ネットワーク及び演習

第 7 回 hey と HTTP サーバの負荷テスト

佐野 彰

講義パートは今回が最終回

本日の到達目標

- 負荷テストを**かけてよい相手と、かけてはいけない相手**を区別できる（今日いちばん大事）
- `hey` の `-n` と `-c` が何を指定しているかを説明できる
- `hey` の出力から、どこで時間がかかっているかを説明できる
- 自分で書いたサーバの性能を測り、結果から仮説を立てられる

> これまでは自分の PC の中だけでプログラムを動かしていた。今日は PC の外にもリクエストを送る。
> ツールの使い方より先に、**使ってよい範囲**を確認する。

1. 負荷テストのルール

ツールの使い方より先に、ルールを確認する

測ってよい相手・いけない相手

○ 測ってよい相手

① 自分の PC で動かしているサーバ

`http://localhost:50080/` まずここで測る。誰にも迷惑をかけない

② 自分が管理しているサーバ

自分で借りた VPS、自分の Raspberry Pi、同じ LAN 内の自分の PC など

③ この授業で許可された学内サーバ

`www.math.ryukoku.ac.jp` `whale2.math.ryukoku.ac.jp`

この 2 つだけ。ほかの学内ホストも対象外

× 測ってはいけない相手

- × 大学の公開サイト（www.ryukoku.ac.jp など）
- × 企業・公共機関・他人のサイト
- × 学内の他の実習室・研究室のマシン
- × 上の ①〜③ 以外のあらゆるサーバ

過去に、許可のないサーバへ過大なリクエストを送って大学から指導が入った例がある。

DoS 攻撃と判断されると、龍谷大学全体からそのサービスへアクセスできなくなることもある。

迷ったら localhost。許可された相手でも -n は 100 程度から始め、必要以上に繰り返さない。

なぜルールが要るのか

DoS 攻撃（Denial of Service attack）— サーバやネットワークに意図的に過剰な負荷をかけ、
そのサービスを使えなくする攻撃。情報セキュリティでいう「可用性」を侵害する行為。

- 負荷テストと DoS 攻撃は、コマンドとしては区別がつかない。違うのは**許可があるかどうか**だけ
- 相手のサーバの管理者に分かるのは、大学のネットワークから大量のリクエストが来たという事実だけ
  - 発信者個人ではなく、龍谷大学全体からのアクセスが遮断されることがある
- 実際に、許可のないサーバへ過大なリクエストを送って、大学から指導が入った例がある

この科目のルール: 負荷テストの宛先は、自分の PC 上のサーバ・自分が管理するサーバ・
`www.math.ryukoku.ac.jp` ・ `whale2.math.ryukoku.ac.jp` **のみ**。それ以外は禁止。

2. hey の使い方

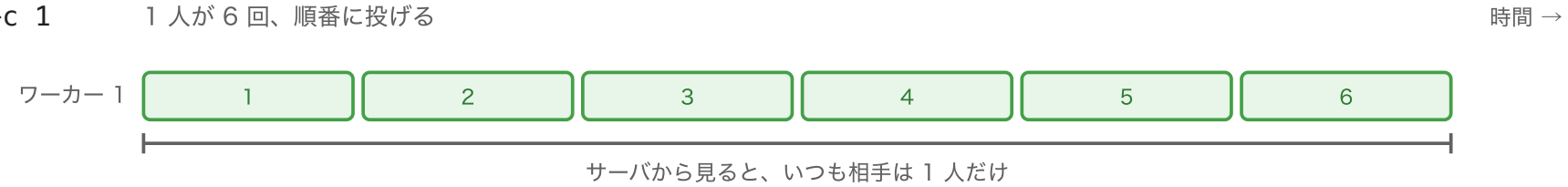
hey — HTTP サーバの性能測定ツール

- `ab`（Apache Bench）と同じ役割のツール。Go 製でクロスプラットフォーム
- https://github.com/rakyll/hey からコンパイル済みバイナリを入手できる
  - Windows: ダウンロードしたファイル名に `.exe` を付ける（`hey.exe`）
  - WSL / Linux は `apt`、macOS は `brew install hey` でも入る

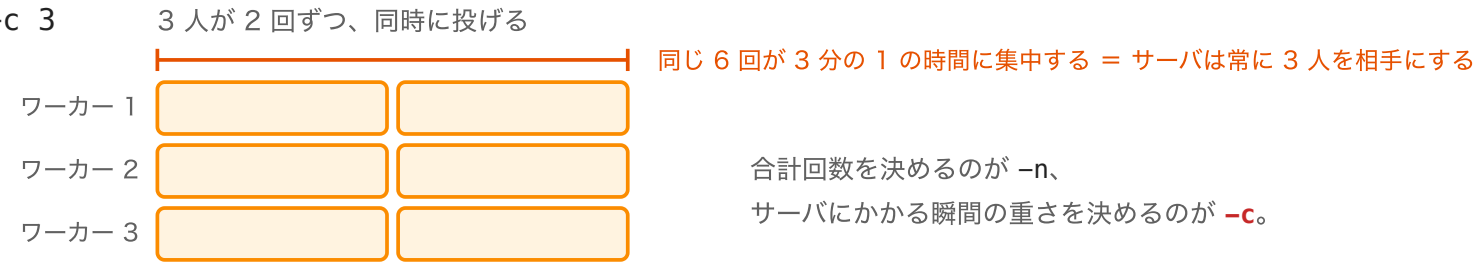
```
$ hey http://localhost:50080/server.html
Summary:
  Total:        0.3821 secs      # 全リクエストが終わるまでの合計時間
  Slowest:      0.0472 secs      # いちばん遅かった 1 回
  Fastest:      0.0032 secs      # いちばん速かった 1 回
  Average:      0.0189 secs      # 1 回あたりの平均
  Requests/sec: 523.4012         # 1 秒あたりに処理できたリクエスト数
```

-n と -c — 何を指定しているのか

`hey -n 6 -c 1`　1 人が 6 回、順番に投げる



`hey -n 6 -c 3`　3 人が 2 回ずつ、同時に投げる



合計回数を決めるのが **-n**、
サーバにかかる瞬間の重さを決めるのが **-c**。

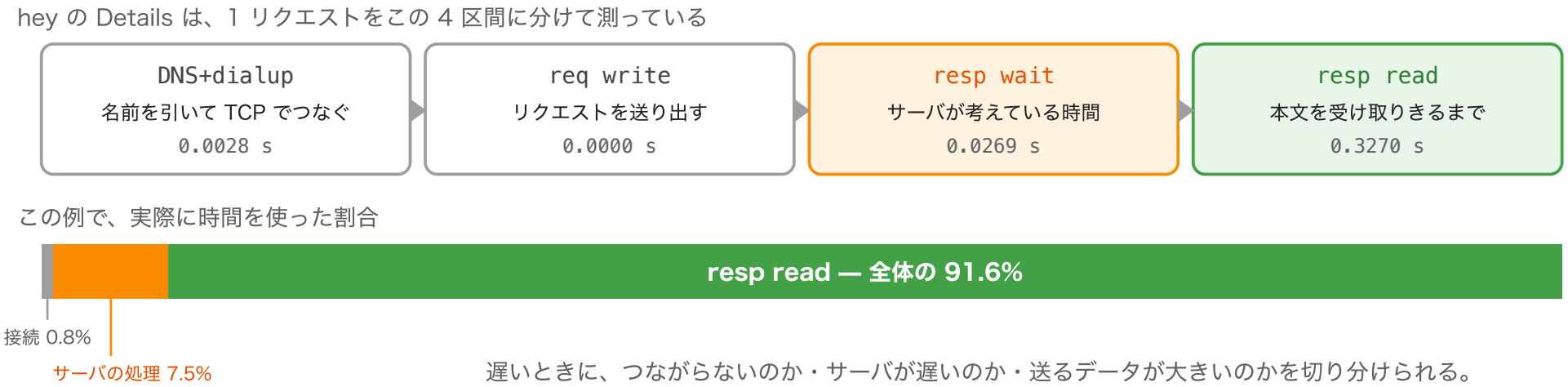
よく使うオプション

オプション	意味	既定値
`-n`	送るリクエストの**合計回数**	200
`-c`	**同時に**走らせるワーカー数	50
`-z`	回数ではなく**時間**で指定（`-z 10s`）	—
`-q`	ワーカー 1 つあたりの毎秒リクエスト数の上限	無制限
`-h2`	HTTP/2 を使う	off

```
$ hey -n 100 -c 5 http://localhost:50080/server.html
```

