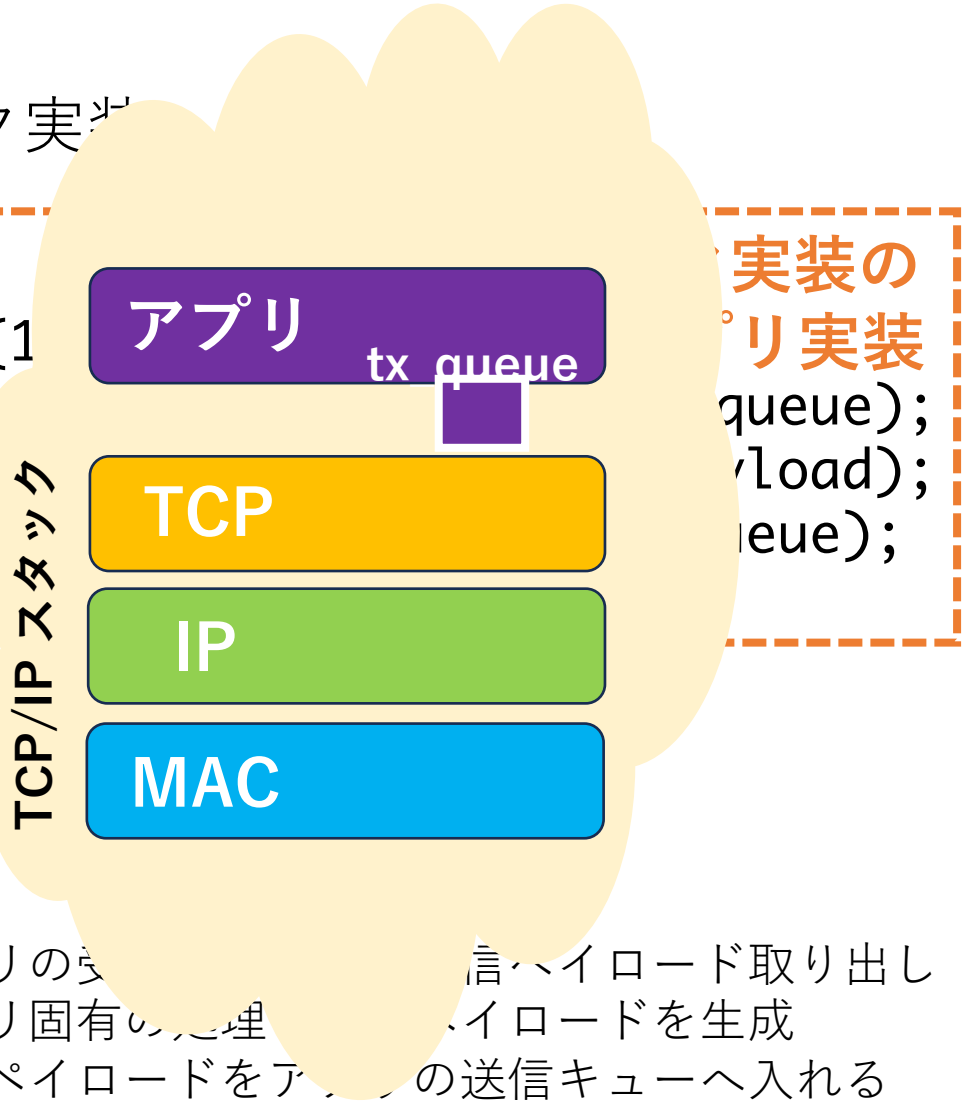
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

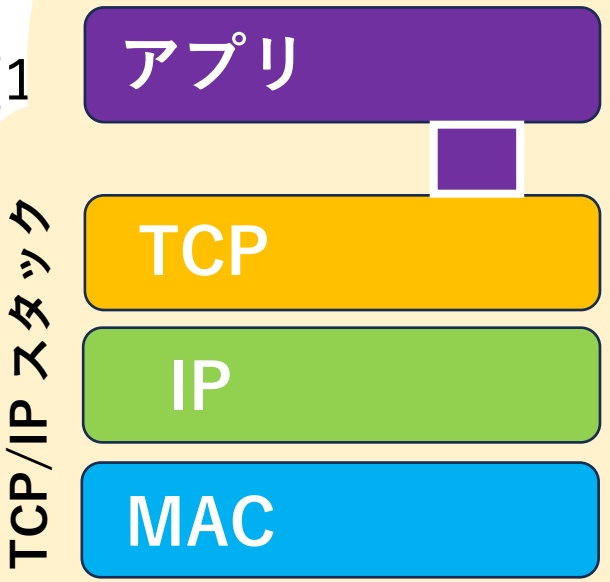
1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

実装の
プリ実装

```
while (1
1.rx_pa
2.tx_pc
3.enque
}
```

```
queue);
yload);
eue);
```



1. アプリの受　　信ペイロード取り出し
2. アプリ固有の処理　　イロードを生成
3. 送信ペイロードをア　　の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

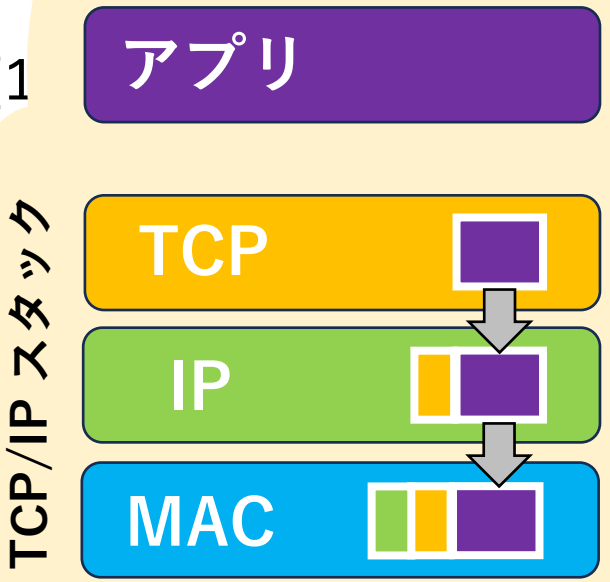
1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

実装の
プリ実装

```
while (1
1.rx_pa
2.tx_pc
3.enque
}
```

```
queue);
yload);
ueue);
```



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

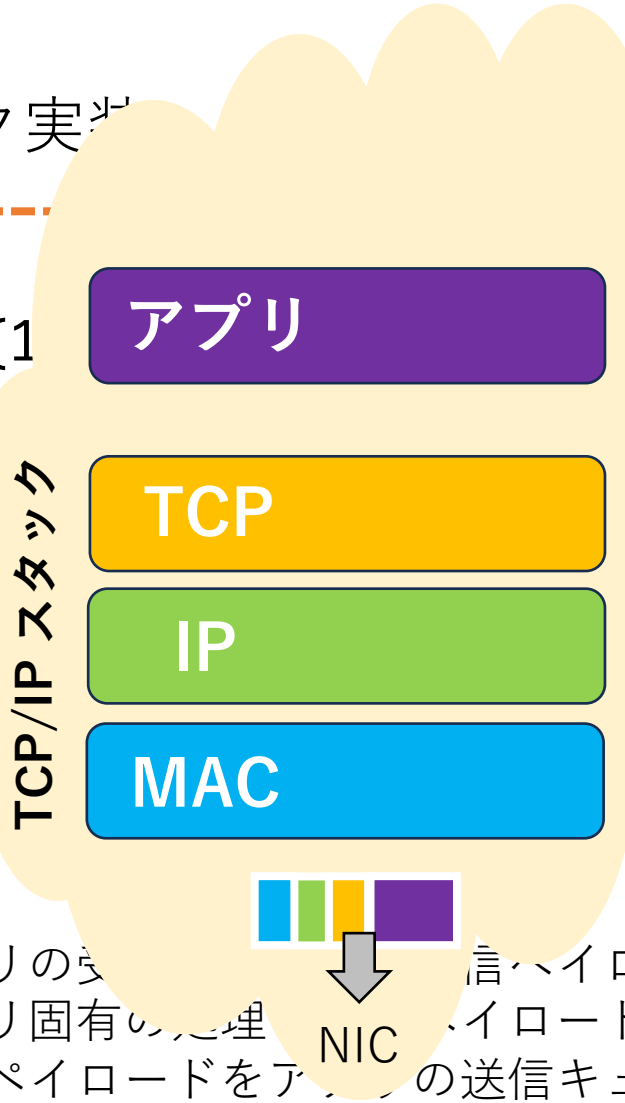
1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

実装の
プリ実装

```
while (1
1.rx_pa
2.tx_pc
3.enque
}
```

```
queue);
yload);
ueue);
```



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 ペイロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

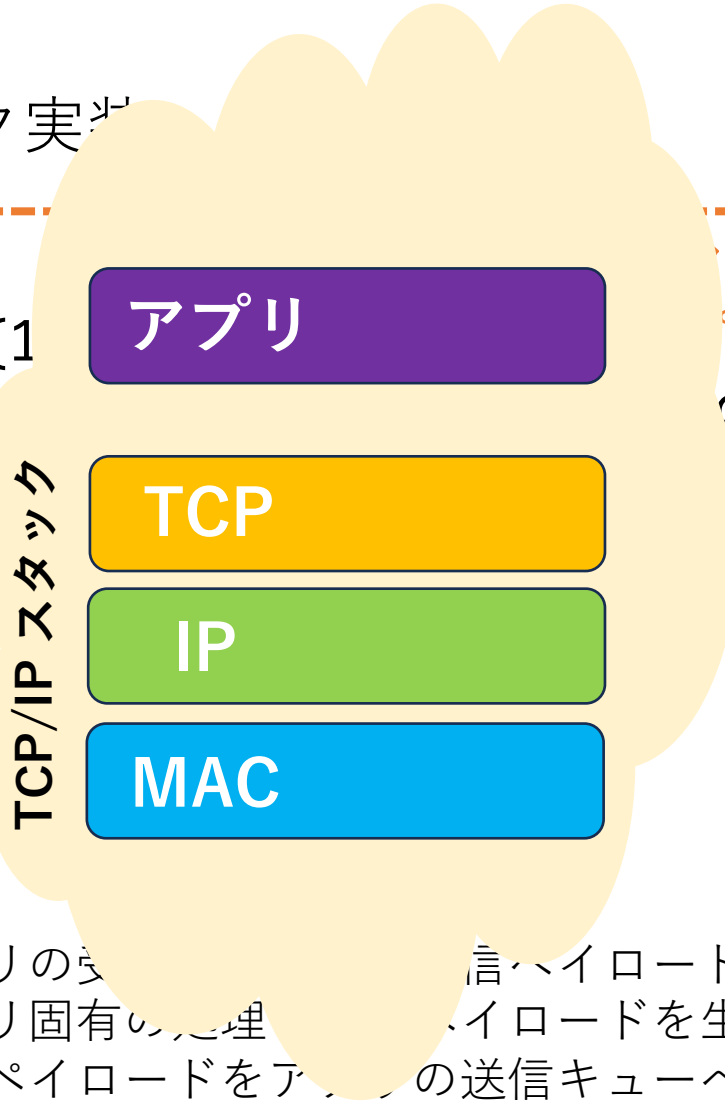
TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_rx(rx_buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_tx(tx_payload);
6. nic_tx(tx_pkt);
}
```

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

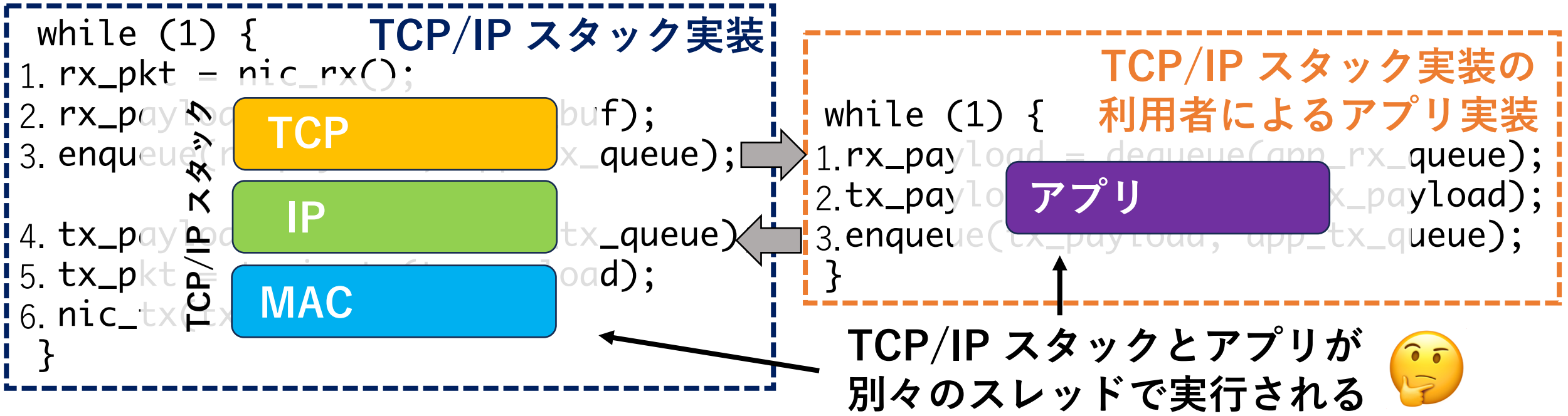
4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信



1. アプリの受 信ペイロード取り出し
2. アプリ固有の処理 イロードを生成
3. 送信ペイロードをア の送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装



1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

パターン２

それぞれ同じCPUコアで動かす

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

パターン２

それぞれ同じCPUコアで動かす

欠点

片方のスレッドが使っていない
CPU 時間を
もう片方が使うことができない

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

欠点

片方のスレッドが使っていない
CPU 時間を
もう片方が使うことができない

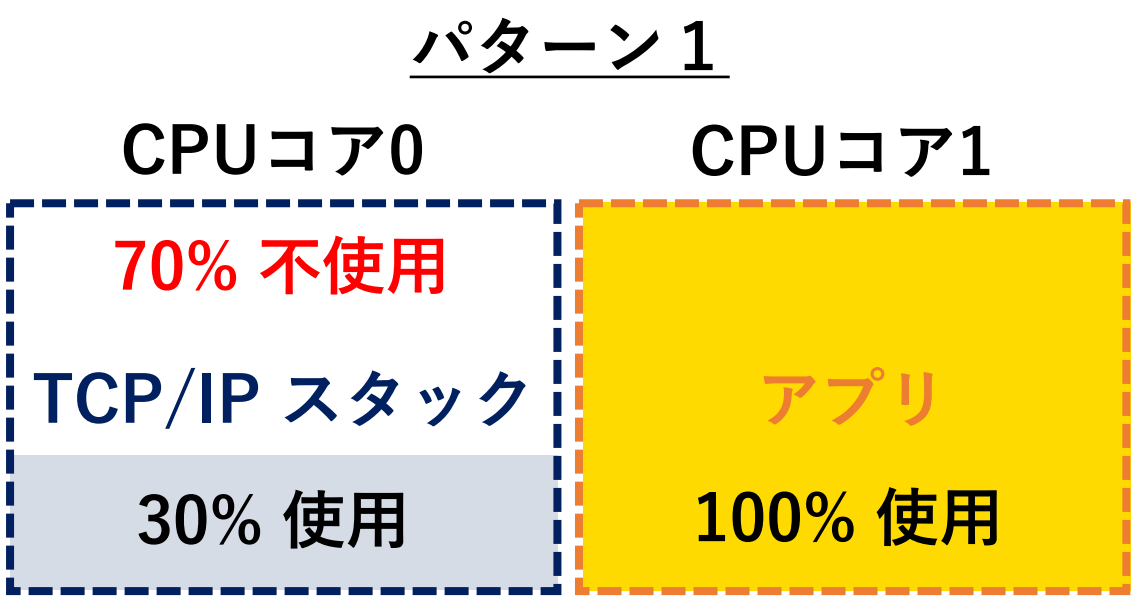
パターン２

それぞれ同じCPUコアで動かす

スレッド実行形式の制約

CPUコア割り当て

パターン１

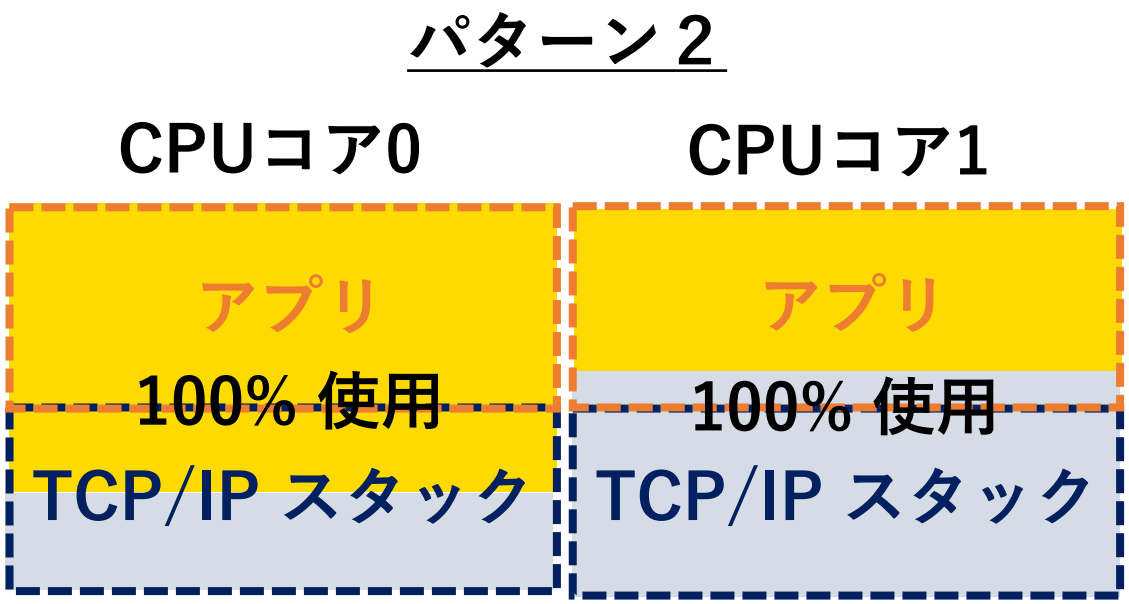


それぞれ別のCPUコアで動かす

欠点

片方のスレッドが使っていない
CPU 時間を
もう片方が使うことができない

パターン２



それぞれ同じCPUコアで動かす

利点

片方のスレッドが使っていない
CPU 時間を
もう片方が使うことができる

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

パターン２

それぞれ同じCPUコアで動かす

欠点

実行コンテキスト切り替えに
カーネルによる
スレッドスケジューリングが必要

スレッド実行形式の制約

CPUコア割り当て

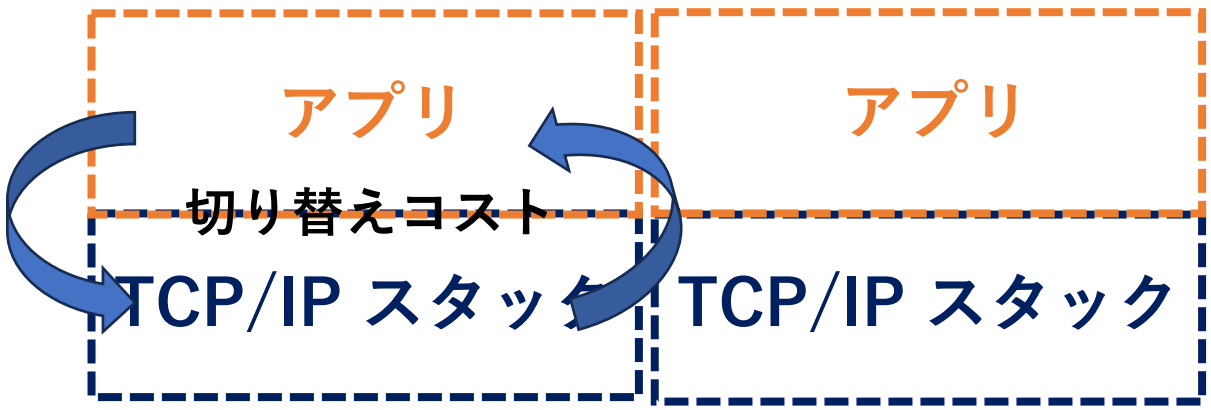
パターン１

CPUコア0　　**CPUコア1**

それぞれ別のCPUコアで動かす

パターン２

CPUコア0　　**CPUコア1**



それぞれ同じCPUコアで動かす

欠点

実行コンテキスト切り替えに
カーネルによる
スレッドスケジューリングが必要

スレッド実行形式の制約

CPUコア割り当て

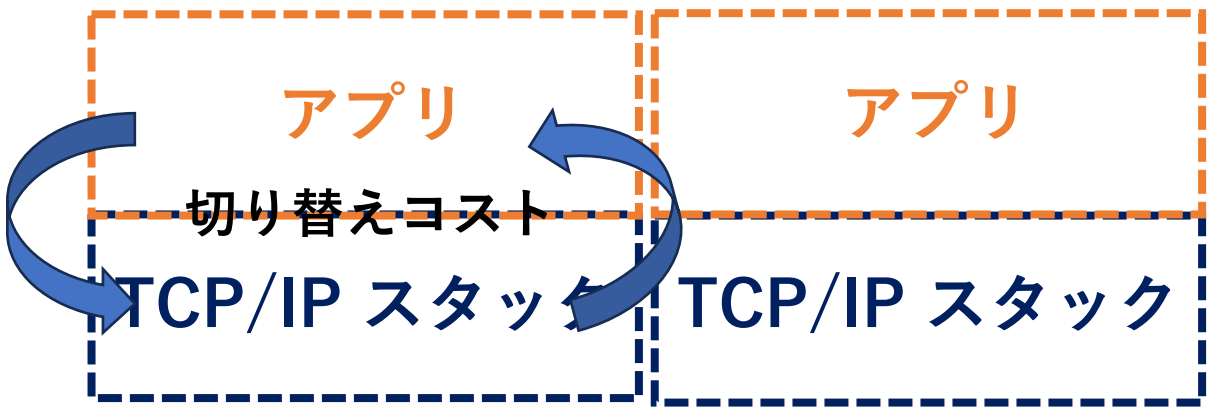
パターン１

それぞれ別のCPUコアで動かす

利点

実行コンテキストの
切り替えが不要

パターン２



それぞれ同じCPUコアで動かす

欠点

実行コンテキスト切り替えに
カーネルによる
スレッドスケジューリングが必要

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

パターン２

それぞれ同じCPUコアで動かす

どちらも欠点がある

スレッド実行形式の制約

CPUコア割り当て

パターン１

それぞれ別のCPUコアで動かす

パターン２

それぞれ同じCPUコアで動かす

どちらも欠点がある

一方、これらの欠点のない方法もある

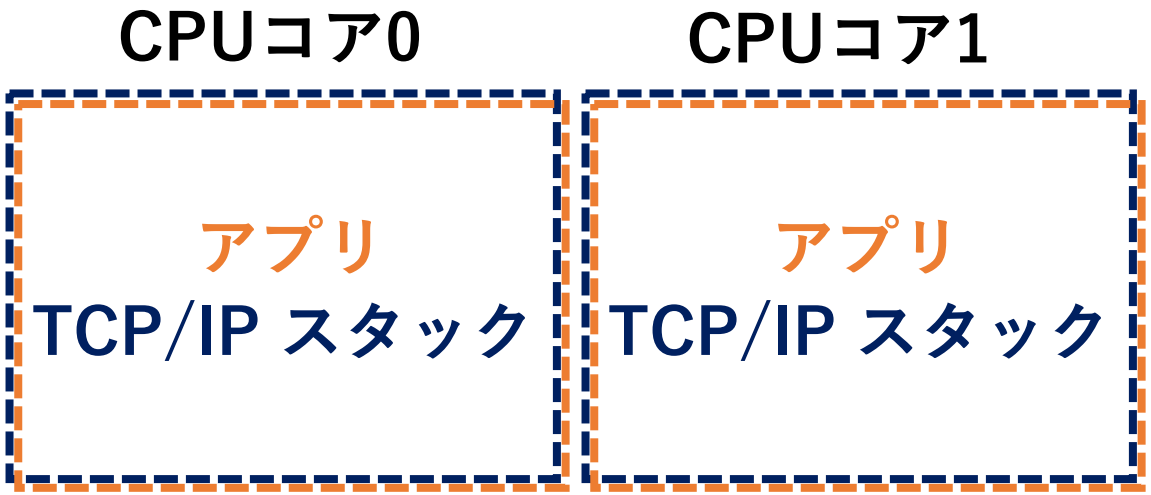
スレッド実行形式の制約

先ほどの欠点のない方法

余っていても使えない
CPU 時間が発生しない

スレッド切り替えの
コストが不要

CPUコア割り当て
パターン３

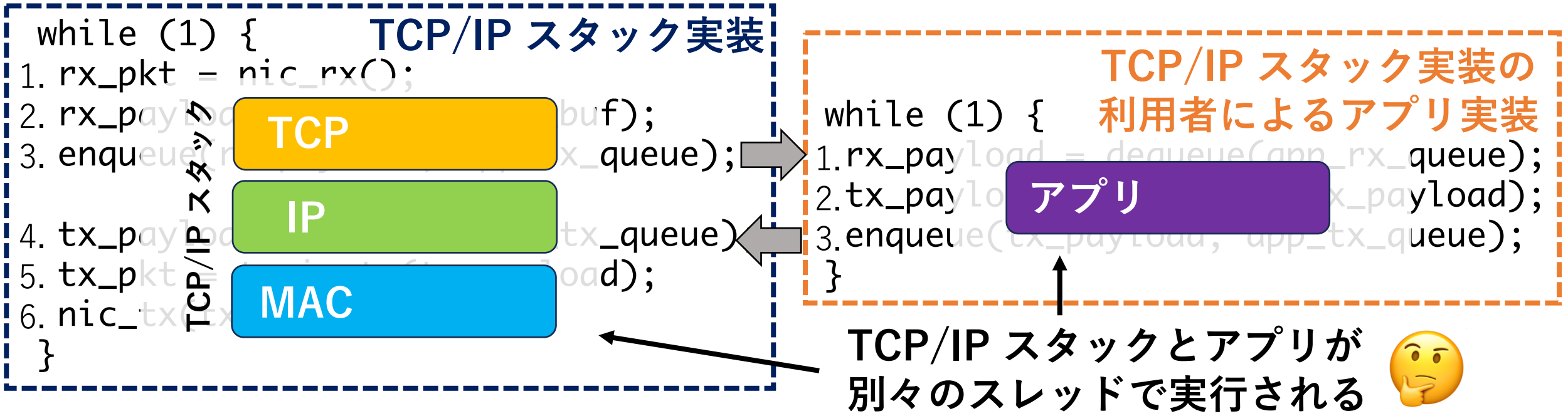


アプリと TCP/IP スタックを
同じスレッドで動かす

本来、TCP/IP スタック実装はこの形式をサポートすべき

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装



1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

```
while (1) {
1. rx_pkt = nic_rx();
2. rx_payload = tcpip_input(rx_pkt, buf);
3. enqueue(rx_payload, app_rx_queue);

4. tx_payload = dequeue(app_tx_queue);
5. tx_pkt = tcpip_output(tx_payload);
6. nic_tx(tx_pkt);
}
```

TCP/IP スタック: TCP / IP / MAC

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1. rx_payload = dequeue(app_rx_queue);
2. tx_payload = app_proc(rx_payload);
3. enqueue(tx_payload, app_tx_queue);
}
```

アプリ

TCP/IP スタックとアプリが別々のスレッドで実行される 🤔

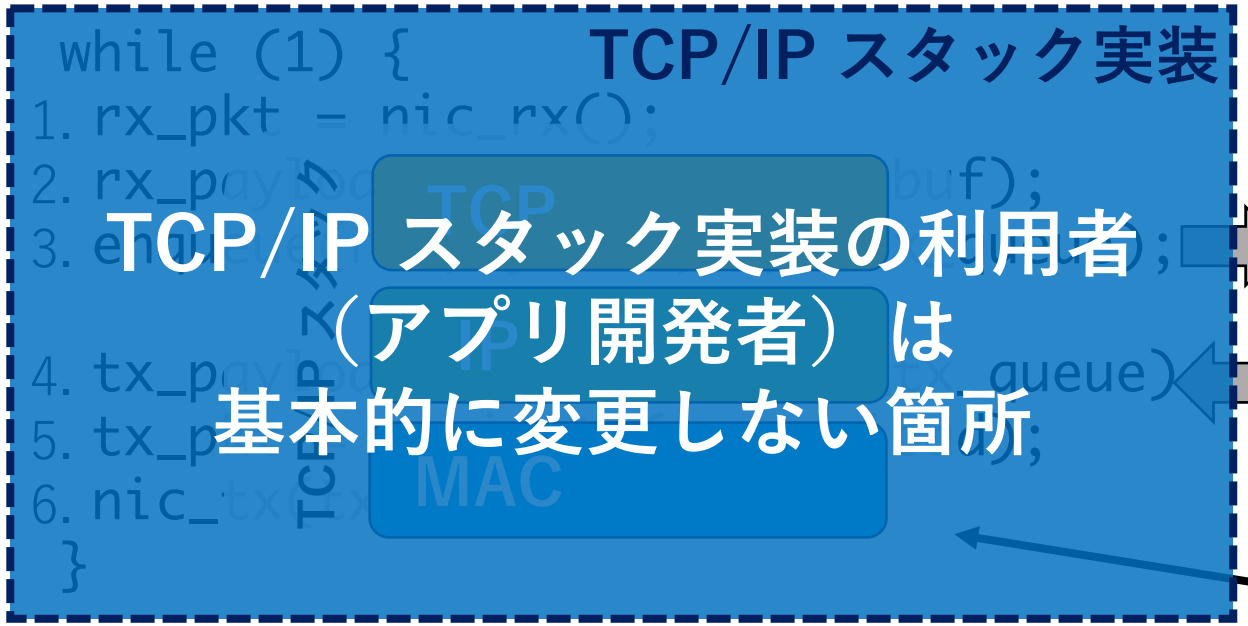
1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装



TCP/IP スタックとアプリが
別々のスレッドで実行される 🤔

1. NIC から受信したパケットを取り出し
2. TCP/IP スタックが受信処理
3. 受信ペイロードをアプリの受信キューへ入れる

4. アプリの送信キューから送信ペイロードを取り出し
5. TCP/IP スタックが送信処理
6. NIC からパケットを送信

1. アプリの受信キューから受信ペイロード取り出し
2. アプリ固有の処理で送信ペイロードを生成
3. 送信ペイロードをアプリの送信キューへ入れる

スレッド実行形式の制約

比較的一般的な性能に最適化された TCP/IP スタック実装

TCP/IP スタック実装

TCP/IP スタック実装の利用者（アプリ開発者）は基本的に変更しない箇所

TCP/IP スタック実装の利用者によるアプリ実装

```
while (1) {
1.rx_payload = dequeue(app_rx_queue);
2.tx_payload = アプリ                yload);
3.enqueue(tx_payload, app_tx_queue);
}
```

