Operating Systems and Networks SoSe 25 Notes

Igor Dimitrov

2024-12-18

Table of contents

Preface

Part I

Operating Systems

1 Process Management

1.1 Condition Variables and Producer / Consumer Problem

Condition variables are employed **together** with mutexes when synchronizing producers and consumers. It woul be incorrect ot only use a condition variable without a mutex, or a mutex with busy waiting without a condition varible.

Incorrect Variant 1: Condition Variable Without Mutex

```
ready = False
condition = ConditionVariable()

def wait_thread():
    if not ready:
        condition.wait()  # Incorrect: no mutex guarding shared state
    print("Condition met!")

def signal_thread():
    ready = True
    condition.notify()
```

Why It's Wrong:

- Access to `ready` is unprotected — race conditions may occur.
- `condition.wait()` must always be used with a mutex.

Incorrect Variant 2: Mutex Without Condition Variable (Busy Waiting)

```
ready = False
mutex = Mutex()
```

```
def wait_thread():
    while True:
        mutex.lock()
        if ready:
            mutex.unlock()
            break
        mutex.unlock()
        sleep(0.01)  # Active polling (wasteful)

def signal_thread():
    mutex.lock()
    ready = True
    mutex.unlock()
```

Why It's Problematic:

- Avoids races, but wastes CPU via busy waiting.
- Also prone to subtle visibility issues if memory barriers aren't enforced.

Correct Variant: Condition Variable with Mutex

```
ready = False
mutex = Mutex()
condition = ConditionVariable()

def wait_thread():
    mutex.lock()
    while not ready:
        condition.wait(mutex)  # Atomically unlocks and waits
    mutex.unlock()
    print("Condition met!")

def signal_thread():
    mutex.lock()
    ready = True
    condition.notify()
    mutex.unlock()
```

Why It Works:

- Shared state is properly guarded.

- No busy waiting.
- Safe signaling and waking.

Another question is why to use `while not ready` and not simply `if not ready`:

```
def wait_thread():
    mutex.lock()
    if not ready:
        condition.wait(mutex)
    mutex.unlock()
```

Problem:

- May miss spurious wakeups or situations where multiple threads wait and only one should proceed.
- A `while` loop is necessary to recheck the condition after being woken up.

Producer/Consumer Problem

Variant A: Unbounded Queue (No Buffer Limit)

```
queue = []
mutex = Mutex()
not_empty = ConditionVariable()

def producer():
    while True:
        item = produce()
        mutex.lock()
        queue.append(item)
        not_empty.notify()
        mutex.unlock()

def consumer():
    while True:
        mutex.lock()
        while not queue:
            not_empty.wait(mutex)
        item = queue.pop(0)
```

```
        mutex.unlock()
        consume(item)
```

Variant B: Bounded Queue (Fixed Buffer Size)

```
queue = []
BUFFER_SIZE = 10
mutex = Mutex()
not_empty = ConditionVariable()
not_full = ConditionVariable()

def producer():
    while True:
        item = produce()
        mutex.lock()
        while len(queue) >= BUFFER_SIZE:
            not_full.wait(mutex)
        queue.append(item)
        not_empty.notify()
        mutex.unlock()

def consumer():
    while True:
        mutex.lock()
        while not queue:
            not_empty.wait(mutex)
        item = queue.pop(0)
        not_full.notify()
        mutex.unlock()
        consume(item)
```

1.2 Summary Table

Case	Uses Mutex	Uses Condition Variable	Blocking	CPU-Efficient	Correct
1. Condition variable without mutex	No	Yes	No	Yes	No
2. Mutex without condition variable	Yes	No	No	No (busy)	Partly
3. Condition variable with mutex	Yes	Yes	Yes	Yes	Yes
4. If instead of while	Yes	Yes	Yes	Yes	Risky
5. Producer/Consumer (unbounded)	Yes	Yes (`not_empty`)	Yes	Yes	Yes
6. Producer/Consumer (bounded)	Yes	Yes (`not_empty`, `not_full`)	Yes	Yes	Yes

---

Operations of a Bounded Queue

Step	Operation	`in`	`out`	Buffer State	Count == (`(in - out + 5) % 5`)
0	Start	0	0	`[_ _ _ _ _]`	0
1	Produce A	1	0	`[A _ _ _ _]`	1
2	Produce B	2	0	`[A B _ _ _]`	2
3	Produce C	3	0	`[A B C _ _]`	3
4	Consume → A	3	1	`[_ B C _ _]`	2
5	Consume → B	3	2	`[_ _ C _ _]`	1
6	Produce D	4	2	`[_ _ C D _]`	2
7	Produce E	0	2	`[_ _ C D E]`	3
8	Consume → C	0	3	`[_ _ _ D E]`	2
9	Produce F	1	3	`[F _ _ D E]`	3

where

- `in`: the write position / index
- `out`: the read position /index
- `count == (in - out + 5) % 5` is the invariant of the data structure, giving the number of elements in the buffer

2 Memory Management

2.1 Virtual Memory

Paging

Translating Logical to Physical Addresses

Context

In paging, the operating system divides:

- Logical (virtual) memory into fixed-size pages
- Physical memory (RAM) into same-size frames

Each process has a page table that maps page numbers to frame numbers.

Our goal is:

> Given a virtual address, compute the corresponding physical address.

Example Setup

- Virtual address $V = 7000$
- Page size = 4096 bytes = $2^{12} \quad k = 12$
- Assume the page table maps page 1 to frame 9: $F(1) = 9$

Step 1: Manual (Arithmetic) Calculation

To translate a virtual address manually, we need to answer two questions:

1. **Which page** is the address in?
2. **Where within that page** is the address?

This is done by:

- Dividing the address by the page size to get the **page number**
- Taking the remainder (modulo) to get the **offset** within the page

Apply this to $V = 7000$ with page size 4096:

- Page number $p = \left\lfloor \frac{7000}{4096} \right\rfloor = 1$
- Offset $d = 7000 \mod 4096 = 2904$

Now we look up page 1 in the page table:

- Frame number $f = F(1) = 9$

To get the final physical address, we compute the base address of frame 9 and add the offset:

- Physical address $= f \cdot 4096 + d = 9 \cdot 4096 + 2904 = 39768$

Result: 39768

Step 2: Bitwise Calculation (Optimized for Hardware)

For power-of-two page sizes, the address can be efficiently split using bitwise operations:

- Page number $= V \gg 12$ (right shift by 12 bits is equivalent to dividing by 4096)
- Offset $= V \& (2^{12} - 1) = V \& 0xFFF$ (bit mask keeps the lower 12 bits)
- Page number 1 maps to frame number $f = F(1) = 9$

To compute the **frame's starting address**, we use a left shift:

- $f \ll 12 = 9 \ll 12 = 36864$, which is equivalent to $9 \cdot 4096$

Final physical address:

- Physical Address $= 36864 + 2904 = 39768$

Same result, now using fast bit operations.

Bit Sequence Visualization

Let's visualize how the virtual address is split in binary:

- Virtual address $V = 7000$
- Binary representation (14 bits): `0001 1011 0101 1000`

Split into:

- Page number (upper 2 bits): `00 01` $\rightarrow 1$
- Offset (lower 12 bits): `1011 0101 1000` $\rightarrow 2904$

This split works because:

- The lower 12 bits represent the offset for a 4 KB page
- The upper bits index into the page table

Why This Works Mathematically

The logic behind using bit shifts and masks instead of division and modulo is based on how numbers are represented in binary.

Decimal Analogy (Base 10)

Consider dividing `1375` by powers of 10:

- $137\mathbf{5} \div 10^1 = 137 \quad (\text{modulo: } \mathbf{5})$
- $13\mathbf{75} \div 10^2 = 13 \quad (\text{modulo: } \mathbf{75})$
- $1\mathbf{375} \div 10^3 = 1 \quad (\text{modulo: } \mathbf{375})$

The rightmost digits are the remainder (modulo); the left are the quotient (division).

Binary Example (Base 2)

Take the binary number `1011` ($= 11$):

- $101\mathbf{1} \div 2^1 = 101 = 5 \quad (\text{modulo: } \mathbf{1})$
- $10\mathbf{11} \div 2^2 = 10 = 2 \quad (\text{modulo: } \mathbf{11} = 3)$
- $1\mathbf{011} \div 2^3 = 1 = 1 \quad (\text{modulo: } \mathbf{011} = 3)$

In both systems, the rightmost digits/bits represent the **offset**, and the leftmost represent the **page number**.

This is why in binary:

- $V \gg k$ is equivalent to $\lfloor V/2^k \rfloor$
- $V\&(2^k - 1)$ is equivalent to $V \mod 2^k$
- $f \ll k$ is equivalent to $f \cdot 2^k$, which gives the frame base address

These operations are both mathematically correct and hardware-efficient.

Final Formula

$$\text{Physical Address} = (F(V \gg k) \ll k) + (V \& (2^k - 1))$$

This computes:

- The page number via right shift
- The frame number from the page table
- The frame base via left shift (i.e., multiplying by page size)
- The final physical address by adding the offset

Additional Example for Practice and Clarity

Let's now take another address and apply all three methods for reinforcement.

Setup

- Virtual address $V = 13{,}452$
- Page size $= 4096 = 2^{12}$
- Page table:

Page #	Frame #
0	3
1	7
2	1
3	6

Manual Calculation

- Page number: $13{,}452 \div 4096 = 3$
- Offset: $13{,}452 \mod 4096 = 1164$
- Frame number: $F(3) = 6$
- Physical address $= 6 \cdot 4096 + 1164 = 24{,}576 + 1164 = 25{,}740$

Bitwise Calculation

- $V = 13{,}452 = 0b0011\ 0100\ 1001\ 1100$
- Page number $= V \gg 12 = 3$
- Offset $= V\&0xFFF = 1164$
- Frame number $= F(3) = 6$
- Frame base $= 6 \ll 12 = 24{,}576$
- Physical address $= 24{,}576 + 1164 = 25{,}740$

Using the Formula

$$\text{Physical Address} = (F(V \gg 12) \ll 12) + (V\&0xFFF)$$

$$= (6 \ll 12) + 1164 = 24{,}576 + 1164 = 25{,}740$$

Conclusion

When the page size is a power of two, address translation can be performed using fast bit operations instead of division and modulo. This is possible because of how binary numbers encode positional value. We saw that the lower bits give the offset and the upper bits the page number. Whether done manually, with bit operations, or using the translation formula, all approaches yield the same physical address — and this consistency is what makes paging both robust and efficient.

Absolutely — here is the **complete regenerated summary**, now incorporating:

- The **updated "Inverted Page Tables"** section with size explanation and example
- The **updated "Hierarchical Page Tables"** section with both the 32-bit and 64-bit address resolution examples and definitions of each table level
- Consistent formatting throughout:
  - **No bold in headers**
  - **Minimal boldface emphasis** in the text — used only where strictly useful for clarity

This version is fully ready for integration into your Quarto notes.

Page Tables

Single-Level (Direct) Page Tables

In the simplest form of paging, each process has its own single-level page table, which directly maps virtual page numbers to physical frame numbers.

For example, in a system with:

- A 32-bit virtual address space (4 GB total)
- A page size of 4 KB = 2^{12} bytes

The number of virtual pages is:

$$2^{32}/2^{12} = 2^{20} = 1{,}048{,}576 \text{ entries}$$

If each page table entry (PTE) is 4 bytes, the total size of the page table is:

$$2^{20} \times 4 = 4 \text{ MB per process}$$

In a 64-bit system, even with larger pages (e.g. 4 MB), the number of virtual pages is so large (e.g., 2^{52}) that flat page tables become completely impractical.

Why Single-Level Tables Are Impractical

Main issues:

- Memory usage per process becomes excessive (e.g., 4 MB/page table $\times$ hundreds of processes)
- Scaling issues as address spaces grow
- Most processes use only a small part of their virtual address space, so allocating full page tables is wasteful

Thus, alternative paging strategies are needed.

Frame Table (Global Physical Memory Tracking)

The OS maintains a global frame table, which tracks:

- Which physical frames are in use
- What each frame is used for (user page, kernel structure, page table, etc.)
- Associated metadata: dirty bit, reference count, owner process

This allows the OS to allocate and deallocate physical memory intelligently and safely. It is also crucial for page replacement algorithms, memory protection, and managing shared pages or I/O buffers.

Inverted Page Tables

In a traditional page table system, each process maintains its own page table, which maps virtual pages to physical frames. In contrast, an inverted page table uses a fundamentally different approach:

- There is a single global page table for the entire system
- It contains one entry for each physical frame, not for each virtual page
- Each entry records:
  - The process ID that owns the frame
  - The virtual page number that maps to it
  - Any additional metadata (e.g., access flags, validity)

This approach dramatically reduces memory overhead, especially in systems with large virtual address spaces.

Size of the inverted page table

The size of an inverted page table depends only on the number of physical frames, which is determined by:

$$\text{Number of entries} = \frac{\text{RAM size}}{\text{frame size}}$$

Each entry stores fixed-size metadata (such as PID and VPN), so the total size is:

$$\text{Table size} = \frac{\text{RAM size}}{\text{frame size}} \times \text{entry size}$$

The ratio of the table size to RAM size simplifies to:

$$\frac{\text{Table size}}{\text{RAM size}} = \frac{\text{entry size}}{\text{frame size}}$$

This ratio is independent of total RAM size. In other words, the memory overhead of the page table scales proportionally with RAM but is bounded by the frame size and the entry size.

Concrete example

Consider a 32-bit system with the following properties:

- Physical RAM: 4 GB = 2^{32} bytes
- Page/frame size: 4 KB = 2^{12} bytes
- Page table entry size: 8 bytes (to store PID, VPN, flags, etc.)

Step-by-step:

1. Number of physical frames:

$$\frac{2^{32}}{2^{12}} = 2^{20} = 1{,}048{,}576 \text{ frames}$$

2. Total inverted page table size:

$$2^{20} \times 8 = 8 \text{ MB}$$

3. Relative overhead:

$$\frac{8 \text{ MB}}{4 \text{ GB}} = \frac{1}{512}$$

This means the page table occupies only about 0.2% of RAM.

Summary of trade-offs

Inverted page tables offer substantial memory savings, especially on systems with large or sparsely used virtual address spaces. However, the downside is that address translation becomes more complex:

- The system must search (or hash) the page table to find the matching (process ID, virtual page) pair
- This lookup is slower than direct indexing
- TLB caching becomes less straightforward

For this reason, inverted page tables are rarely used in modern general-purpose OSes, though they are still valuable in embedded or resource-constrained systems.

Hierarchical Page Tables

Modern systems (e.g., x86, Linux, Windows) use multi-level page tables to avoid allocating massive single-level tables for sparse address spaces. The key idea is to divide the virtual address into multiple segments, each of which indexes a level in the page table hierarchy. This allows the OS to only allocate memory for regions that are actually used.

Each level of the hierarchy resolves part of the virtual address and points to the next level down. The final level contains the physical frame number. The remaining bits (the offset) are added to form the final physical address.

This approach reduces memory overhead and supports sparse, large virtual address spaces.

32-bit Two-Level Paging Example

Assume a 32-bit virtual address space with:

- Page size = 4 KB = 2^{12}
- 10 bits for the page directory index
- 10 bits for the page table index
- 12 bits for the offset

The virtual address layout is:

```
[ 10 bits | 10 bits | 12 bits ]
   PD index  PT index   Offset
```

Suppose the virtual address is:

`VA = 0x1234ABCD`

Convert to binary:

`0001 0010 0011 0100 1010 1011 1100 1101`

Split:

- Page Directory index = `0001001000` = 0x048 = 72
- Page Table index = `1101001010` = 0x34A = 842
- Offset = `101111001101` = 0xBCD = 3021

Assume:

- The page directory is located at physical address `0x00100000`
- Entry 72 in the page directory points to a page table at `0x00200000`
- Entry 842 in that page table points to a physical frame at `0x00ABC000`

Final physical address:

$$0x00ABC000 + 0xBCD = 0x00ABCBCD$$

This example illustrates how a 2-level table hierarchy resolves the virtual address to a physical address through two indirections and an offset.

64-bit Four-Level Paging Example

Modern x86-64 systems typically support a 48-bit virtual address space, split across four paging levels. The page size remains 4 KB = 2^{12}.

Each level of the hierarchy resolves 9 bits (since $2^9 = 512$ entries per table), so the full 48-bit address is broken into:

```
[ 9 bits | 9 bits | 9 bits | 9 bits | 12 bits ]
  PML4     PDPT     PD       PT       Offset
```

The levels are defined as follows:

- PML4 (Page Map Level 4): The root of the page table hierarchy; indexed by the top 9 bits of the virtual address. Each entry points to a PDPT.
- PDPT (Page Directory Pointer Table): Intermediate level; each entry points to a Page Directory.

- PD (Page Directory): Each entry points to a Page Table.
- PT (Page Table): Final level; each entry contains a physical frame number.
- Offset: Specifies the exact byte within the 4 KB page.

Suppose the virtual address is:

```
VA = 0x00007F34_1234ABCD
```

Breaking down the lower 48 bits:

- PML4 index = bits 47–39 = 0
- PDPT index = bits 38–30 = 505
- PD index = bits 29–21 = 322
- PT index = bits 20–12 = 210
- Offset = bits 11–0 = 0xAF3D = 44861

Assume the following physical mappings:

- CR3 register points to PML4 at `0x00100000`
- PML4 entry 0 → PDPT at `0x00200000`
- PDPT entry 505 → PD at `0x00300000`
- PD entry 322 → PT at `0x00400000`
- PT entry 210 → frame at `0x00ABC000`

Final physical address:

$$0x00ABC000 + 0xAF3D = 0x00B06F3D$$

This example demonstrates how a virtual address is translated step-by-step through four levels of indirection. The layered structure supports extremely large address spaces (up to 256 TB) without requiring full allocation of all intermediate tables.

Summary

Hierarchical page tables solve the scalability problem of flat page tables by breaking the virtual address into multiple segments. Each level of the hierarchy is a smaller table, and lower levels are allocated only when needed. This provides a sparse, memory-efficient structure for address translation.

The cost of additional indirection is mitigated by the use of TLBs, which cache recent address translations to avoid repeated page walks.

When and How Page Tables Are Allocated

Page tables are allocated in two situations:

1. At program load time The OS allocates top-level tables and reserves virtual address regions for code, data, stack, etc., but not necessarily all intermediate tables.
2. On demand via page faults When a process accesses a virtual address with no current mapping, the CPU triggers a page fault. If the access is valid, the OS allocates missing intermediate page tables and a physical frame, updates the page table entries, and resumes execution.

This approach enables sparse memory allocation and efficient use of physical memory.

Physical Memory: Frames and Their Usage

RAM is divided into fixed-size frames (e.g., 4 KB). Each frame can hold:

- A user page (code, stack, heap)
- A page table (of any level)
- A kernel structure
- Other memory-resident objects

Contiguous physical frames may contain completely unrelated contents, as physical memory management is modular and page-based. The OS tracks frame usage via the global frame table.

Kernel Mapping and Access

The kernel is mapped into the upper region of each process's virtual address space (e.g., from `0xC0000000` upward in 32-bit systems). This allows:

- Fast system calls and interrupt handling
- Avoiding page table switches on mode transitions

This region is protected by page table flags, preventing access in user mode. The kernel itself runs entirely in kernel mode. Its physical location is determined at boot and may vary across systems. Techniques like KASLR (Kernel Address Space Layout Randomization) add further protection.

Translation Lookaside Buffer

The TLB is a hardware-managed cache used by the MMU to store recently used virtual-to-physical page translations.

Why it's important:

- Page walks involve multiple memory accesses
- The TLB allows near-instant translation on a hit
- Reduces the average cost of memory access in the presence of multi-level page tables

TLBs are small (typically 16–512 entries) but highly effective due to temporal and spatial locality in most program behavior.

Part II

Networks

3 Network Fundamentals - Summary of Slides 1 - 45

3.1 Overview

Topic	Slide Range	Notes
1. History & Fundamentals of the Internet	Slides 1–16	ARPANET, TCP/IP, WWW, client-server, HTTP statelessness
2. Circuit vs. Packet Switching	Slides 17–23	FDM, TDM, statistical multiplexing, delay tradeoffs
3. Delays, Loss, Throughput	Slides 24–35	d_proc, d_queue, La/R, traceroute, throughput bottlenecks
4. **Protocol Layers and Encapsulation**	**Slides 36–44**	Layer model, encapsulation, host/router/switch roles

3.2 Historical Background and Internet Foundations (Slides 1–16)

- Early developments: ARPANET, Cyclades, and ALOHANet pioneered **packet switching**.
- The **Internet** emerged as a global interconnection of autonomous systems using the **TCP/IP protocol suite**.
- **Tim Berners-Lee's World Wide Web (1989–1991)** introduced:
  - A unified model of hyperlinked documents (HTML)
  - The HTTP protocol (stateless)
  - URLs for addressing
  - The browser-server interaction model
- The **stateless nature of HTTP** means each request is handled independently — servers do not retain memory of previous interactions.

3.3 Transmission Media and Infrastructure (Slides 16–20)

- Data transmission can occur over:
  - **Copper (UTP)**: electrical signals
  - **Fiber optics**: light pulses
  - **Radio**: electromagnetic waves (Wi-Fi, LTE)
- Fiber-optic links offer high bandwidth and low latency — widely used in backbone and undersea cables.
- **Satellite communication** has higher propagation delay (500 ms round-trip) due to distance (~36,000 km geostationary orbit).
- Real-world systems combine many media and technologies in layered infrastructure.

