Summary for Advanced Computer Systems at University of Copenhagen 2021/2022. These notes are mostly based on the lecture slides and reading material.

Fundamentals

Common Problems of Systems

- **Emergent Properties**
  properties not showing up in individual components, but when combining those components
- **Propagation of Effects**
  what looks at first to be a small disruption or a local change can have effects that reach from one end of a system to the other
- **Incommensurate Scaling**
  as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working
- **Trade-offs**
  waterbed effect: pushing down on a problem at one point causes another problem to pop up somewhere else

System technical definition:

A **system** is a set of interconnected components that has an expected behavior observed at the interface with its environment.

Divide all the things in the world into two groups:

- those under discussion (part of the system)
- those that are not part **(environment)**
- the interactions between system and its environment are the **interface** between the system and environment

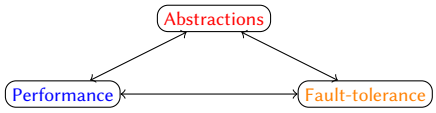
Fundamentals

- Abstractions: interpreters, memory, communication links
- Modularity with clients and services (RPC)
- Techniques for performance

Learning goals

- Identify the fundamental abstractions in computer systems
- Explain how names are used in the fundamental abstractions
- Being able to design a top-level abstraction, based on lower-level abstractions
- Discuss performance and fault-tolerance of a design

Central Trade-off: Abstractions, Performance, Fault-Tolerance



Examples for Trade-off

- To improve performance one might has to ignore the abstraction and take the behavior of the underlying concrete implementation into account
- when introducing another layer of abstraction we might introduce new kinds of errors (for example when introducing RPC, we can have communication errors)
- introducing mechanisms for fault-tolerance can have a negative effect on performance

Names

Names make connections between the different abstractions.

- Examples
  - IP-address
  - IR
- Names require a mapping scheme
- How can we map names?
  - Table lookup (e.g. Files inside directories)
  - Recursive lookup
  - Multiple lookup

Memory

- READ(name) → value
- WRITE(name, value)

Examples of Memory

- Physical memory (RAM)
- Multi-level memory hierarchy
- Address spaces and virtual memory with paging
- Key-value stores
- Database storage engines

Interpreters

Interpreter has:

- Instruction repertoire
- Environment
- Instruction pointer

Interpretation Loop:

```
do forever
  instruction <- READ(instruction_pointer)
  perform instruction in environment context
  if interrupt_signal = True
    instruction_pointer <- entry of INTERRUPT_HANDLER
    environment <- environment of INTERRUPT_HANDLER
```

Examples of Interpreters:

- Processors (CPU)
- Programming language interpreters
- Frameworks (e.g. MapReduce, Spark)
- layered programs (RPCs)

Communication links

- SEND(linkName, outgoingMessageBuffer)
- RECEIVE(linkName, incomingMessageBuffer)

Examples of Communication Links:

- Ethernet interface
- IP datagram service
- TCP sockets
- Message-Oriented Middleware (MOM)
- Multicast (e.g. CATOCS Causal and Totally-Ordered Communication System)

Other Abstractions

- Synchronization
  - Locks
  - Condition variables & monitors
- Data processing
  - Data transformations
  - Operators

Modularity through Clients and Services, RPC

Layers and Modules

- Interpreters often organized in layers
- Modules
  - Components that can be separately designed / implemented / managed / replaced (*Saltzer & Kaashoek glossary*)
  - "Instructions" of higher-level interpreters
  - Recursive: can be whole interpreters themselves!

Isolating Errors: Enforced Modularity

Problem: What happens when modules fail with (unintended) errors?