TCP/IP スタックとアプリが別々のスレッドで実行される 🤔

既存の性能へ最適化された実装の多くは
アプリと TCP/IP 処理が別スレッドになることを想定しており
効率の良くないコア割り当てパターンを利用者に強制する

組み込みやすさと性能についての要件

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. **NIC のオフロード機能を利用できる**
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - Checksum Offload
  - TCP Segmentation Offload (TSO)

プロトコル処理に必要な
- CPU での計算量
- CPU からのメモリアクセス
を削減できる

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

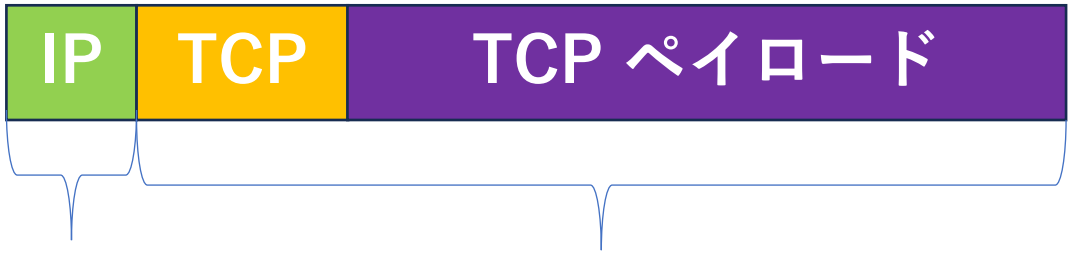
- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - **Checksum Offload**
  - TCP Segmentation Offload (TSO)

プロトコル処理に必要な
- CPU での計算量
- CPU からのメモリアクセス
を削減できる

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - **Checksum Offload**
  - TCP Segmentation Offload (TSO)



NIC がチェックサム計算をしてくれる

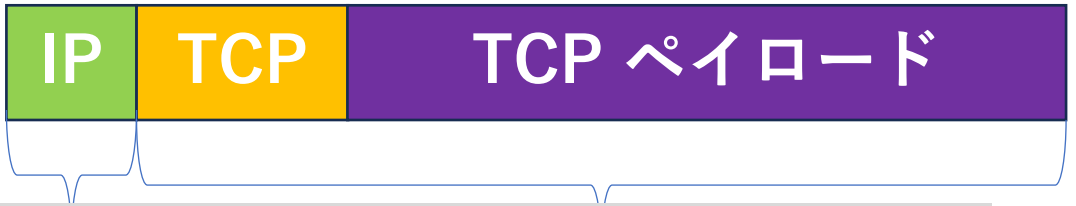
プロトコル処理に必要な
- CPU での計算量
- CPU からのメモリアクセス
を削減できる

ペイロードが大きいほどチェックサム計算に必要な
- CPU での計算量と
- メモリアクセスのバイト数も多くなるため、
それを NIC に肩代わりしてもらえるのはありがたい

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - **Checksum Offload**
  - TCP Segmentation Offload (TSO)



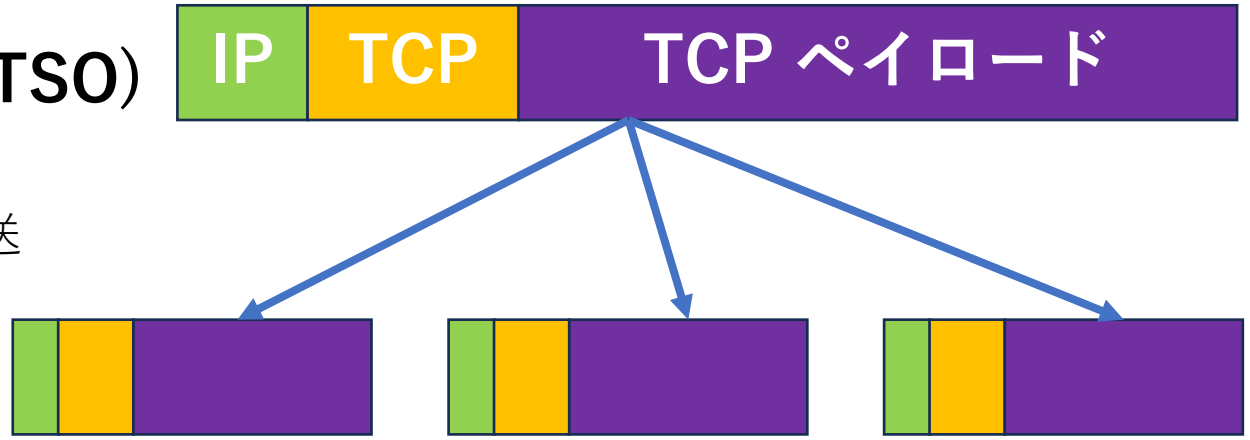
例えば、ペイロードが 64 KB あれば、
CPU は 64 KB 分データを**メモリからスキャン**してチェックサム**計算**が必要
一方、
NIC のオフロード機能を使えば、
メモリからのデータのスキャンと
チェックサム計算のための CPU サイクルが不要

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - Checksum Offload
  - **TCP Segmentation Offload (TSO)**

MTU サイズを超えるパケットを
NIC が分割＋ヘッダを付与して転送



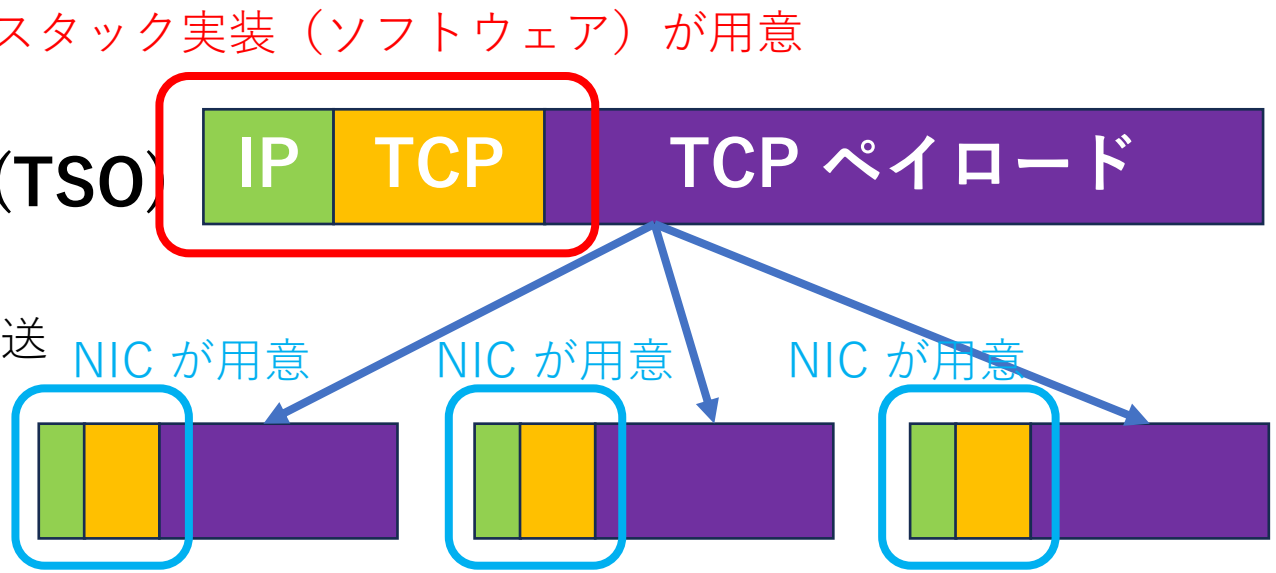
プロトコル処理に必要な
- CPU での計算量
- CPU からのメモリアクセス
を削減できる

CPU が処理するパケットの個数が減るため
CPU での計算量を削減できる

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - Checksum Offload
  - **TCP Segmentation Offload (TSO)**



MTU サイズを超えるパケットを
NIC が分割＋ヘッダを付与して転送

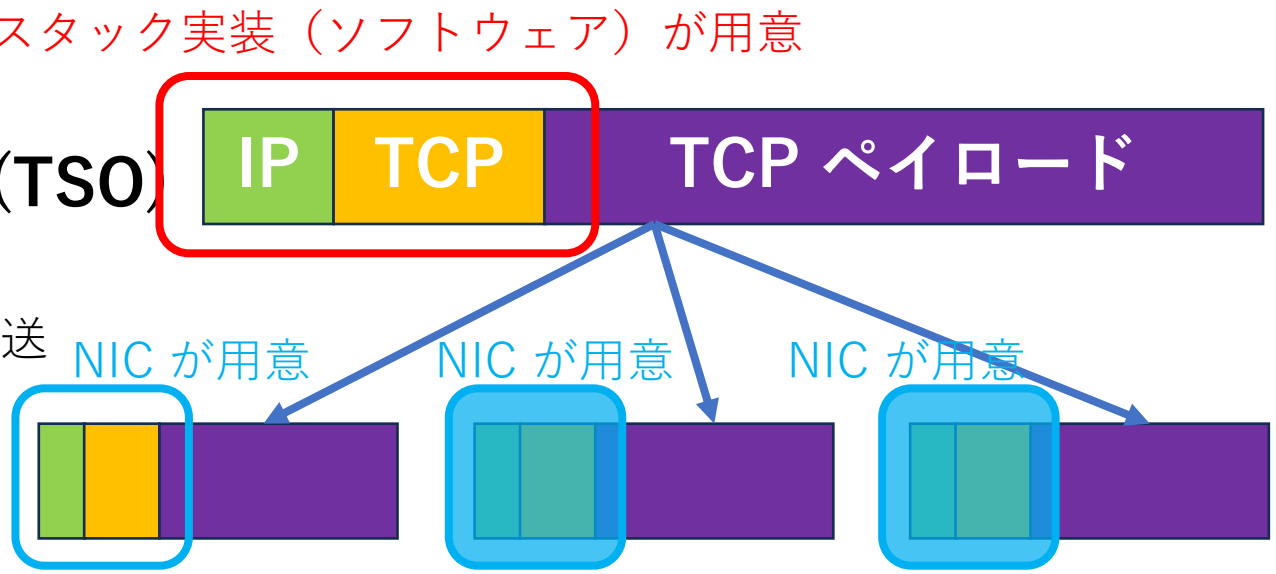
プロトコル処理に必要な
CPU での計算量

例えば、TCP/IP スタック実装が4500 バイトのパケットを一つ用意して
そのパケットをTSO 機能 が 1500 バイトのパケット 3 つに分けると
TCP/IP スタック実装はパケット 2 つ分のヘッダの用意に必要な CPU での計算減らせる

特に、大きなデータの移動（e.g., ファイル転送）で重要

NIC のオフロード機能が重要な理由

- 最近の高性能な NIC で一般的にサポートされているオフロード機能がいくつかある
  - Checksum Offload
  - **TCP Segmentation Offload (TSO)**



MTU サイズを超えるパケットを
NIC が分割＋ヘッダを付与して転送

プロトコル処理に必要な
CPU での計算量

例えば、TCP/IP スタック実装が4500 バイトのパケットを一つ用意して
そのパケットをTSO 機能 が 1500 バイトのパケット 3 つに分けると
TCP/IP スタック実装は**パケット 2 つ分のヘッダの用意に必要な CPU での計算減らせる**

特に、大きなデータの移動（e.g., ファイル転送）で重要

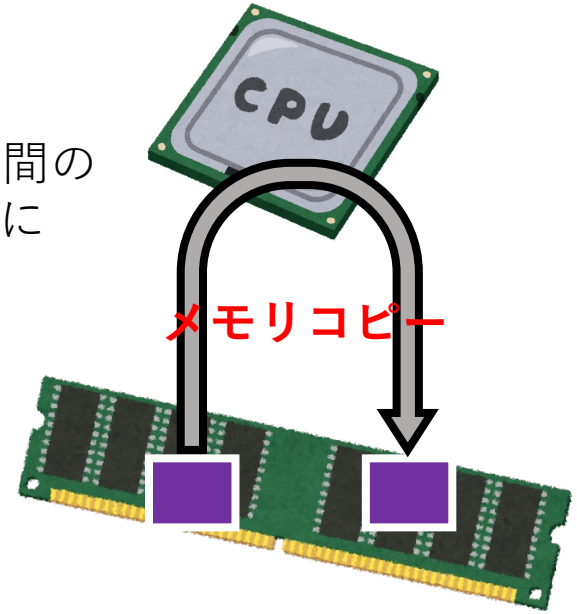
組み込みやすさと性能についての要件

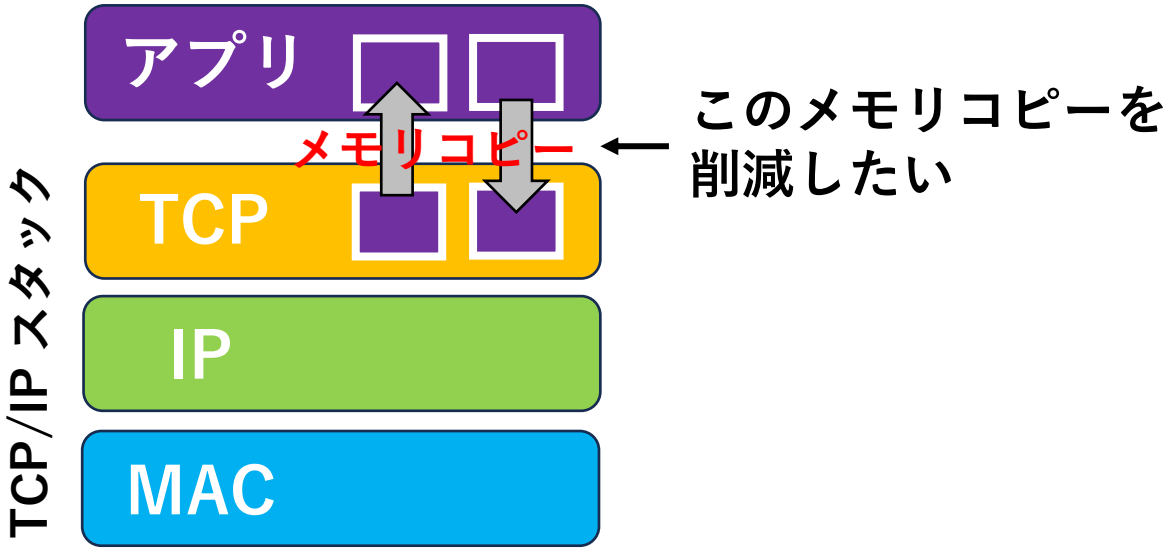
1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. **メモリコピー回数を抑えられる**
5. マルチコア環境でのスケーラビリティ

メモリコピー削減が重要な理由

- メモリコピーは時間がかかる処理　（特にデータがキャッシュに乗らない場合）
- TCP/IP スタックとアプリ間でメモリコピーを要する実装も多い（ポータブルな実装は概ね該当）

CPU・メモリ間のデータの移動に時間がかかる





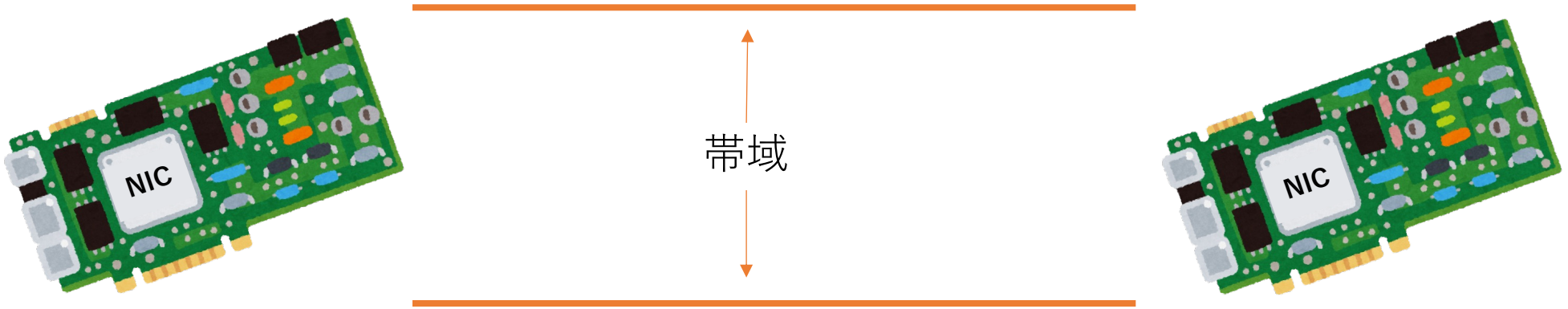
特に、大きなデータの移動（e.g., ファイル転送）で重要

組み込みやすさと性能についての要件

- CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
- 色々なスレッド実行形式に対応可能
- NIC のオフロード機能を利用できる
- メモリコピー回数を抑えられる
- **マルチコア環境でのスケーラビリティ**

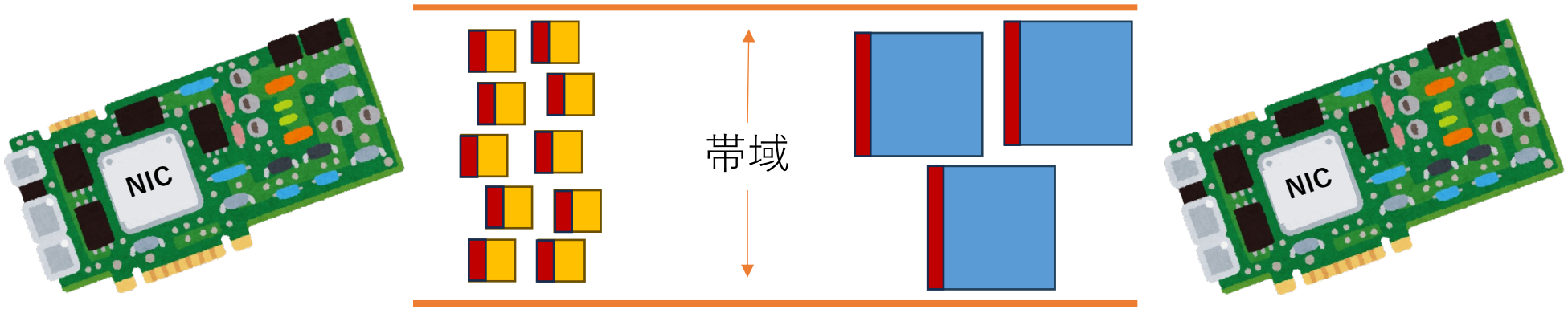
複数 CPU コアの利用が必要な理由

- パケットが小さいほど NIC は多くのパケットを送受信できる



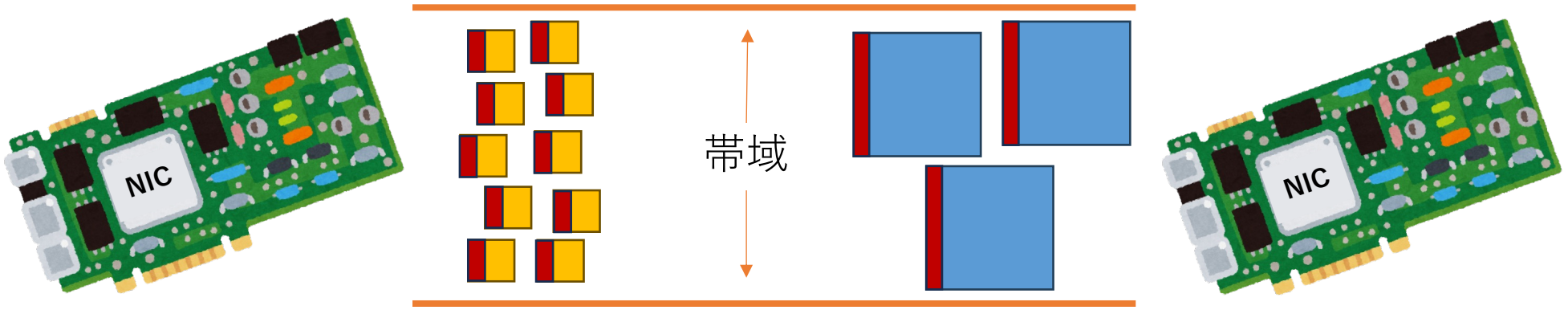
複数 CPU コアの利用が必要な理由

- パケットが小さいほど NIC は多くのパケットを送受信できる



複数 CPU コアの利用が必要な理由

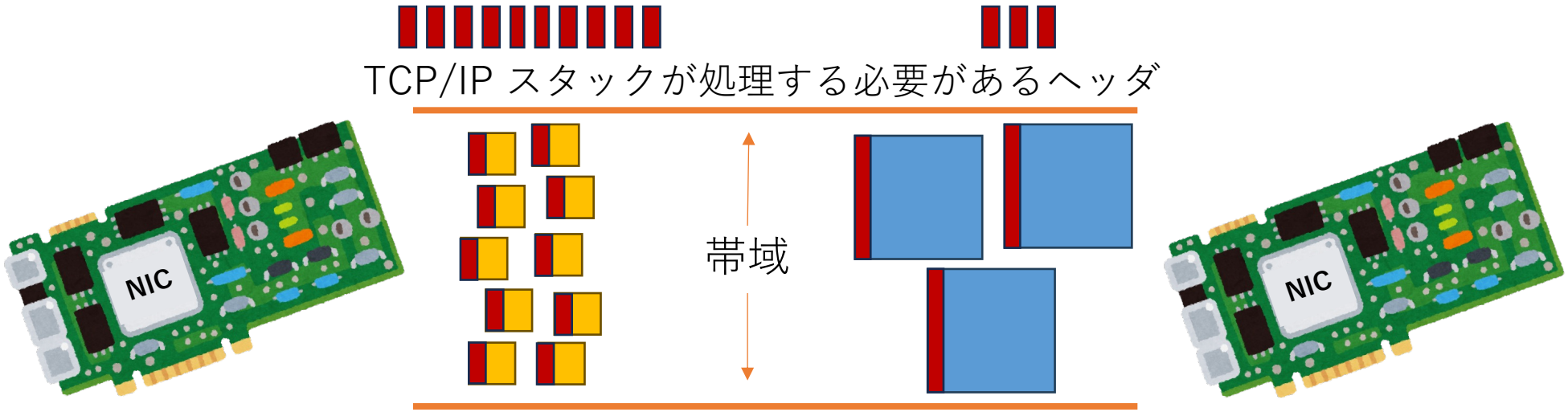
- パケットが小さいほど NIC は多くのパケットを送受信できる
- パケットが小さいワークロードほど TCP/IP スタックが処理しなければならないヘッダの数が増える



複数 CPU コアの利用が必要な理由

- パケットが小さいほど NIC は多くのパケットを送受信できる
- パケットが小さいワークロードほど TCP/IP スタックが処理しなければならないヘッダの数が増える

CPU での計算量が増える



複数 CPU コアの利用が必要な理由

- パケットが小さいほど NIC は多くのパケットを送受信できる
- パケットが小さいワークロードほど TCP/IP スタックが処理しなければならないヘッダの数が増える

CPU での計算量が増える

複数 CPU コアの利用が必要な理由

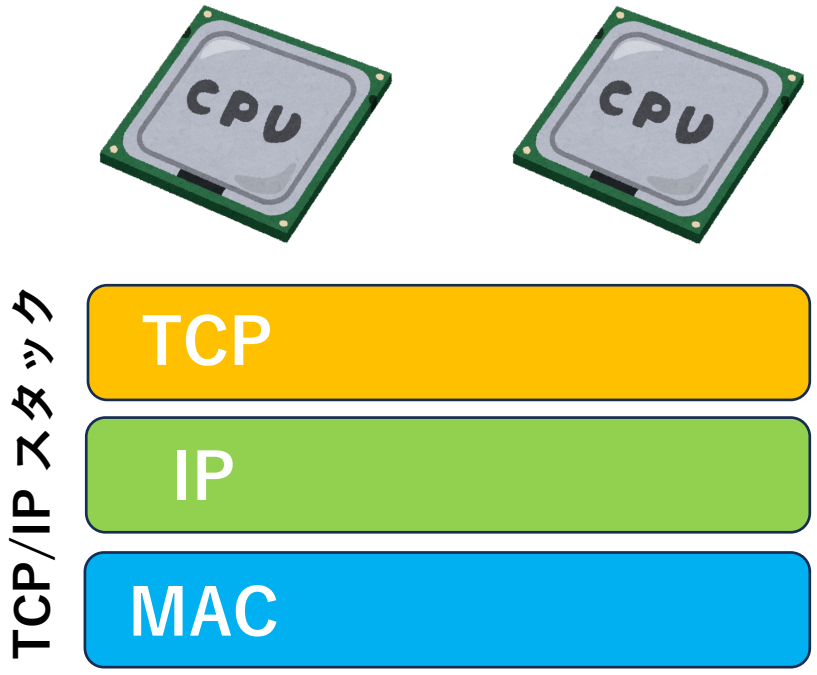
- パケットが小さいほど NIC は多くのパケットを送受信できる
- パケットが小さいワークロードほど TCP/IP スタックが処理しなければならないヘッダの数が増える

CPU での計算量が増える

特に、パケットサイズが小さい RPC のようなワークロードで重要

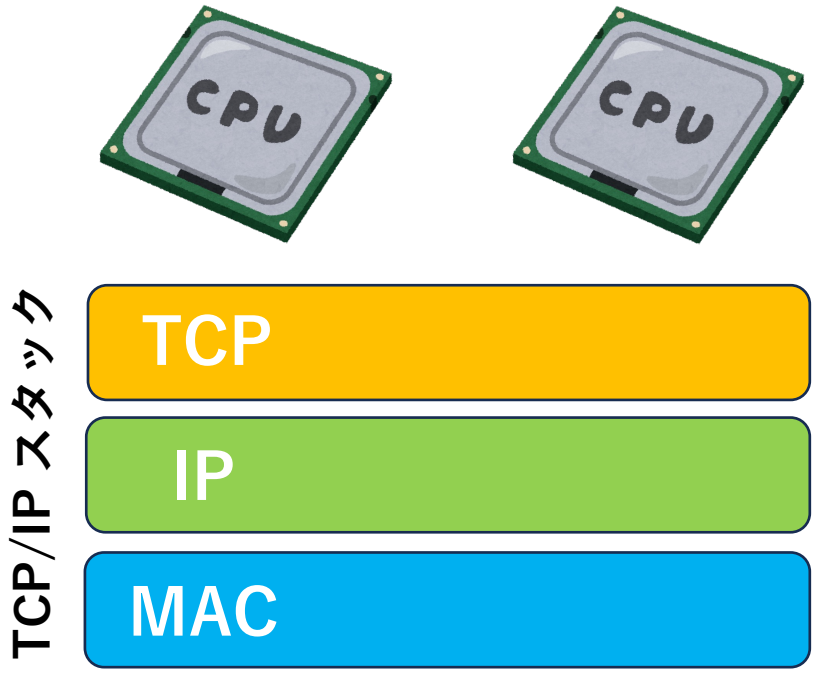
複数 CPU コア利用時に必要な配慮

- TCP/IP スタック実装は複数 CPU コアで処理が行えることを想定して実装されるべき



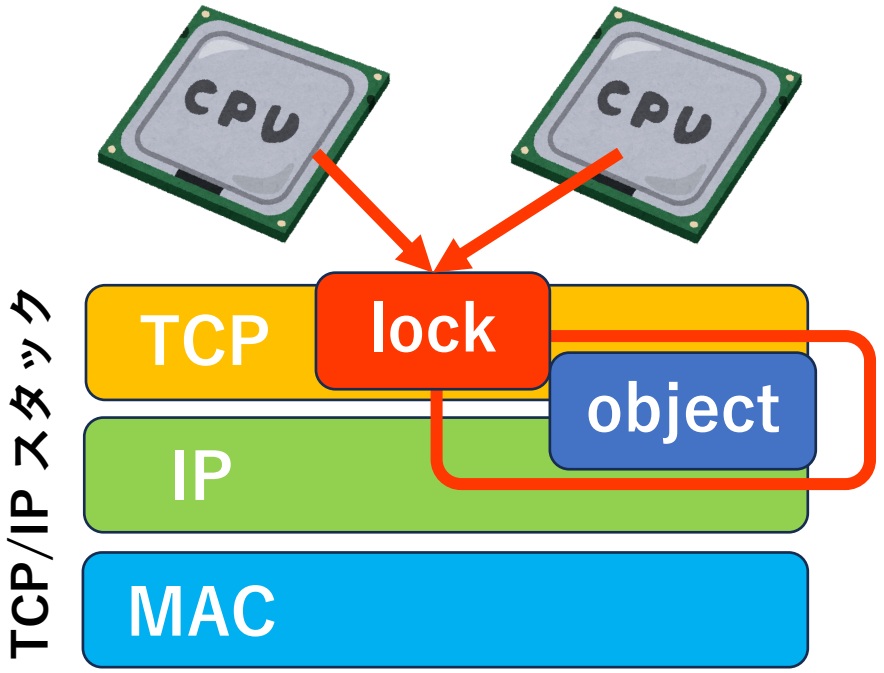
複数 CPU コア利用時に必要な配慮

- TCP/IP スタック実装は複数 CPU コアで処理が行えることを想定して実装されるべき
- CPU コアの増加に合わせて性能を高めるには CPU コア間で競合する箇所をなくすなどの配慮が必要



複数 CPU コア利用時に必要な配慮

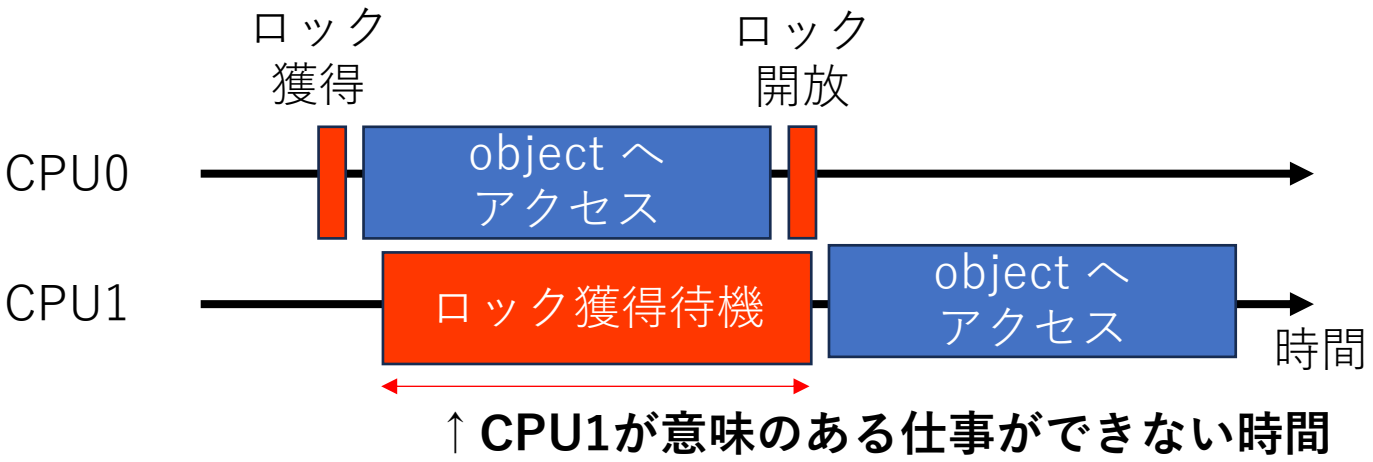
- TCP/IP スタック実装は複数 CPU コアで処理が行えることを想定して実装されるべき
- CPU コアの増加に合わせて性能を高めるには CPU コア間で競合する箇所をなくすなどの配慮が必要