3.4 Circuit Switching vs. Packet Switching (Slides 18–23)

- **Circuit switching**: fixed, reserved paths (e.g. telephony)
  - Uses **TDM (Time Division Multiplexing)** or **FDM (Frequency Division Multiplexing)**
- **Packet switching**: data is broken into packets routed independently
  - Uses **statistical multiplexing**
  - No reservation of bandwidth; packets share the link dynamically
- Trade-offs of packet switching:
  - More efficient use of bandwidth under bursty traffic
  - Potential for packet delay, loss, and reordering

3.5 Network Performance Metrics (Slides 24–32)

Four types of delay:

1. **Processing delay**: time to examine packet header and perform checks
2. **Queueing delay**: time waiting in the router buffer

3. **Transmission delay**:

$$d_{\text{trans}} = \frac{L}{R}$$

where:

- L: packet size (bits)
- R: link bandwidth (bps)

4. **Propagation delay**:

$$d_{\text{prop}} = \frac{d}{s}$$

where:

- d: physical distance (meters)
- s: signal propagation speed (m/s)

Total node delay:

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

Traffic intensity and queue behavior:

Let:

- a: average packet arrival rate (packets/sec)
- L: packet size (bits)
- R: link bandwidth (bps)

Then:

$$\text{Traffic intensity} = \frac{aL}{R}$$

Interpretation:

- If $\frac{aL}{R} \geq 1$: the queue grows without bound
- As $\frac{aL}{R} \to 1$: delay increases sharply

End-to-end delay over multiple hops:

$$d_{\text{end-to-end}} = \sum_{i=1}^{N} (d_{\text{proc},i} + d_{\text{queue},i} + d_{\text{trans},i} + d_{\text{prop},i})$$

where N is the number of routers.

Traceroute:

- Uses IP TTL (Time-To-Live) field to probe each hop
- When TTL reaches zero, routers send an ICMP "Time Exceeded" message
- Allows measurement of **round-trip time (RTT)** per hop

3.6 Throughput (Slides 33–35)

- **Throughput**: the rate at which data is successfully delivered (bps)

Two cases (Slide 34):

If:

- R_S: server's sending rate
- R_C: client-side link rate

Then:

- If $R_S < R_C$, then Throughput $= R_S$
- If $R_S > R_C$, then Throughput $= R_C$

$$\text{Throughput} = \min(R_S, R_C)$$

Multi-user sharing (Slide 35):

If 10 users share a backbone link of rate R, and each has:

- Sender link: R_s
- Receiver link: R_c

Then per-connection throughput is:

$$\text{Throughput} = \min(R_s, R_c, \frac{R}{10})$$

Yes — definitely. The current version is clean and comprehensive, but if your goal is **clarity + conciseness for study purposes**, we can streamline it without sacrificing completeness.

Here is a **more succinct version** of the same summary, optimized for use in study notes:

3.7 Summary: Protokollschichten und ihre Dienstmodelle (Slides 36 - 45)

(Slides 36–44)

This final chapter introduces the **layered architecture of the Internet**. It explains how each protocol layer serves the one above and relies on the one below, and how encapsulation enables structured communication.

Layering Motivation (Slide 36–37)

- Networks are complex (hosts, routers, media, apps).
- Solution: **Schichtenarchitektur** for modular design.
- Each layer performs actions and uses only the services of the layer below.

Protocol Layering: Foundations (Slides 36–38)

- Due to network complexity, functionality is divided into **layers**, each with clear responsibilities.
- A layer k uses only the services of layer $k-1$:

$$\text{Layer } k \longrightarrow \text{uses services of Layer } (k-1)$$

- Each layer communicates **vertically** (service interface) and defines **horizontal protocols** (with its counterpart on the remote host).
- Layering enables:
  - Modularity
  - Replaceability
  - Interoperability
  - Abstraction from hardware details

The Internet Stack (Slide 38–39)

Layer	Function	Examples
Application	Application protocols, user data	HTTP, FTP, SMTP
Transport	Process-to-process delivery	TCP, UDP
Network	Host-to-host delivery, routing	IP, ICMP
Data Link	Frame-level delivery on local links	Ethernet, Wi-Fi, PPP
Physical	Transmission of bits over the medium	Fiber, DSL, 5G

- Layers are identified by **who communicates** (e.g. processes, hosts, links).
- Data is encapsulated step by step as it moves downward.

Protocol Scope by Device (Slide 40)

Device	Implements Up To
Host	All 5 layers
Router	Network layer (IP)
Switch	Data Link layer (MAC)

Encapsulation (Slides 41–43)

Each layer adds its own header (and possibly trailer). The result:

```
Frame = [Data Link hdr] + [IP hdr] + [TCP hdr] + Message + [Trailer]
```

At the receiver, each layer removes its own header.

- Only **hosts** process all layers.
- **Routers** read only IP headers.
- **Switches** forward based on MAC addresses.

OSI Model (Slide 44)

A 7-layer reference model defined by ISO, used mostly for conceptual clarity.

OSI Layer	**Added vs. Internet Model**
7: Application	Matches Internet's application layer
6: Presentation	Data format, compression, encryption
5: Session	Dialog management
4–1: Transport → Physical	Same as in Internet stack

Internet model simplifies OSI: layers 5–7 are often merged into the application.

This version keeps all major points but trims redundant explanation and tightens the wording for effective study reference.

Would you like it exported to markdown or added to your ongoing Quarto notes?

3.8 Unified Protocoll Stack Overview

Layer	Communication Endpoint	Data Unit Name	What It Contains	Adds Header/Footer?	Can Split Data?	Typical Protocols
Application	Applications or processes (e.g., browser web server)	**Message**	App-level data (e.g. HTTP, SMTP)	No	Yes — application logic (e.g. file chunks)	HTTP, FTP, SMTP, DNS, TLS, SSH, POP, IMAP
Transport (TCP/UDP)	Sockets on end hosts (process process)	**Segment**	Message + TCP/UDP header	Yes — transport header	**Yes** — TCP segments long messages	TCP, UDP
Network (IP)	Hosts or end systems (host host, abstracting from processes)	**Packet** (or Datagram)	Segment + IP header	Yes — network header	**Yes** — IP may fragment large packets	IP (v4/v6), ICMP, IGMP
Data Link	Directly connected devices (e.g. Host Router)	**Frame**	Packet + MAC header + trailer (e.g. CRC)	Yes — frame header and trailer	**No** — one packet per frame	Ethernet, Wi-Fi (802.11), PPP, ARP
Physical	Physical interfaces (e.g., NICs, cables, radio) exchanging raw bits	**Bits**	Encoded electrical/optical/radio signals	N/A (not in software)	No — transmits one bit at a time	DSL, Optical Fiber, Ethernet Cable, 5G

4 Intro to the Application Layer, Web and HTTP

4.1 Overview

Slide	Title / Chapter Heading	Content Covered in This Chapter
1	*Title Slide*	Vorlesung N02 – Artur Andrzejak
2	**Protokollschichten und ihre Dienstmodelle** *(Review)*	Motivation for layering, protocol stack overview, encapsulation, OSI model
11	**Das Web und HTTP**	Web as an application, HTTP request/response, URLs, message formats
20	**HTTP – Fortgeschrittene Konzepte**	Persistent connections, cookies, status codes, conditional GET
30	**Grundlagen von Netzwerkanwendungen**	Application architectures (Client-Server, P2P), sockets, addressing
37	**Zwei Grundlegende Internet-Protokolle**	TCP and UDP compared: properties, use cases
38	*(End of main material)*	Table summarizing example applications and their underlying protocols

---

Visual Flow:

1. **Slides 2–10**: Foundations of the Internet stack (5-layer model), encapsulation, and implementation across hosts/routers.
2. **Slides 11–19**: Introduction to the **Web and HTTP** — how web clients and servers communicate, HTTP syntax.
3. **Slides 20–29**: Deeper into HTTP — efficiency issues, cookies, status handling, caching mechanisms.
4. **Slides 30–36**: What constitutes a **network application** — roles of processes, clients/servers, socket interfaces.

5. **Slides 37–38**: **TCP vs UDP** — core transport-layer protocols and which applications use them.

Progression:

- It begins with **protocol layering** as a general framework,
- Moves to a real-world application: the **Web**,
- Adds depth by covering **advanced HTTP behavior**,
- Then generalizes back to all network apps: **architecture, sockets, addressing**,
- And ends with **transport-level choices** (TCP vs UDP).