→ only that particular module should fail rest of system should still work (for this need enforced modularity)

Example: Clients & Services

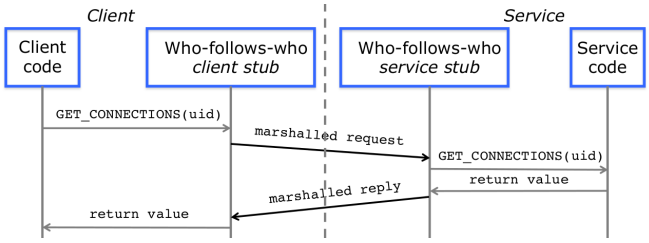
- Restrict communication to *message only*
- Client request / Service response (or reply)
- Conceptually client and service in different computers

Example: OS Virtualization

- Create virtualized version of fundamental abstraction
- Client and services remain isolated even on same computer
- VMs: virtualize the virtualizer

RPC: Remote Procedure Call

- Client-service request / response interactions
- Automate marshalling and communication



RPC Semantics

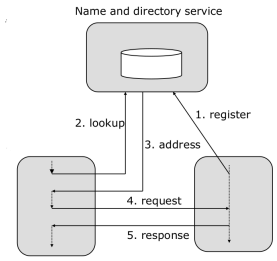
- **At-least-once**
  - Operation is *idempotent* (naturally occurs if side-effect free)
  - Stub just retries operation → failures can still occur!
  - Example: calculate SQRT
- **At-most-once**
  - Operation does have side-effects
  - Stub must ensure duplicate-free transmission
  - Example: transfer $100 from my account to yours
- **Exactly-once**
  - possible for certain classes of failures
  - Stub & service keep track (*durably*) of requests and responses
  - Example: bank cannot develop amnesia!

How to achieve RPCs?

- Special-purpose **request-reply protocol** e.g. DNS
  - Developer must design protocol and marshalling scheme
- **Classic RPC** protocols, DCE, Sun RPC
  - Special APIs and schemes for marshalling
- **RMI: Remote Method Invocation**
  - RPCs for methods in OO languages
  - Compiler-generated proxies
- **Web Services**
  - many modes of communication possible, including RPC-style communication
  - Tools available to compile proxies, e.g., JAX-WS
  - Generic marshalling (e.g., XML, JSON, Protocol Buffers) over HTTP transport
    → **programming-language Independence**

RPC and Naming

- Most basic extension to the synchronous interaction pattern
  - Avoid having to name the destination
  - Ask where destination is
  - then bind to destination
- Advantages:
  - Development is independent of deployment properties (e.g. network address)
  - more flexibility
    - change of address
  - Can be combined with
    - Load balancing
    - Monitoring
    - Routing
    - Advanced service search



Common Issues in Designing Services

- Consistency
  - How to deal with *updates* from multiple clients?
- Coherence
  - How to refresh caches while respecting consistency?
- Scalability
  - What happens to resource usage if we increase the #clients or the #operations?
- Fault Tolerance
  - Under what cirumstances will the service be unavailable?

Other Examples of Services

- File systems: NFS, GFS
- Object stores: Dynamo, PNUTS
- Database: relational DB
- Configuration: Zookeeper
- Even whole computing clouds!
  - Infrastructure-as-a-service (IaaS): e.g. Amazon EC2
  - Platform-as-a-service (PaaS): e.g. Windows Azure
  - Software-as-a-service (SaaS): e.g. Salesforce, Gmail

Techniques for Performance

Motivation: Abstractions, Implementation and Performance

Let I_1 and I_2 be two implementations of an abstraction

- Examples
  - Web service with or without HTTP proxies
  - Virtual memory with or without paging
  - Transactions via concurrency or serialization

⇒ **How can we choose between I_1 and I_2?**

Performance Metrics

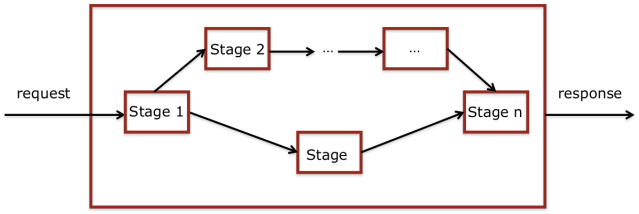
- **latency:** The delay between a change at the input to a system and the corresponding change at its output. From the client/service perspective, the latency of a request is the time from issuing the request until the time the response is received from the service.
- **throughput:** Is a measure of the rate of useful work done by a service for some given workload of requests. If processing is serial, then throughput is inversely proportional to the average time to process a single request

$$throughput = \frac{1}{latency}$$

  If processing is concurrent, then no direct relationship between latency and throughput.
- **scalability:** