ネットワーク及び演習

第 6 回 スレッドとネットワーク通信の暗号化

佐野 彰

本日の到達目標

スレッド

- `accept()` や `recv()` が止まって待つこと（Blocking I/O）で何が起きるかを説明できる
- `threading.Thread` で関数を別のスレッドとして実行できる
- `httpServer.py` を `httpThreadServer.py` に書き換えられる

暗号化

- 共通鍵暗号と公開鍵暗号の違いを、鍵の配り方の観点から説明できる
- HTTPS が「相手の確認」と「暗号化」をどの仕組みで実現しているかを説明できる
- パスワードとパスキーの違いを、サーバに何を預けるかの観点から説明できる

1. Blocking I/O とスレッド

1 つのクライアントしか処理できないサーバを、どう直すか

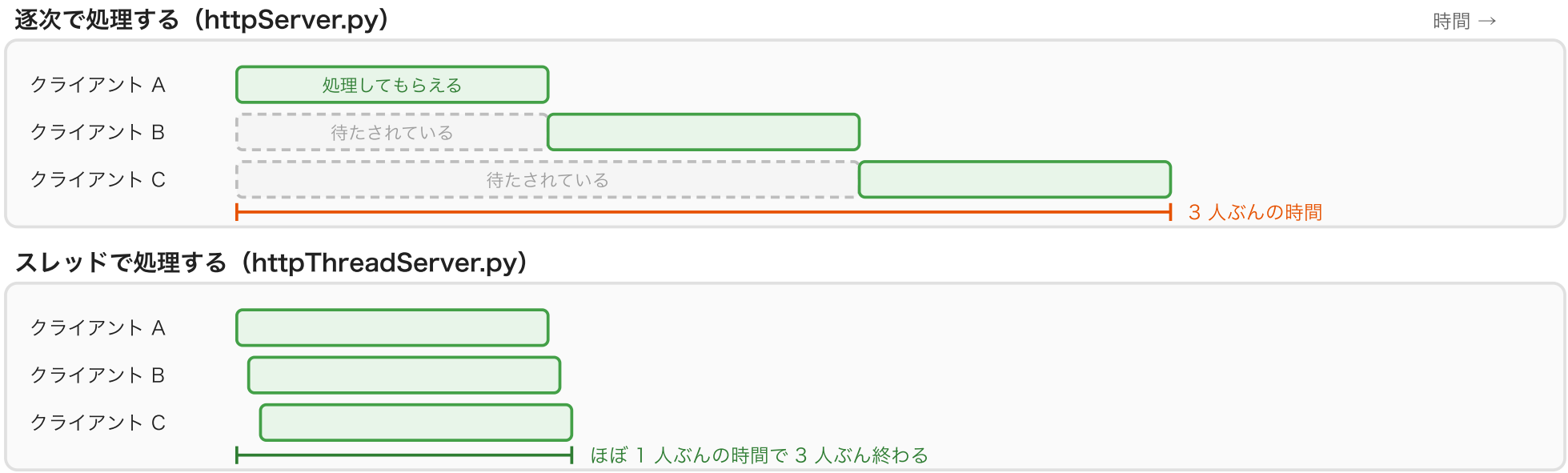
ソケットの呼び出しは「止まって待つ」

```
while True:
    conn, addr = s.accept()      # 接続が来るまで、ここで止まる
    with conn:
        data = conn.recv(1024)   # データが来るまで、ここで止まる
        conn.sendall(rData)
```

- 呼び出した処理が終わるまで次の行に進まない。これを Blocking I/O という
- 上から順に読めるので分かりやすい。ただし、待っている間はほかの処理ができない
- サーバが 1 つ目の接続を処理している間は、2 つ目の接続を受け付ける `accept()` が呼ばれない

解決方法は 2 つ。スレッドで処理の流れを増やす（今日扱う）か、ノンブロッキング I/O とイベントループで待たずに済む書き方にするか（この科目では扱わない。`asyncio` などがこちら）。