Absolutely. Below is a more **comprehensive and didactically structured** summary of `vlN02-ibn.pdf`, suitable for integration into your note-taking system. It covers the **main thread** (slides 11–29) and integrates relevant **clarifying detours** for long-term retention.

4.2 IBN – Vorlesung 02: HTTP und die Anwendungsschicht

Source: `vlN02-ibn.pdf` (slides 11–29) **Focus**: Practical introduction to the **Application Layer** of the Internet stack using **HTTP** and the **Web** as central examples.

Context and Approach

You started at **slide 11**, skipping the review of protocol layering (slides 2–10), which was already covered in `vlN01-ibn.pdf`. The lecture introduces how **HTTP** functions as a real-world example of an **application-layer protocol** built on top of **TCP**.

Slide 11 – Das Web und HTTP

- The Web is a **distributed application** using the **HTTP protocol**, layered over **TCP**.
- A single web page typically includes:
  - A main HTML file
  - Additional embedded resources (images, CSS, JS, etc.)

Slide 12 – Übersicht und Begriffe

- Defines key terms:
  - **Browser**: the HTTP client
  - **Web server**: provides content via HTTP
  - **URL**: encodes hostname, protocol, and path to resource
- HTTP clients usually connect to servers on **port 80**.

Slide 14 – Hypertext Transfer Protocol (HTTP)

- HTTP runs over **TCP** and is a **stateless** protocol.
- In its basic form (HTTP/1.0), each request requires a **new TCP connection**.
- Later versions (HTTP/1.1+) support **persistent connections**.

Slide 15 – HTTP-Request (Beispiel)

- Structure of a simple `GET` request:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: ...
```

- `GET` requests contain **no body**.

Slide 16 – Allgemeines Format der Request-Nachricht

- Every HTTP request consists of:
  - **Request line** (method, path, HTTP version)
  - **Header fields** (e.g. Host, User-Agent, Content-Length)
  - Optional **body** (only for `POST`, `PUT`, etc.)

Slide 17 – HTTP-Request-Methoden

- Supported methods:
  - `GET`: parameters in URL
  - `POST`: parameters in body
  - `HEAD`: like `GET` but without response body
  - `PUT`, `DELETE`: less common, used in RESTful APIs
- HTTP is **semantically extensible**, but the method determines message structure.

Slide 18 – HTTP-Response (Beispiel)

- Response structure includes:
  - **Status line** (e.g., `HTTP/1.1 200 OK`)
  - **Header fields** (e.g., `Content-Type`, `Content-Length`)
  - **Optional body** (HTML, image, etc.)

Slide 19 – Allgemeines Format der Response-Nachricht

- Response message layout:
  - **Status line**
  - **Headers**
  - Blank line
  - Optional **entity body**

Slide 21 – HTTP mit Telnet simulieren

- Demonstrates that HTTP is **plain-text based**.
- You can use `telnet` to manually type HTTP requests.
- This reinforces the protocol's **human-readable** structure.

Slide 22 – Statuscodes und Statusnachrichten

- HTTP status line contains:
  - Protocol version
  - Numeric status code (e.g., 200, 404)
  - Optional text message (e.g., "OK", "Not Found")
- Clients should rely on the **numeric code**, not the message text.

Slide 23 – HTTP Verbindungstypen

- Distinguishes between:
  - **Nichtpersistente Verbindungen**: new TCP connection per object (inefficient)
  - **Persistente Verbindungen**: single TCP connection reused for multiple objects (default in HTTP/1.1)

Slide 24 – Antwortzeit bei nichtpersistenten Verbindungen

- Non-persistent setup incurs at least:
  - **1 RTT** for TCP handshake
  - **1 RTT** for HTTP request-response
- This adds up to **2 RTTs + transfer time per object**.
- For pages with many small resources, the overhead becomes significant.

Slide 25 – Vergleich: Persistente vs. Nichtpersistente Verbindungen

- **Persistent connections** reduce:
  - Latency
  - Resource usage
  - Network congestion
- **Non-persistent connections** introduce repeated setup and teardown.

Slide 26 – Benutzerzustand via Cookies

- HTTP is stateless, but **cookies** allow servers to maintain state.
- Mechanism:
  - Server sets a cookie via `Set-Cookie` header
  - Browser stores it and includes it in future requests using the `Cookie` header
- Enables **sessions**, **user tracking**, and **authentication**

Related Discussion: Cookies and Logins

- Login systems rely on cookies to associate a **session ID** with a user.
- The cookie itself does **not contain sensitive data**, only an identifier.
- The session data lives on the server.

Slide 27 – Beispiel: HTTP und Cookies

- First visit: server sets a cookie (`Set-Cookie: ID=12345`)
- Second visit: client automatically sends (`Cookie: ID=12345`)
- Server identifies the session using this ID.

Slide 28 – HTTP: Bedingtes GET

- Introduces the `If-Modified-Since` header:
  - Client asks: *"Has this file changed since* `<timestamp>`*?"*
  - If **no change**, server responds with `304 Not Modified` and no body.
  - Saves bandwidth and improves performance.
- Basis for **browser caching** and **proxy validation**.

Slide 29 – Internet-Protokollstapel

- Overview of the **5-layer Internet protocol stack**:

Layer	Example Protocols	Function
Application Layer	HTTP, FTP, SMTP	Network applications
Transport Layer	TCP, UDP	Reliable or best-effort process communication
Network Layer	IP, Routing Protocols	Packet forwarding across networks
Data Link Layer	Ethernet, PPP	Local frame delivery between adjacent nodes
Physical Layer	(depends on medium)	Bit transmission over hardware links

- Emphasizes **modularity** and **encapsulation**.

Slide 30 – Abschnittsübergang: Grundlagen von Netzwerkanwendungen

Title slide:

Grundlagen von Netzwerkanwendungen *(Architekturen, Sockets, Protokolle)*

Function:

- Marks the **transition** from a concrete case study (HTTP) to **general concepts** in the application layer.
- Sets the stage for understanding **how any application-level protocol is built**, not just HTTP.

Slide 31 – Netzwerkanwendungen sind ...

Core Insight:

- Network applications consist of **processes** (not just hosts) that **exchange messages**.
- These processes run at the **end systems** only — routers and switches do **not** participate in the application logic.

Architecture View:

- Application logic exists **only at the network edges** (hosts), reinforcing the **end-to-end principle**.
- Routers forward packets; they **do not run web servers or browsers**.

Slide 32 – Prozesskommunikation

Key Idea:

> "It's not the computers, but the **processes** that communicate."

- On the same host → **Interprocess Communication (IPC)**
- Across hosts → **Message exchange over TCP or UDP**
- Processes use **sockets** to communicate via the transport layer.

Terminology:

- **Application process** communicates over a socket.
- **Socket** is the OS-provided API to send/receive messages across the network.

Slide 33 – Architektur: Client-Server vs. Peer-to-Peer (P2P)

Client-Server:

- Central server, always on, often with a fixed IP.
- Clients initiate communication.
- Examples: HTTP, SMTP, FTP

Peer-to-Peer:

- No central server; every peer can be both client and server.
- More scalable, but more complex (NAT traversal, coordination).
- Examples: BitTorrent, VoIP (classic Skype)

Slide 34 – Prozesse: Client und Server

Roles defined by behavior:

- **Client**: initiates communication
- **Server**: waits to be contacted

Important:

- These are **roles**, not hardware definitions.
- In P2P, a node can act as **client in one interaction, server in another**.

Slide 35 – Sockets: Schnittstellen

Sockets:

- The **programming interface** between the application process and the transport layer.
- A socket is like a **door** through which the process sends/receives data.

OS abstraction:

- OS handles TCP/UDP details.
- Application uses `send()`, `recv()`, or `read()/write()` on the socket.

Slide 36 – Addressieren von Prozessen

Problem:

- IP address identifies a **host**, not a **specific process**.

Solution:

- Use `(IP address, Port number)` to identify a process.
- Known ports (e.g. `80` for HTTP, `443` for HTTPS) identify standard services.

Slide 37 – Zwei grundlegende Internet-Protokolle: TCP und UDP

Feature	TCP	UDP
Connection setup	Yes (3-way handshake)	No
Reliability	Yes	No
Ordering	Guaranteed	No
Congestion control	Yes	No
Use case examples	HTTP, FTP, email	DNS, VoIP, streaming

- TCP is used when **reliability and ordering** matter.
- UDP is used when **speed and simplicity** are more important than reliability.

Slide 38 – Beispiele für Anwendungen

Protocol usage by application:

Application	Protocol
HTTP, FTP, SMTP	TCP
DNS (queries)	UDP
DNS (zone transfer)	TCP
VoIP, video stream	UDP

- Some apps use **both**, depending on use case (e.g. DNS).

4.3 Clarifying Detours and Triggers

1. "Do HTTP servers need to know the client's IP address?"

Triggered by: slide 35 (Sockets) **Clarification**:

- No, not usually. The OS handles delivery.
- The server can access the client IP via `accept()` if needed, but it's optional.

2. "Is it realistic to build your own HTTP server?"

Triggered by: understanding sockets in slide 35 **Discussion**:

- Yes — very realistic.
- A minimal HTTP server can be written in under 50 lines of Python using just the socket API.
- Focus is on parsing requests, forming responses, serving files, and handling concurrency.

3. "Why do Apache and Nginx exist if HTTP is simple?"

Triggered by: realization that building a server is feasible **Clarification**:

- Industrial servers are engineered for performance, security, flexibility, and scaling.
- They handle advanced features like TLS, compression, load balancing, and dynamic routing.

4. "Can an HTTP server listen on multiple ports?"

Triggered by: slide 36 (port numbers) **Clarification**:

- Yes — it's technically trivial (e.g. `bind()` on multiple sockets).
- Often done for development (`8080`), reverse proxies, or alternate services.

5. "Do P2P applications prefer UDP over TCP?"

Triggered by: slide 33 (P2P architectures) **Clarification**:

- Many do prefer UDP due to NAT traversal, lower latency, and flexible retransmission control.
- But some, like BitTorrent, still use TCP for reliability.

Practical Experiment

You tested this on your **Quarto preview server** (`localhost:3475`) and used `telnet` to manually send a `GET /` request. The server responded with:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: ...
```