- 既定のまま実行すると `-c 50` になり、いきなり 50 並列でリクエストを送る。**必ず `-c` を明示する**
- 詳しくは `hey -h`

出力の読み方 — 1 リクエストの内訳



Details の読み方

```
$ hey -n 100 -c 5 http://localhost:50080/server100.html

Details (average, fastest, slowest):
  DNS+dialup: 0.0028 secs, 0.0011 secs, 0.0093 secs   # 接続するまで
  req write:  0.0000 secs, 0.0000 secs, 0.0001 secs   # 送信はすぐ終わる
  resp wait:  0.0269 secs, 0.0233 secs, 0.0411 secs   # 応答の最初のバイトが届くまで（ほぼサーバの処理時間）
  resp read:  0.3270 secs, 0.2592 secs, 0.4269 secs   # 本文の受信

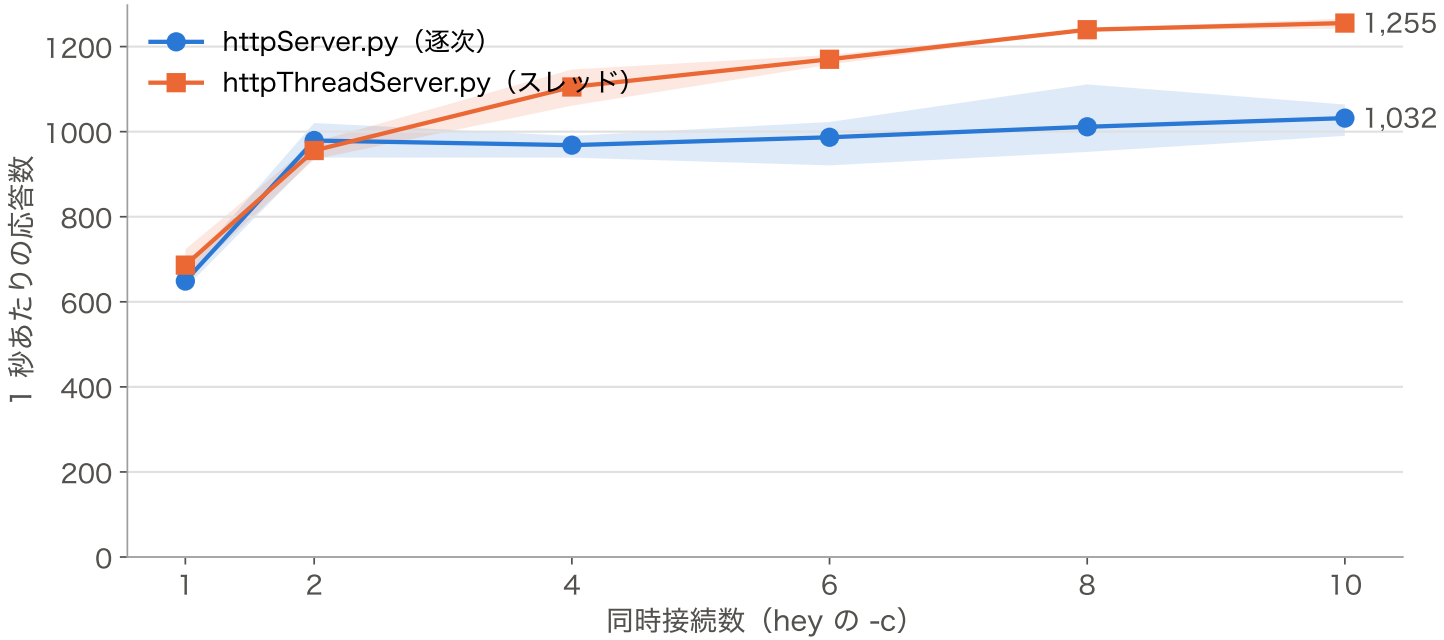
Status code distribution:
  [200] 100 responses                                  # 全部成功
```