待たされている様子



スレッドは 1 件を速くするわけではない。待っている間に別の件を進められるようにするだけ。

threading — 関数を別のスレッドで実行する

```python
import threading, time

def work(name, sec):
    print(name, 'start')
    time.sleep(sec)
    print(name, 'done')

t1 = threading.Thread(target=work, args=('A', 3))
t2 = threading.Thread(target=work, args=('B', 1))
t1.start()
t2.start()
t1.join()      # t1 が終わるまで待つ
t2.join()
```

- `Thread(target=関数, args=引数のタプル)` で作る
  - **関数を呼ぶのではなく、関数そのものを渡す**（`work` であって `work()` ではない）
- `start()` でスレッドが動き始める。呼び出した側はすぐ次の行へ進む
- `join()` は、そのスレッドが終わるまで待つ
- 実行順は保証されない。上の例では、待ち時間の短い B が先に終わる

> 第 3 回の「なぜ関数にするのか」で予告したとおり、処理を関数にまとめておくと、スレッドに関数ごと渡せる。

httpThreadServer.py — 1 接続ぶんの処理を関数にしてスレッドに渡す

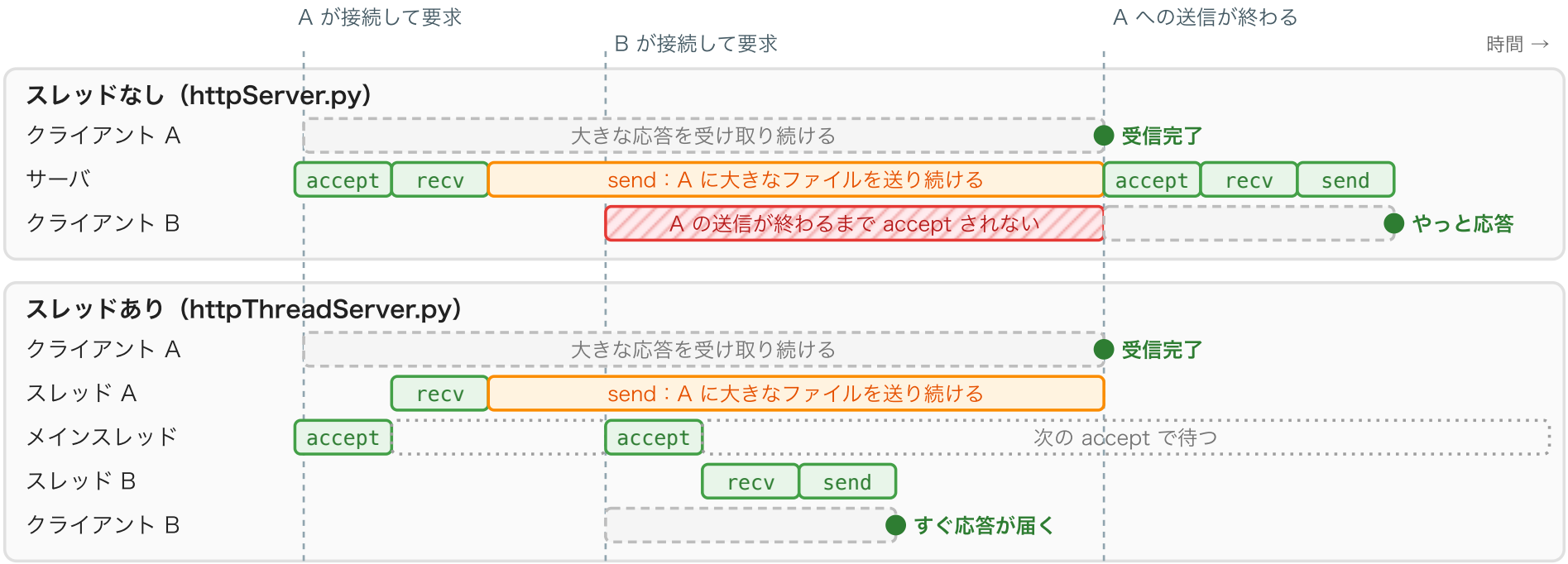
```
def httpResponse(connection, address):
    with connection:
        print('Connected by', address)
        # ... (httpServer.py の中身をそのまま移しただけ)

if __name__ == '__main__':
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        s.bind((HOST, PORT))
        s.listen(128)                  # 待たせておける接続の数を増やす
        while True:
            conn, addr = s.accept()
            threading.Thread(target=httpResponse, args=(conn, addr)).start()
```