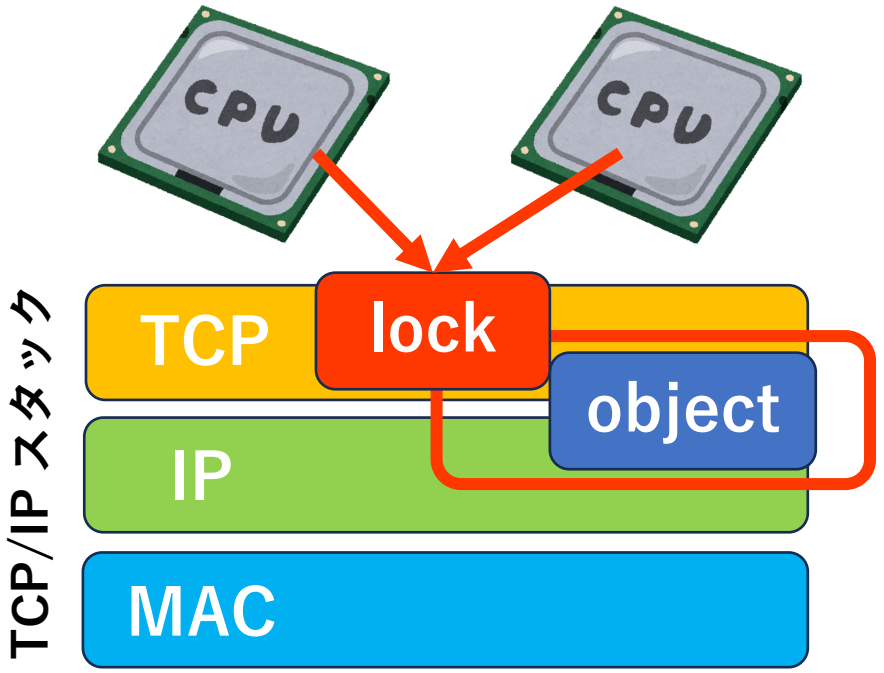


複数 CPU コア利用時に必要な配慮

- TCP/IP スタック実装は複数 CPU コアで処理が行えることを想定して実装されるべき

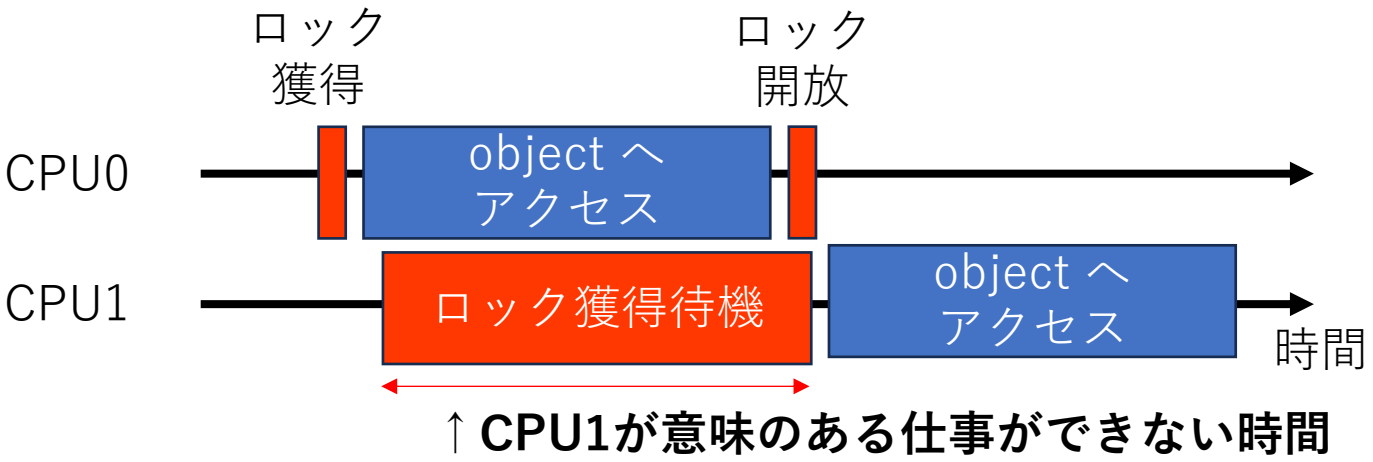
- CPU コアの増加に合わせて性能を高めるには CPU コア間で競合する箇所をなくすなどの配慮が必要

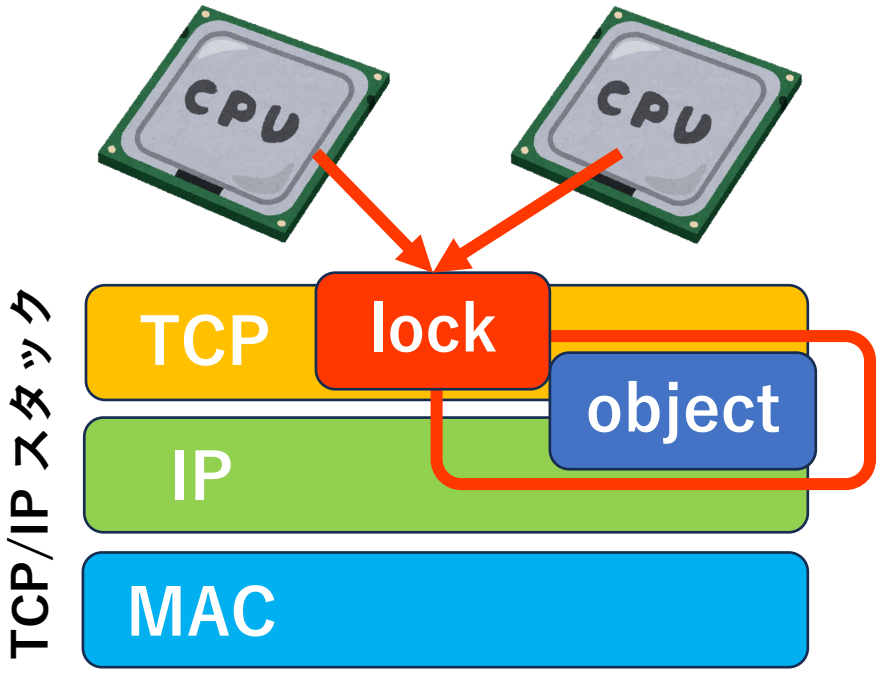




複数 CPU コア利用時に必要な配慮

- TCP/IP スタック実装は複数 CPU コアで処理が行えることを想定して実装されるべき
- CPU コアの増加に合わせて性能を高めるには CPU コア間で競合する箇所をなくすなどの配慮が必要





既存のポータブルな実装の多くはこの点についての配慮が不足

設計

- ５つの要件を満たす実装の模索

提案する TCP/IP 実装

1. **CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い**
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

他要素への依存度を抑える

- 依存関係を最低限にするために、
  - CPU：アセンブリ命令を**直接**利用しない
  - NIC：NIC 固有機能を**直接**利用しない
  - OS：システムコールを**直接**利用しない
  - ライブラリ：外部のライブラリを**直接**利用しない
  - コンパイラ：コンパイラ固有機能を**直接**利用しない

- 環境依存要因になる部分は TCP/IP スタック実装利用者が実装するようにする

他要素への依存度を抑える

- TCP/IP スタック実装利用者が実装する機能の例
  - パケットバッファ含むメモリ確保・開放
    - アトミック操作を含むため CPU 命令依存
    - NUMA への配慮も環境+ライブラリ依存
  - NIC の I/O
    - I/O を担当する実装はOS、ライブラリ、利用者の選択に依存
      - e.g., DPDK, netmap, tap device
  - その他、現在時刻取得、プログラムを実行している CPU コア取得等
    - CPU 命令、OS、ライブラリ依存

パケットの表現

- TCP/IP スタック実装内でパケットを表現するデータ構造を独自に定義・保持（Linux の sk_buff のようなもの）
  - DPDK 等は独自にパケットを表現する構造体 (e.g., rte_mbuf) 等を定義しているが、それを直接利用すると依存関係ができてしまうため
  - TCP/IP スタック実装内部では、それら rte_mbuf 等を void のポインタに紐づけて取り扱う

```
struct netstack_packet {
  void *packet;
  struct netstack_packet *prev, *next;
  // ...
};
```

DPDK であれば rte_mbuf のアドレスを代入

TCP/IP スタック実装内で連結リストを構成するためのポインタ

提案する TCP/IP 実装

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. **色々なスレッド実行形式に対応可能**
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

スレッド実行形式の自由度を担保

TCP/IP スタック実装

```
while (1) {
  rx_pkt = nic_rx();
  rx_payload = tcpip_rx(rx_buf);
  enqueue(rx_payload, app_rx_queue);

  tx_payload = dequeue(app_tx_queue);
  tx_pkt = tcpip_tx(tx_payload);
  nic_tx(tx_pkt);
}
```

TCP/IP スタック実装の
利用者によるアプリ実装

```
while (1) {
  rx_payload = dequeue(app_rx_queue);
  tx_payload = app_logic(rx_payload);
  enqueue(tx_payload, app_tx_queue);
}
```

スレッド実行形式の自由度を担保

TCP/IP スタック実装

```
while (1) {
    rx_pkt = nic_rx();
    rx_payload = tcpip_rx(rx_buf);

    tx_payload = app_logic(rx_payload);

    tx_pkt = tcpip_tx(tx_payload);
    nic_tx(tx_pkt);
}
```

TCP/IP スタック実装の利用者によるアプリ実装

スレッド実行形式の自由度を担保

提案の実装

TCP/IP スタック実装が提供

TCP/IP スタック実装の利用者（アプリ開発者）がループを実装

```
while (1) {
  rx_pkt = nic_rx();
  rx_payload = tcpip_rx(rx_buf);

  tx_payload = app_logic(rx_payload);

  tx_pkt = tcpip_tx(tx_payload);
  nic_tx(tx_pkt);
}
```

OS・ライブラリへの依存回避のため TCP/IP スタック実装利用者が実装

TCP/IP スタック実装自体は、
TCP/IP 処理を行うスレッド（while ループ）を実装しない

提案する TCP/IP 実装

1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. **NIC のオフロード機能を利用できる**
4. メモリコピー回数を抑えられる
5. マルチコア環境でのスケーラビリティ

NIC オフロード機能

- TCP/IP 実装利用者が実装する機能の一部として、
  - NIC オフロード機能のサポートの有無確認と
  - サポートされている場合にそれを適用する機能を含める

- TCP/IP スタック実装内での利用例

```
if (is_nic_tx_tcp_checksum_offload_available()) {
  nic_tx_tcp_checksum(packet);
} else {
  // software checksum implementation
}
```

NIC オフロード機能

- TCP/IP 実装利用者が実装する機能の一部として、
  - **NIC オフロード機能のサポートの有無確認**と
  - サポートされている場合にそれを適用する機能を含める

- TCP/IP スタック実装内での利用例

```
if (is_nic_tx_tcp_checksum_offload_available()) {
  nic_tx_tcp_checksum(packet);
} else {
  // software checksum implementation
}
```

NIC オフロード機能

- TCP/IP 実装利用者が実装する機能の一部として、
  - NIC オフロード機能のサポートの有無確認と
  - サポートされている場合に**それを適用する機能**を含める

- TCP/IP スタック実装内での利用例

```
if (is_nic_tx_tcp_checksum_offload_available()) {
  nic_tx_tcp_checksum(packet);
} else {
  // software checksum implementation
}
```

提案する TCP/IP 実装

- CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
- 色々なスレッド実行形式に対応可能
- NIC のオフロード機能を利用できる
- **メモリコピー回数を抑えられる**
- マルチコア環境でのスケーラビリティ

ゼロコピー送信

NIC の Scatter Gather 機能を利用

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

TCP/IP スタック実装

ヘッダを用意

ゼロコピー送信

NIC の Scatter Gather 機能を利用

① **TCP/IP スタック実装利用者の実装**

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

パケットバッファを確保

パケットバッファ

TCP/IP スタック実装

ヘッダを用意

ゼロコピー送信

② **TCP/IP スタック実装利用者の実装**

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

TCP/IP スタック実装

ヘッダを用意

NIC の Scatter Gather 機能を利用

送信用ペイロードを配置

ゼロコピー送信

③ **TCP/IP スタック実装利用者の実装**

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

TCP/IP スタック実装

ヘッダを用意

NIC の Scatter Gather 機能を利用

TCP/IP スタック実装に対して
以下のペイロードを送信するよう
リクエスト

ゼロコピー送信

NIC の Scatter Gather 機能を利用

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

TCP/IP スタック実装

④ ヘッダを用意

ゼロコピー送信

NIC の Scatter Gather 機能を利用

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

パケットバッファ上にヘッダを用意

⑤ **TCP/IP スタック実装**

ヘッダを用意

ゼロコピー送信

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

TCP/IP スタック実装

⑥ ヘッダを用意

NIC の Scatter Gather 機能を利用

ヘッダとペイロードの接続を記録

ゼロコピー送信

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

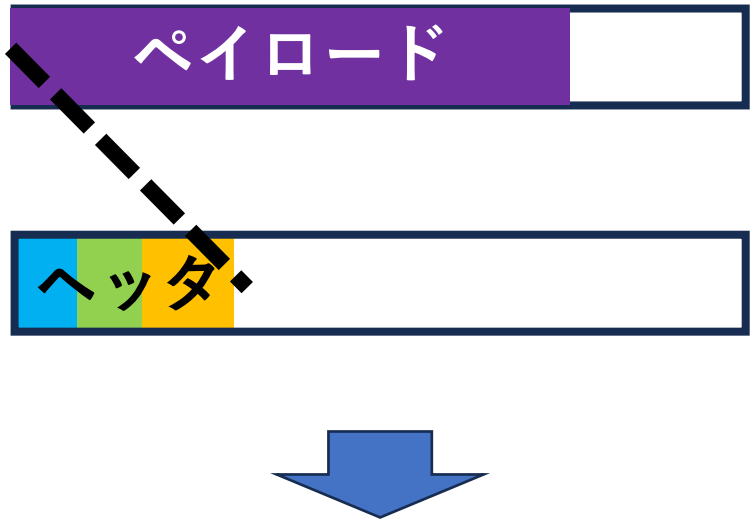
NIC へパケット送信をリクエスト

TCP/IP スタック実装

⑦ ヘッダを用意

NIC の Scatter Gather 機能を利用

NIC からパケットの転送をリクエスト



ゼロコピー送信

TCP/IP スタック実装利用者の実装

パケットバッファ確保実装

送信用ペイロードの用意

NIC の Scatter Gather 機能の適用

NIC へパケット送信をリクエスト

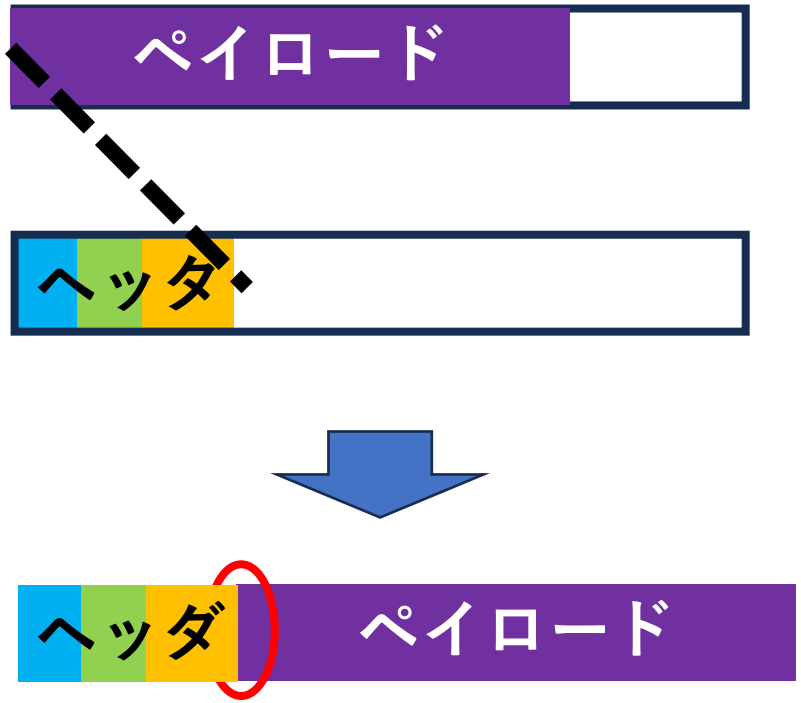
TCP/IP スタック実装

⑦ ヘッダを用意

NIC の Scatter Gather 機能によって
送信時にヘッダとペイロードが接続される

NIC の Scatter Gather 機能を利用

NIC からパケットの転送をリクエスト

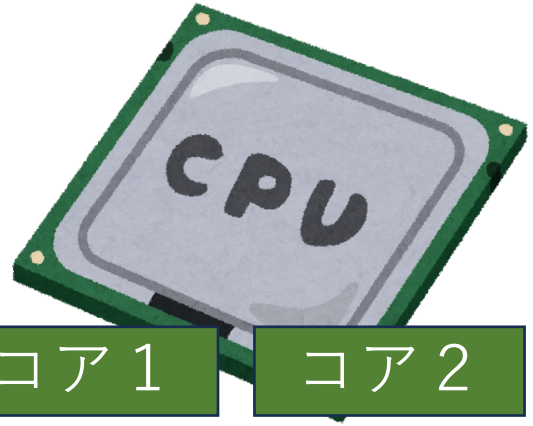


提案する TCP/IP 実装

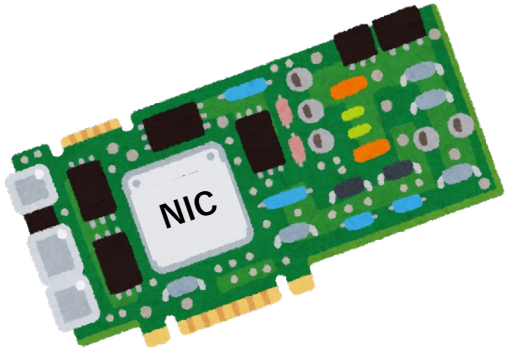
1. CPU、NIC、OS、ライブラリ、コンパイラへの依存度が低い
2. 色々なスレッド実行形式に対応可能
3. NIC のオフロード機能を利用できる
4. メモリコピー回数を抑えられる
5. **マルチコア環境でのスケーラビリティ**

マルチコア環境でのスケーラビリティ

- 方針１：CPU コア間で共有するオブジェクトをなくす

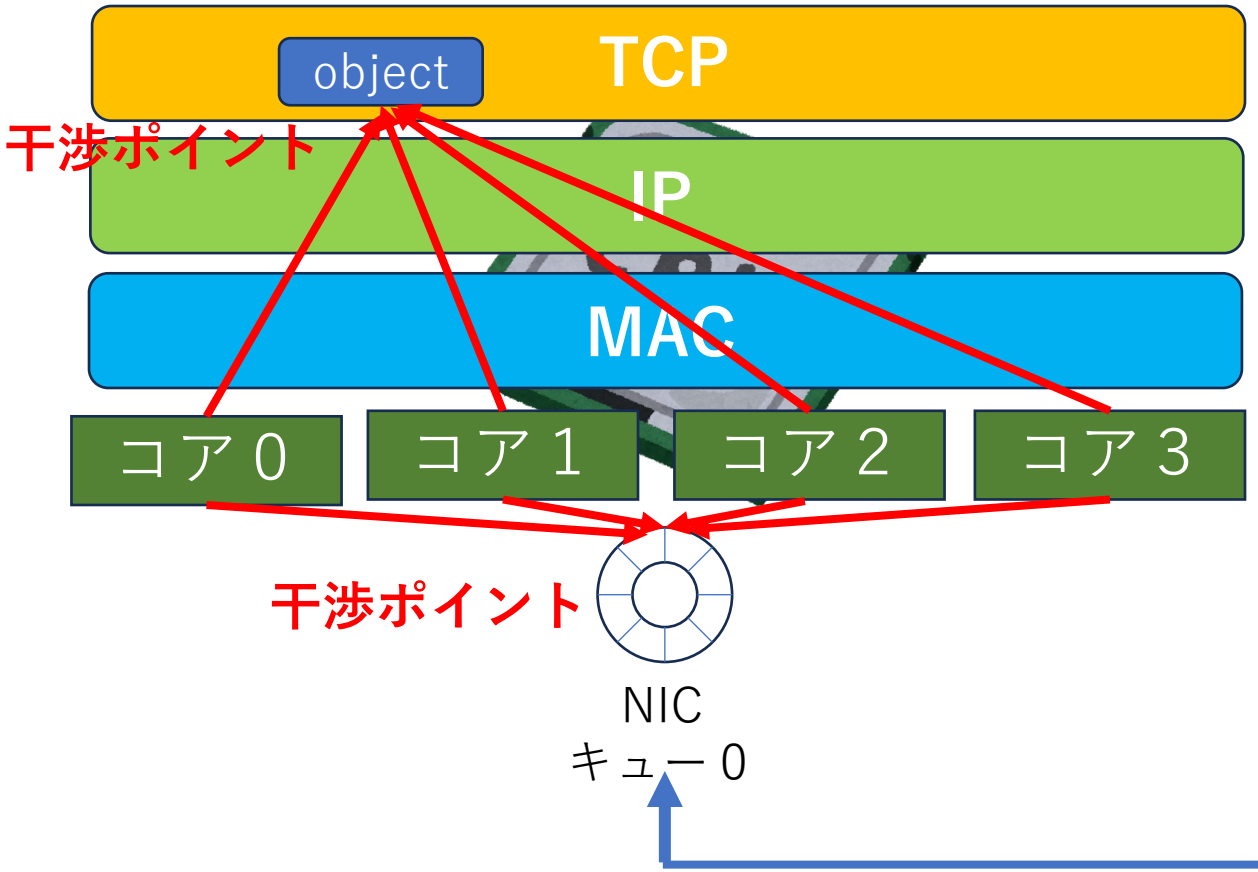


コア０ コア１ コア２ コア３

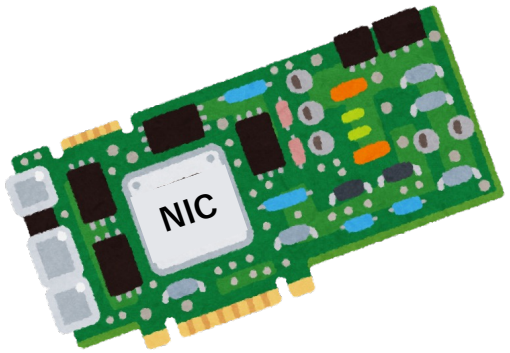


マルチコア環境でのスケーラビリティ

- 方針１：CPU コア間で共有するオブジェクトをなくす



特に配慮しない場合

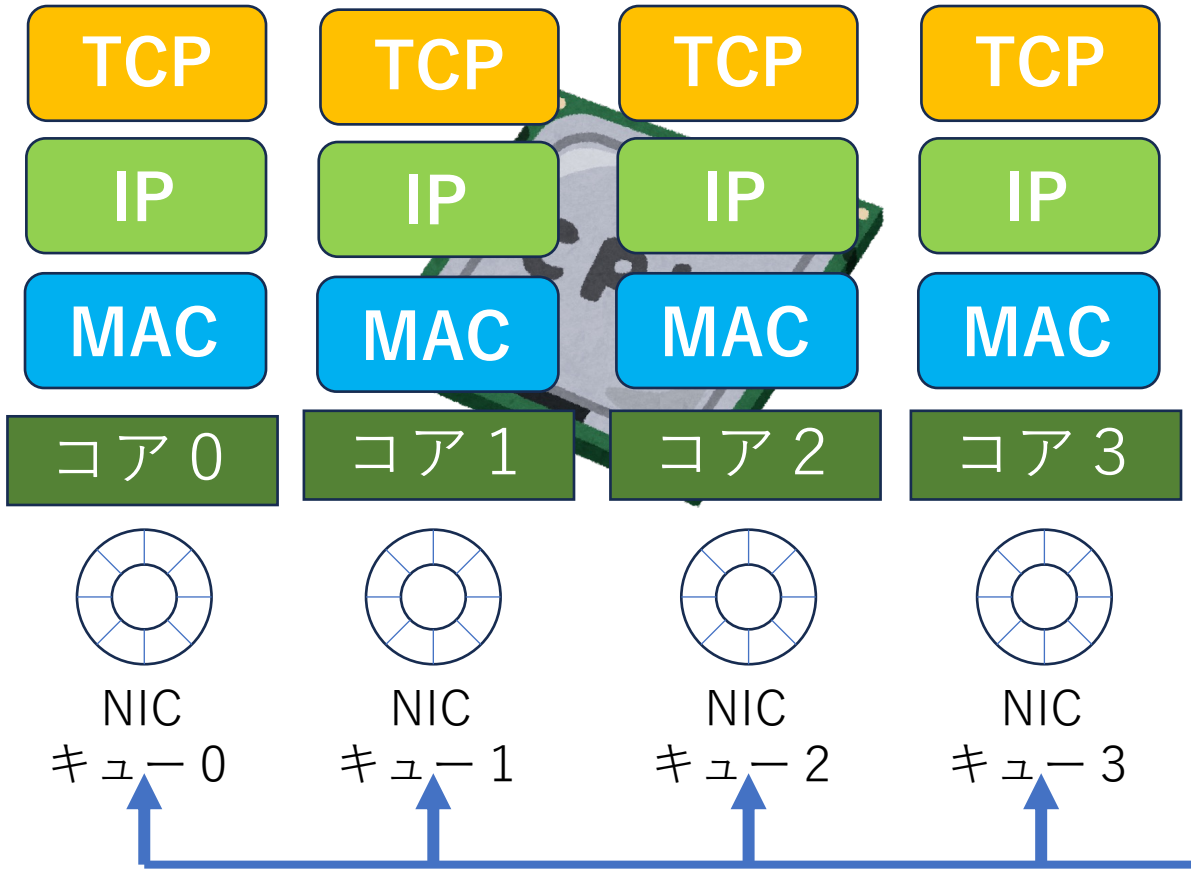


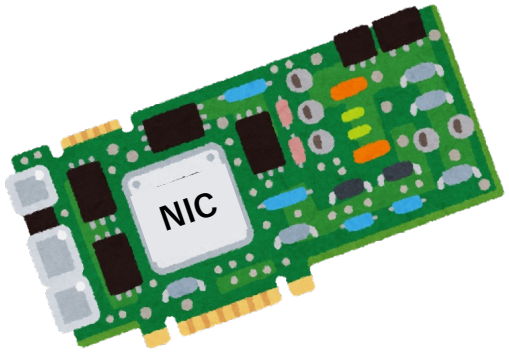
マルチコア環境でのスケーラビリティ

- 方針１：CPU コア間で共有するオブジェクトをなくす

e.g., mTCP (NSDI'14)

今回の実装



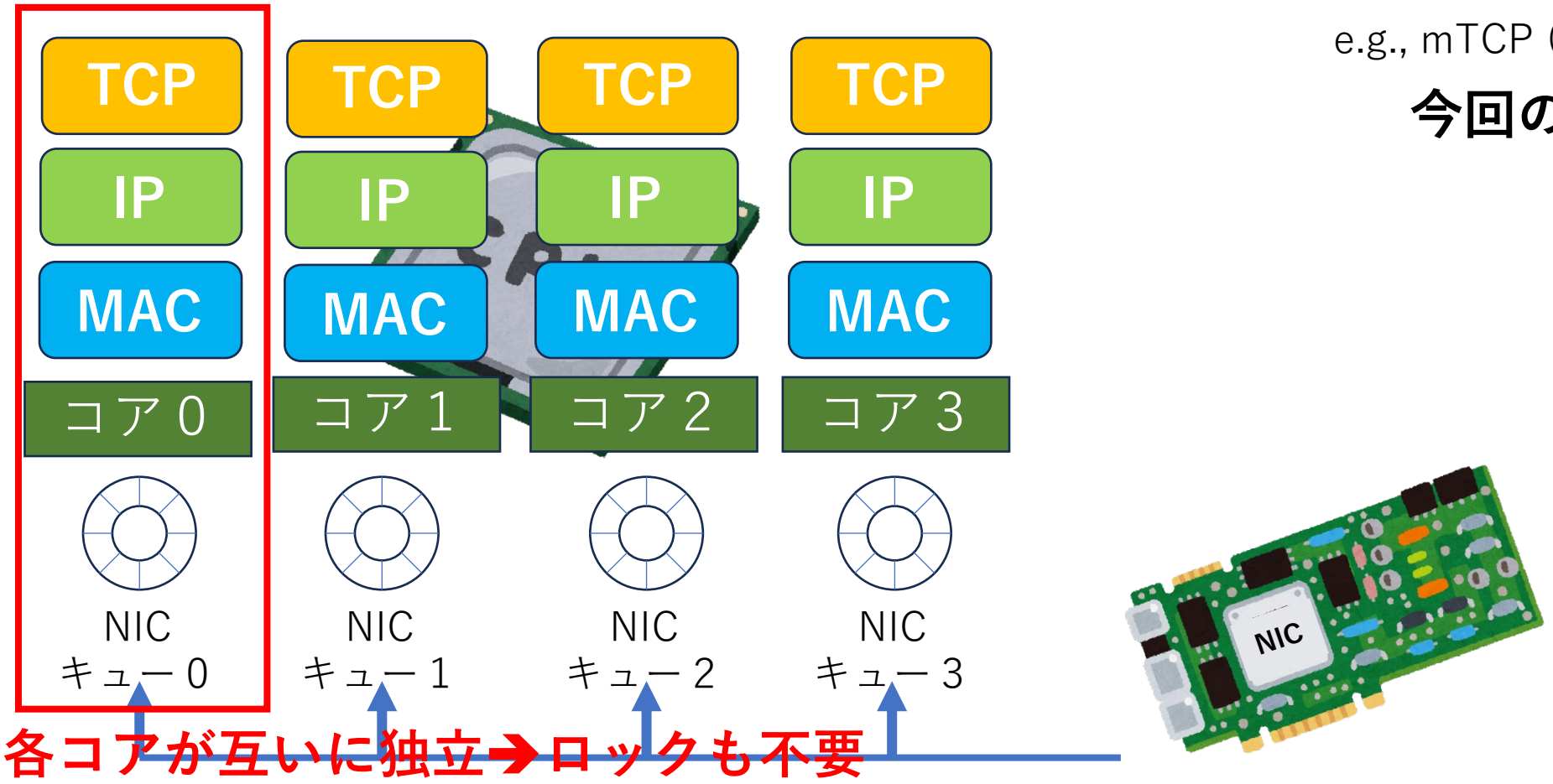


マルチコア環境でのスケーラビリティ

- 方針１：CPU コア間で共有するオブジェクトをなくす

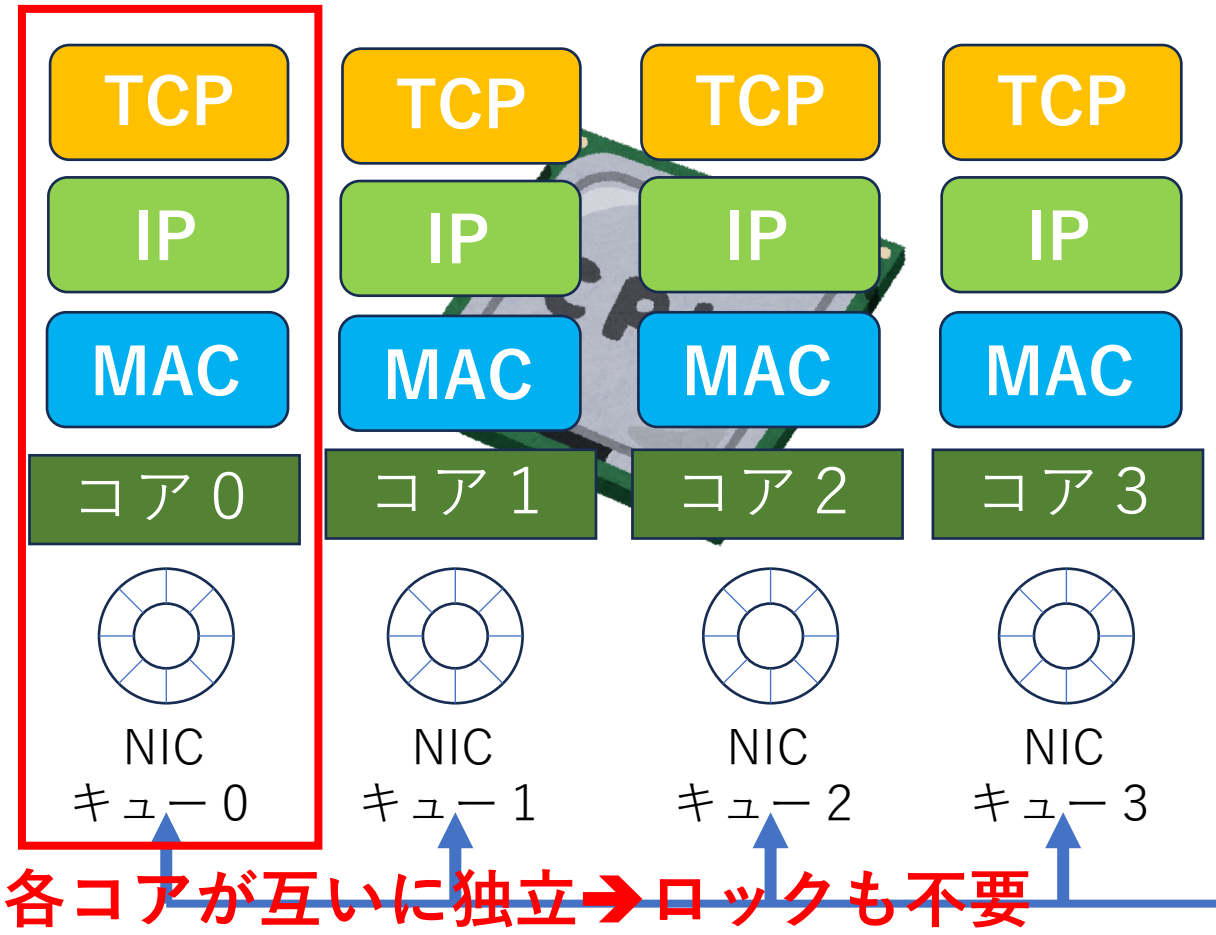
e.g., mTCP (NSDI'14)