- `resp wait` が大きい → サーバの処理に時間がかかっている（スレッド化で改善できる場面）
- `resp read` が大きい → 応答のデータが大きい、または回線の帯域が狭い
- `DNS+dialup` が大きい → 名前解決や接続確立に時間がかかっている
- `[200]` 以外が混じったら、**結果を読む前に原因を調べる**

3. 測ってみる

測定例その 1 — localhost

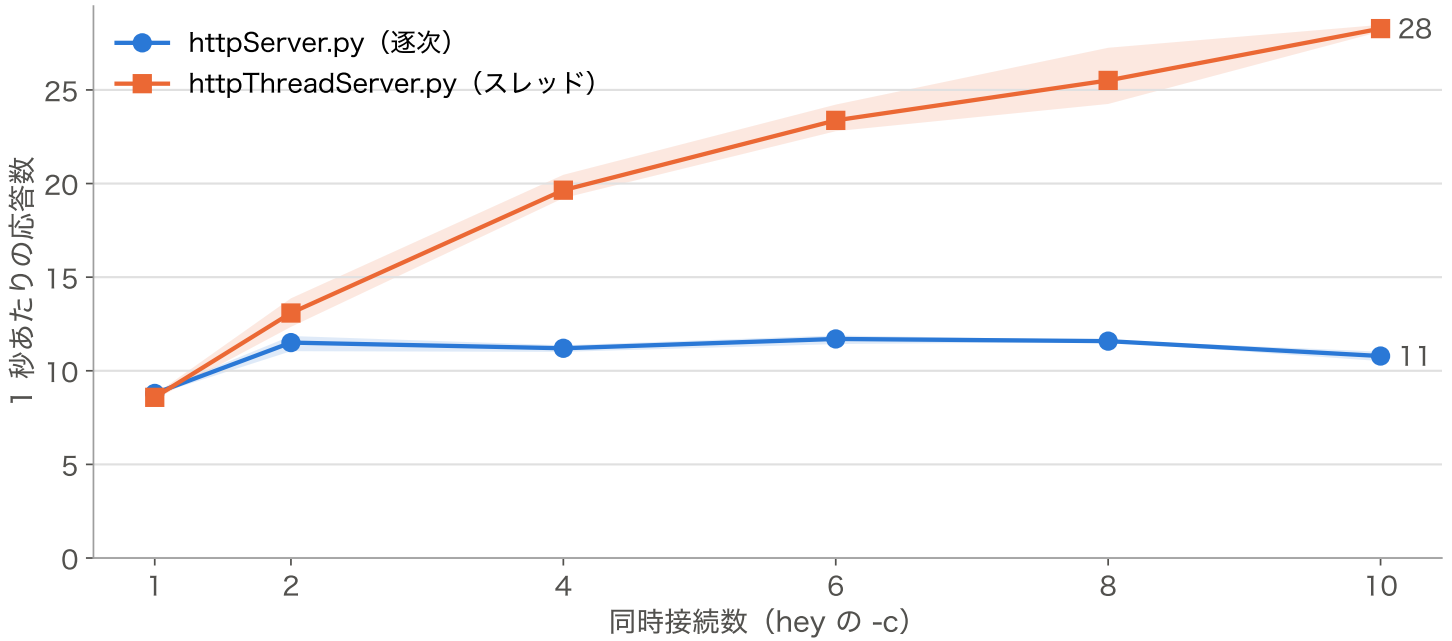
- 接続先: 自分の PC の `httpServer.py` と `httpThreadServer.py`（どちらも `listen(128)` にしておく）
- 条件: `server10000.html`（`server.html` を 10000 回繰り返した 1.1 MB）、N = 40、C = 1〜10、各 3 回