- 接続を受けたあとの呼び出しを 2 行に替えただけで、受信・送信の処理は `httpServer.py` と同じ
- `accept()` で受け付けた接続をすぐスレッドに渡し、次の `accept()` に戻るのがポイント

どこで止まるか — スレッドなしとスレッドあり

A に大きなファイルを送っている途中で、B が要求を送った場合を比べる。



- スレッドなし: 流れが 1 本なので、A への送信が終わるまで B の `accept()` に進めない
- スレッドあり: スレッド A が送信を続ける間に、スレッド B が B の送受信を進める

スレッドで気をつけること

- 実行の順番は決まらない。出力が混ざったり、実行のたびに順序が変わったりする
- 複数のスレッドが同じ変数を同時に書き換えると、値が壊れることがある。カウンタの加算も安全ではない
  - 共有する変数は `threading.Lock()` で保護する
- スレッドは無限には作れない。接続のたびにスレッドを作ると、数千接続で処理しきれなくなる
  - 実際のサーバはスレッドを使い回す（**スレッドプール**）
- 標準の CPython では、スレッドを使っても CPU の計算は速くならない（GIL のため）
  - ただし、ネットワークやファイルの待ち時間は重ねられるので、サーバの用途では効果がある

> 今日の目的は、待ち時間を重ねられるようにすること。計算を速くすることではない。

2. 暗号化と HTTPS

途中で読まれても内容が分からない形で送る

なぜ暗号化が要るのか

- データは、バケツリレーのように経路上のたくさんの機器を順に渡って届く（第 2 回の階層の話）
- 各機器は通常、自分の層のヘッダしか見ない。ただし、中身を**読むことはできる**
  - 経路上の機器の管理者や、同じ Wi-Fi にいる誰かが読める
- パスワードやカード番号を平文のまま送ると、経路の途中で読まれてしまう

> そのため、送信側が送る前に暗号化し、受信側が受け取ってから復号する。経路の途中では中身を読めない。
> `http://` で送ったパスワードは平文。`https://` なら暗号化されている。

共通鍵暗号と公開鍵暗号

共通鍵暗号

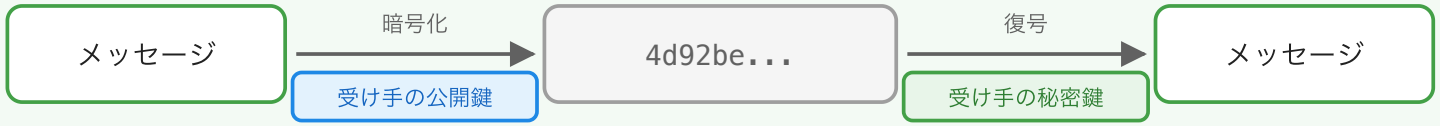
同じ鍵で施錠も解錠もする



速い。ただし、その鍵を相手にどうやって安全に渡すのかという問題がまるごと残る。

公開鍵暗号

施錠用と解錠用が別の鍵



公開鍵は誰に見られてもよい（配れる）。秘密鍵は本人が持ったまま動かさない。だから鍵配送の問題が解ける。

RSA — 鍵を作る（手順と小さな例）

Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman（1977）。左の手順を、右の小さな数で追う。

	手順	例
1	素数 p, q を選び、n = pq とする	p = 5, q = 11 → **n = 55**
2	L = LCM(p − 1, q − 1) を求める	L = LCM(4, 10) = **20**
3	GCD(r, L) = 1 となる r を選ぶ → **公開鍵 (n, r)**	GCD(3, 20) = 1 → **公開鍵 (55, 3)**
4	rs ≡ 1 (mod L) となる s を求める → **秘密鍵 (n, s)**	3 × 7 = 21 ≡ 1 (mod 20) → **秘密鍵 (55, 7)**

- 公開するのは n と r だけ。s を求めるには L が要り、L を求めるには p と q が要る
- n = 55 なら 5 × 11 とすぐ分かる。実際の n は 2048 ビット（10 進で約 617 桁）以上あり、現実的な時間では素因数分解できない。これが安全性の根拠