This confirmed:

- HTTP is **text-based**
- The Quarto server is a **fully functional HTTP server**
- The response body was a real HTML document rendered by the browser

---

1. Chunking and Large Files

What initiated it: Slide 18 showed a large HTML response. You asked:

> "If the file is large, is it split into multiple HTTP messages?"

What we explored:

- HTTP responses are **not split** at the application level
- **TCP handles segmentation**
- **Chunked transfer encoding** exists for streaming, but still within one HTTP response

---

2. Opening Local HTML Files in Browser

What initiated it: You asked how browsers handle local `.html` files opened by double-clicking.

What we explored:

- If all resources (CSS, JS) are **embedded**, the file renders correctly
- If external files are **fetched via HTTP**, they fail unless a **local server is running**
- Browsers block `fetch()` from `file://` for security

---

3. What is an SPA (Single Page Application)?

What initiated it: You observed that SPAs avoid reloading full HTML pages and instead dynamically update content.

What we explored:

- SPAs load one HTML file and update the DOM using JavaScript
- No page reloads = smoother experience
- Commonly powered by frameworks like **React**, **Vue**, etc.

---

4. JSON, REST, and GraphQL

What initiated it: You asked how SPAs fetch data and render views without loading new HTML pages.

What we explored:

- SPAs fetch **JSON** from the server
- REST APIs expose fixed endpoints (`/api/posts`)
- GraphQL allows flexible, structured queries (`{ post { title } }`)
- Backend sends **data**, not HTML

---

5. Role of PostgreSQL and Backend

What initiated it: You asked how GraphQL or REST interfaces connect to a **PostgreSQL database**.

What we explored:

- The **backend server** receives GraphQL or REST requests
- It queries **PostgreSQL** for data
- Responds with JSON to the frontend

---

6. Authentication and Password Handling

What initiated it: You asked how user login works — and whether passwords are visible to the server.

What we explored:

- Passwords are hashed using **bcrypt** and stored in the database
- Only hashed values are stored; plain passwords are never saved
- After login, the server issues:
  - A **session cookie**, or
  - A **JWT** (JSON Web Token)
- The frontend uses the token to authenticate future requests

---

7. HTTPS and Security

What initiated it: You noticed the password is sent in plain text within the HTTP body.

What we explored:

- Without HTTPS, this is dangerous — passwords can be intercepted
- HTTPS encrypts the entire transmission (headers + body)
- Modern login flows **require HTTPS** for secure password handling

---

8. Final Integration: Full-Stack SPA Project

What initiated it: You asked whether this could be integrated into a learning project.

What we designed:

- A minimal blog platform (**MiniPost**) with:
  - SPA frontend (React or Svelte)
  - GraphQL backend (Node.js)
  - PostgreSQL database
  - Login/authentication (JWT + bcrypt)
- Designed as a hands-on way to tie everything together

---

1. What is a TCP segment?

- A **TCP segment = TCP header + data payload**
- Sent over IP; reassembled on the receiving side using SEQ/ACK numbers

2. What is a byte stream?

- TCP provides a **reliable, ordered byte stream**
- No inherent message boundaries — just a flow of bytes
- Contrast with message-oriented protocols like UDP

3. How does TCP know transmission is over?

- Sender signals with a **FIN** segment
- Connection is closed using a **4-step FIN/ACK handshake**

4. What does "contiguous bytes received" mean?

- TCP only acknowledges data that has been received **in order**, without gaps
- If segments arrive out of order, ACK does **not advance** until missing parts are filled in

5. Duplicate segment handling

- If the same segment arrives twice (e.g. delayed + retransmitted), TCP detects overlap using SEQ numbers and **silently discards** duplicates

6. Do segments usually arrive in order?

- Yes, most of the time — but TCP is built to tolerate:
  - **Moderate reordering**
  - **Delayed or duplicate segments**

7. When and how retransmission occurs

- **Fast Retransmit** is triggered by **3 duplicate ACKs**
- **Timeout-based retransmission (RTO)** occurs when ACKs don't arrive
- Fast retransmit is called "fast" because it reacts quicker than waiting for RTO

5 IP Layer & Subnetting

5.1 Summary of VLN03 – Internet Protocol and Addressing (Slides 3–17)

Overview

This lecture begins the deep dive into the **Network Layer** of the Internet stack — specifically the **Internet Protocol (IP)**. We analyze how data is transmitted between hosts across networks, how datagrams are structured, and how IP addresses define logical and physical boundaries within networks.

Several **side explorations** were included to clarify underlying technologies (e.g. ARP, DHCP, NAT) and real-world applications (e.g. dormitory network issues, personal Wi-Fi routing).

---

1. HTTP Traffic and Wireshark (Slides 3–7)

Wireshark is used to capture and inspect real HTTP traffic.