今回の実装



マルチコア環境でのスケーラビリティ

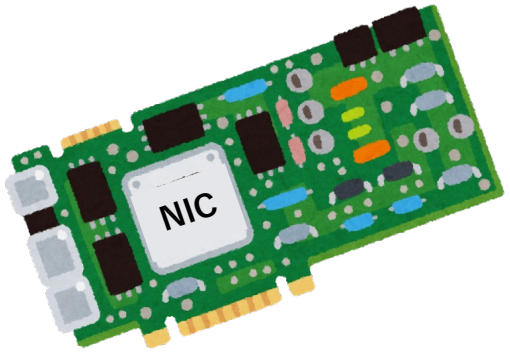
- 方針１：CPU コア間で共有するオブジェクトをなくす



e.g., mTCP (NSDI'14)

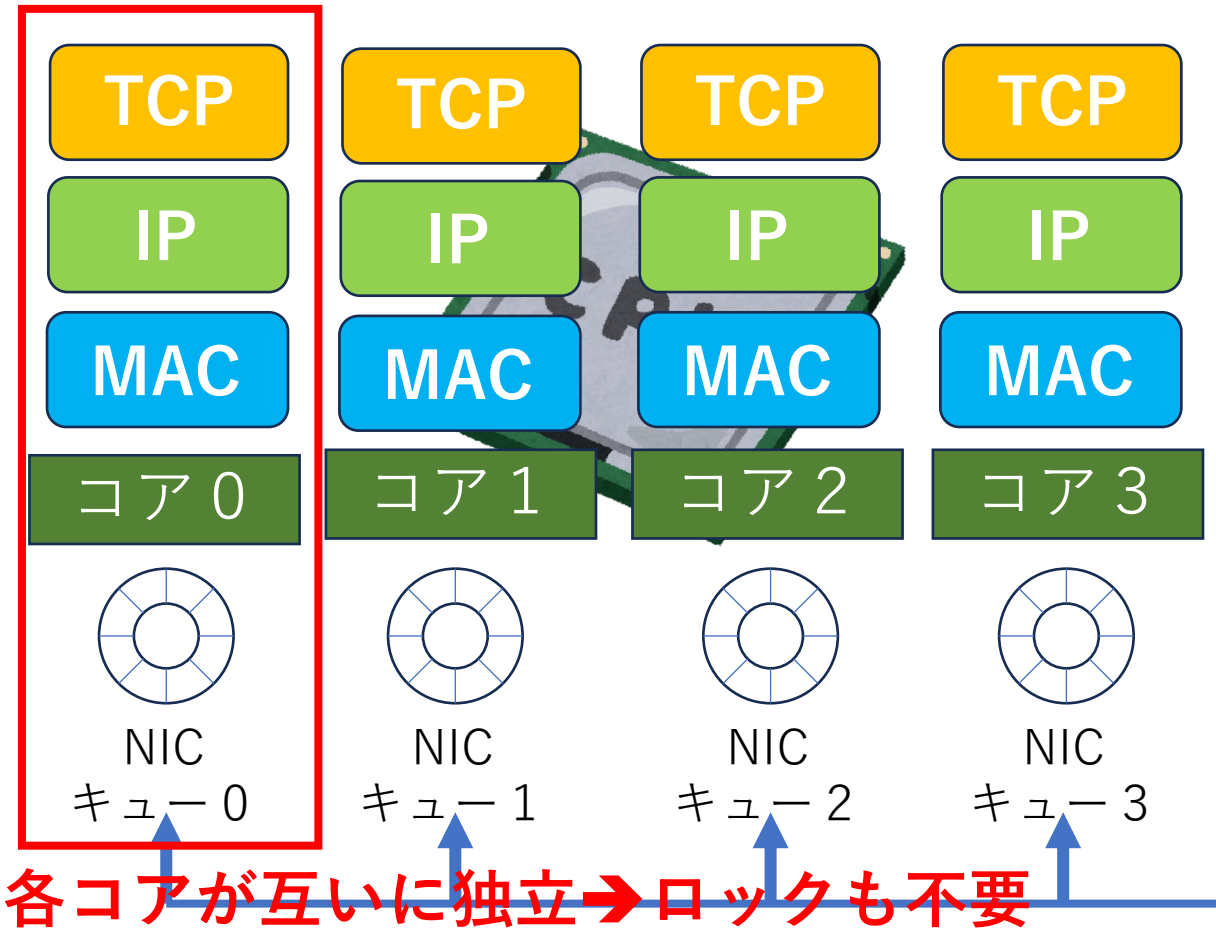
今回の実装

NIC のハードウェア機能である Receive Side Scaling (RSS) を利用して、受信パケットを複数の NIC キューへ振り分け



マルチコア環境でのスケーラビリティ

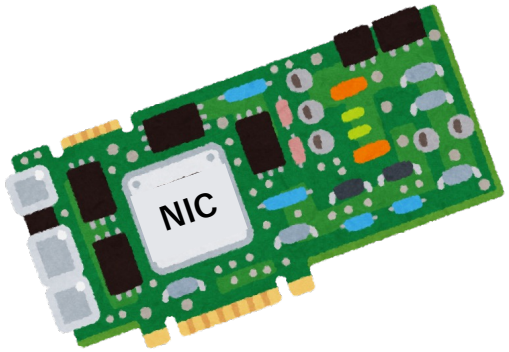
- 方針１：CPU コア間で共有するオブジェクトをなくす



e.g., mTCP (NSDI'14)

今回の実装

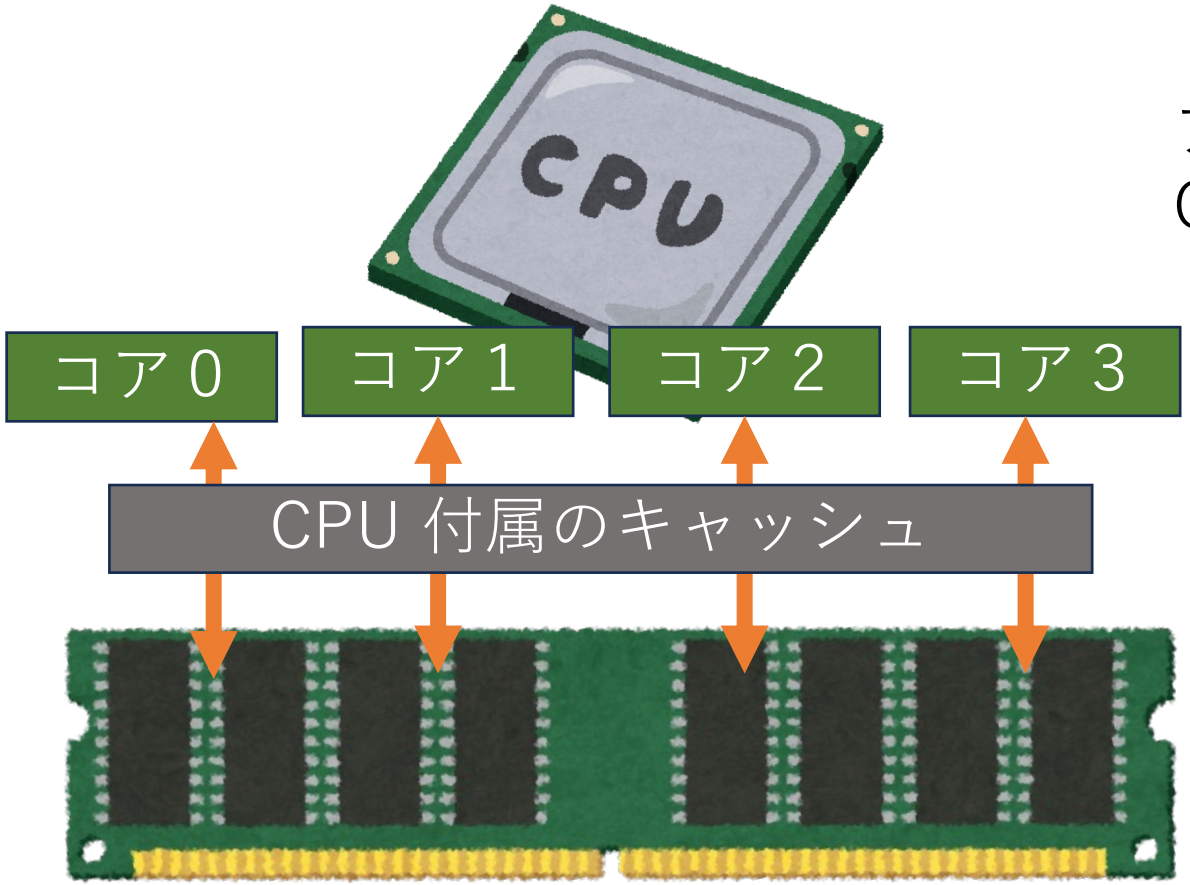
NIC のハードウェア機能である Receive Side Scaling (RSS) を利用して、受信パケットを複数の NIC キューへ振り分け