RSA — 暗号化と復号（手順と小さな例）

鍵は前のスライドの例を使う。**公開鍵 (n, r) = (55, 3)**、**秘密鍵 (n, s) = (55, 7)**

	手順	例（m = 8 を送る）
暗号化（**公開鍵** r で）	`c = m ** r % n`	`8 ** 3 = 512`、512 を 55 で割った余りは **17** → 17 を送る
復号（**秘密鍵** s で）	`m = c ** s % n`	`17 ** 7 = 410338673`、55 で割った余りは **8** → 元に戻る
公開鍵では戻せない	`c ** r % n`	`17 ** 3 = 4913`、55 で割った余りは 18 → 8 にならない

- 送る数 m は n より小さい整数。文字列は `encode()` でバイト列にしてから数として扱う
- Python では `pow(8, 3, 55)` と書くと、巨大な数を作らずに余りを求められる

```
>>> pow(8, 3, 55), pow(17, 7, 55)
(17, 8)
```

RSA — 署名（鍵を逆向きに使う）

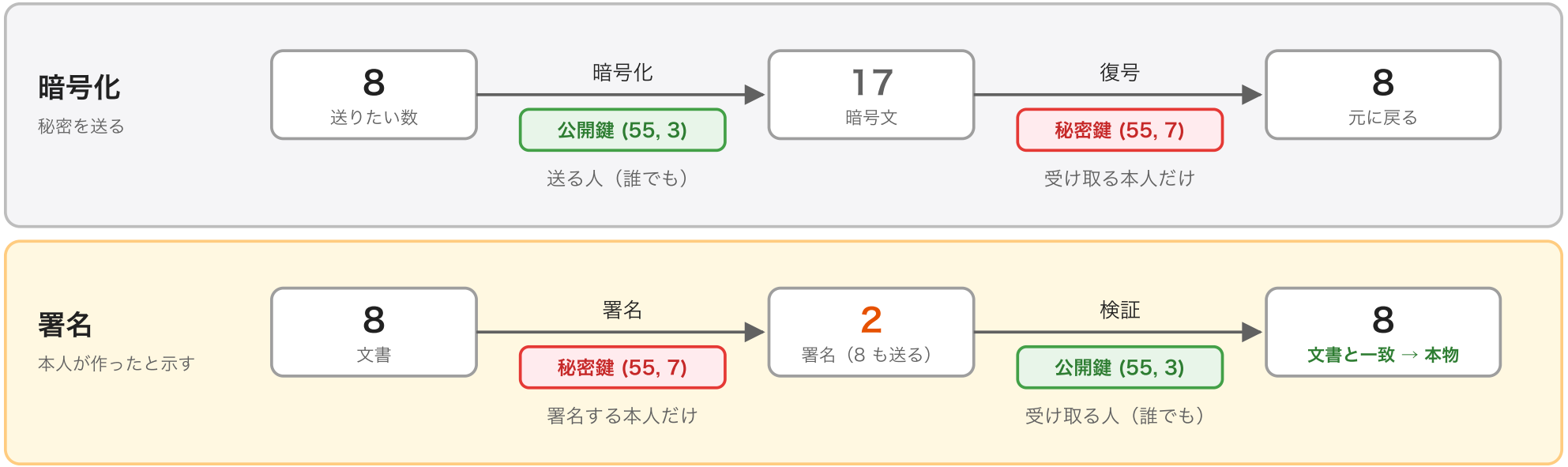
鍵は前のスライドの例を使う。**公開鍵 (n, r) = (55, 3)**、**秘密鍵 (n, s) = (55, 7)**

	手順	例（m = 8 に署名する）
署名（**秘密鍵** s で）	`sig = m ** s % n`	`8 ** 7 = 2097152`、55 で割った余りは **2** → m と 2 を送る
検証（**公開鍵** r で）	`sig ** r % n == m`	`2 ** 3 = 8` → m と一致するので、署名は本物

