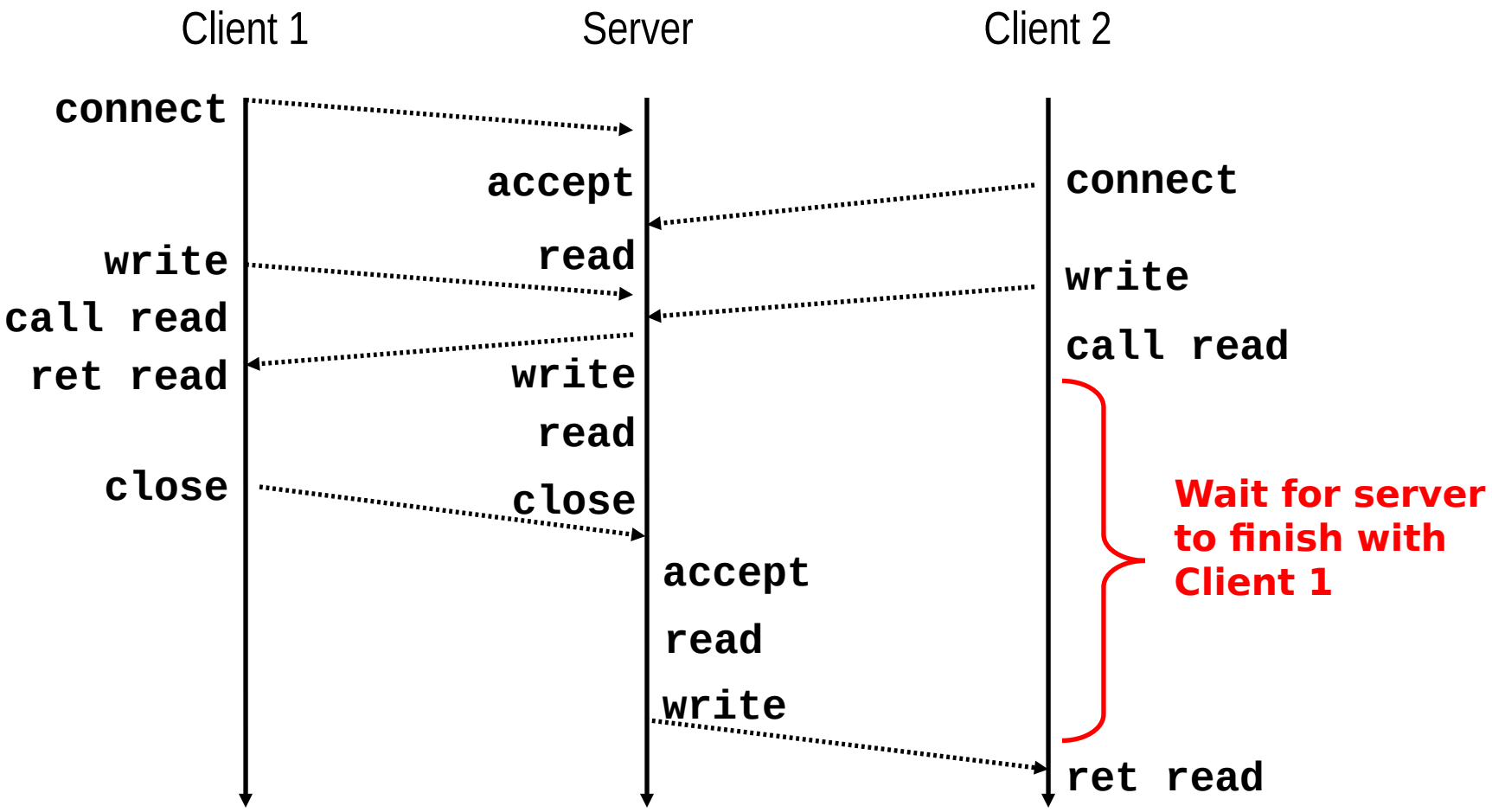
UNIVERSITY OF COPENHAGEN

Non-blocking Network Programming and Introducing Security: Hashes and Salt

David Marchant

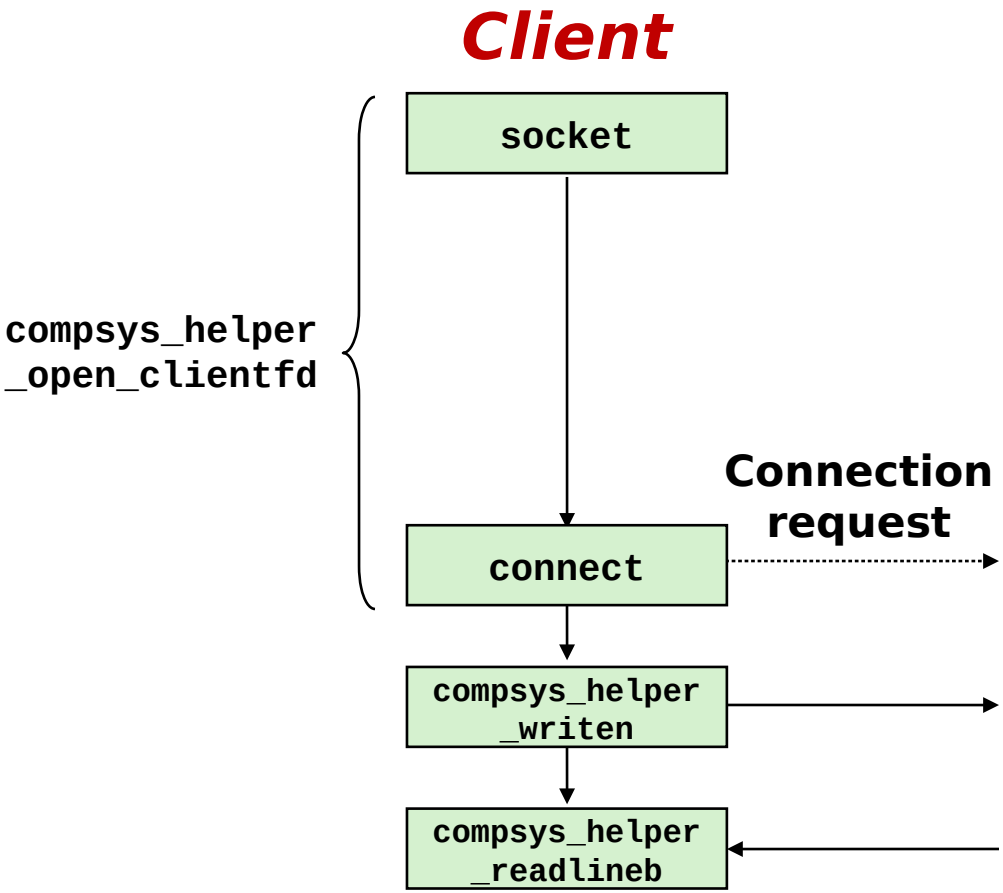
Iterative Servers

- Iterative servers process one request at a time



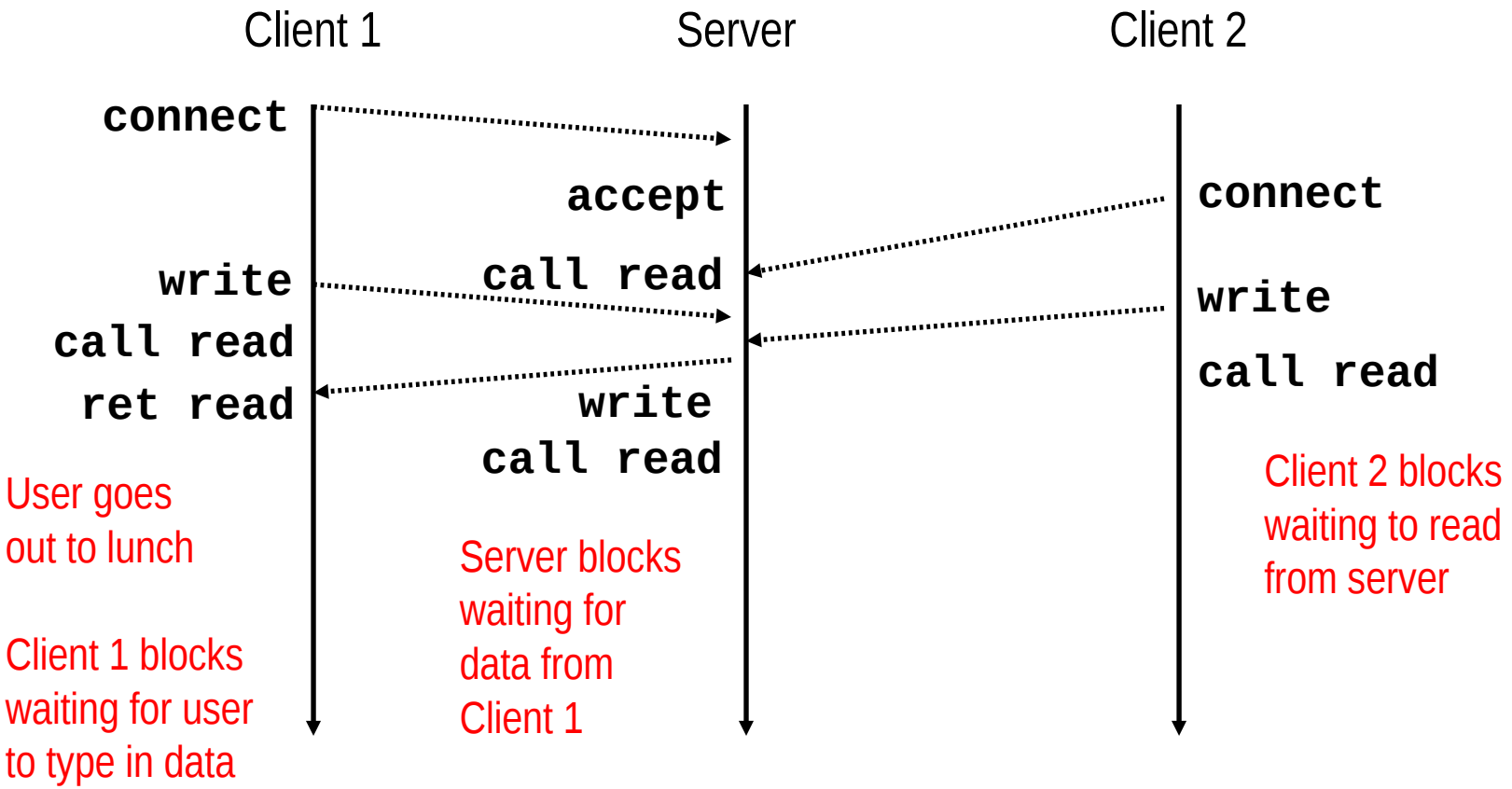
Where Does Second Client Block?

- Second client attempts to connect to iterative server



- Call to connect returns
  - Even though connection not yet accepted
  - Server side TCP manager queues request
  - Feature known as “TCP listen backlog”