RSS のおかげで、特定のTCP 接続のパケットは常に同一のキューへ常に届けられる

マルチコア環境でのスケーラビリティ

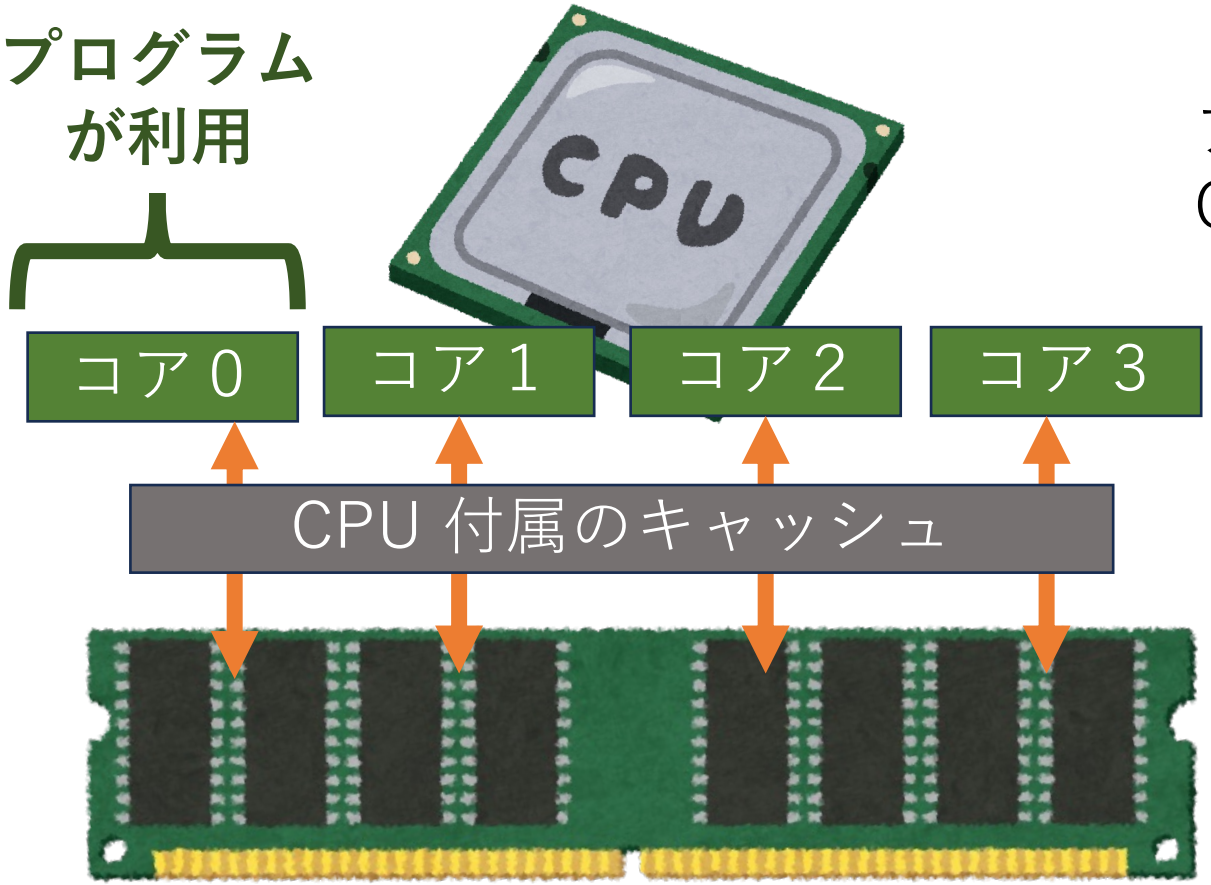
- 方針２：１CPUコアがアクセスするメモリ領域を小さくする



プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

マルチコア環境でのスケーラビリティ

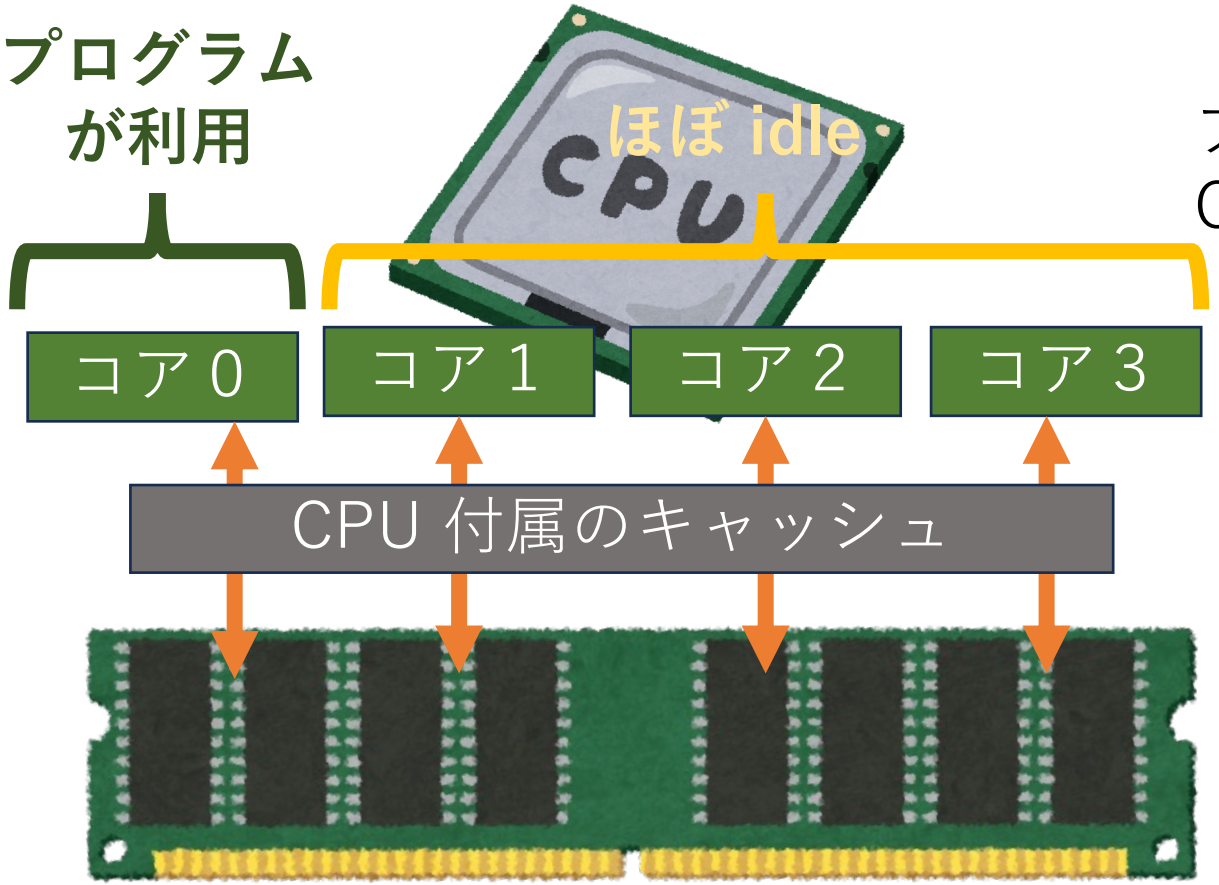
- 方針２：１CPUコアがアクセスするメモリ領域を小さくする



プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

マルチコア環境でのスケーラビリティ

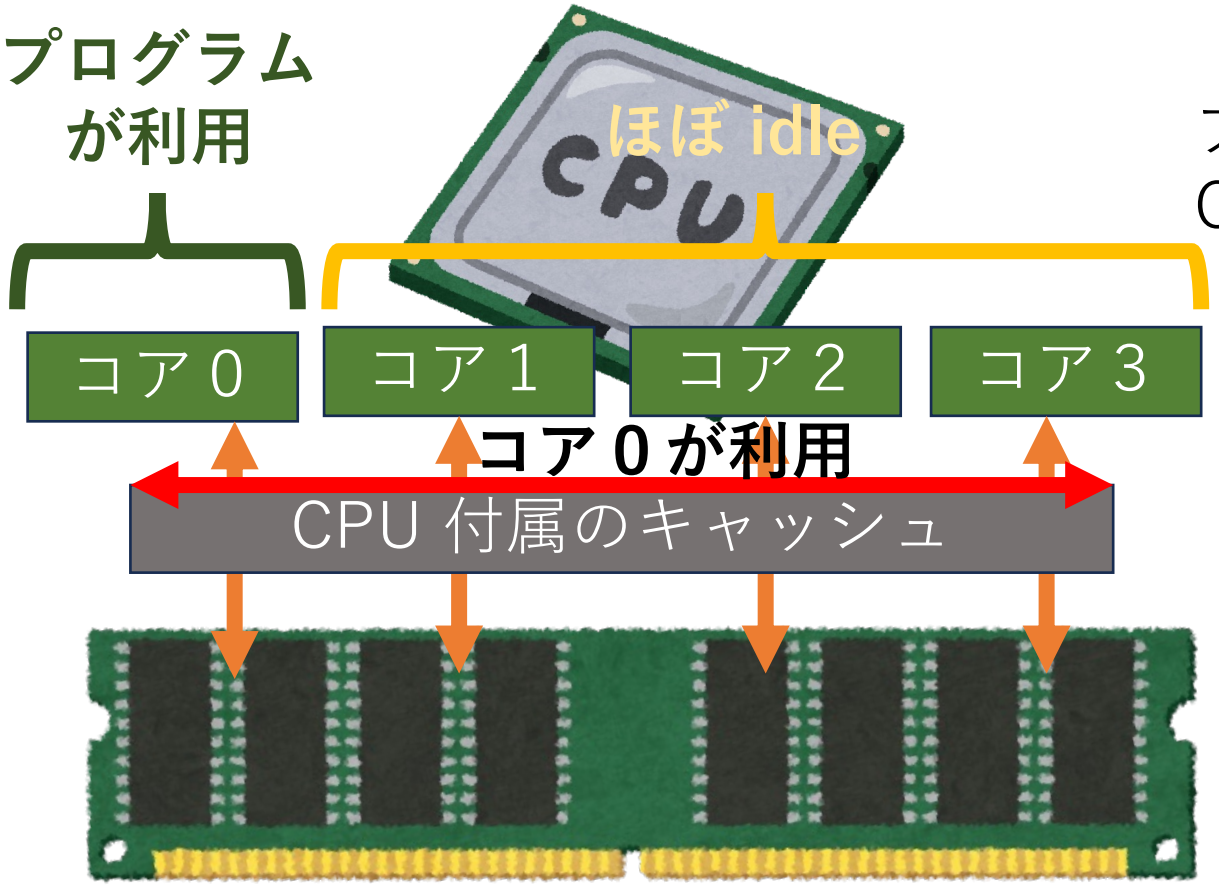
- 方針２：１CPUコアがアクセスするメモリ領域を小さくする



プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

マルチコア環境でのスケーラビリティ

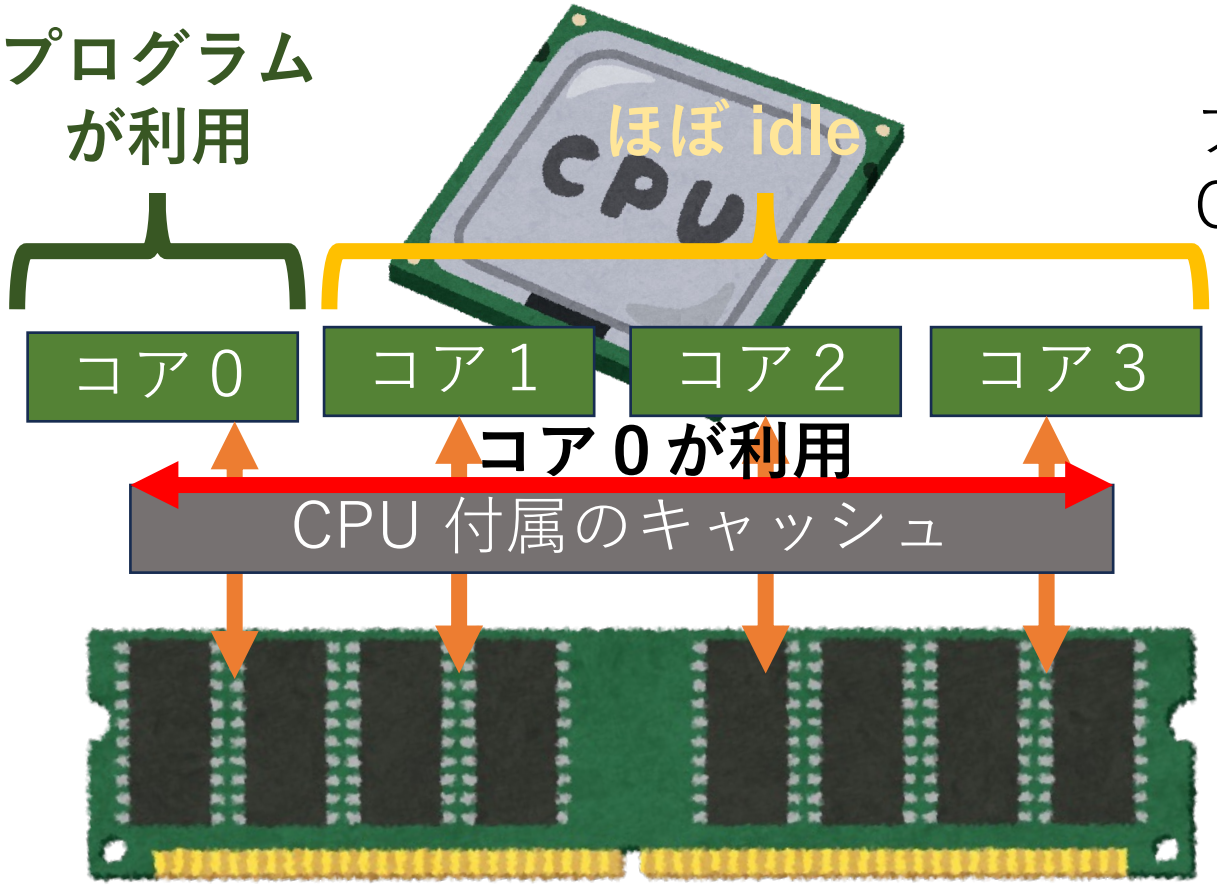
- 方針２：１CPUコアがアクセスするメモリ領域を小さくする



プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

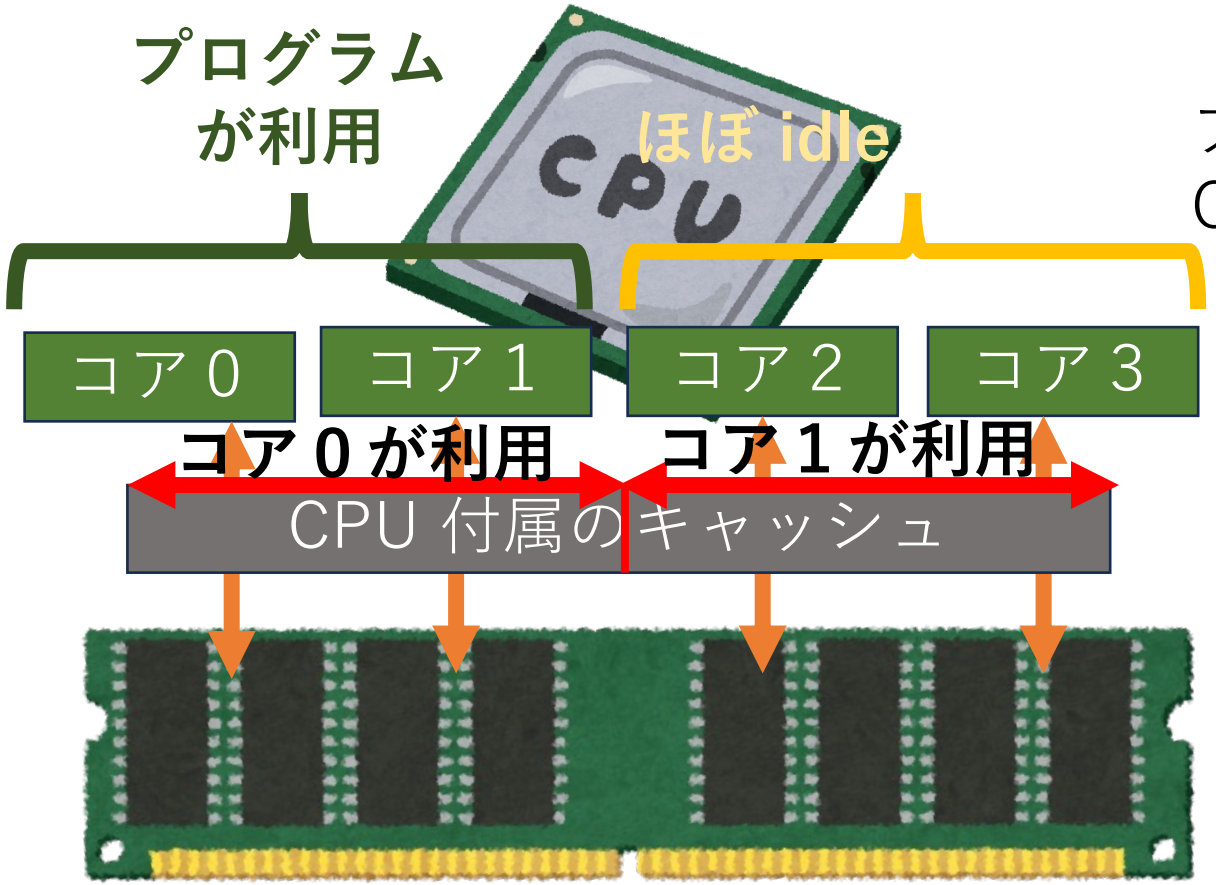


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

１CPUの中で１コアだけが
重点的に稼働している場合は、
そのコアが付属のキャッシュを
ほぼ占有できる

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

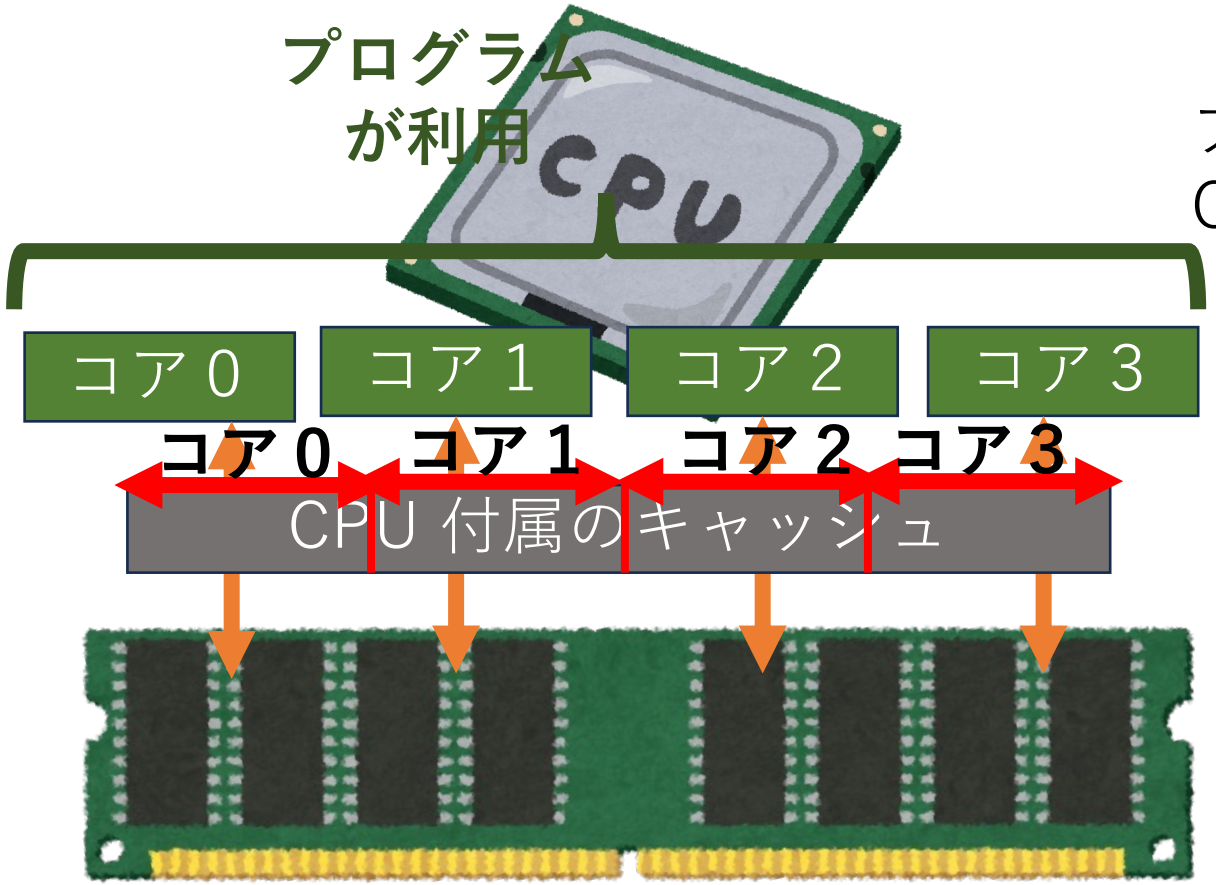


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

１CPUの中で２コアが
重点的に稼働している場合は、
付属のキャッシュをその２コアで
共有する：一つのコアが利用できる
キャッシュサイズが約半分になる

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

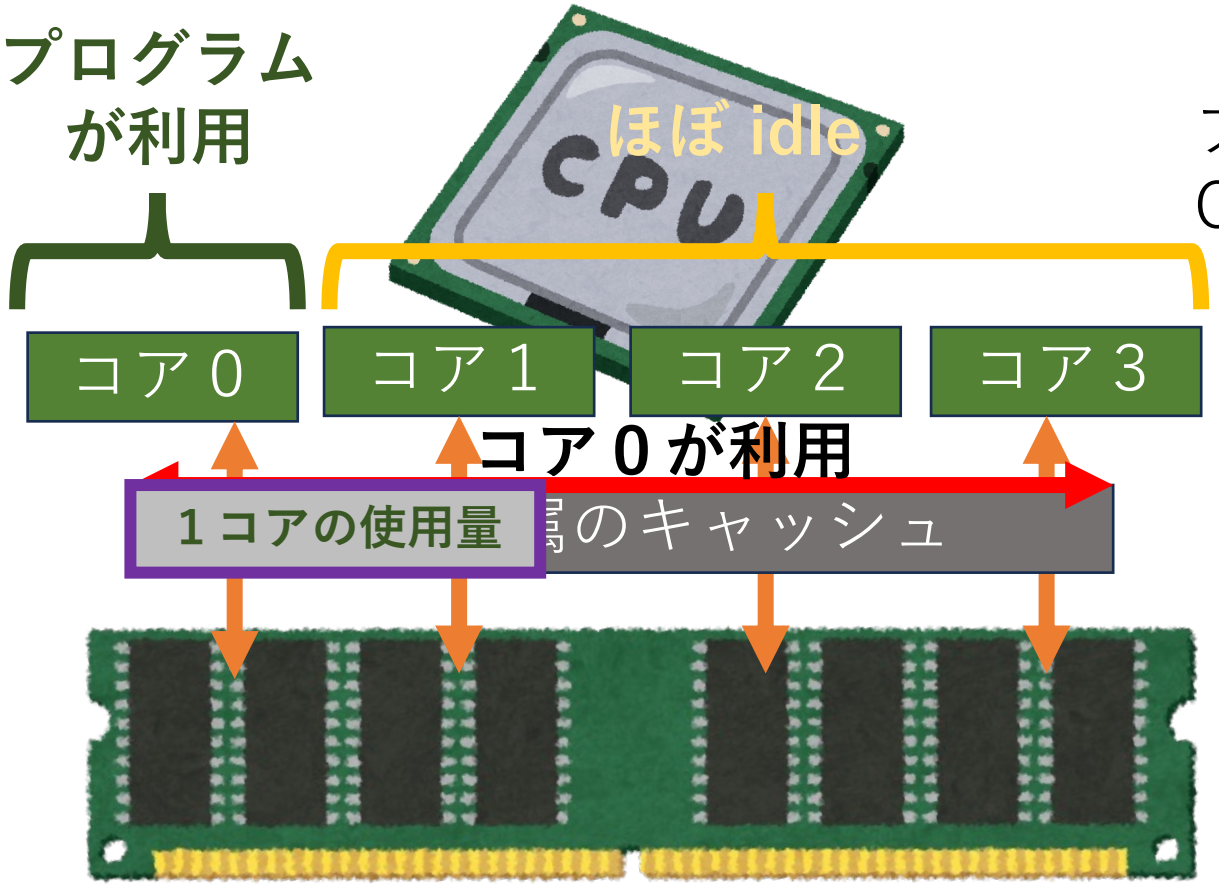


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

１CPUの中で４コアが
重点的に稼働している場合は、
付属のキャッシュをその４コアで
共有する：一つのコアが利用できる
キャッシュサイズが約１/４になる

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

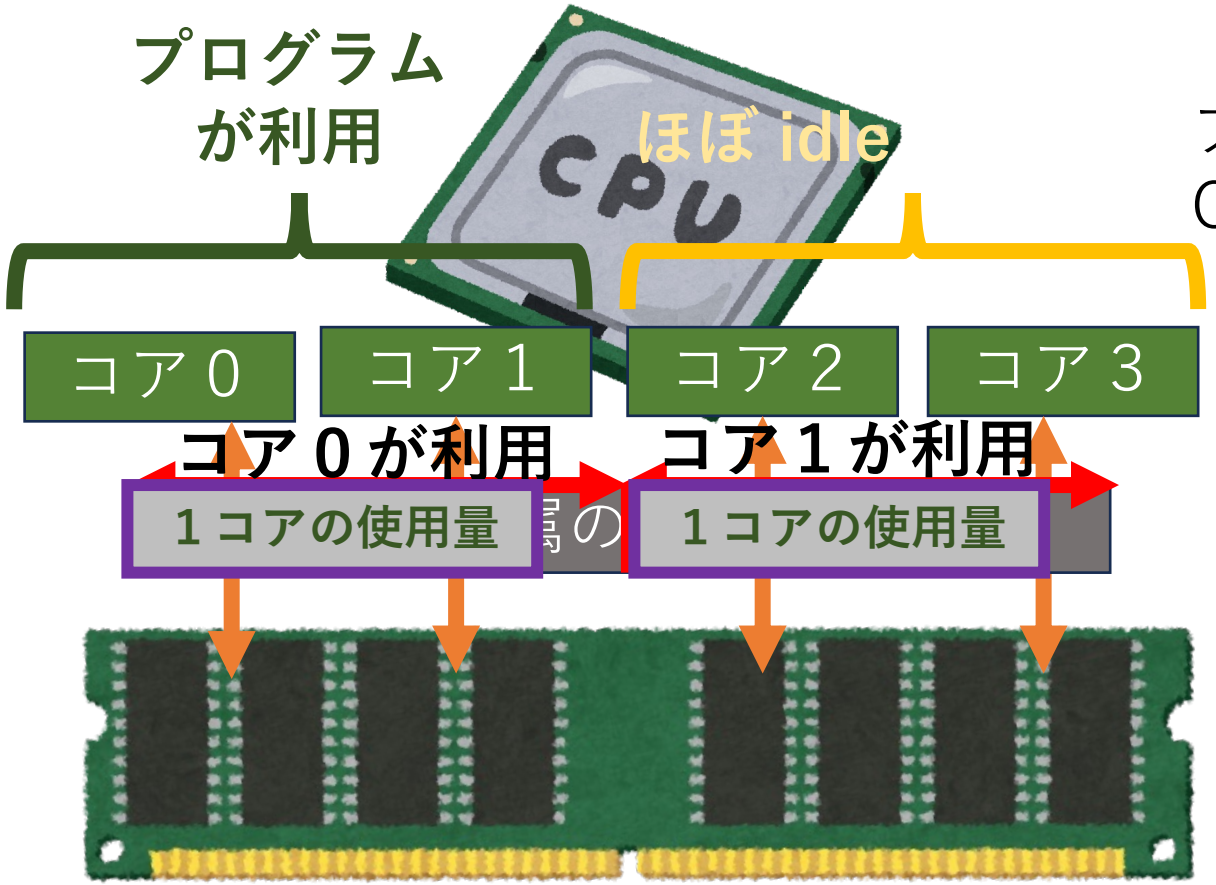


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

１CPUの中で１コアだけが
重点的に稼働している場合は、
そのコアが付属のキャッシュを
ほぼ占有できる

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

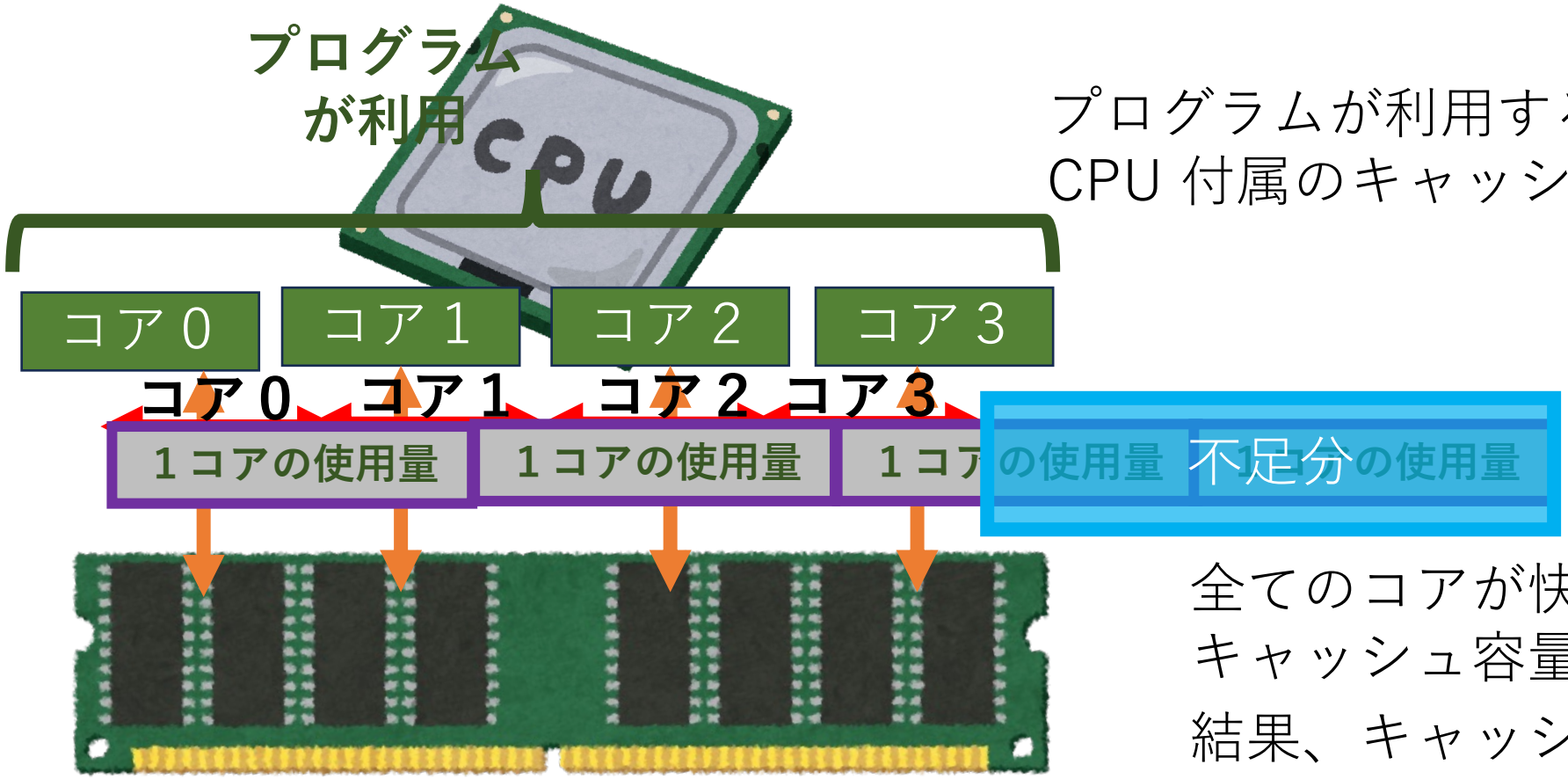


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

１CPUの中で２コアが
重点的に稼働している場合は、
付属のキャッシュをその２コアで
共有する：一つのコアが利用できる
キャッシュサイズが約半分になる

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

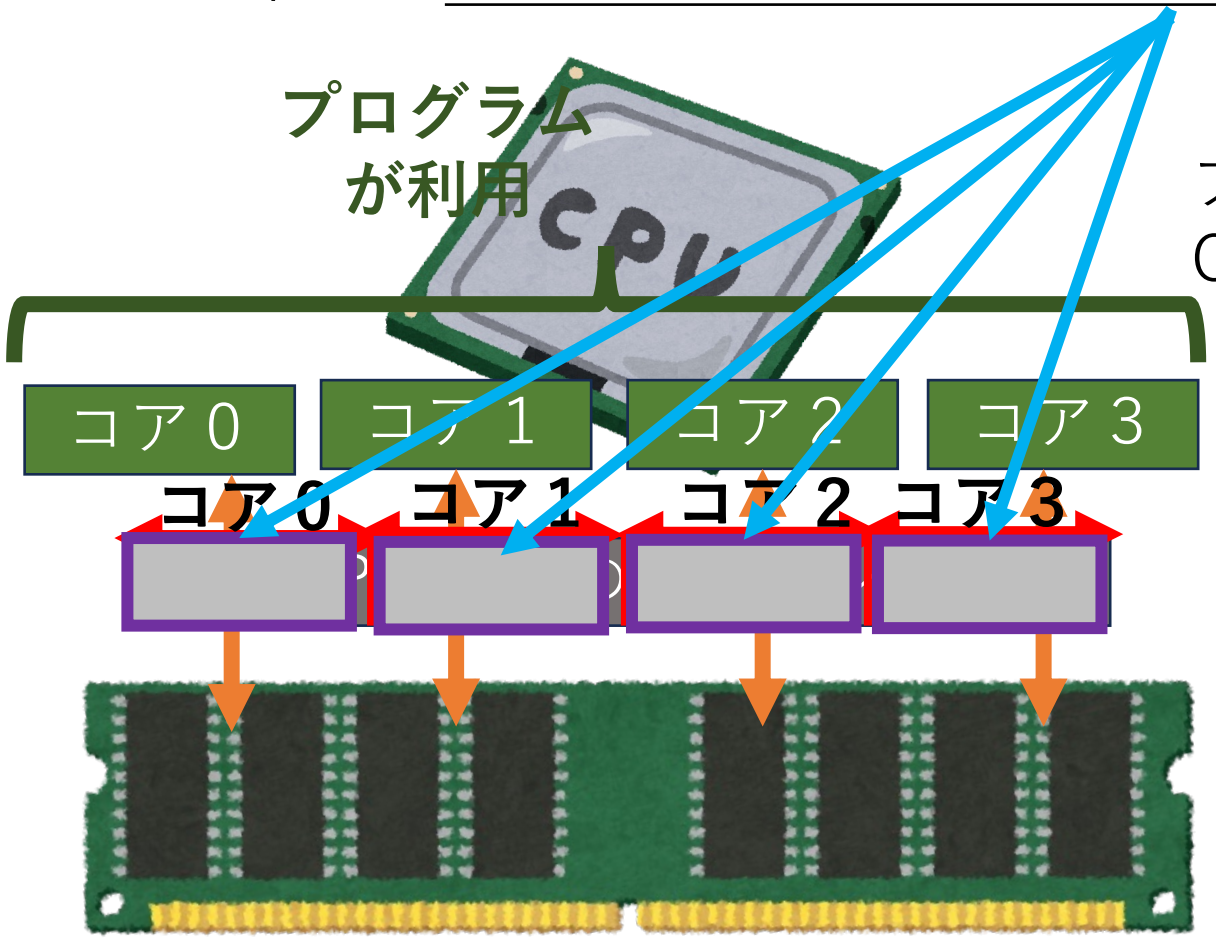


プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

全てのコアが快適に動作するには
キャッシュ容量が足りなくなる
結果、キャッシュミスが多発する

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする



プログラムが利用するコア数を増やしても
CPU 付属のキャッシュサイズは増えない

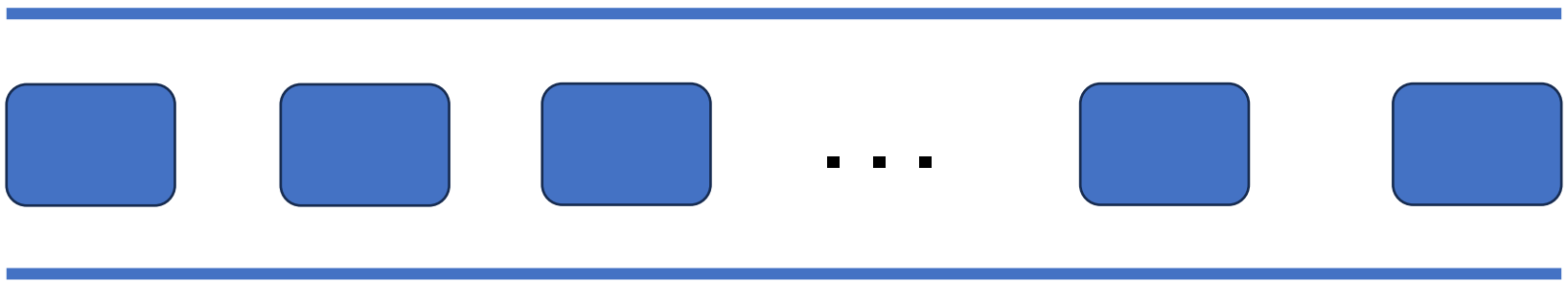
１CPUコアがアクセスするメモリ領域を小さく保つことで、１CPUコアあたりの利用キャッシュサイズを小さく抑え、なるべく多くのコアが動いてもデータがキャッシュに乗るようにする

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



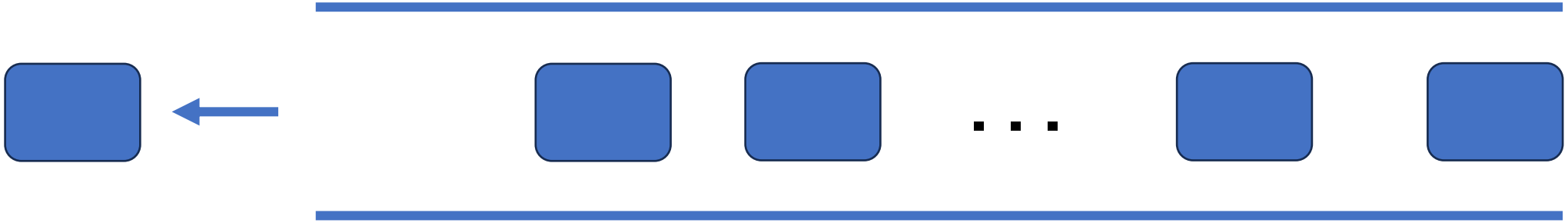
データ構造確保時：先頭から取り出し

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



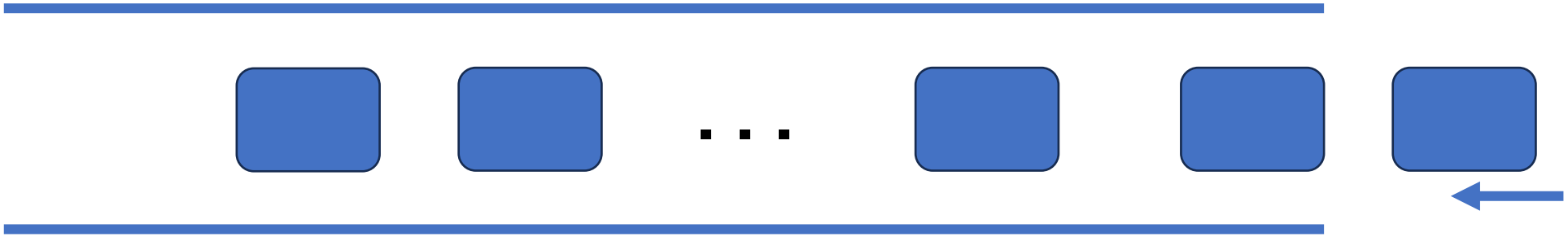
データ構造確保時：先頭から取り出し

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

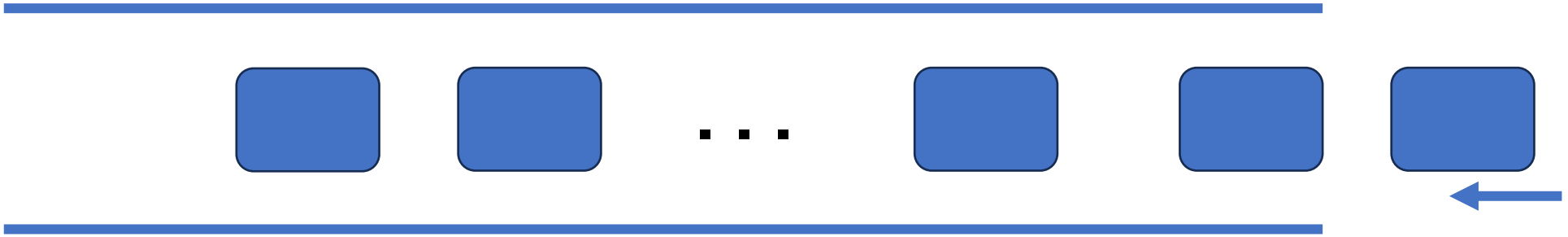
データ構造解放時：末端に追加

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

データ構造解放時：末端に追加

16 CPU コアあたりから
性能がスケールしない、、、

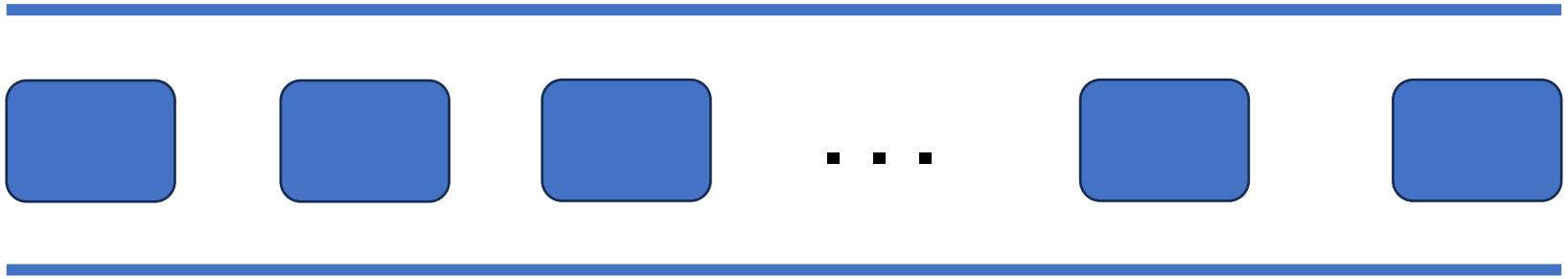
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善方法

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

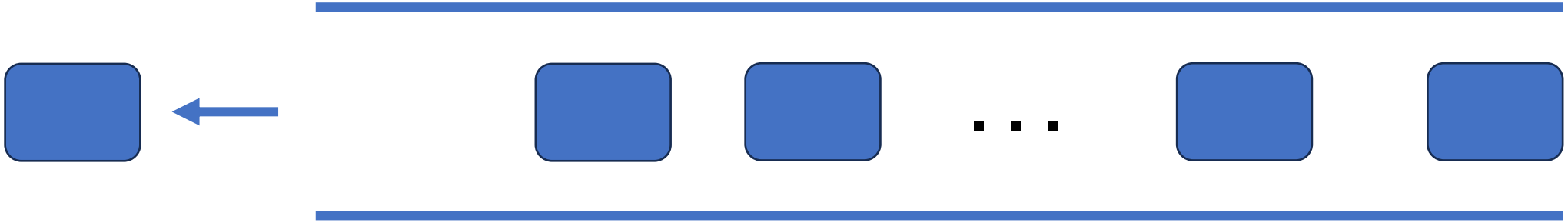
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善方法

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

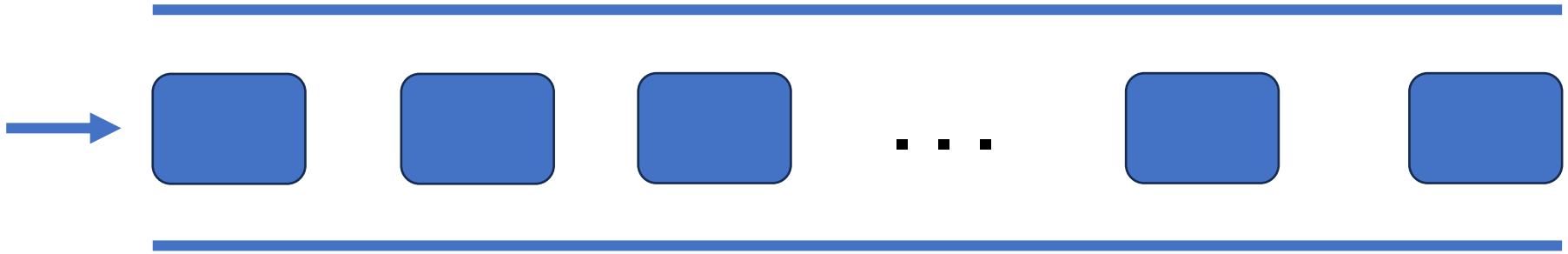
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善方法

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

データ構造解放時：先頭に戻す

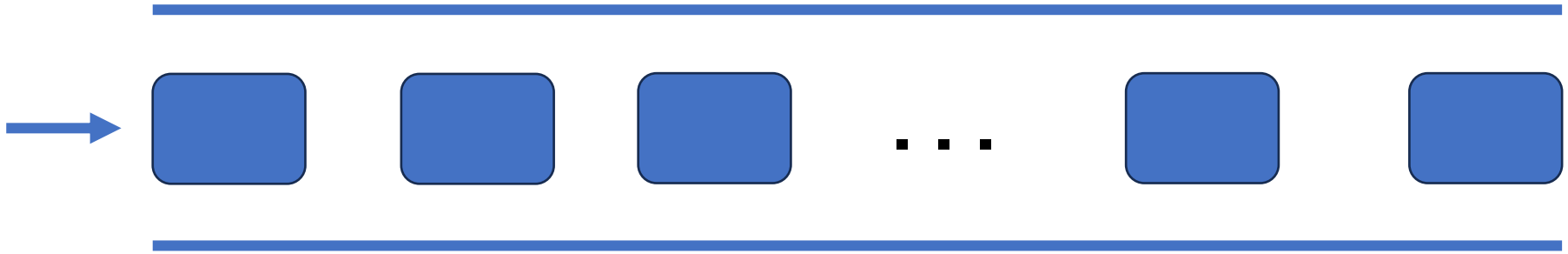
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善方法

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



データ構造確保時：先頭から取り出し

データ構造解放時：先頭に戻す

これだけで 32 CPU コアまで
性能がスケールするようになった

改善前と比べて約３倍高速化

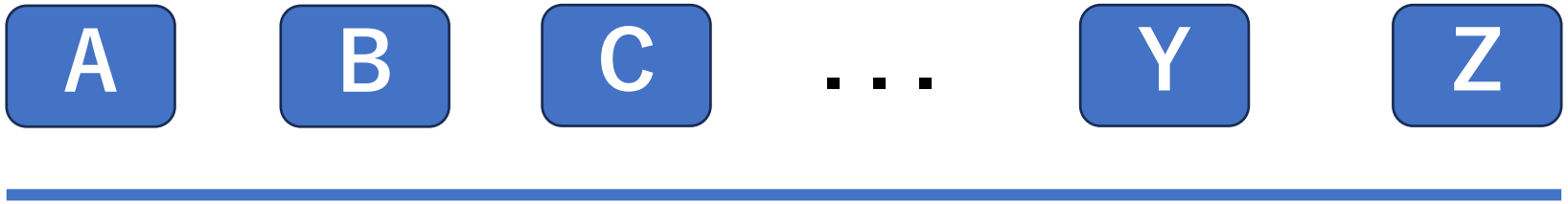
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



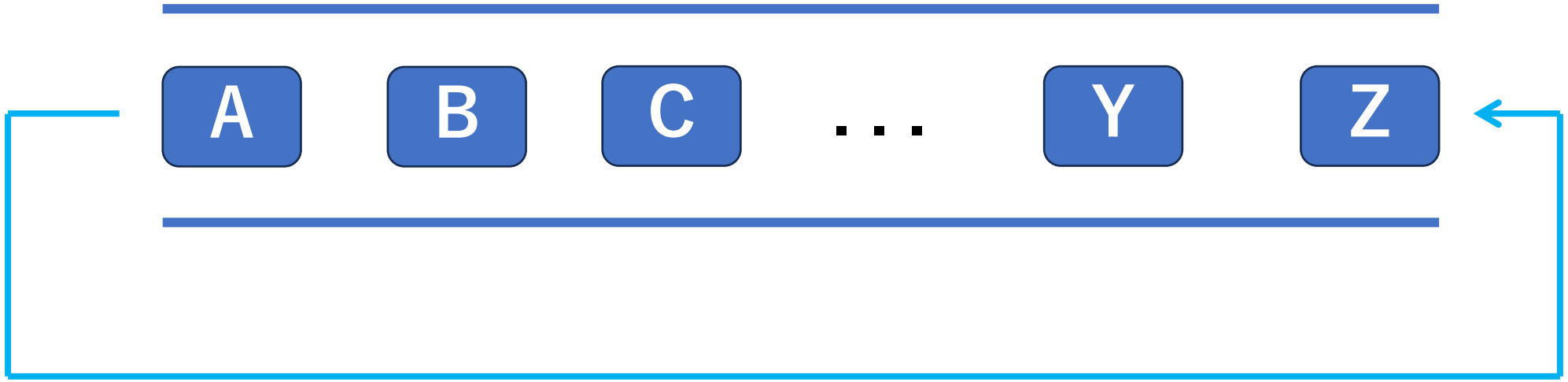
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



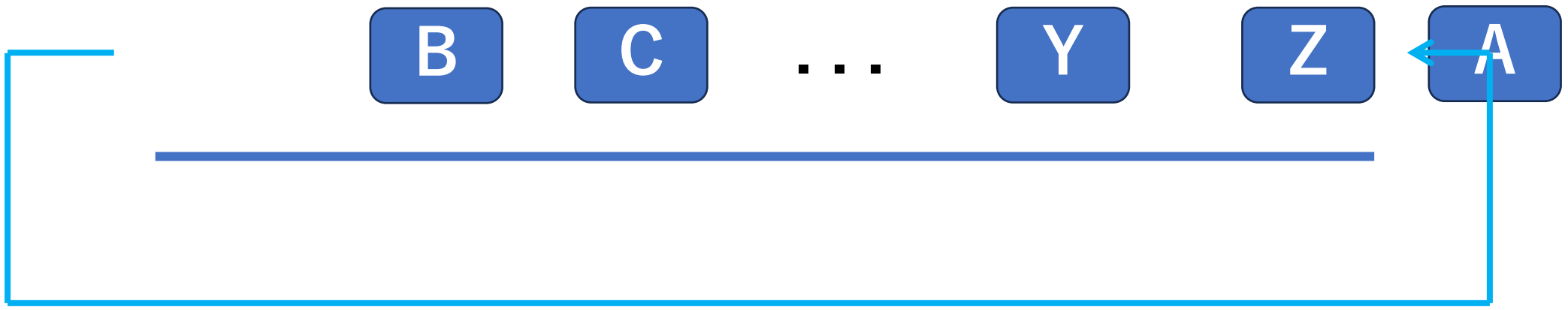
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



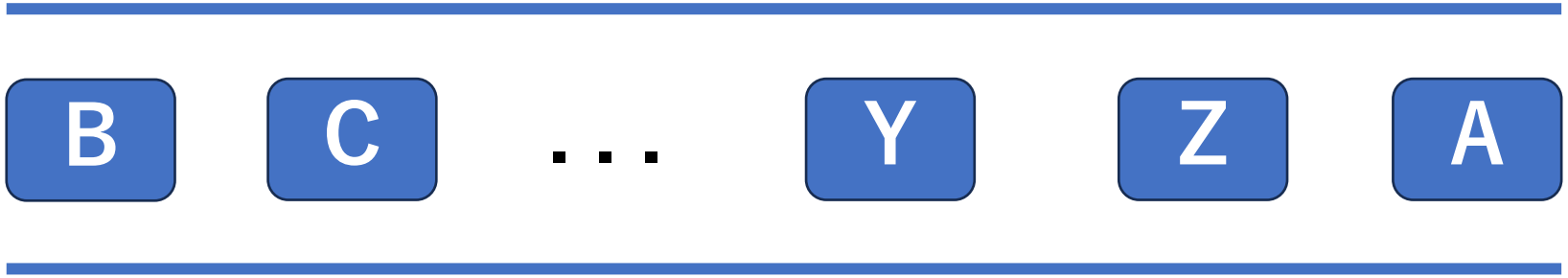
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト

マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト

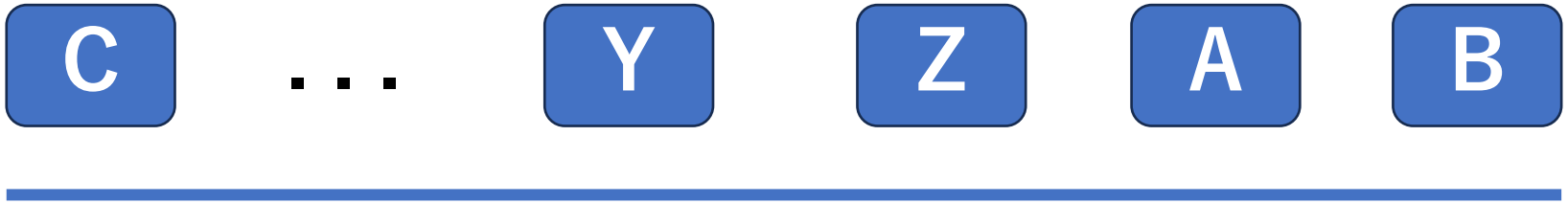
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



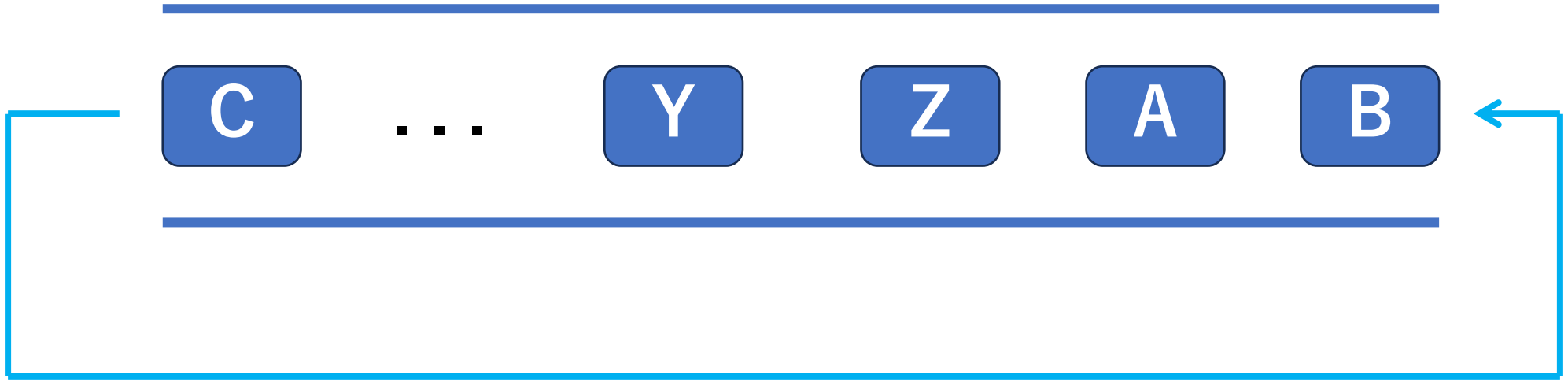
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



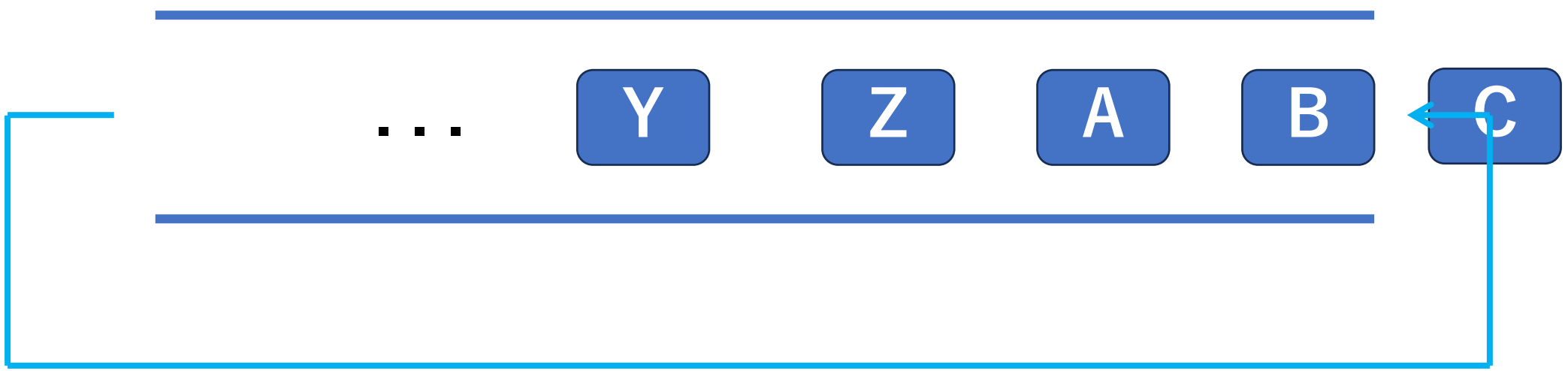
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



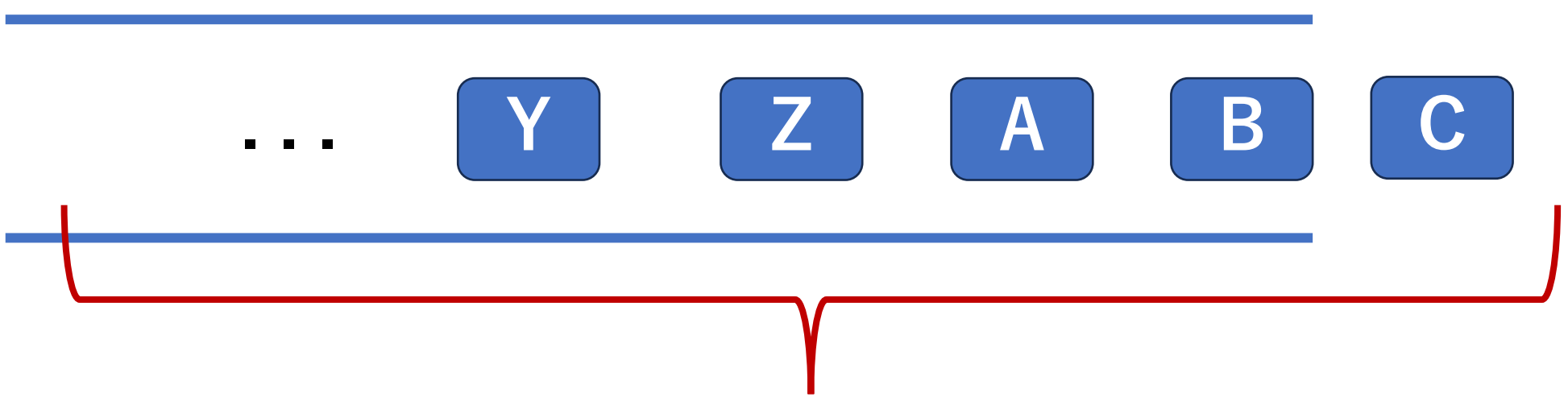
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善前

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



確保・解放を繰り返すと、これら全てにアクセスすることになる

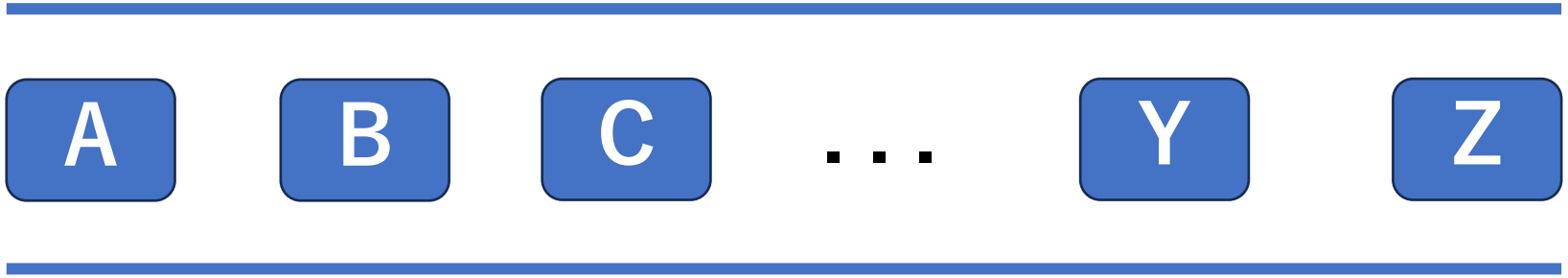
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善後

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



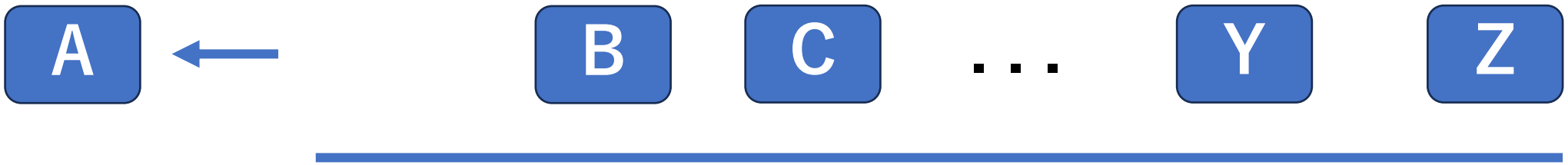
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善後

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



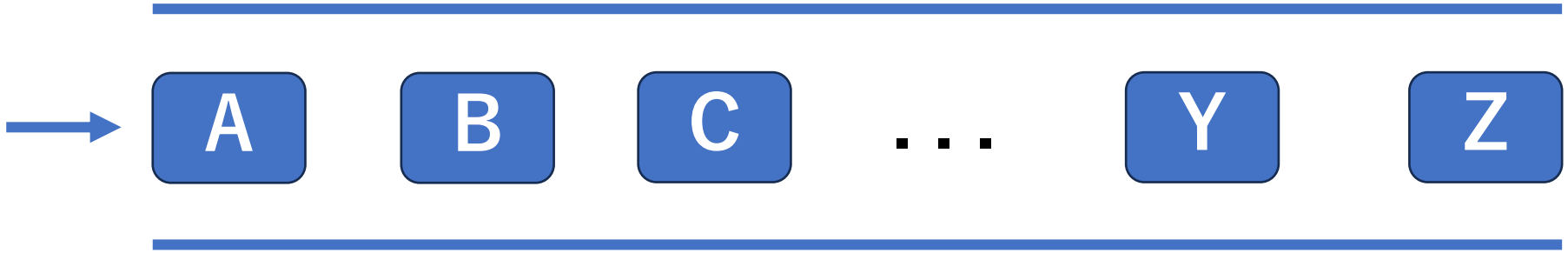
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善後

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



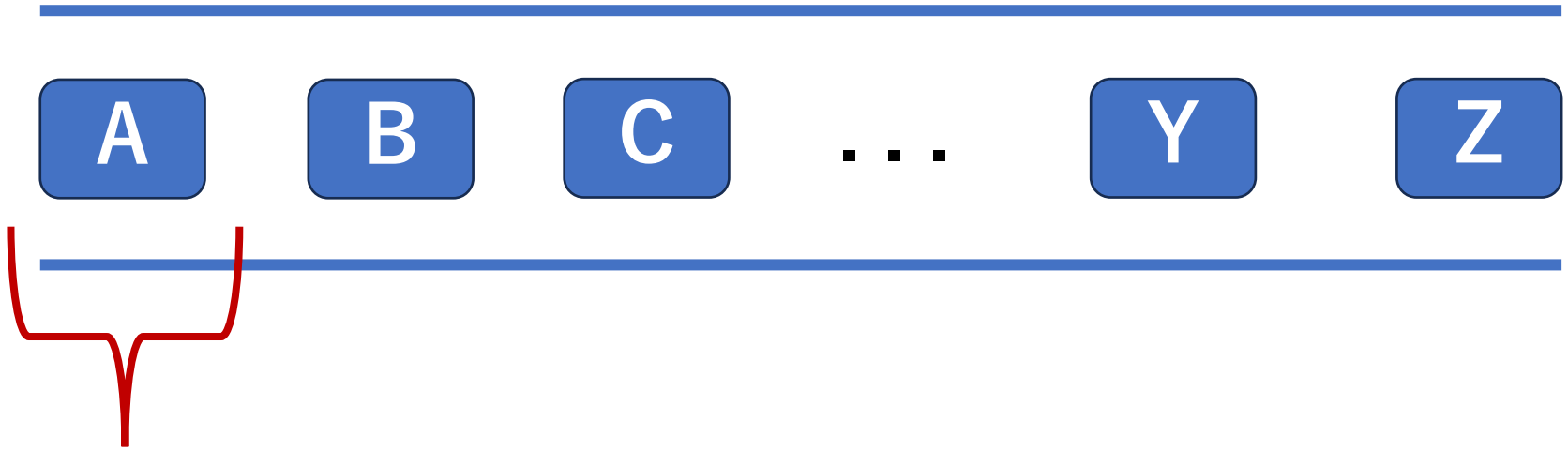
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善後

予めメモリ領域を確保したパケットを表現するデータ構造のリスト



先頭に戻すと、なるべく同じアドレスのオブジェクトが利用できる
＝＞１CPUコアが利用するキャッシュサイズを減らすことができた

実装

- https://github.com/yasukata/iip

- リポジトリ内のファイル
  - iip/LICENSE
  - iip/README.md
  - iip/main.c ← このファイル自体は以下のフラグでコンパイル可能

```
-m32 -std=c89 -nostdlib -nostdinc
```

- 現状の依存要素は C 言語と 32 以上のビットサイズの CPU

環境依存実装とアプリ

- 環境依存である NIC I/O 関連
  - DPDK : https://github.com/yasukata/iip-dpdk
  - AF_XDP : https://github.com/yasukata/iip-af_xdp

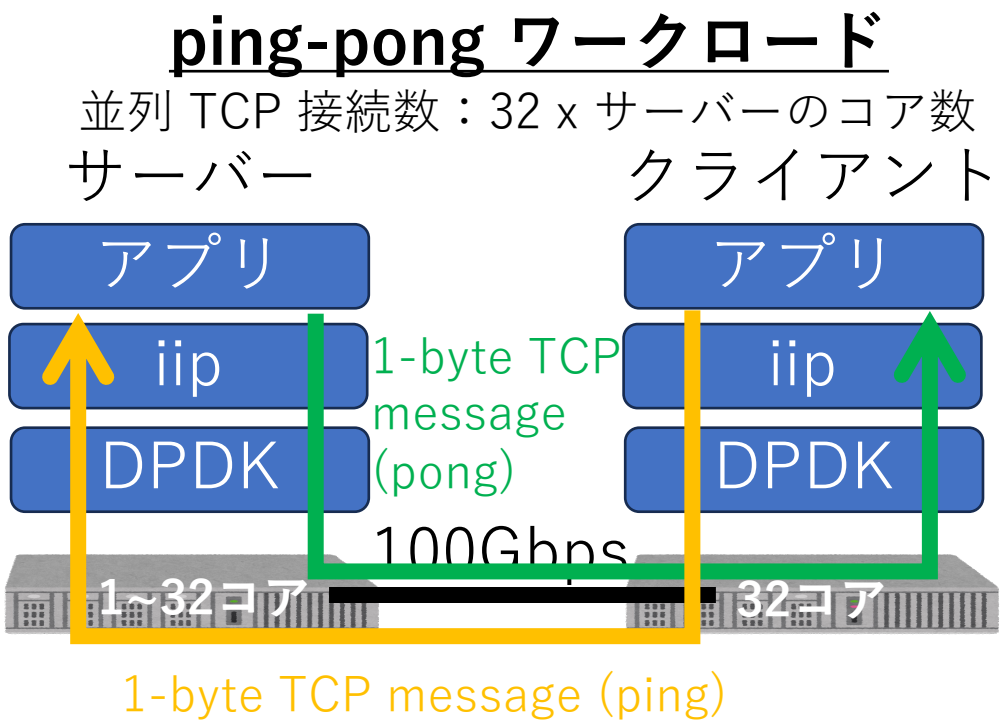
- アプリ
  - 簡易ベンチマークツール：https://github.com/yasukata/bench-iip

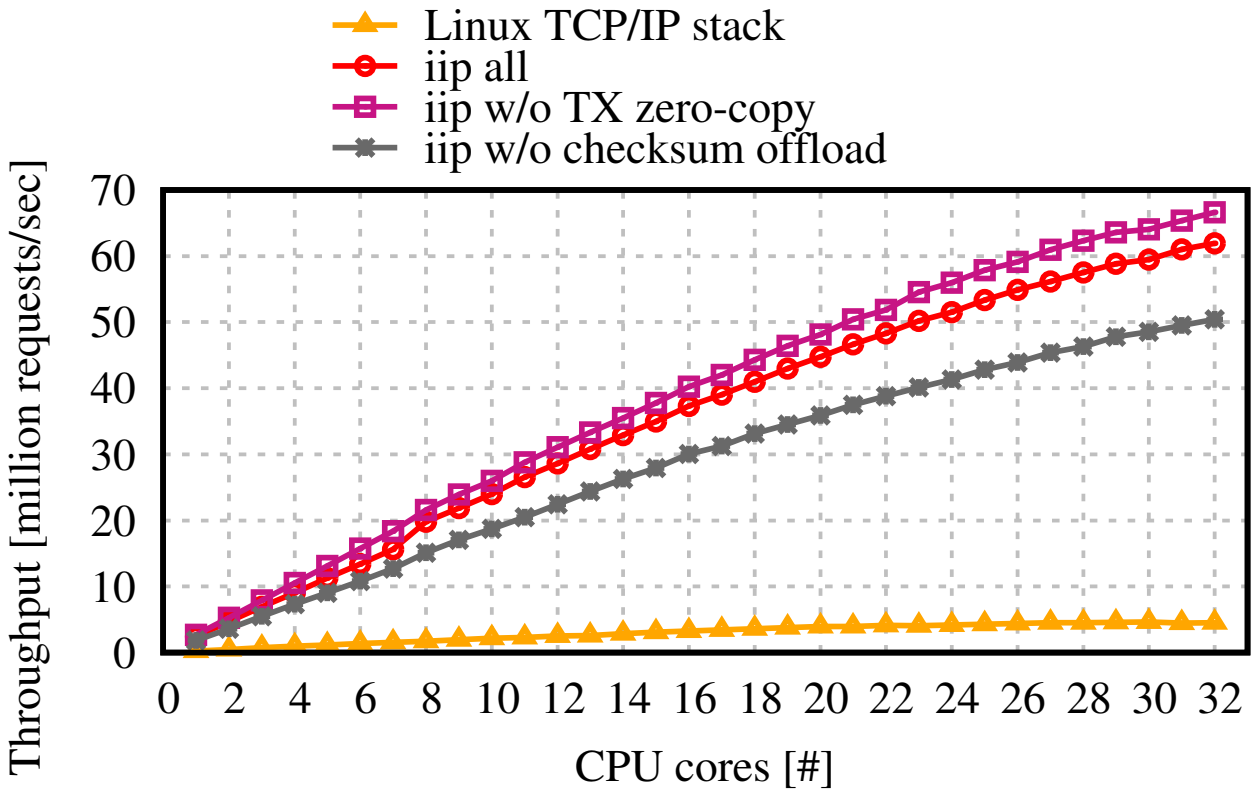
実験環境

- CPU： 2 x 16-core Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz (合計 32 CPU コア)： 1 CPUあたり 24MB キャッシュ
- DRAM：128 GB (DDR4-3200 64GB x 2)
- NIC : Mellanox ConnectX-5 100 Gbps NIC
- OS : Linux 6.2

小さいメッセージの交換 (e.g., RPC)

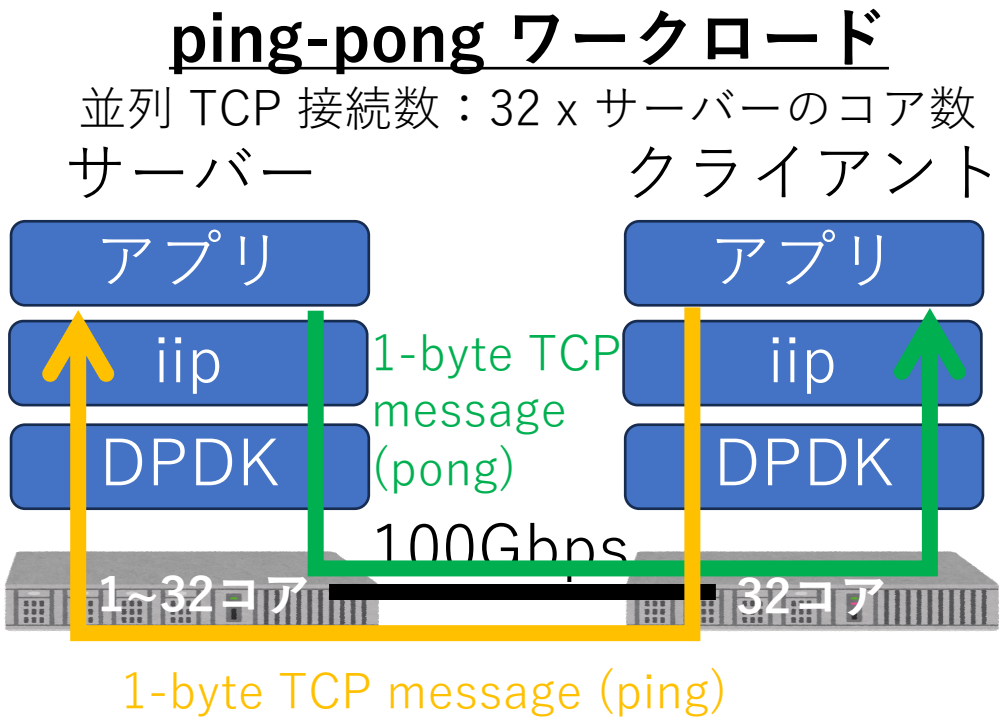
- マルチコアでのスケーラビリティが重要

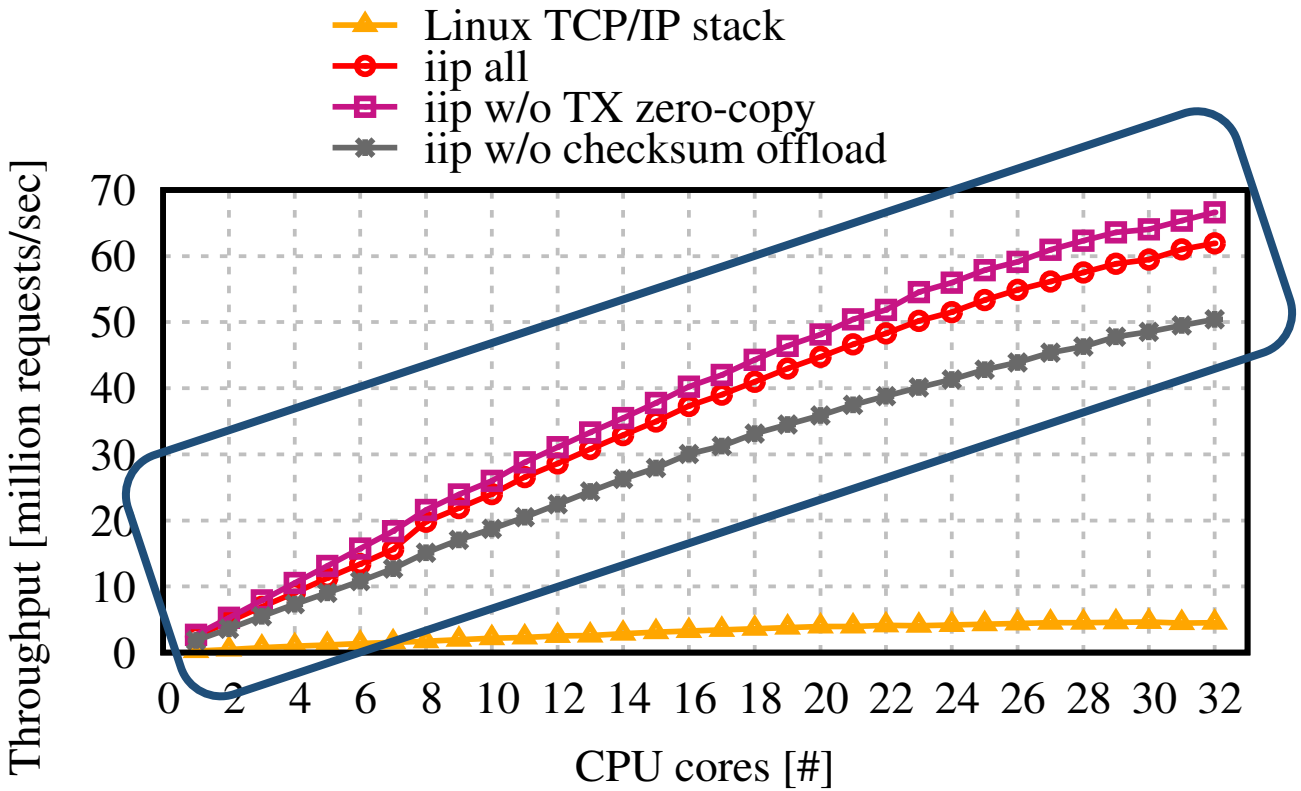




小さいメッセージの交換 (e.g., RPC)