Key points:

- HTTP messages are captured and decoded by Wireshark.
- These are transported over **TCP**, which is encapsulated in **IP**, which is wrapped in **Ethernet frames**.

Reassembly in Wireshark:

Wireshark reconstructs **complete HTTP messages** from **multiple TCP segments** using internal state tracking.

Key term: PDU (Protocol Data Unit)

A PDU is the **unit of data** defined at a specific protocol layer.

Layer	PDU Name	Example
Application Layer	Message	HTTP request or response
Transport Layer	Segment / Datagram	TCP segment, UDP datagram
Network Layer	Datagram	IP packet
Link Layer	Frame	Ethernet frame
Physical Layer	Bits	Electrical or optical signals

---

2. Introduction to the IP Layer (Slides 8–10)

We zoom into the **Network Layer** of the Internet stack.

Main components of the network layer:

1. **IP protocol**:
    - Packet format, addressing, and forwarding
    - Stateless, best-effort delivery
2. **Routing protocols**:
    - Determine forwarding tables (e.g. RIP, OSPF, BGP)
3. **ICMP**:
    - Used for diagnostics (e.g. `ping`, `traceroute`)

Key term: Datagram

A **datagram** is a self-contained, independently routed network-layer packet. In IP networks, it refers to an **IP packet** that includes both header and payload.

- Connectionless
- May be fragmented
- Carries no session state

IP packets = IP datagrams (used interchangeably)

---

3. Structure of an IP Datagram (Slide 11)

We study the fields of the IPv4 header.

IP Header Fields (simplified):

Field	Size (bits)	Purpose
Version	4	IP version (usually 4)
Header Length	4	In 32-bit words
DS/ECN	8	Differentiated Services / Congestion
Total Length	16	Header + payload length
Identification	16	Fragmentation ID
Flags	3	Fragment control
Fragment Offset	13	Position of fragment
TTL	8	Time to live
Protocol	8	Higher-layer protocol (e.g. TCP = 6)
Header Checksum	16	Detects corruption in header
Source IP	32	IP of sender
Destination IP	32	IP of receiver
Options (optional)	variable	Rarely used

---

4. IP Fragmentation (Slides 12–13)

Why fragmentation exists:

- Ethernet and other link-layer technologies impose an **MTU (Maximum Transmission Unit)**.
- IP allows a large datagram to be split into smaller **fragments** if the MTU is too small.

Fragmentation fields:

- **Identification**: Shared across all fragments of a datagram
- **Fragment Offset**: Position (in 8-byte units)
- **MF (More Fragments)**: Flag set to 1 for all fragments except the last

  Only the **destination host** reassembles fragments. Routers never do.

Example (simplified):

A 3072-byte datagram sent over a 1200-byte MTU link:

Fragment	Offset	MF	Data bytes
1	0	1	960
2	120	1	960
3	240	0	1020

---

5. Interfaces and Addressing (Slides 14–15)

Key concepts:

- **An IP address is assigned to a network interface**, not the host as a whole.
- A **router has multiple interfaces**, each on a different subnet.
- A **host usually has one IP**.
- Devices in the same subnet share the same **address prefix**.

Reference network diagram from slide 15:

- **Router interfaces**:
  - `223.1.1.4` (Subnet A, left)
  - `223.1.2.9` (Subnet B, right)
  - `223.1.3.27` (Subnet C, bottom)
- **Subnets and hosts**:

Subnet	Hosts	Router IP
A	223.1.1.1 – 1.3	223.1.1.4
B	223.1.2.1 – 2.2	223.1.2.9
C	223.1.3.1 – 3.2	223.1.3.27

All IPs within a subnet share the same `/24` prefix.

---

6. LANs, WLANs, and Subnets (Discussion and Detours)

Triggered by slide 15 and related questions.

Clarifications:

- A **LAN** is a link-layer broadcast domain (Ethernet, WLAN).
- A **subnet** is a logical grouping of IPs — typically matches a LAN, but not always.
- A **WLAN** is a type of LAN using wireless physical media.

Key term: ARP (Address Resolution Protocol)

ARP resolves IP addresses to MAC addresses on a LAN.

- A device sends a broadcast:

  ```
  Who has 192.168.1.5?
  ```

- The target replies with its MAC address.

- The result is cached in the **ARP table**.

  Without ARP, IP-based communication over Ethernet is impossible.

ARP Spoofing

A malicious device sends fake ARP replies:

- "I am 192.168.1.1 — here's my MAC"
- Can hijack traffic (man-in-the-middle)

This only works on **shared LANs** without port isolation.

---

7. Dorm Network Architecture (Applied Detour)

Triggered by IP addressing and subnet questions

- Dorm connections are likely per-port VLANs
- Ethernet switch: exposes multiple MACs/IPs to Nexabit
- Wi-Fi AP (with NAT): hides multiple devices behind one IP
- Nexabit bans switches/APs due to fragility, not inherent danger

  Your dorm disconnection incident may have occurred due to multiple devices behind a dumb switch overwhelming fragile infrastructure — not misuse on your part.

---

8. Subnet Hardware Definition (Slide 17)

A subnet consists of **all interfaces** connected to the **same physical link-layer medium**, excluding routers.

Visual trick: if you "cut away" each router's interfaces, the **remaining connected islands** are subnetworks.

In the slide:

- Regions 1–3: LAN subnets (hosts + router)
- Regions 4–6: router-router links (point-to-point subnets)

5.2 Summary of VLN03 — Slides 18–23

(IP Address Blocks, Subnets, and Classful Addressing)

Subnets and Minimal Address Blocks

We revisit the subnet diagram from slide 15 and ask: **What is the minimal CIDR block that covers all required IPs for Subnet A and B?**

Although it might seem sufficient to use the numerical range of required IPs (e.g. `.1-.4`), IP subnets must follow strict rules:

- A subnet must be a **CIDR-aligned block**: its size must be a power of two, and its starting address must be divisible by that size.
- Therefore, to include `223.1.1.4`, Subnet A requires a `/29` block: `223.1.1.0 - 223.1.1.7`
- To include `223.1.2.9`, Subnet B requires a `/28` block: `223.1.2.0 - 223.1.2.15`

These blocks ensure binary-aligned network prefixes suitable for routing.

NetID and HostID Structure

An IP address is logically split into:

- **NetID** — determines the network or subnet
- **HostID** — identifies a host within that network

The boundary is defined by the CIDR prefix (e.g. `/24`), where the first `n` bits are the NetID, and the remaining `32-n` bits are the HostID.

This structure enables:

- **Scalable routing** — routers forward based only on the NetID
- **Local autonomy** — host addresses are managed internally

Routing efficiency increases because routers don't need entries for individual hosts — only for the networks.

Address Allocation and Prefix Sizing

How long should the NetID be? It depends on the use case:

- A **long NetID** (e.g. `/27`) allows many networks with few hosts
- A **short NetID** (e.g. `/16`) allows fewer networks but many hosts per network

This trade-off determines how many subnets or hosts fit in a given address block. Organizations receiving a larger block (e.g. `/16`) can internally create finer subdivisions using longer prefixes (`/24`, `/28`, etc.).

Classful Addressing

Historically, IP addresses were divided into rigid **classes**, which defined fixed NetID/HostID splits:

Class	NetID Bits	HostID Bits	Address Range	Notes
A	8	24	`1.0.0.0 - 126.255.255.255`	Large orgs
B	16	16	`128.0.0.0 - 191.255.255.255`	Mid-size
C	24	8	`192.0.0.0 - 223.255.255.255`	Small orgs
D	—	—	`224.0.0.0 - 239.255.255.255`	Multicast

This system was easy for routers to interpret (based on the first few bits), but led to massive address waste. For example, a company needing 500 IPs couldn't use a `/24` (too small), so it had to request a `/16` block — wasting over 60,000 addresses.