結果: どちらも 1 秒に 1000 件ほど。C = 10 でもスレッドは約 1.2 倍にとどまる（線は 3 回の平均、帯は最小〜最大）。

測定例その 2 — 同じ LAN 内の別の PC（Wi-Fi）

- サーバも条件もその 1 と同じ（1.1 MB、N = 40、C = 1〜10、各 3 回）
- 変えるのは `hey` を動かす場所だけ。同じ LAN の別の PC から Wi-Fi で接続する（往復 20〜100 ms）



結果: 逐次は 1 秒に約 11 件で頭打ちになる。スレッドは同時接続数とともに伸び、C = 10 で約 2.6 倍。

なぜその 2 ではスレッドが効いたのか

- 1.1 MB の応答は、OS の送信バッファ（128 KB）に一度には入らない。`sendall()` は、相手から「受け取った」の返事（ACK）を待ちながら少しずつ送るので、送り終わるまで戻ってこない
- 逐次サーバは、その待ち時間の間 `accept()` に戻れない。クライアントが何人いても、1 件ずつしか送れない
- スレッドなら、あるスレッドが ACK を待っている間に、別のスレッドが別のクライアントへ送れる。**待ち時間が重なる**（第 6 回の図のとおり）
- その 1 も同じ 1.1 MB を送るが、localhost は物理的な装置を通らないので、ACK がすぐに返る。待ち時間がほとんどないので、スレッドの差は小さい

> スレッドの効果を見るには、**サーバが何かを待つ条件**を作る。この例では、宛先を遠くし、送るものを大きくした。
> 条件を変えても差が出ないときは、「どこにも待ち時間がないのでは」と考える。

4. おまけのレポート課題

2 つのテーマから 1 つを選ぶ

	テーマ A: 測定実験	テーマ B: 新しいプロトコルの設計と実装
やること	`hey` で HTTP サーバの性能を測り、結果を説明する	新しいプロトコルを自分で設計し、クライアントとサーバを実装する
使うもの	`httpServer.py`・`httpThreadServer.py`・`hey`	Python のソケット（第 3〜6 回の内容）
レポートの中心	測定の条件・結果のグラフ・**結果の理由**	プロトコルの仕様・実装・**動かして確かめた結果**

- どちらも任意提出。提出方法は科目ページで案内する
- テーマ A の例は「3. 測ってみる」、テーマ B の設計例はこのあとのスライドで示す

テーマ A: 測定条件をいろいろ変えてみる

基本編

- リクエスト数（`-n`）、同時接続数（`-c`）を変える
- `httpServer.py` と `httpThreadServer.py` を比べる
- ネットワーク上の距離を変える — localhost ／ 同じ LAN 内の自分の PC ／ 許可された学内サーバ
- 送るファイルのサイズを変える（`server.html`、`server100.html`、1 MB ほどのファイル）

応用編

- 待受けキューのサイズ（`listen()` の引数）、受信バッファのサイズ（`recv()` の引数）
- サーバ側・クライアント側のハードウェア（メモリ、ディスク速度）
- 自分のサーバの処理能力の上限を探る、時刻や曜日による違いを調べる

> 上限を探る実験は、**自分が管理しているサーバに対してのみ**行うこと。

テーマ A: 測定結果から何をみるか

- ネットワークも計算機もほかの通信や処理の影響を受けるので、結果は必ず**ばらつく**
  - 1 回の測定で結論を出さない。複数回測って、平均・分散・最大最小を見る
- 予想と違うデータに注目する
  - 条件を変えても結果が変わらない → なぜ？（どこかが頭打ち？ 測れていない？）
  - あるところで急に値が変わる → なぜ？（キューがあふれた？ 別の要因？）
- 「おかしい」と言うためには、おかしくない結果とはどういうものかを先に説明できる必要がある
  - この説明がレポートの中心になる。グラフを貼るだけでは説明にならない

テーマ B: 新しいプロトコルを設計・実装する

HTTP は「どんな文字列を、どの順に送り合うか」の取り決めだった（第 5 回）。同じように、新しい取り決めを自分で設計する（既存のプロトコルを使うのではない）。

1. **目的を決める** — 何をするためのやり取りか（メモの保存、チャット、ゲーム、ファイル転送 など）
2. **仕様を書く** — メッセージの形式、送り合う順序、エラーのときの返事
3. **実装する** — クライアントとサーバを、Python のソケットで書く
4. **確かめる** — 正しい使い方だけでなく、間違った命令や途中で切れた接続も試す

レポートに書くこと

- 仕様（メッセージの一覧、やり取りの順序の図、エラーの決め方）と、そう決めた理由
- クライアント・サーバのコードと、実行例
- 試した結果と、仕様の限界（何ができないか、どう直せるか）

設計例: メモを預ける MEMO プロトコル

名前を付けてメモをサーバに預け、あとで取り出す。

クライアント → サーバ（命令）

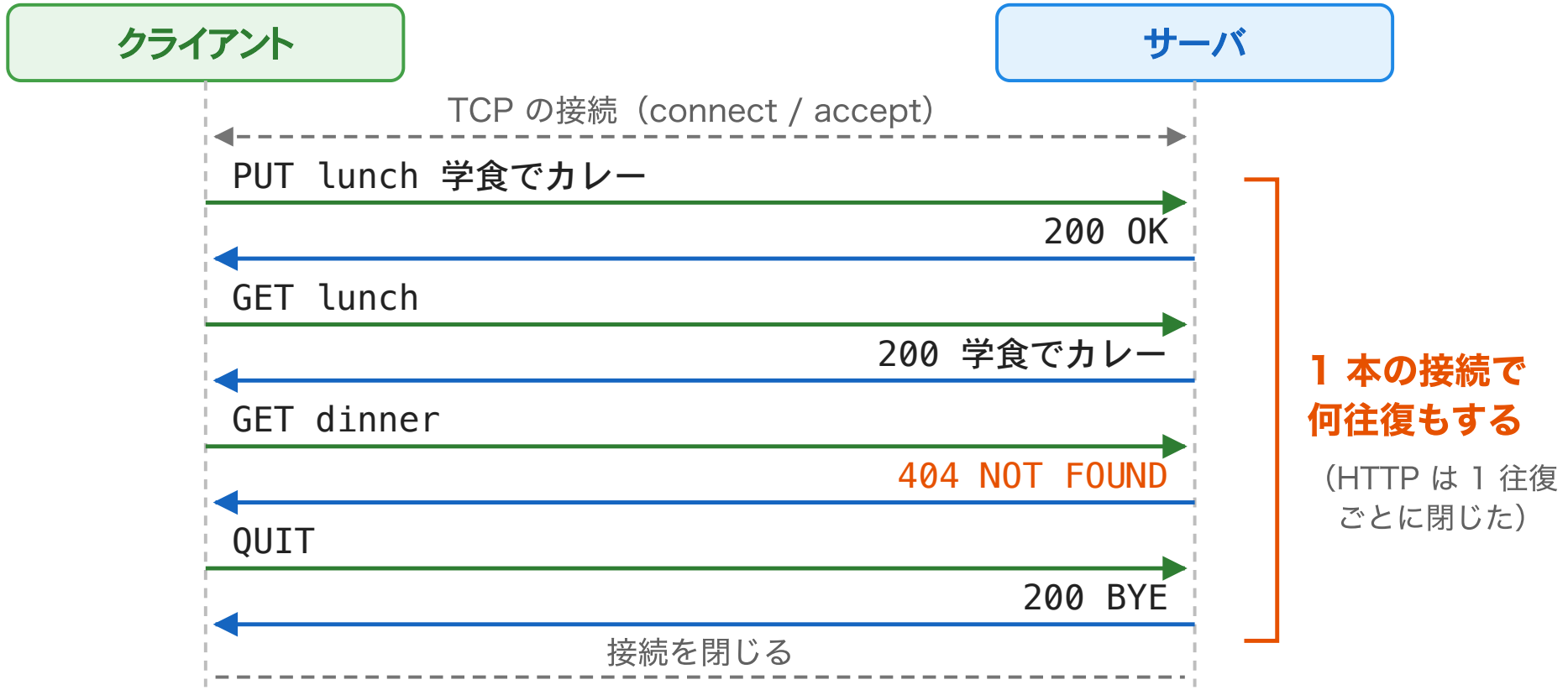
命令	意味
`PUT 名前 内容`	メモを預ける
`GET 名前`	メモを取り出す
`QUIT`	やり取りを終える

サーバ → クライアント（返事）

返事	意味
`200 OK` ／ `200 内容`	成功
`404 NOT FOUND`	その名前のメモがない
`400 BAD REQUEST`	命令の形が正しくない
`200 BYE`	終了（このあと接続を閉じる）

- 命令も返事も **UTF-8 の 1 行**（末尾は `\n`）。返事の先頭の数字は HTTP のステータスコードをまねた

設計例: やり取りの順序


クライアント
サーバ
TCP の接続（connect / accept）
PUT lunch 学食でカレー
200 OK
GET lunch
200 学食でカレー
GET dinner
404 NOT FOUND
QUIT
200 BYE
接続を閉じる
1 本の接続で
何往復もする
（HTTP は 1 往復
ごとに閉じた）

設計例: 決めておくこと

どんなプロトコルでも、次のことを仕様として決めておく。MEMO ではこう決めた。

- **メッセージの区切り** — TCP はバイトの流れで、`recv()` は送った単位で届くとは限らない（第 5 回）。MEMO は「1 行 = 1 メッセージ」とした
- **文字コード** — MEMO は UTF-8 とした。決めないと日本語が化ける
- **エラー** — 知らない命令や形の正しくない命令に、どう返すか。MEMO は `400` を返し、接続は切らずに次の命令を待つ
- **接続の終わり** — どちらが、どの合図で切るか。相手が何も言わずに切ったらどうするか
- **同時接続** — 複数のクライアントを受け付けるか。データを全員で共有するか

> 仕様を文章で書いておけば、別の人が書いたクライアントとも通信できる。それがプロトコルを決める意味。

テーマ B: 設計のヒント

- **状態を持たせる** — ログインしてから使える、じゃんけんや数当てのように順番が決まっている
- **複数のクライアントをつなぐ** — チャット（1 人の発言をほかの全員に送る）
- **異常に備える** — 長すぎる行、途中で切れた接続、同じデータを 2 人が同時に書き換えたとき
- **相互運用を試す** — 仕様だけを友人に渡し、相手が書いたクライアントと自分のサーバで通信できるか

> サーバを動かすのは**自分の PC か自分が管理するサーバ**で。ポートを開けたまま放置しない。

お知らせ・今日やること

次回（11/13）：まとめの小テスト（4 講時 15:30〜16:30、1-608 対面。科目ページの ex8 を参照）

- 講義パート・通常の小テスト・演習課題はありません

今日の 5 講時（実習）

1. `hey` をインストールし、自分の `httpServer.py` に対して負荷テストを行う
   - 実行したコマンドと出力を含むスクリーンショットを、aiJudge の **ex7** へ提出
2. aiJudge の **ex7** の `threadList.py`（スレッドをリストにためて全部 join する）に取り組む
   - ex7 は**まとめの小テストのリハーサル**。1-608 の実習室 PC の Web エディタ（画面全体を共有）からだけ提出できる（締切 11/13 0:00）。hey の画像は実習室 PC に移してから提出する
   - まとめの小テストでは自分のコードを参照できる。今のうちに実習室 PC へコピーしておく
3. 任意提出のレポート課題（テーマ A・B から 1 つ）について、質問があれば TA・教員へ

> 宛先は自分の PC（127.0.0.1）／ 自分が管理するサーバ ／ `www.math.ryukoku.ac.jp` ／ `whale2.math.ryukoku.ac.jp` **のみ**。

質問は **Teams「★ 質問用チャネル」** か、教室の TA へ。