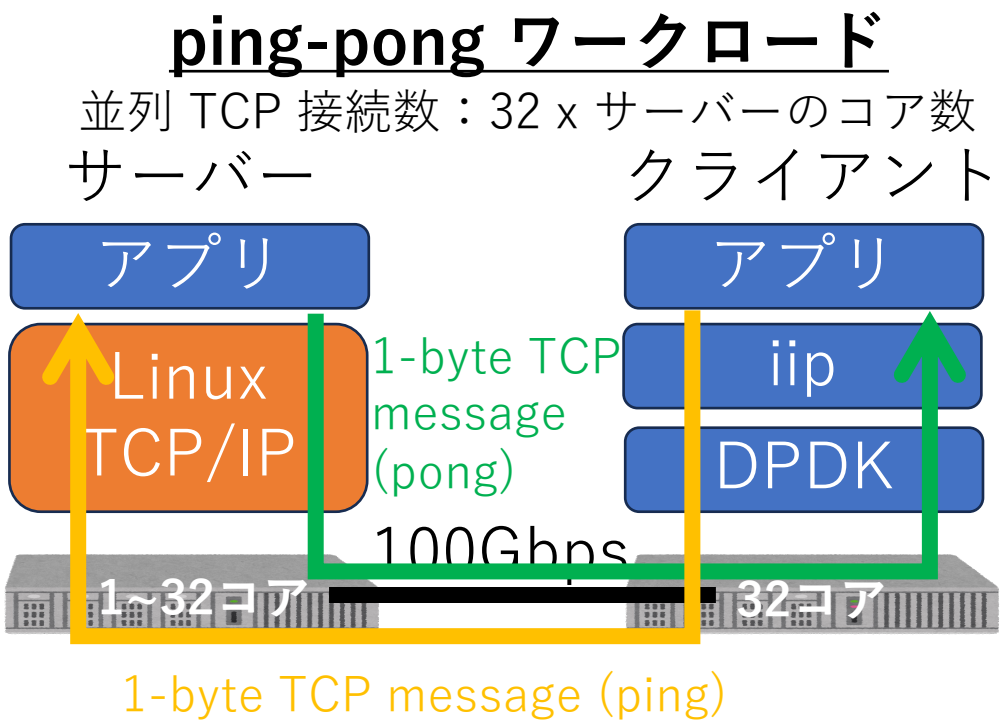
- マルチコアでのスケーラビリティが重要

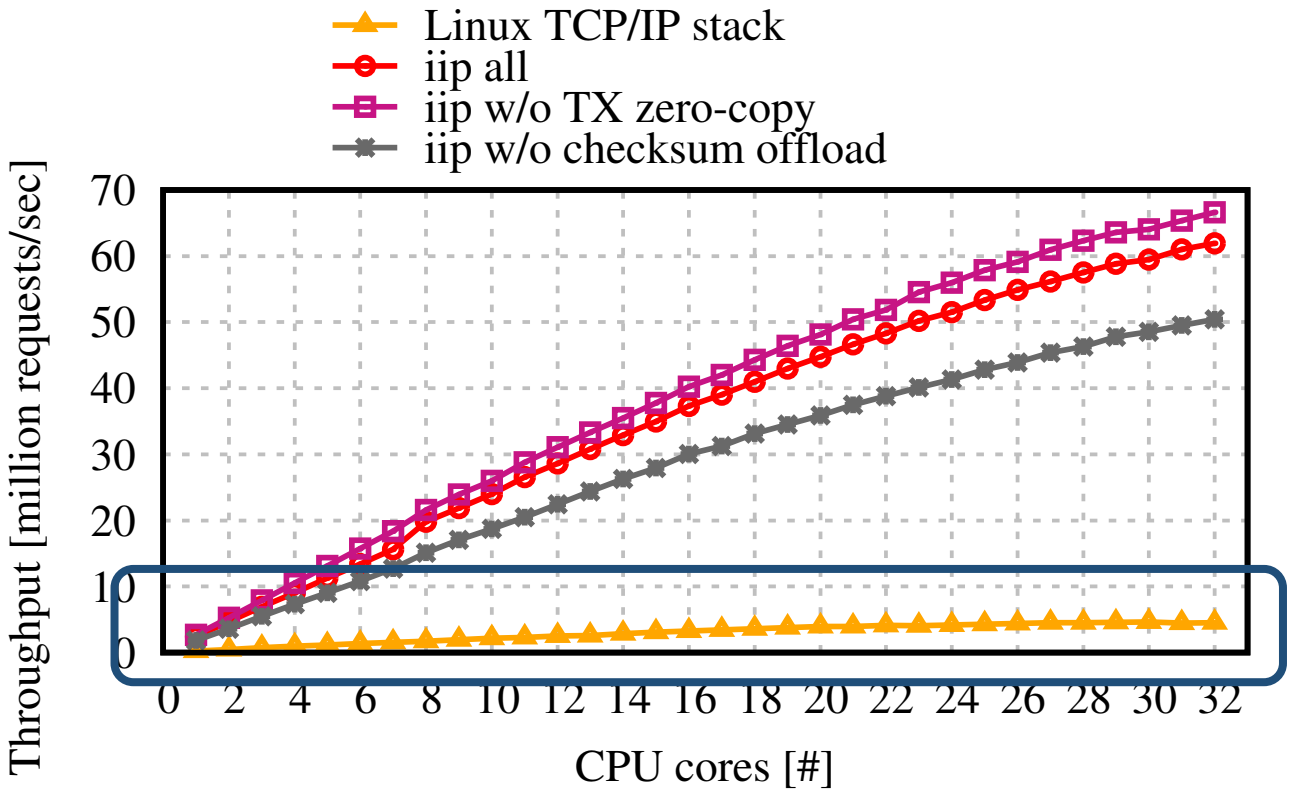




小さいメッセージの交換 (e.g., RPC)

- マルチコアでのスケーラビリティが重要

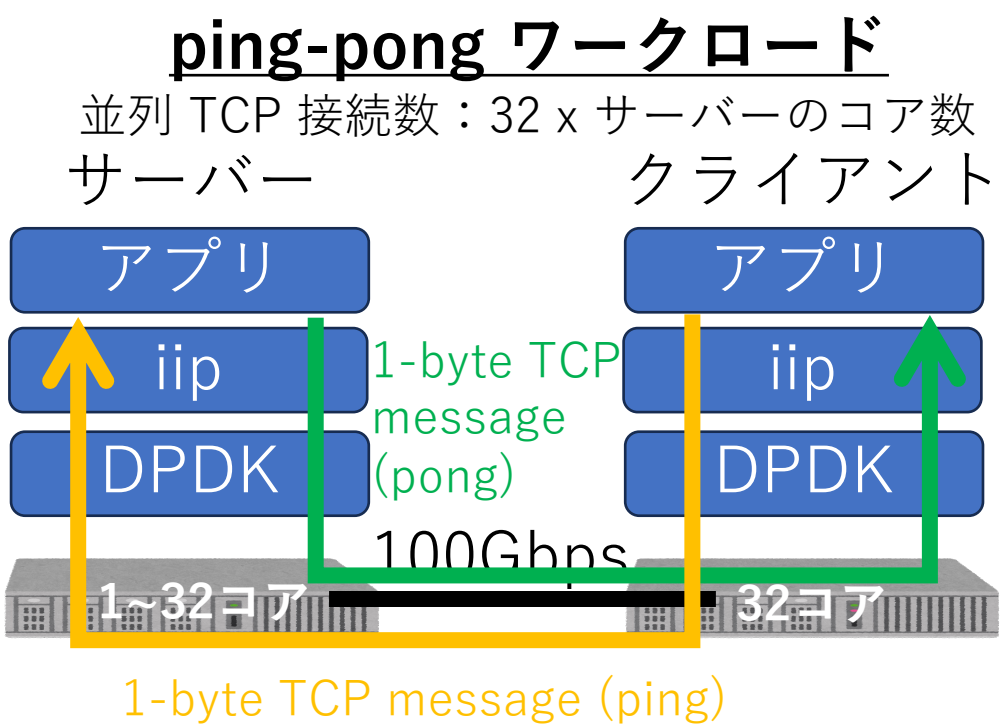


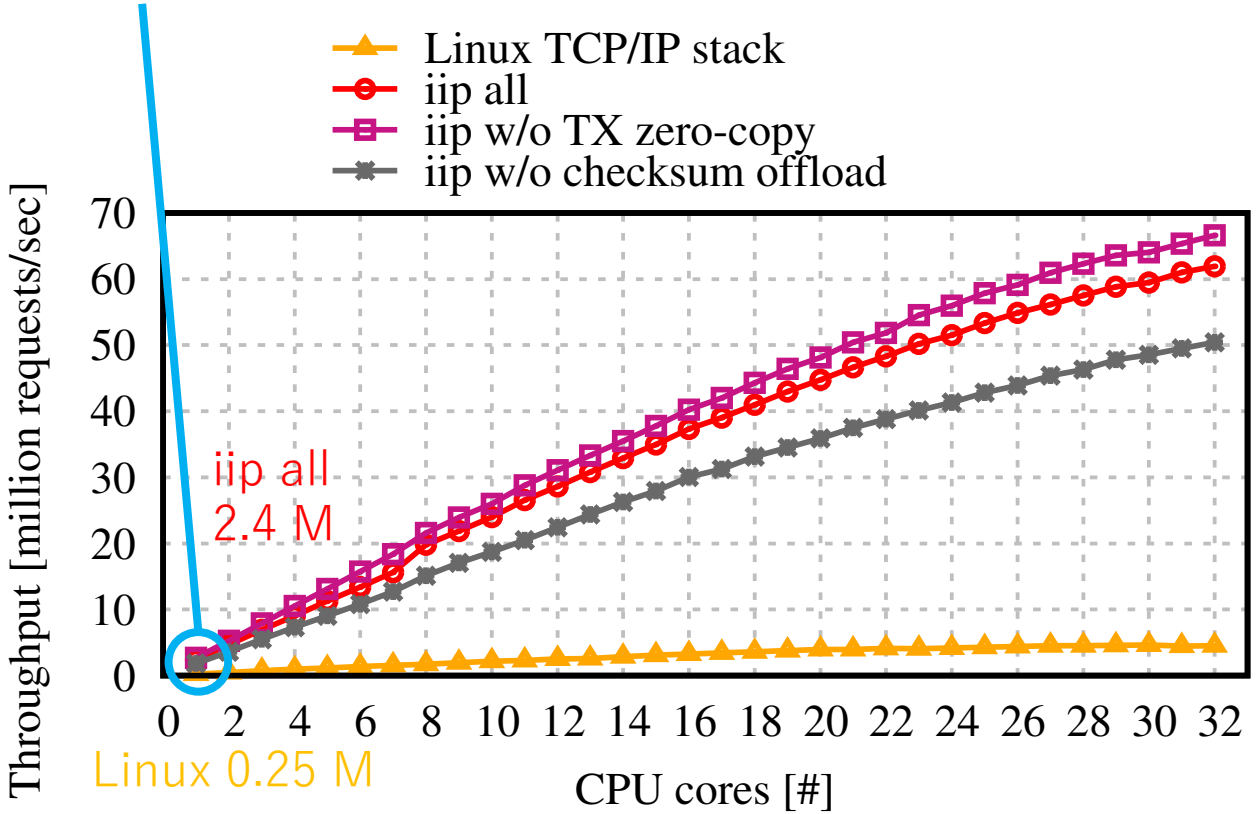


小さいメッセージの交換 (e.g., RPC)

- マルチコアでのスケーラビリティが重要

マルチコア環境を活かすことができない実装の性能の目安

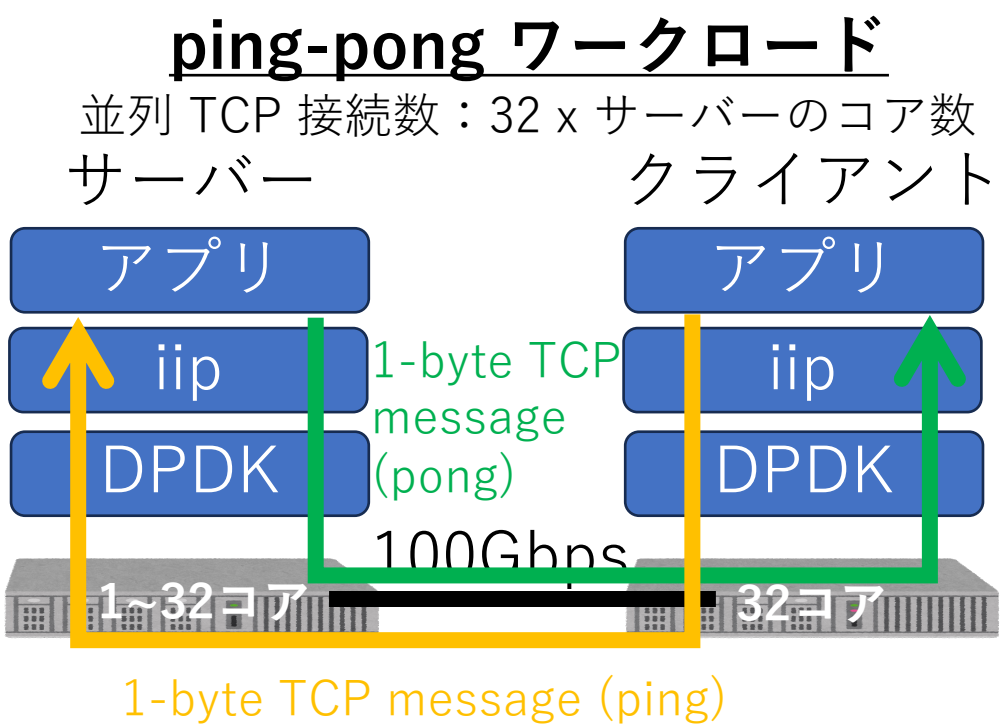


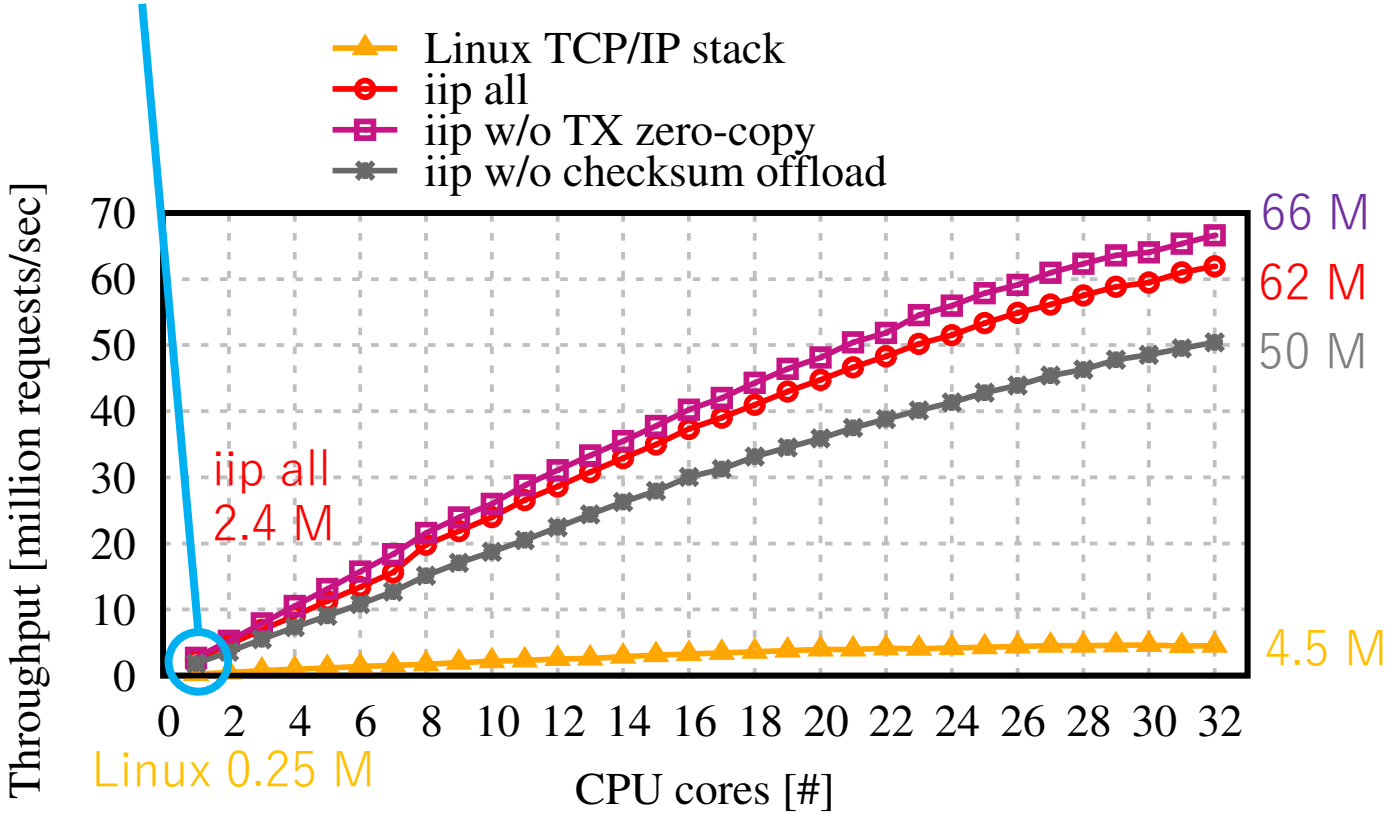


小さいメッセージの交換 (e.g., RPC)

- マルチコアでのスケーラビリティが重要

マルチコア環境を活かすことができない実装の性能の目安





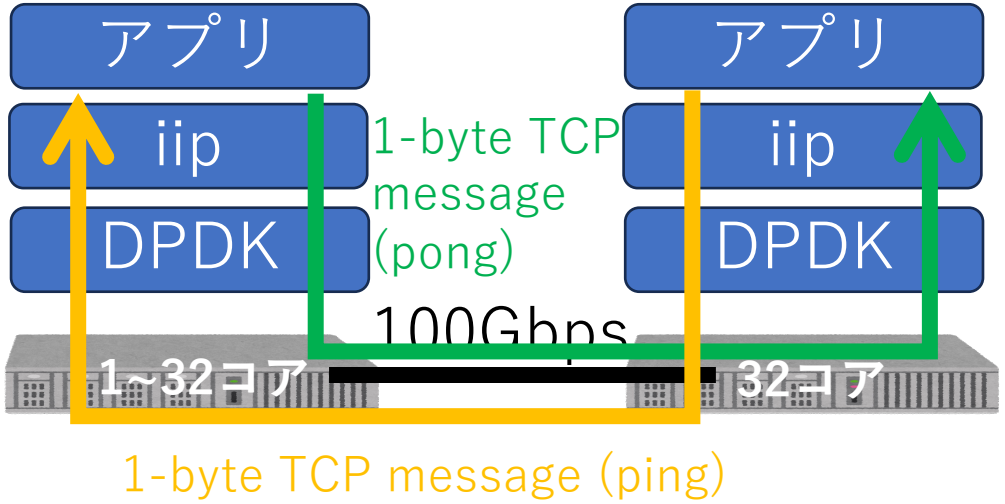
小さいメッセージの交換 (e.g., RPC)

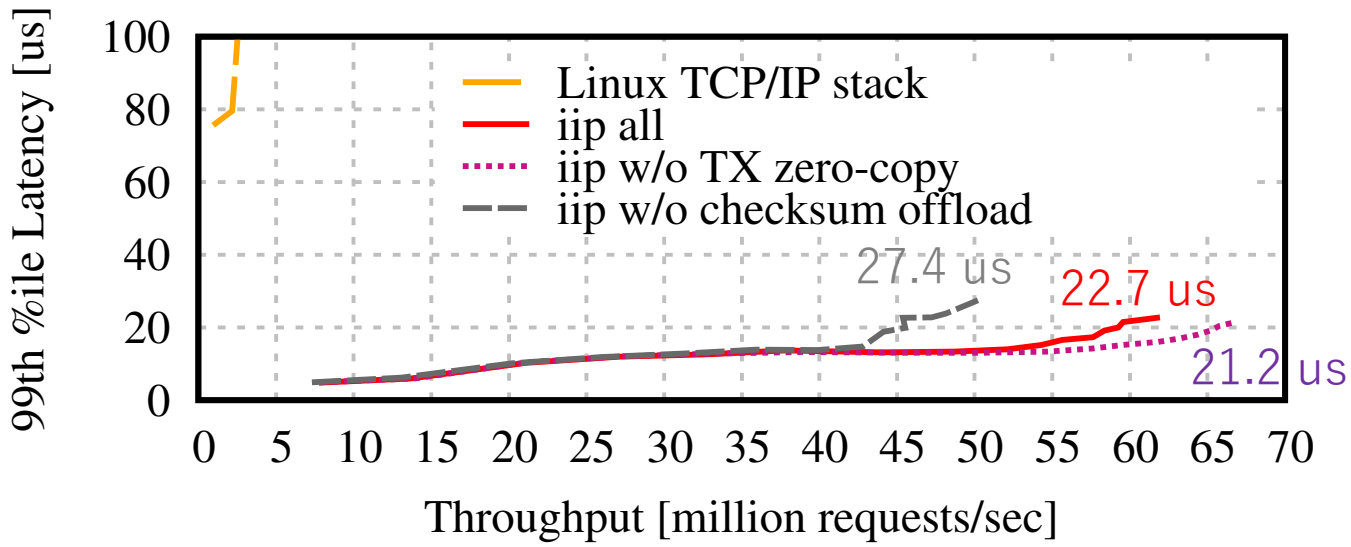
- マルチコアでのスケーラビリティが重要

32 CPU コア利用時の遅延対スループット

ping-pong ワークロード

並列 TCP 接続数：32 x サーバーのコア数





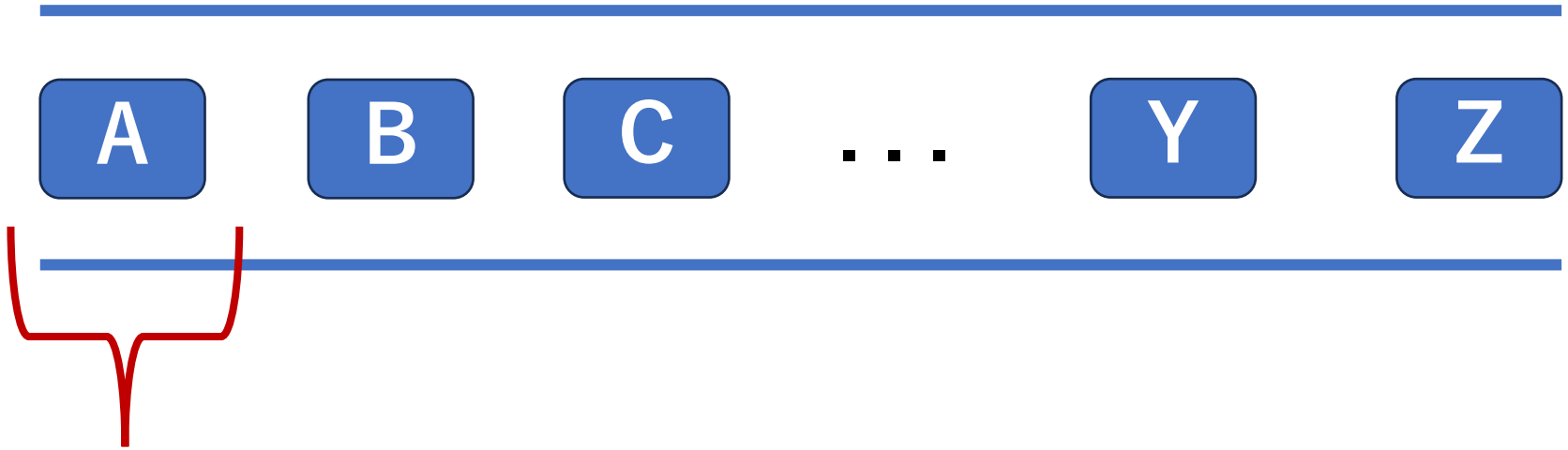
マルチコア環境でのスケーラビリティ

- 方針２：１CPUコアがアクセスするメモリ領域を小さくする

実装中にあったこと

改善後

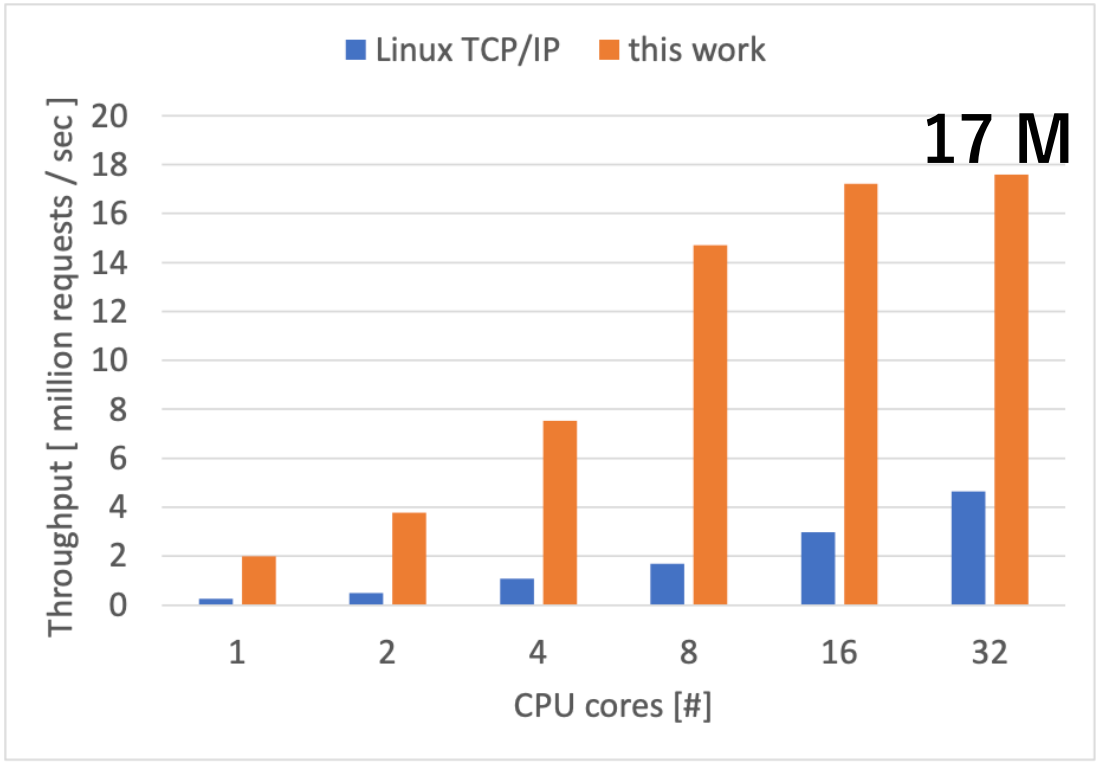
予めメモリ領域を確保したパケットを表現するデータ構造のリスト



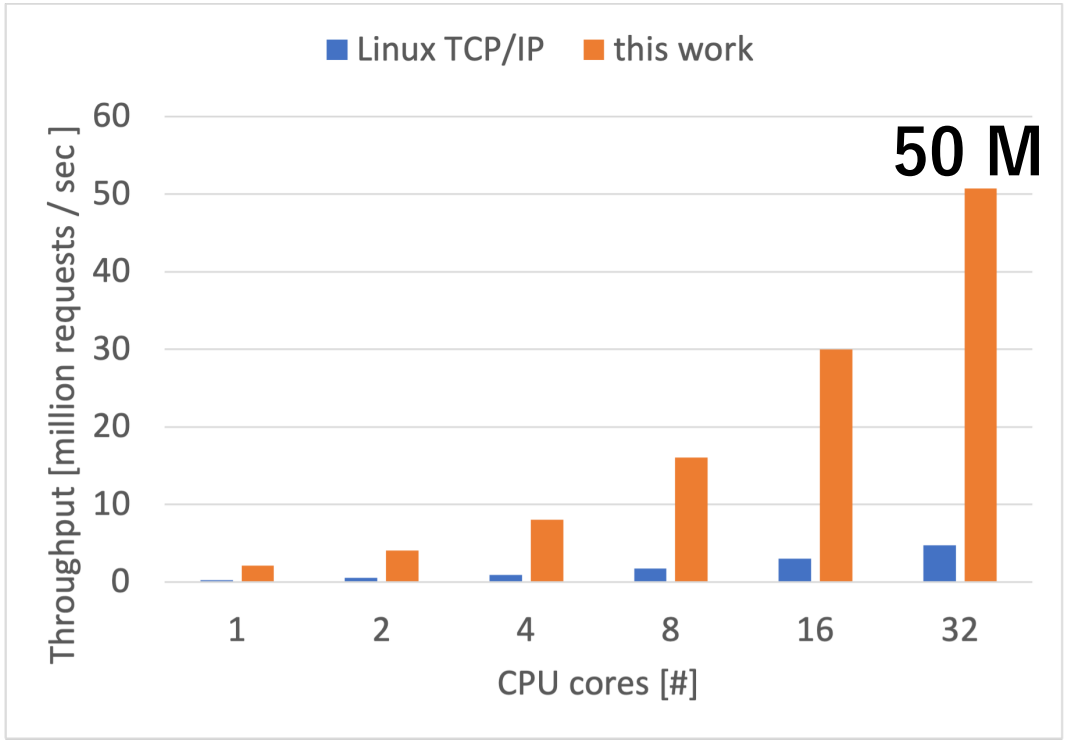
先頭に戻すと、なるべく同じアドレスのオブジェクトが利用できる
＝＞１CPUコアが利用するキャッシュサイズを減らすことができた

小さいメッセージの交換 (e.g., RPC)

開放時：末尾に追加



開放時：先頭に追加



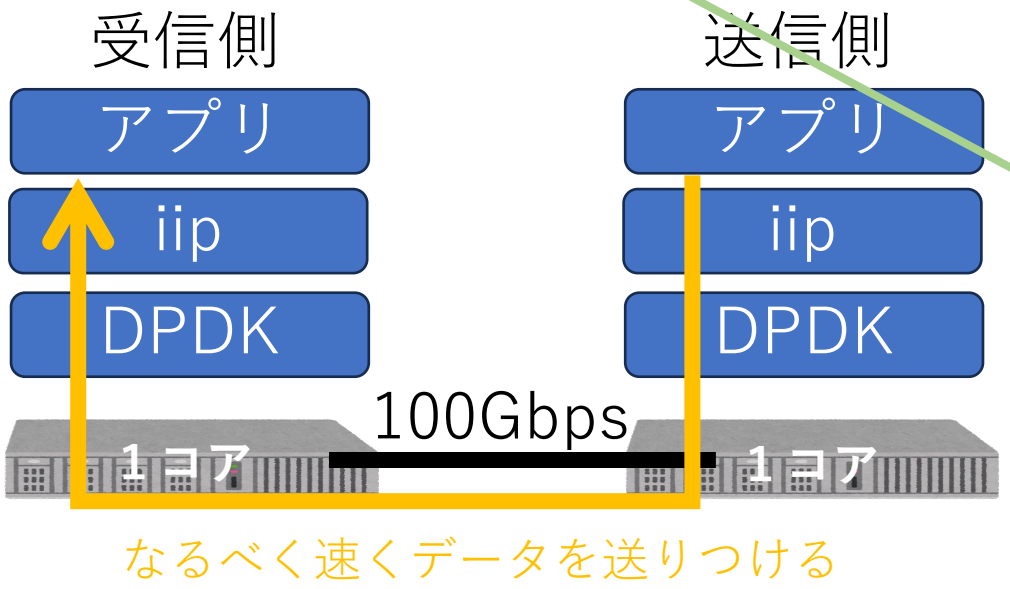
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要