Detours and Clarifications

What is a CIDR-aligned block?

A block of size 2^k must begin at an address divisible by 2^k and span exactly that many addresses. Arbitrary ranges like `.1-.4` are not valid subnet definitions.

What is `x.x.x.x/y` notation?

CIDR notation explicitly states the NetID length. `/24` means 24 bits of network, 8 bits of host. The prefix length determines both address block size and routing behavior.

Why longest prefix match?

When multiple routing entries match a destination, routers choose the one with the **longest (most specific) prefix**. This ensures fine-grained routing within larger aggregate blocks.

How does a packet reach the final host?

When a packet reaches the destination subnet, the router uses **ARP** to resolve the destination IP to a MAC address, then sends the packet as an Ethernet frame. The target host accepts the frame based on MAC address filtering.

What happens on a shared medium without a switch?

On a shared Ethernet or Wi-Fi network, all hosts receive all frames. Only the device whose NIC matches the destination MAC address accepts the packet; the rest discard it silently.

Does each NetID correspond to a single subnet?

Not necessarily. A large NetID (e.g. `/16`) can be internally **subnetted** into multiple smaller blocks (`/24`, `/27`, etc.). Subnetting is hierarchical and flexible under CIDR.

Could organizations subnet class-based address blocks?

Yes. Even under the old class-based system, organizations could **borrow bits from the HostID portion** to create subnets internally. For example, a Class B `/16` block could be subnetted into 256 `/24` blocks using a subnet mask like `255.255.255.0`. This allowed structured internal networks despite the rigidity of the external class system.

Certainly — here is the same answer, reformatted according to your project's preferences (no boldface in headings, clean structure, sparing emphasis in body text).

---

6 Summary of VLN03 — Slides 24–38

(From classless addressing to routing and network segmentation at Uni Heidelberg)

6.1 1. From Classful to Classless Addressing (Slides 24–25)

Problems with classful addressing

- Waste of addresses: e.g. a Class B allocation (65k addresses) for an org needing only a few hundred
- No aggregation: every network required a separate route — routing tables became large and unscalable
- Chaotic distribution: no topological coherence; blocks were allocated sequentially
- Frequent updates: every new network affected global routing tables

Solution: CIDR (Classless Inter-Domain Routing)

CIDR was introduced in 1993 to allow arbitrary prefix lengths:

```
a.b.c.d/x   →  x = number of NetID bits
```

CIDR enables:

- Fine-grained allocation (e.g. /23, /26, /30)
- Route aggregation in global routing tables:
  - e.g. 192.168.0.0/22 covers 4 contiguous /24 networks
- Scalable design: organizations get only what they need; global tables remain compact

---

6.2 2. Routing hierarchy and subnetting (Slides 26–28)

Hierarchical routing

- Global routers (ISPs) see only top-level blocks (e.g. /16)
- Institutional routers handle internal subnetting and distribution
- Local routers (e.g. in buildings) know the details of subnet-to-interface mapping

Subnetting strategy

From a CIDR block (e.g. /16), an institution can create multiple smaller subnets (/24, /22, etc.) Bits from the HostID are borrowed to form a SubnetID.

Address structure with subnetting

Field	Bit Count	Description
NetID	e.g. 16	Assigned by provider (e.g. 129.206)
SubnetID	e.g. 8	Internal routing (e.g. Informatik = 78)
HostID	e.g. 8	Host within the subnet (e.g. .42)

Example: `129.206.78.42` NetID = 129.206, SubnetID = 78, HostID = 42

6.3 3. Subnet masks and CIDR application (Slides 29–31)

What is a subnet mask?

A 32-bit value used to separate:

- Network + Subnet part (leftmost bits)
- Host part (remaining bits)

Notation

- CIDR: `/x`, where `x` = number of 1s in mask
- Dotted decimal: `255.255.255.0` `/24`

Example calculation

```
IP address:      134.155.48.10
Subnet mask:     255.255.255.0 (/24)
→ Subnet base:   134.155.48.0
```

Use case

Hosts check if another IP is in the same subnet using a bitwise AND with the subnet mask. If not → packet forwarded to default gateway.

6.4 4. Visibility and internal routing (Slides 32–33)

SubnetID is local

Only the top-level prefix (e.g. 129.206.0.0/16) is visible globally. Internal subnet structure (e.g. .78.0/24, .61.0/24) is managed inside the university.

Example from Uni Heidelberg

- Public /16 blocks:
  - 129.206.0.0/16 (north of Neckar)
  - 147.142.0.0/16 (south of Neckar incl. Mathematikon)
- Subnet allocations:

Department	Subnet	Size
Medizin	/18	16,384 IPs
Physik	/18	16,384 IPs
Informatik	/22	1,024 IPs

- One AG (research group) was assigned:

  `129.206.61.39 - 129.206.61.63`

  → 24 usable public IPs, not a full subnet, but a carved-out range from the /24

6.5 5. Physical topology of the Mathematikon (Slides 34–35)

- Two routers in the basement act as:
  - Gateways to the university backbone
  - Entry points to the building network
- IPs of routers: .1 in each subnet (e.g. 129.206.78.1)
- 13 CISCO Catalyst 4500 switches across floors
- Connected via fiber → act as one logical switching fabric
- Enables VLANs to span all floors

6.6 6. VLANs and subnet spanning (Slide 36)

VLAN (Virtual LAN)

A VLAN is a logically segmented subnet that can span multiple physical switches.

- Switch ports are assigned to VLANs (e.g. VLAN 78 → 129.206.78.0/24)
- Provides:
  - Traffic isolation
  - Security
  - Flexibility: any office on any floor can be part of the same subnet

VLANs allow centralized IP and subnet control across a distributed physical environment.

6.7 7. End of subnetting discussion (Slide 37)

This slide links to a video: **Cisco Router Training 101**, starting at 16:30 — reviewing subnet masks and routing behavior.

6.8 8. Transition to routing (Slide 38)

The final slide begins the topic of **packet forwarding and routing tables**, continued in VLN04.

---

7 Detours and enrichments

7.1 How a packet reaches a subnet host

Prompted by: Slide 28 — internal subnet structure

We analyzed:

- Global routing to `/16` block
- Internal forwarding to subnet router
- Final delivery via ARP and Ethernet to the destination host

---

7.2 Do routers use MAC addresses?

Prompted by: Routing behavior

Clarified:

- Routers have MAC addresses on each interface
- MAC addresses are rewritten per hop (layer 2), while IP stays constant (layer 3)

7.3 Reverse proxies and ngrok

Prompted by: Slide 33 — public IPs in Informatik

We discussed:

- Reverse proxies (nginx, Apache, etc.) forward requests from clients to internal servers
- ngrok acts like a hosted reverse HTTP proxy using a tunnel to expose local services
- Differences between public IP access and NAT + proxy setups

---

7.4 Why research groups might get public IPs

Prompted by: Slide 33 — “meiner AG wurden 24 Adressen zugeteilt”

We explored:

- Universities with large legacy blocks can afford to assign public IPs directly
- Benefits: easy incoming connections, public DNS mapping, no NAT hassle
- Risk: requires strong firewall policies

7.5 What is ARP?

Prompted by: Final packet delivery on LAN

Defined ARP as the Address Resolution Protocol — maps IP to MAC within a subnet using broadcast, enabling packet delivery at layer 2.