- 署名を作れるのは秘密鍵を持つ人だけ。検証は、公開鍵を持つ誰でもできる

> 実際の署名では、m そのものではなく、m のハッシュ値（m を短い値に要約したもの）に署名する。HTTPS の証明書とパスキーは、どちらもこの署名を使う。

暗号化と署名 — 鍵を使う順番が逆



- 暗号化は、公開鍵で閉じて秘密鍵で開く。**誰でも閉じられ、本人だけが開けられる**
- 署名は、秘密鍵で作って公開鍵で確かめる。**本人だけが作れ、誰でも確かめられる**

HTTPS = HTTP over TLS

- **H**yper**T**ext **T**ransfer **P**rotocol **S**ecure。既定のポートは 443
- SSL（Secure Sockets Layer）の後継が TLS（Transport Layer Security）。現在は **TLS 1.3**
- HTTP 自体は何も変わらない。第 5 回で書いたリクエストとレスポンスが、そのまま暗号化されて送られる

役割	使う仕組み
相手が本物かを確かめる	公開鍵暗号（証明書の署名の検証）
共通鍵を安全に作る	鍵共有（Diffie-Hellman）
本文を暗号化して送る	**共通鍵暗号**（処理が速いので本文に使う）

HTTPS の鍵と証明書ができるまで

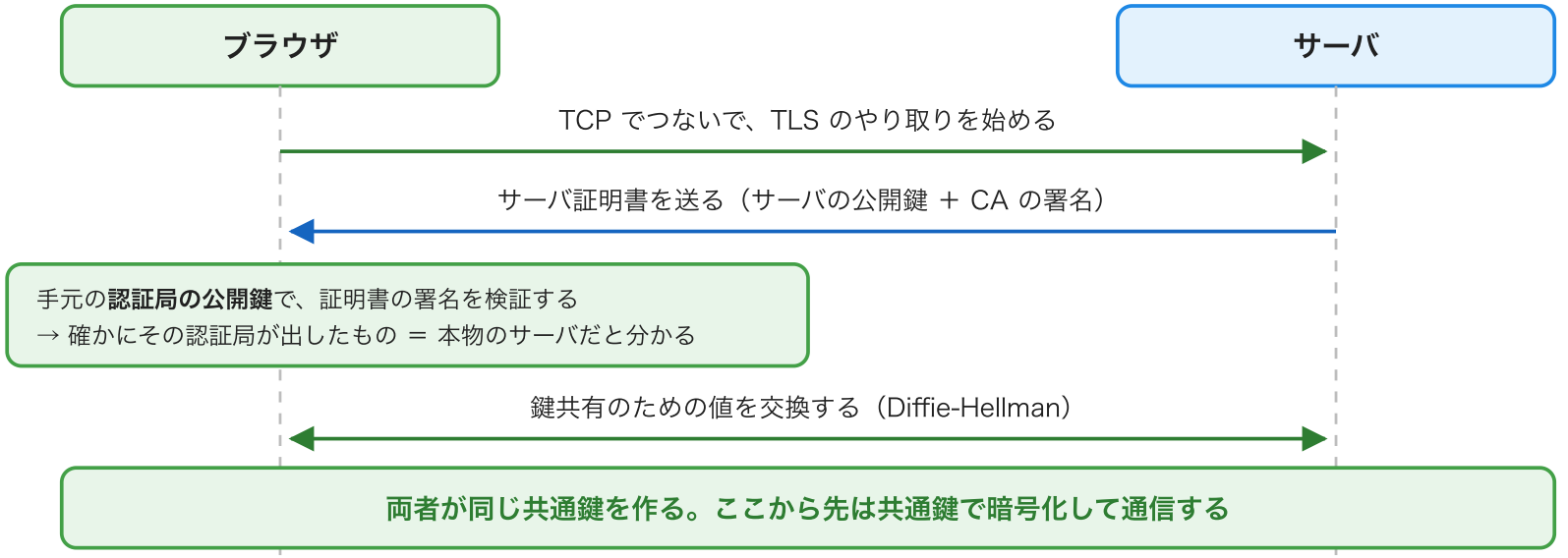


1. サーバの管理者が鍵ペアを作る。秘密鍵はサーバの外に出さない
2. 公開鍵とドメイン名を認証局に送り、証明書を申請する
3. 認証局は、申請者がそのドメインを本当に管理しているかを確かめる。例えば、指定のファイルを Web サーバに置かせ、外から取りに行って確かめる（HTTP-01）
4. 認証局は、公開鍵とドメイン名に**自分の秘密鍵で署名**して、証明書を発行する
5. 管理者は、証明書と秘密鍵をサーバに設置する
6. 認証局の公開鍵（ルート証明書）は、OS やブラウザにあらかじめ組み込まれている