- Call to compsys_helper_writen returns
  - Server side TCP manager buffers input data
- Call to compsys_helper_readlineb blocks
  - Server hasn’t written anything for it to read yet.

Fundamental Flaw of Iterative Servers



- Solution: use *concurrent servers* instead
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

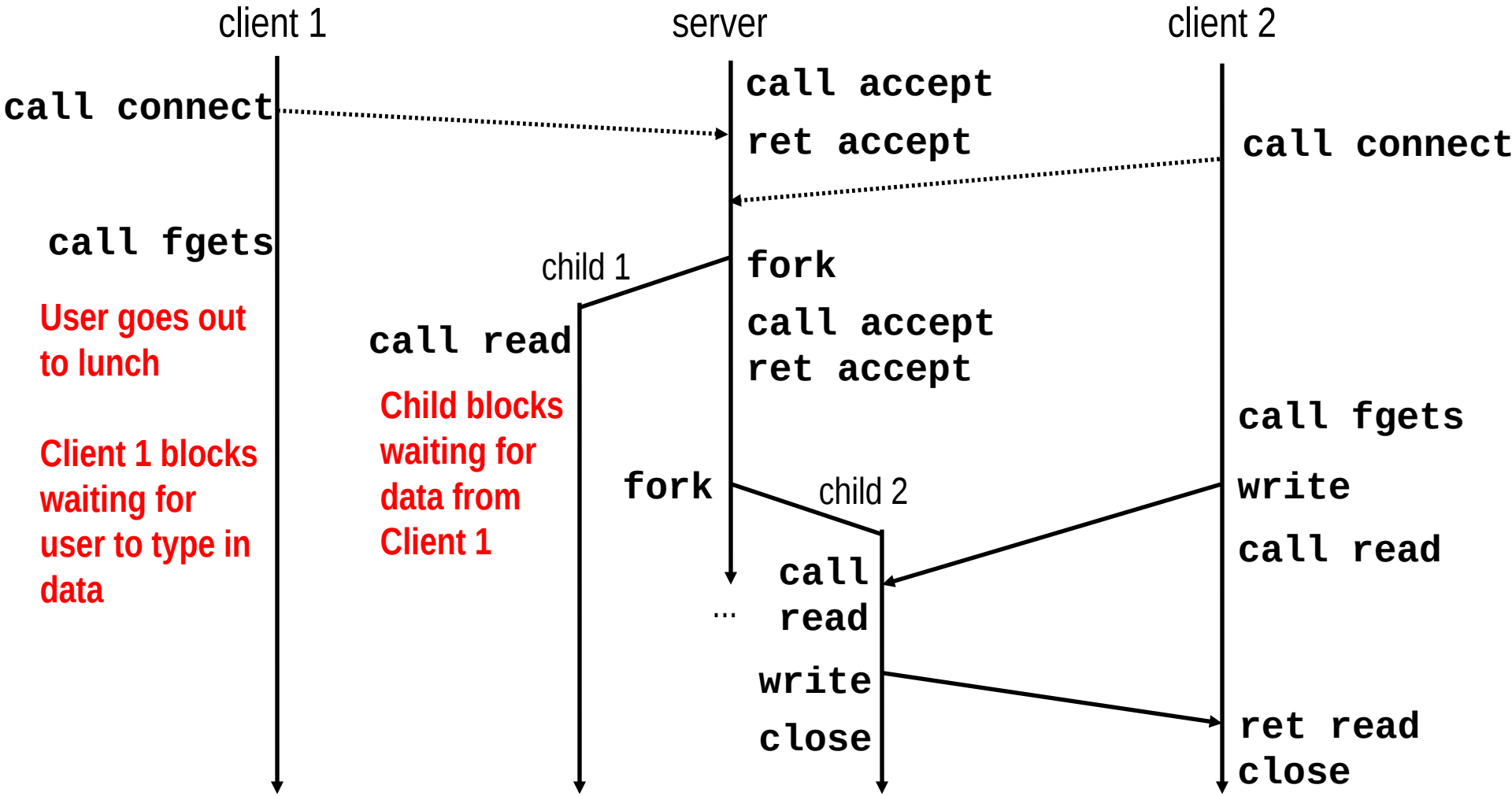
1. Process-based
- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based
- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Approach #1: Process-based Servers