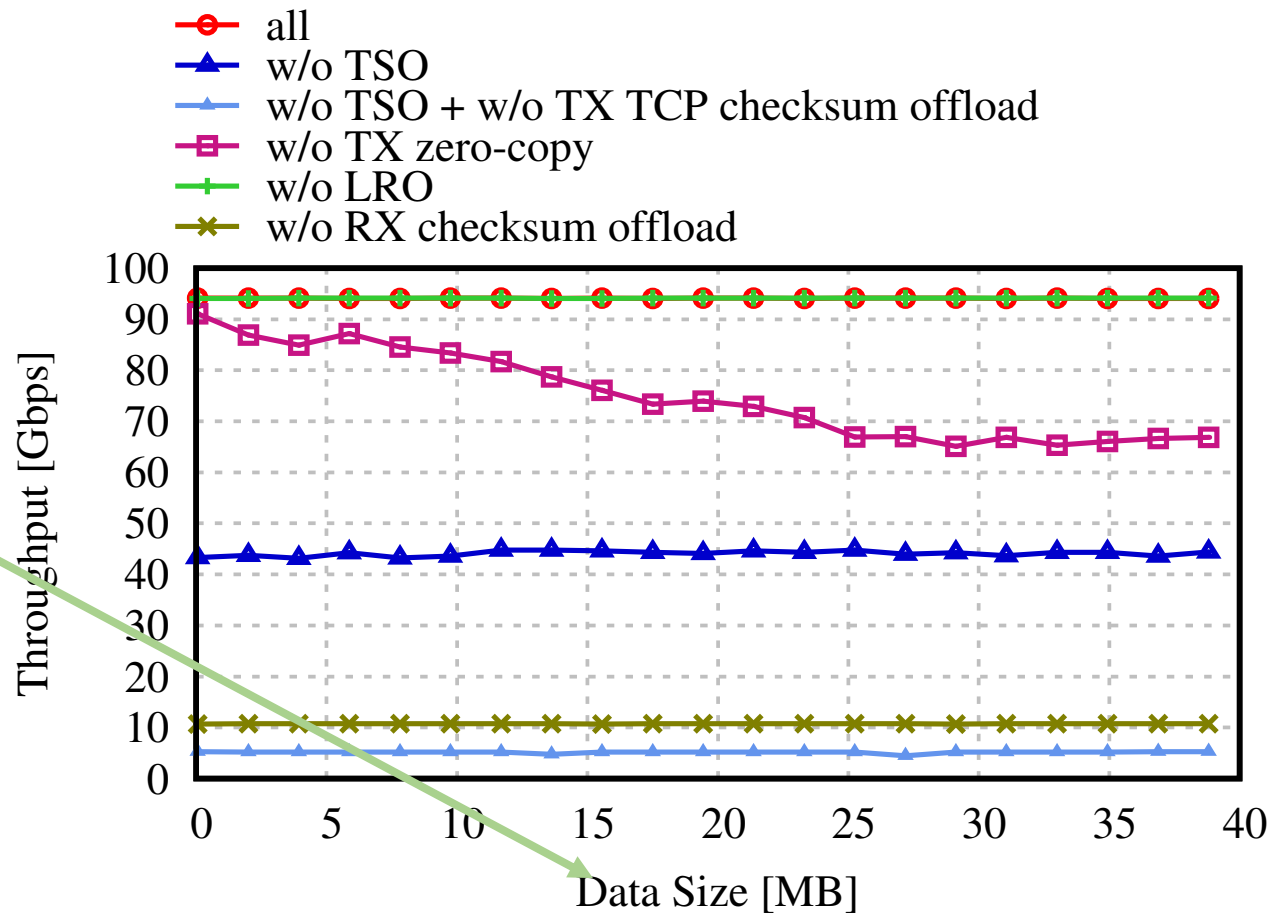
特定サイズのデータの送信が完了したら
再度同じデータを送る、の繰り返しを行う

データ移送ワークロード

並列 TCP 接続数：1

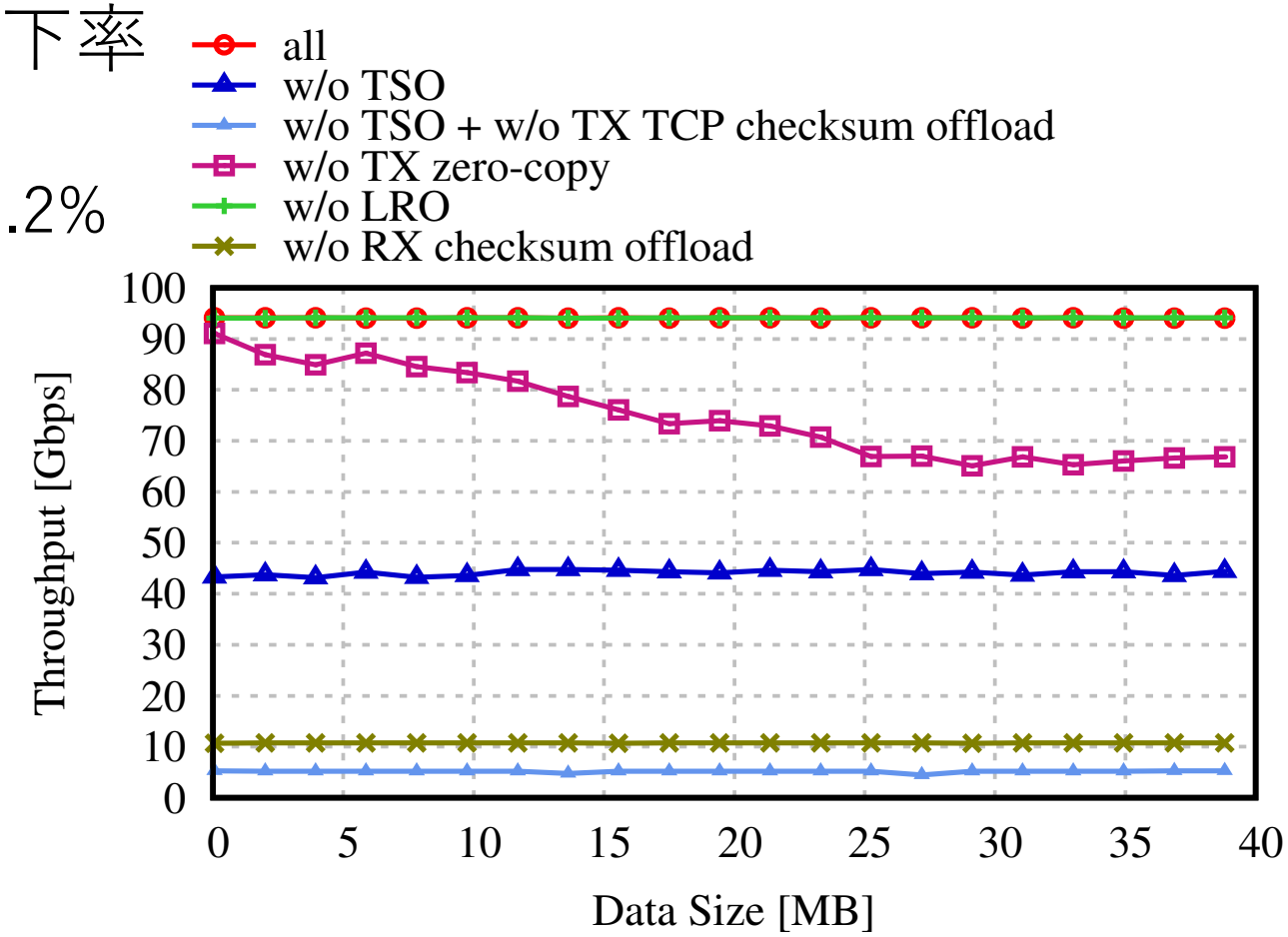


なるべく速くデータを送りつける



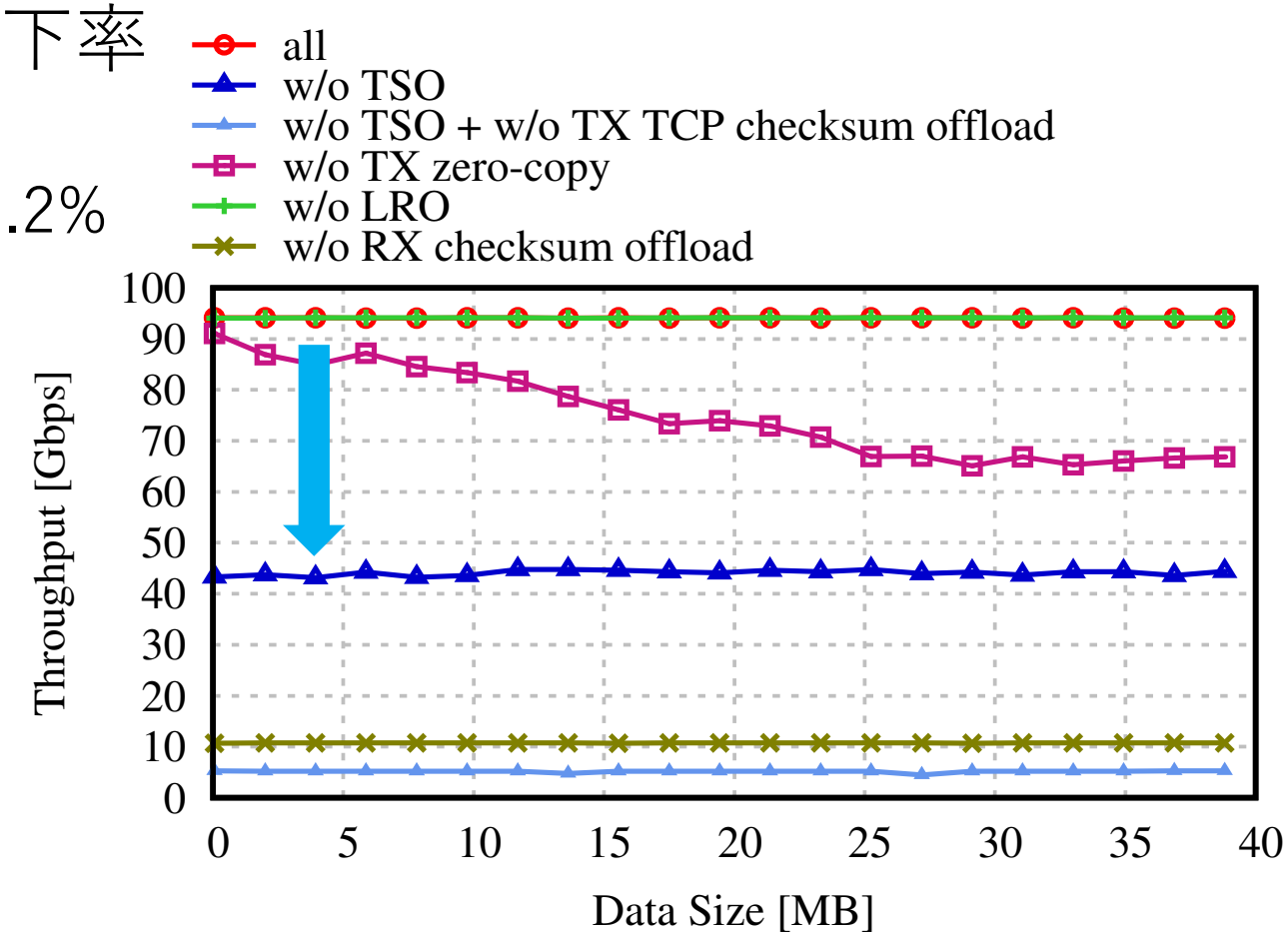
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - TSO : 54.1%
  - TSO + TX TCP checksum : 95.2%
  - ゼロコピー転送 : 30.8%
  - LRO : 0%
  - RX checksum : 88.6%



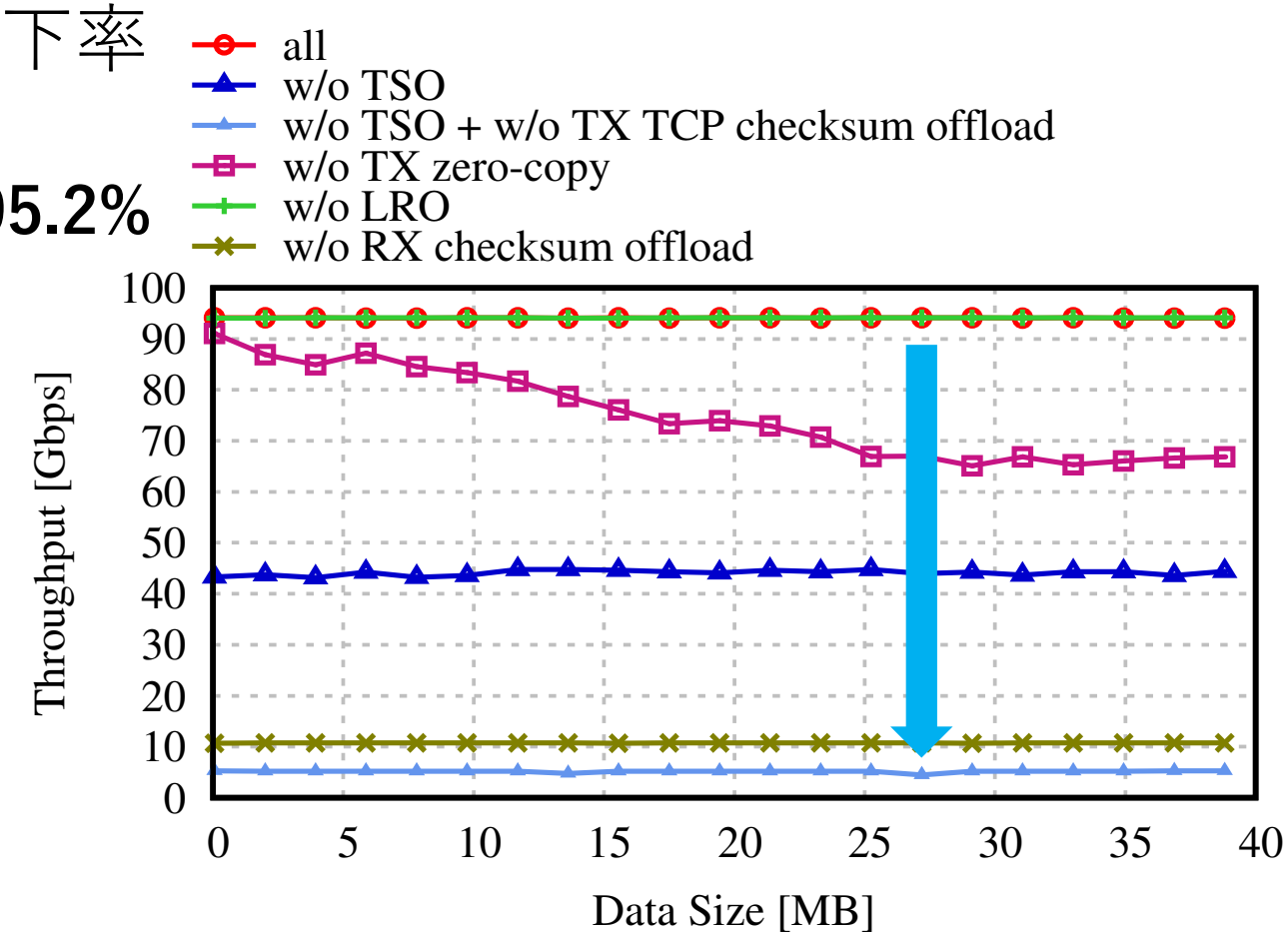
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - **TSO : 54.1%**
  - TSO + TX TCP checksum : 95.2%
  - ゼロコピー転送 : 30.8%
  - LRO : 0%
  - RX checksum : 88.6%



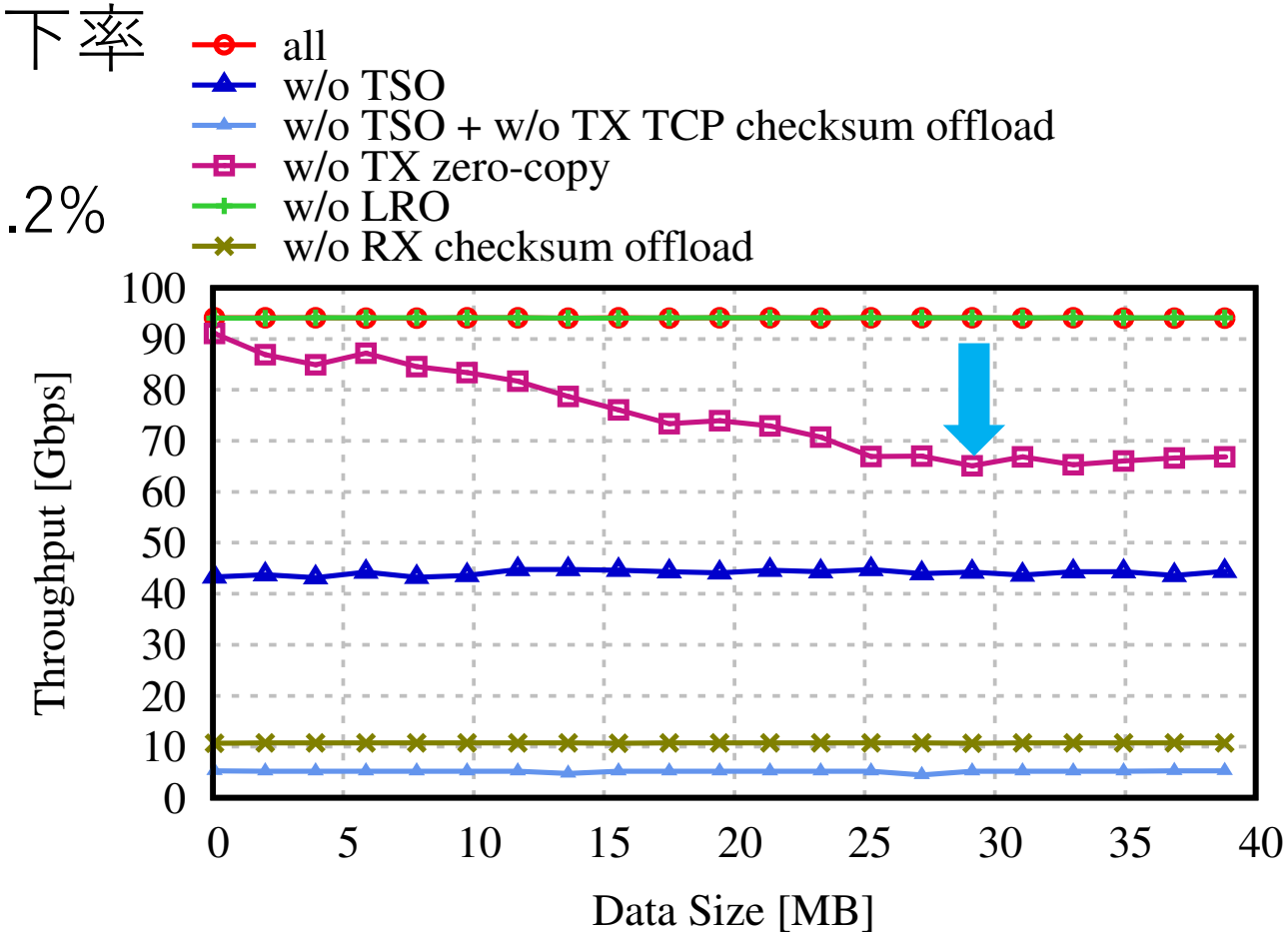
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - TSO : 54.1%
  - **TSO + TX TCP checksum : 95.2%**
  - ゼロコピー転送 : 30.8%
  - LRO : 0%
  - RX checksum : 88.6%



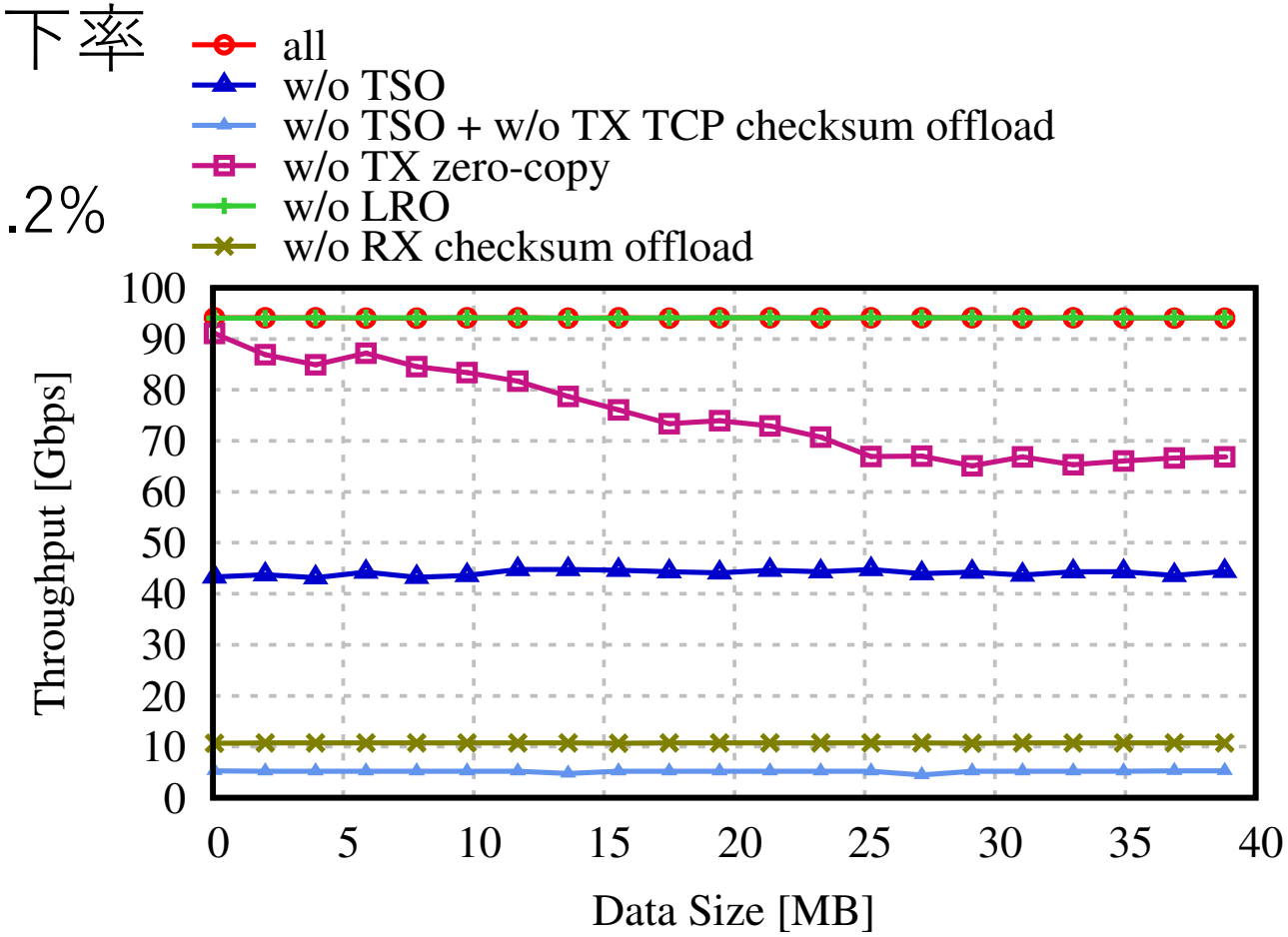
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - TSO : 54.1%
  - TSO + TX TCP checksum : 95.2%
  - **ゼロコピー転送 : 30.8%**
  - LRO : 0%
  - RX checksum : 88.6%



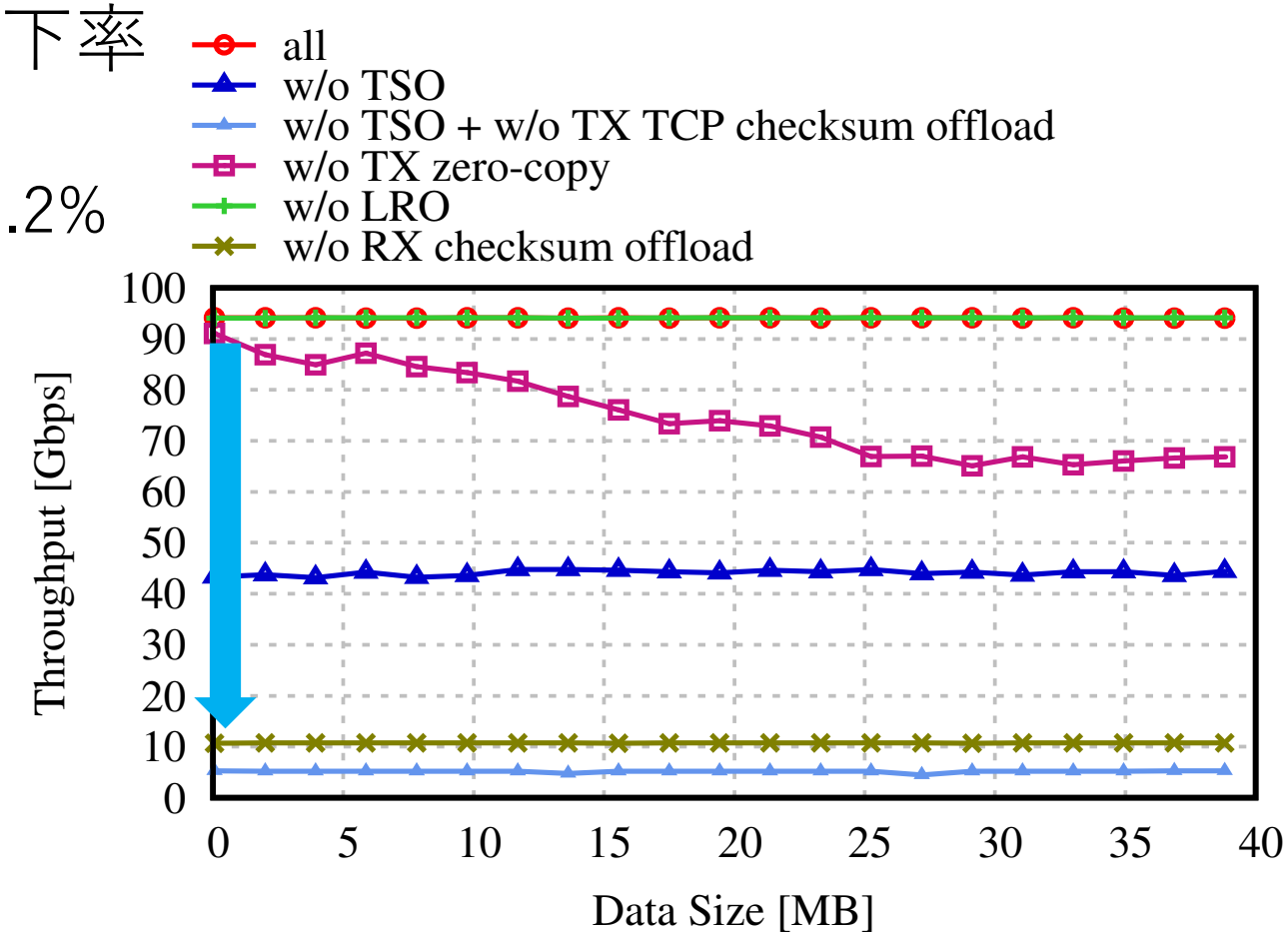
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - TSO : 54.1%
  - TSO + TX TCP checksum : 95.2%
  - ゼロコピー転送 : 30.8%
  - **LRO : 0%**
  - RX checksum : 88.6%



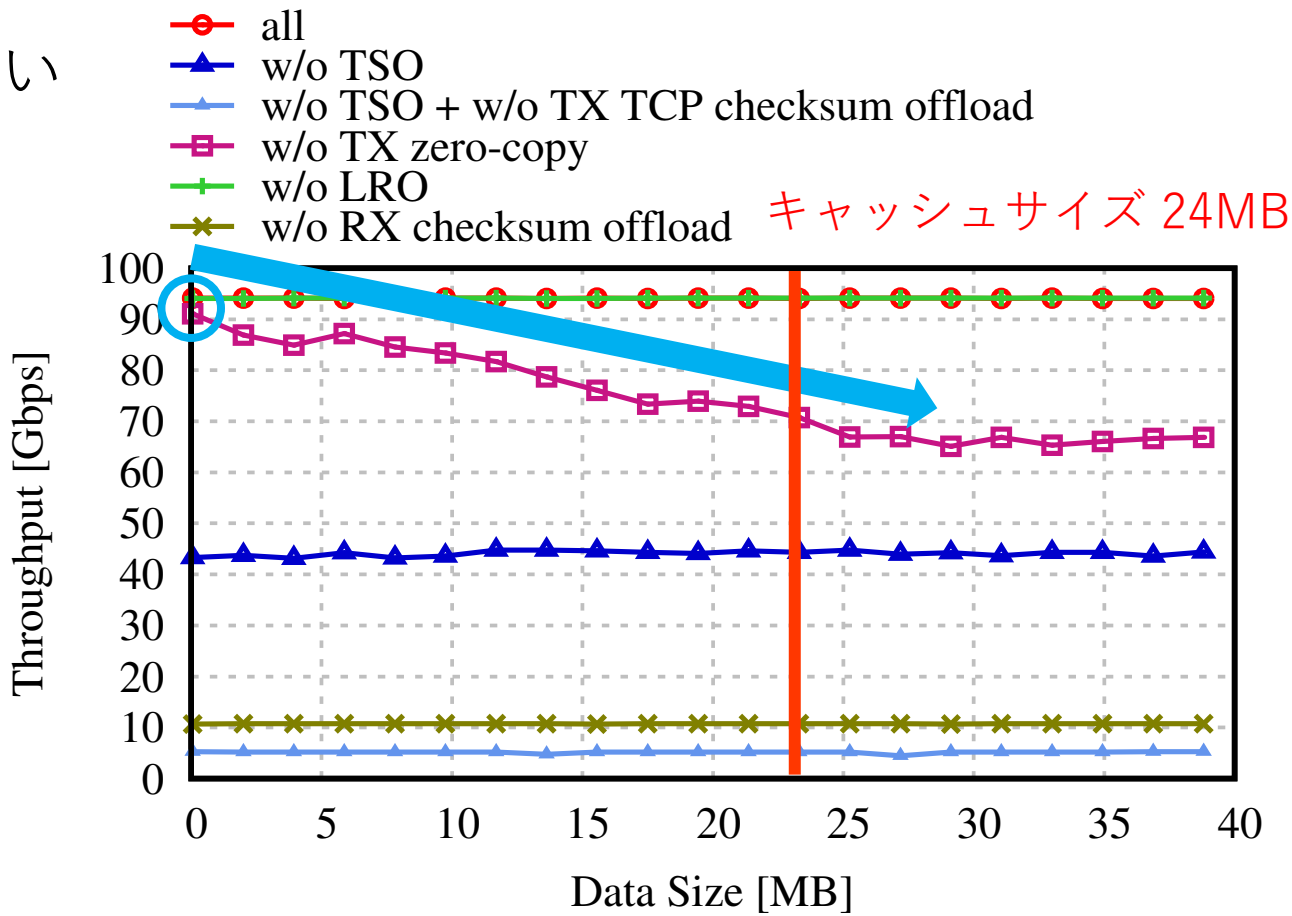
大きなデータの移動 (e.g., ファイル転送)

- NIC のオフロード機能とゼロコピー I/O が重要
- 各機能無効化時の最大性能低下率
  - TSO : 54.1%
  - TSO + TX TCP checksum : 95.2%
  - ゼロコピー転送 : 30.8%
  - LRO : 0%
  - **RX checksum : 88.6%**



大きなデータの移動 (e.g., ファイル転送)

- ゼロコピー転送無効の場合
  - データサイズが小さい場合は
    ゼロコピー有効の場合と遜色ない
  - 一定のデータサイズまで
    サイズ増加とともに
    スループットが低下



得られた知見

- 性能のために扱うデータがキャッシュに乗っていることが大事
  - そもそも CPU がアクセスしなければならないメモリ領域のサイズを小さく保つための配慮が必要
  - TCP/IP スタック実装がペイロードにアクセスしなくてよくなる点で NIC のオフロード機能とゼロコピー送受信は重要
- 計算量を抑えることも重要
  - TSO、checksum オフロードは性能に大きな影響あり
- データが小さければ Scatter Gather よりメモリコピーの方が速い
  - Cornflakes (SOSP'23) 論文にも同様の記載あり

まとめ

- 様々なシステムに組み込みやすく、高い性能を発揮できる TCP/IP スタック実装の模索
  - 既存の性能に最適化された実装の多くは組み込みやすさに課題があり
  - 既存のポータブルな実装は性能への配慮が十分でない

- 実装： https://github.com/yasukata/iip

  まだ作っている途中ですが、よろしければお試しください