サーバ認証と鍵共有

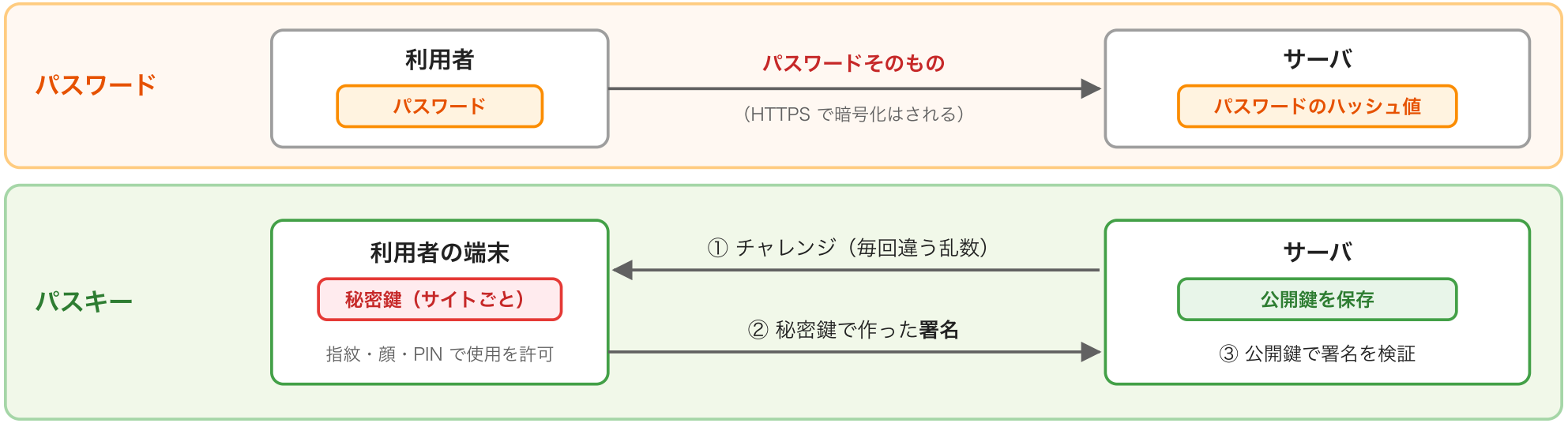
つなぐ前に、すでに済んでいること

① 認証局（CA）がサーバに証明書を発行し、**CA の秘密鍵**で署名しておく
② ブラウザや OS には、信頼する**認証局の公開鍵**があらかじめ組み込まれている