- Spawn separate process for each client



Process-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    listenfd = compsys_helper_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (fork() == 0) {
            close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

Issues with Process-based Servers

- **Listening server process must reap zombie children**
  - to avoid fatal memory leak

- **Parent process must `close` its copy of `connfd`**
  - Kernel keeps reference count for each socket/open file
  - After fork, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) = 0`

Pros and Cons of Process-based Servers

- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - Requires IPC (interprocess communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

Approach #2: Event-based Servers

- **Server maintains set of active connections**
  - Array of connfd's
- **Repeat:**
  - Determine which descriptors (connfd's or listenfd) have pending inputs
    - e.g., using `select` or `epoll` functions
    - arrival of pending input is an *event*
  - If listenfd has input, then `accept` connection
    - and add new connfd to array
  - Service all connfd's with pending inputs
- **Details for select-based server in book**

Pros and Cons of Event-based Servers

- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado
- **– Significantly more complex to code than process- or thread-based designs.**
- **– Hard to provide fine-grained concurrency**
  - E.g., how to deal with partial HTTP request headers
- **– Cannot take advantage of multi-core**
  - Single thread of control

Approach #3: Thread-based Servers

- **Very similar to approach #1 (process-based)**
  - ...but using threads instead of processes

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Posix Threads (Pthreads) Interface

- ***Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs**
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
   int listenfd, *connfdp;
   socklen_t clientlen;
   struct sockaddr_storage clientaddr;
   pthread_t tid;

   listenfd = compsys_helper_open_listenfd(argv[1]);
   while (1) {
     clientlen=sizeof(struct sockaddr_storage);
     connfdp = malloc(sizeof(int));
     *connfdp = accept(listenfd,
              (SA *) &clientaddr, &clientlen);
     pthread_create(&tid, NULL, thread, connfdp);
   }
}
```

echoservert.c

- `malloc` of connected descriptor necessary to avoid deadly race

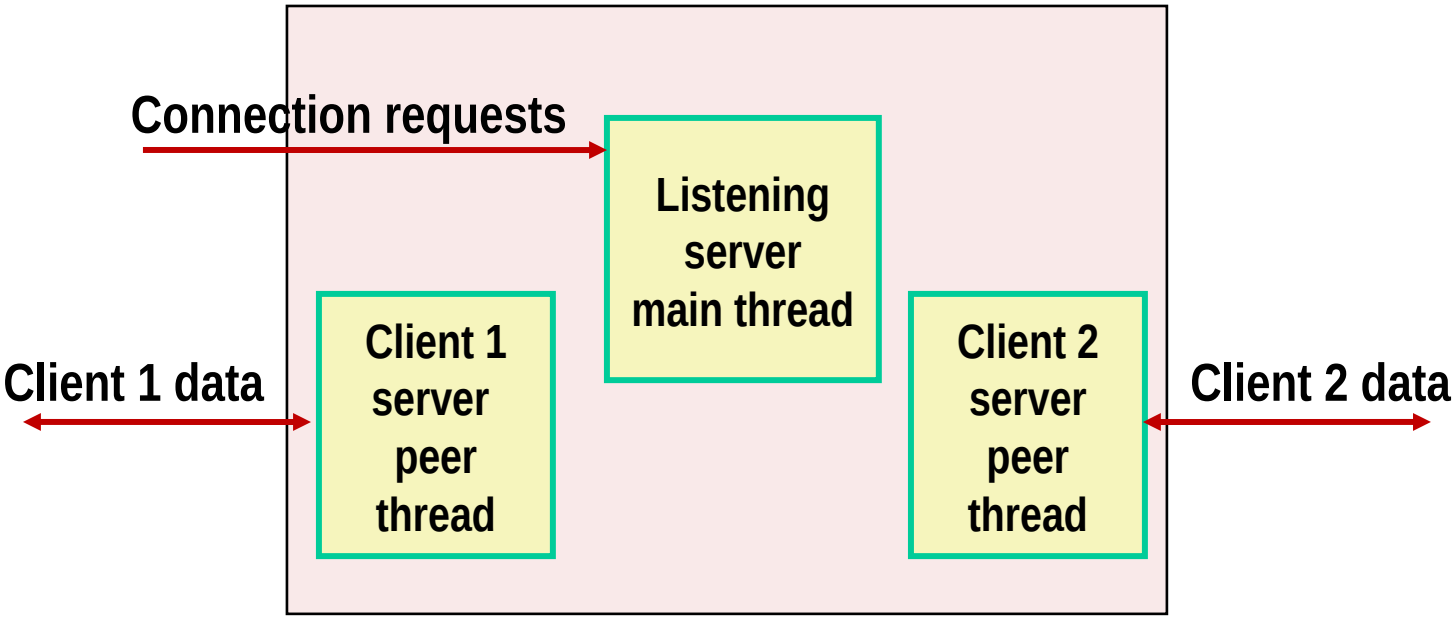
Thread-Based Concurrent Server (cont)

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    pthread_detach(pthread_self());
    free(vargp);
    echo(connfd);
    close(connfd);
    return NULL;
}
```

echoservert.c

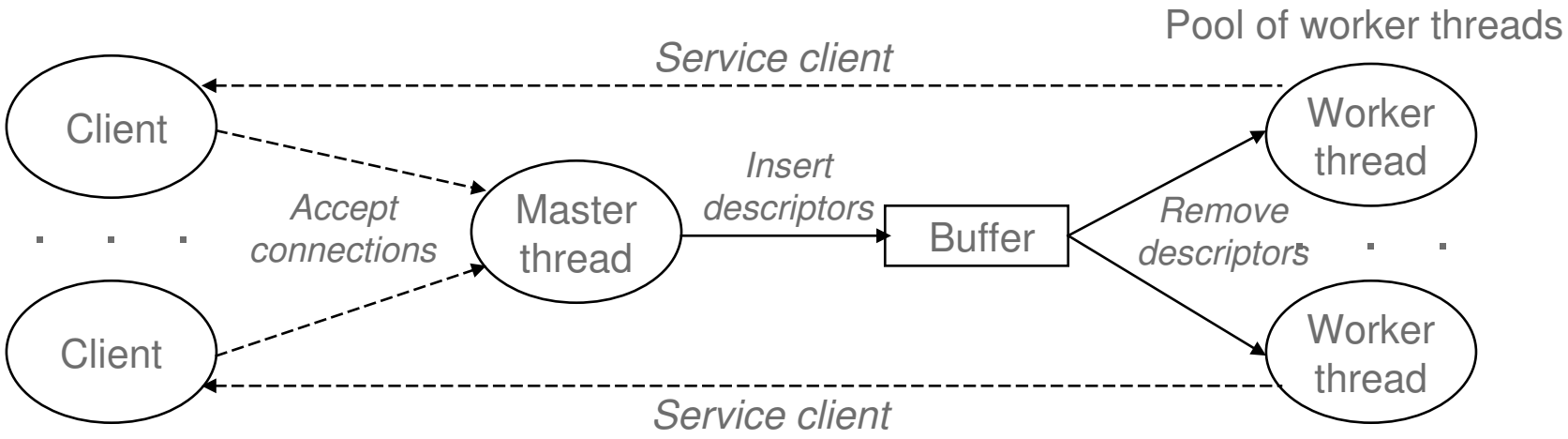
- Run thread in “detached” mode.
  - Runs independently of other threads
  - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold `connfd`.
- Close `connfd` (important!)

Thread-based Server Execution Model



- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables

Pre-threaded Server Model



- Clients handled using a thread-pool architecture
- Bounded/Unbounded buffer can be used for synchronization

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**
- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!

Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- **Event-based**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core
- **Thread-based**
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable

Concurrent Servers in Python

- Use socketserver module
- Few different base servers to choose from, most commonly you'll use **ThreadingTCPServer**, **StreamRequestHandler**
- Can use this either out-of-box, or inherit from them to extend functionality.

```
class handler(socketserver.BaseRequestHandler):
    def handle(self):
        ...
```

```
socketserver.TCPServer(server_address, RequestHandlerClass)
```

Concurrent Server Example

Iterative Server:

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind(("127.0.0.1", 5678))
    server_socket.listen()
    while True:
        connection, connection_address = server_socket.accept()
        with connection:
            message = connection.recv(1024)
            connection.sendall(response)
```

Concurrent Server:

```
from socketserver import ThreadingTCPServer, StreamRequestHandler

class MyHandler(StreamRequestHandler):
    def handle(self) -> None:
        message = self.request.recv(1024)
        self.request.sendall(message)

with ThreadingTCPServer(("127.0.0.1", 5678), MyHandler) as my_server:
    my_server.serve_forever()
```

Introducing Security

Source: Freedman (partial)

A Problem

- Everyone talks about security
- Everyone says its important
- Everyone agrees that its a problem

but

- There is no single simple, or even complex answer here, so it is often forgotten (See the course reading for an example)

This lecture will not cover all of security, but we are introducing it now for context, and will expect you to consider it in both networking courseworks

Source: Freedman (partial)

Nothing is secure forever

“You have 1 minute to design a maze that takes 2 minutes to solve” – some scriptwriter

Source: Freedman (partial)

Internet's Design: Insecure

- Designed for simplicity
- "On by default" design
- Readily available zombie machines
- Attacks look like normal traffic
- Internet's federated operation obstructs cooperation for diagnosis/mitigation

Source: Freedman (partial)

Security Properties

- **Confidentiality:** Concealment of information or resources
- **Authenticity:** Identification and assurance of origin of info
- **Integrity:** Trustworthiness of data or resources in terms of preventing improper and unauthorized changes
- **Availability:** Ability to use desired info or resource
- **Non-repudiation:** Offer of evidence that a party indeed is sender or a receiver of certain information

---

- **Access control:** Facilities to determine and enforce who is allowed access to what resources (host, software, network, ...)

Source: Freedman (partial)

Hash functions

- Function for deterministicly computing a fixed-length output from some variable length input
- Minimise output collisions (eg two inputs producing the same output)
- Idealy is fast to compute, but time consuming to reverse engineer

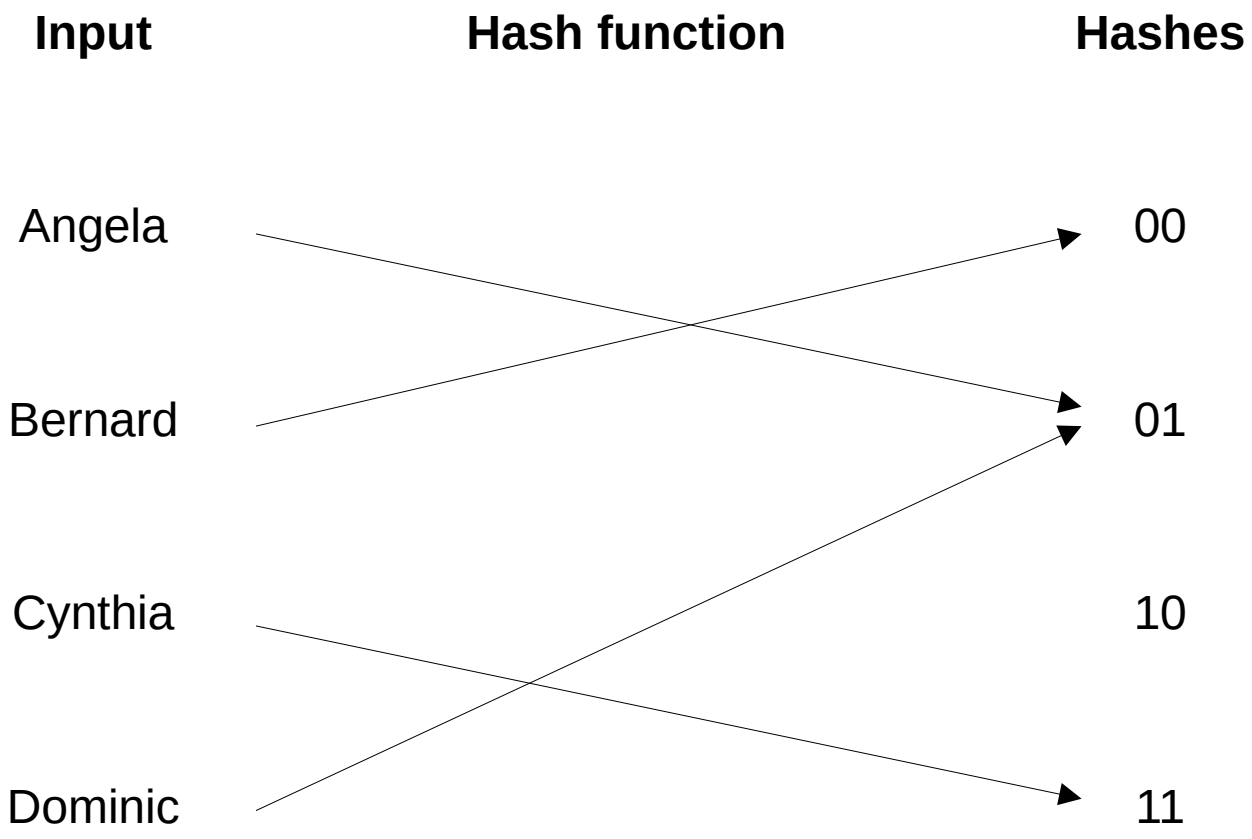
Source: Freedman (partial)

A note on cryptography

- You may encounter the term ‘cryptographic hash functions’
- These are a subset of hash functions, but we will not really discuss them yet. They are much slower and conversely much more difficult to reverse engineer
- We will return to these a few times, later in the course

Source: Freedman (partial)

Hash tables



Source: Freedman (partial)

C Hashing example:

```
typedef uint8_t hashdata_t[32];

void get_data_sha(const char* sourcedata, hashdata_t hash,
    uint32_t data_size, int hash_size) {
  SHA256_CTX shactx;
  unsigned char shabuffer[hash_size];
  sha256_init(&shactx);
  sha256_update(&shactx, sourcedata, data_size);
  sha256_final(&shactx, shabuffer);

  for (int i=0; i<hash_size; i++) {
    hash[i] = shabuffer[i];
  }
}
```

Source: Freedman (partial)

Python Hashing example:

```
import hashlib

def get_sha256(data):
    return hashlib.sha256(data).digest()
```

- Do **not** reinvent the wheel for hashing algorithms, use an existing implementation.
- Many algorithms available achieving different levels of reverse engineerability in differing amounts of time.
- Common quick hashing algorithms are MD5 and SHA256

Source: Freedman (partial)

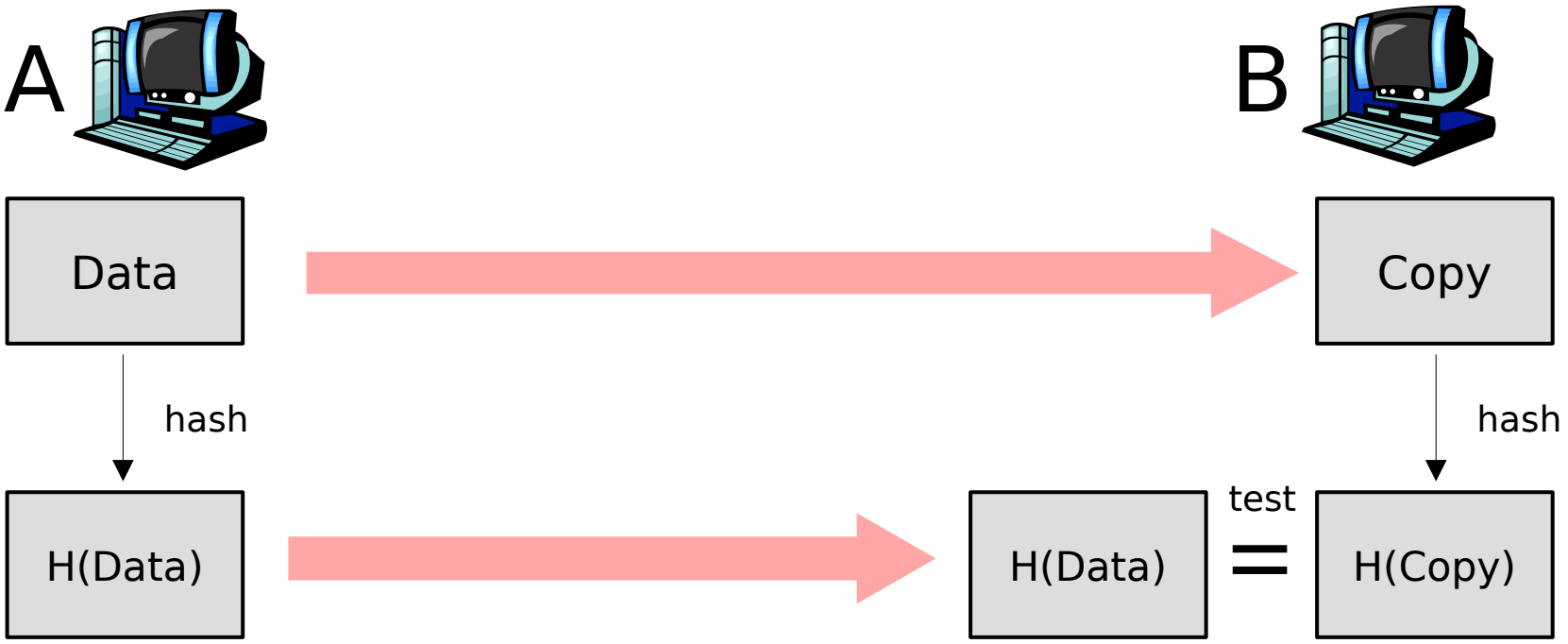
Where we can use hashing

4-bit Version	4-bit Header Length	8-bit Type of Service (TOS)	16-bit Total Length (Bytes)
16-bit Identification		3-bit Flags	13-bit Fragment Offset
8-bit Time to Live (TTL)	8-bit Protocol	16-bit Header Checksum	
32-bit Source IP Address			
32-bit Destination IP Address			
Options (if any)			
Payload			

20-byte header

Source: Freedman

Where we can use hashing: Checksums



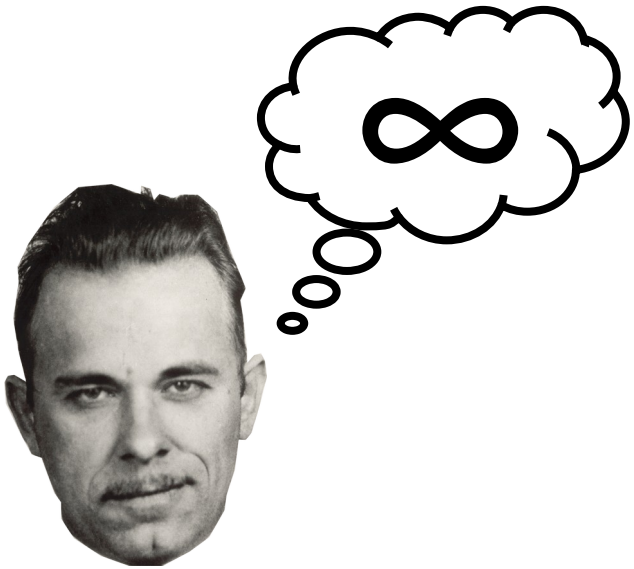
Source: Freedman

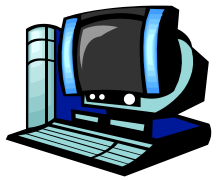
Where we can use hashing: Obfuscation

- Note: Secure algorithms must be used here for this to be effective

Source: Freedman

A sudden tangent about passwords



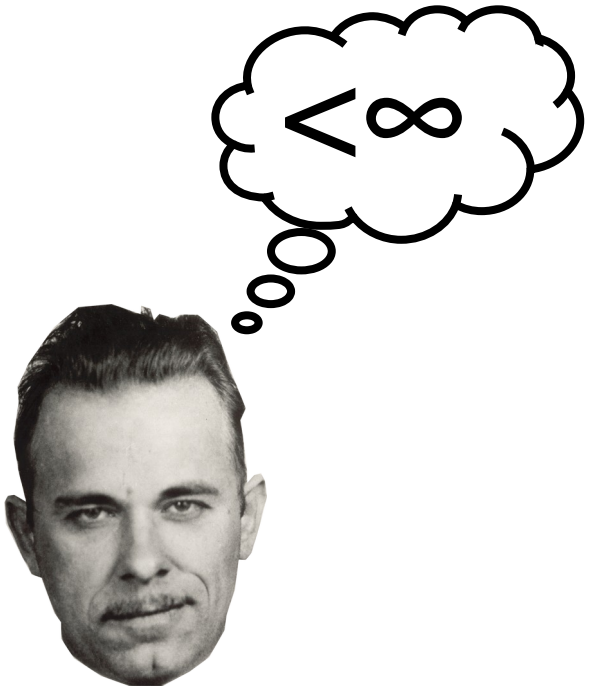
server 

user	password
Edwina	123456
Franklin	password
Gertrude	qwerty
Harry	harry123

- Storing sensitive information like passwords as plaintext is asking for trouble
- Direct intrusion is probably one of the *easier* (note not easy) attacks to prevent, but no reason to make stupid mistakes when the stakes are high
- Many users share details across systems so one breach can affect more than just your unimportant system

Source: Freedman

A sudden tangent about passwords



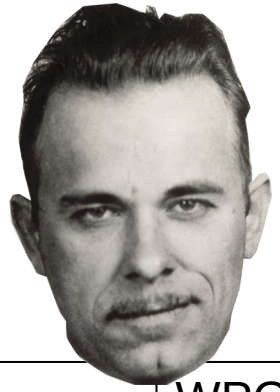
server 

user	hash
Irene	SD0lpCmD
Jacob	NUDYI3BJ
Kirsty	SD0lpCmD
Leviticus	jnHNoO4j

- Hashes help obfuscate passwords, but hashtables of common passwords exist
- We can see that Irene and Kirsty share a password

Source: Freedman

Dictionary Attacks



123456	WPQNfeB4
123456789	tyjtnAOi
qwerty	AWlQArsu
password	Jn0Pefz4

server 

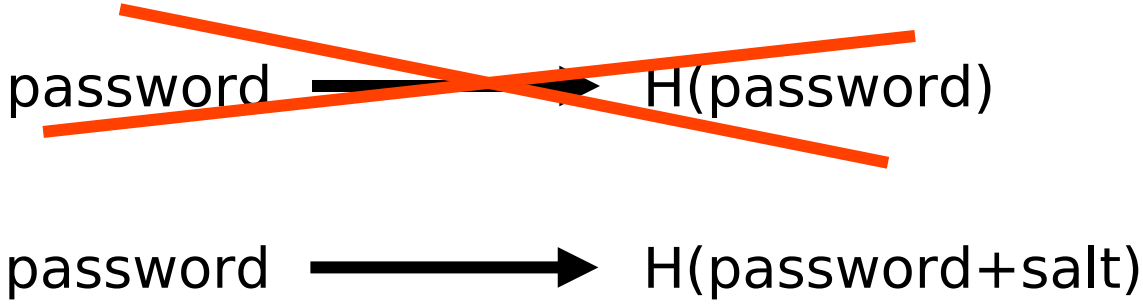
user	hash
Monica	?
Norbet	?
Oana	?
Percy	?

- More intelligent brute force attack, just try common password hashes
- We just can't make users always use unique passwords (And shouldn't tell a user their password isn't unique)

Source: Freedman

Salt to the rescue

- Because hashing is deterministic, the same input will have the same output. Definately a problem when users can't be relied upon to have unique passwords
- Second problem is that rainbow tables of hashes of common passwords defiantely exist and are used



- Salts should idealy be long, random, and unique to ensure minimum chances of collisions

Source: Freedman

Salting code example

Non-salted:

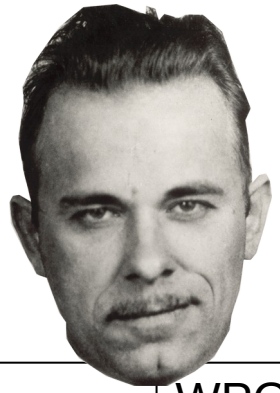
```
import hashlib

def get_sha256(data):
    return hashlib.sha256(data).digest()

hashed = get_sha256("some data")
```

Applying a salt:

```
import hashlib
import string
import random

def get_random_salt():
    return ''.join(random.choices(string.ascii_lowercase, k=64))

def get_sha256(data, salt):
    to_salt = data + salt
    return hashlib.sha256(to_salt).digest()

salt = get_random_salt()
hashed_and_salted = get_sha256("some data", salt)
```

Source: Freedman

A salted password database



server 

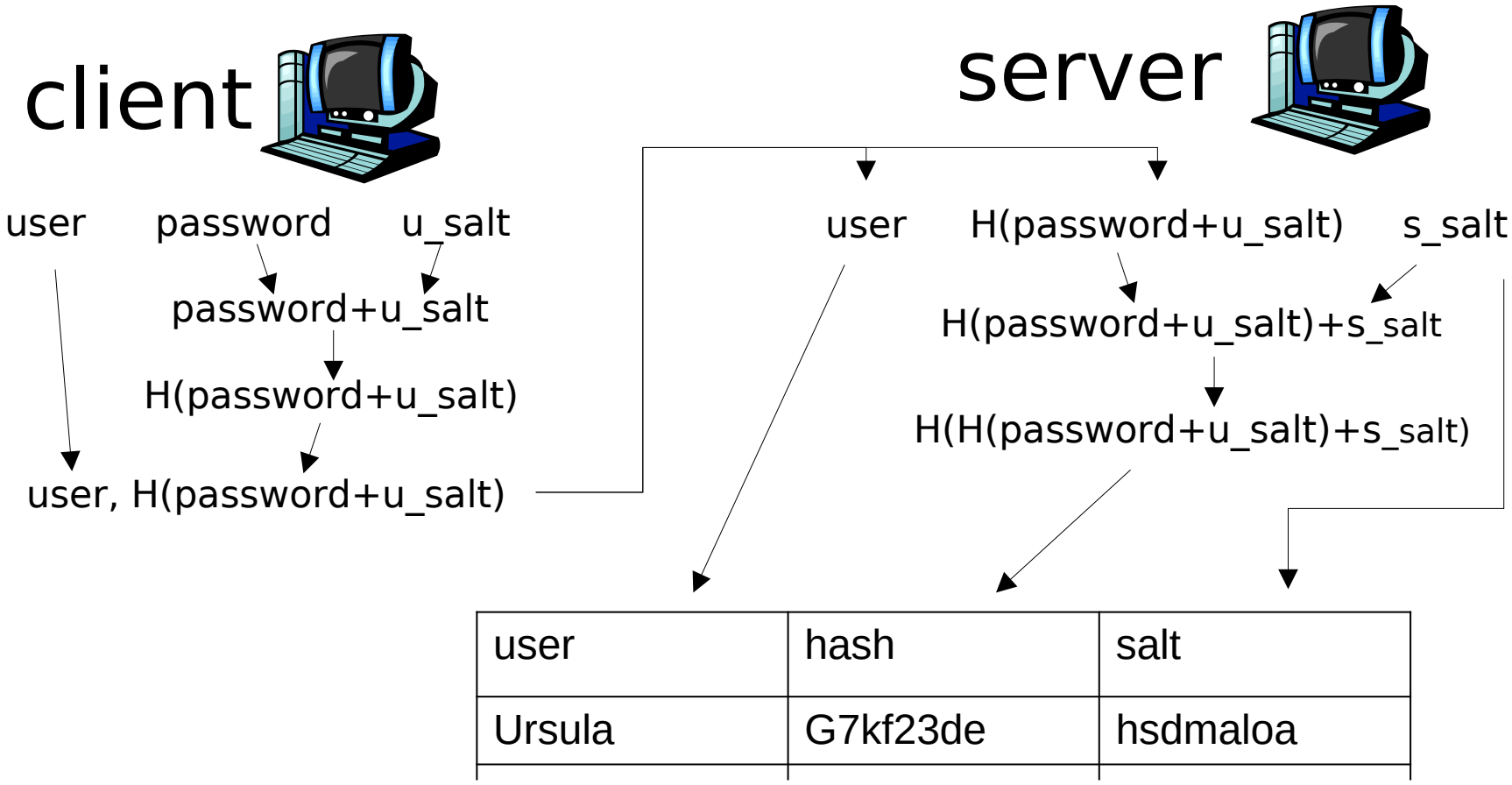
123456	WPQNfeB4
123456789	tyjtnAOi
qwerty	AWlQArsu
password	Jn0Pefz4

user	salted	salt
Quetzalli	AB1h446R	ezncyjgi
Robert	NPmwH9Gh	xhrfanpr
Stacy	9doR1J1Z	mzcixtod
Tarquin	sd1AiaeA	hyesxieg

- All the rainbow tables in the world won't help if our passwords are salted before being hashed
- Note the salt is stored as plaintext as we need it to verify our salted hash, but that isn't a problem really
- This doesn't make it impossible to still brute force, but each hash needs to computed independently

Source: Freedman

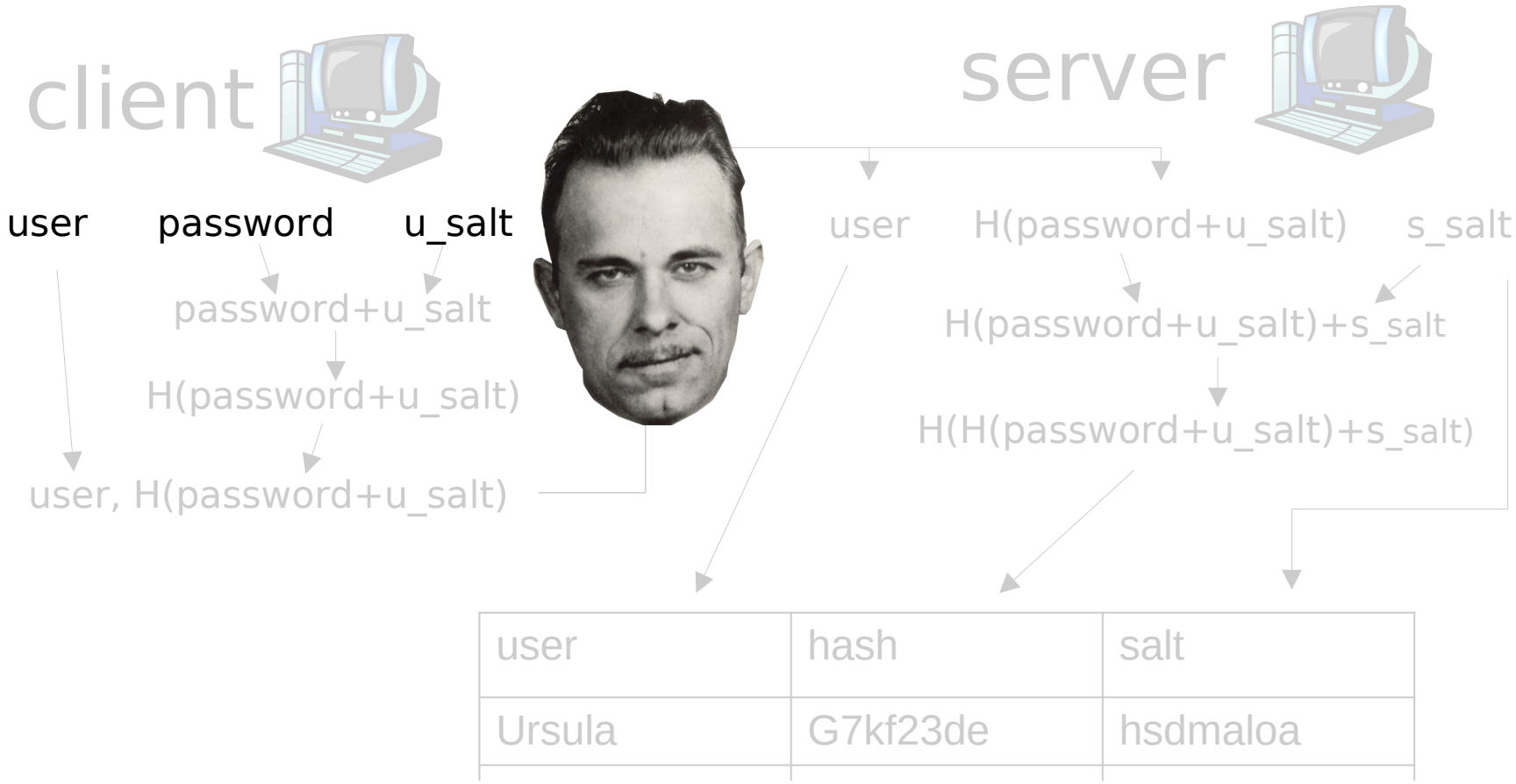
The setup in the assignment



user	hash	salt
Ursula	G7kf23de	hsdmaloa

Source: Freedman

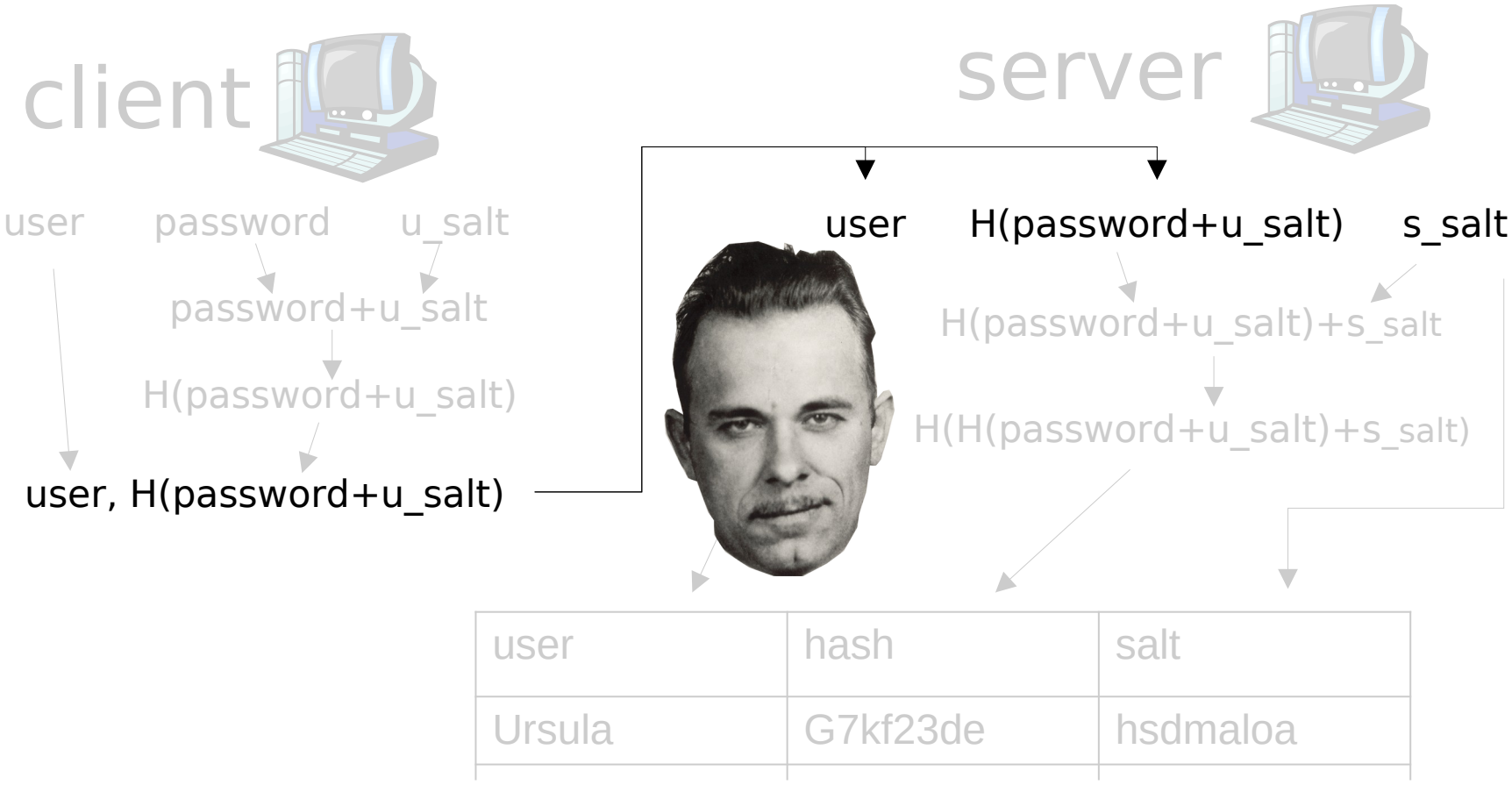
The setup in the assignment



- If a user-remembered password gets compromised, the attacker still needs to find out the salt to gain access
- Password tables become much less effective
- Randomly generating salt means rainbow tables won't help either

Source: Freedman

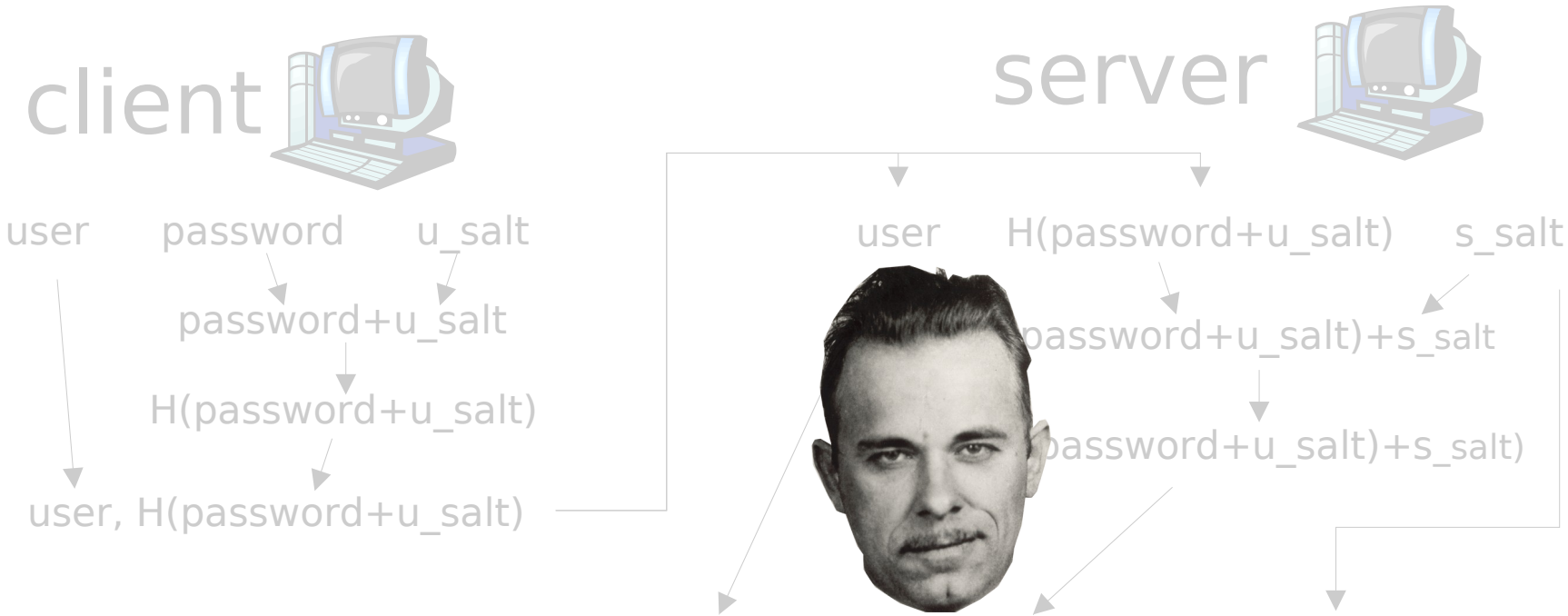
The setup in the assignment



- If the message gets intercepted the user-remembered password is still obscured
- Repeated hashes of salted user passwords are extremely unlikely

Source: Freedman

The setup in the assignment



user	hash	salt
Ursula	G7kf23de	hsdmaloa

- If the user database gets compromised, hashes of salted passwords are obscured
- User passwords never even make it into the server

Source: Freedman

Security

- This is not an airtight system, its just harder for an attacker to get in using certain common techniques
- There are of course other ways into the system
- It’s enough for now though

Nothing is secure forever

“You have 1 minute to design a maze that takes 2 minutes to solve” – some scriptwriter

Source: Freedman (partial)