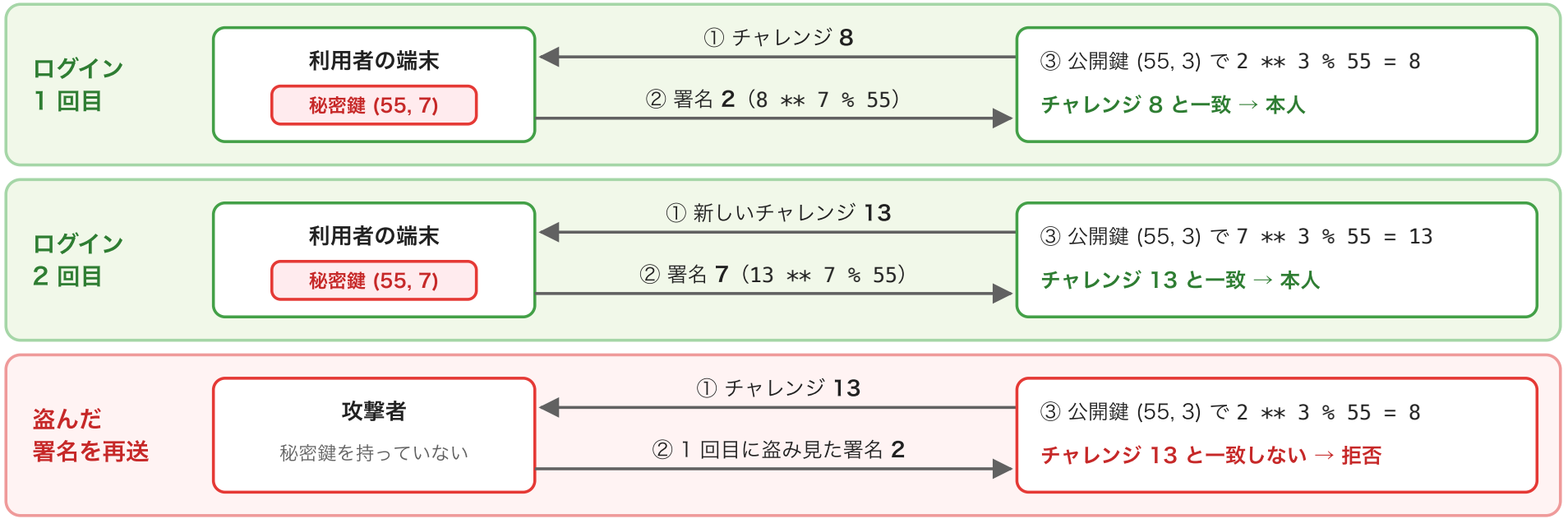
証明書を検証するのは「認証局の公開鍵」。公開鍵暗号は本人確認と鍵の受け渡しに使い、本文は共通鍵で速く暗号化する。

パスワードとパスキー — サーバに何を預けるか



- パスキーは、サイトごとに鍵ペアを作り、サーバには**公開鍵だけ**を登録する。秘密鍵はサーバに送らない
- ログインのたびに、サーバが乱数（チャレンジ）を送り、端末が秘密鍵で署名して返す。毎回違う値なので、盗み見ても使い回せない
- 指紋・顔・PIN は、端末の中で秘密鍵の使用を許可するためのもの。サーバには送らない

パスキー — 毎回ちがうチャレンジに署名する



- 登録のときに、端末がサイト専用の鍵ペアを作り、公開鍵だけをサーバに預ける（鍵は RSA の例と同じ）
- チャレンジが毎回変わるので、署名も毎回変わる。盗み見た署名は、次のログインでは通らない
- 署名する中身には**ドメイン名**も入る。偽サイトが中継した署名は、本物のサーバで通らない

公開鍵暗号から見た、パスワード・パスキー・HTTPS

	パスワード	パスキー
サーバに保存	パスワードのハッシュ値	公開鍵
ログインで送る	パスワードそのもの	チャレンジへの署名（毎回違う）
サーバから漏えい	弱いパスワードは総当たりで割り出される	公開鍵だけなので、ログインには使えない
偽サイト	そのまま盗まれる	鍵がドメインに結びつくので、偽サイトでは使えない
使い回し	しがち。1 か所の漏えいがほかのサイトにも及ぶ	サイトごとに別の鍵ペア

- HTTPS は、ブラウザが**サーバ**を確かめる（認証局の署名を検証）。パスキーは、サーバが**利用者**を確かめる（利用者の署名を検証）。向きは逆だが、どちらも公開鍵で署名を検証する
- HTTPS が守るのは経路だけ。偽サイトへの入力やサーバからの漏えいは防げない。パスキーはこの 2 つを防ぐ

今日やること・次回

5 講時（1-608 で実習）

1. `thread.py` を動かし、実行のたびに出力の順番が変わることを確かめる
2. `httpServer.py` を `httpThreadServer.py` に書き換える
    - ブラウザのタブを何枚も同時に開いて、応答の違いを確かめる
3. aiJudge の **ex6** に取り組む（提出後に自動採点）

次回（11/06）：hey と HTTP サーバの負荷テスト（講義パート最終回）

- 自分のサーバがどれくらいのリクエストを処理できるかを測る。**測ってよい相手のルール**も確認する

質問は **Teams「★ 質問用チャネル」** か、教室の TA へ。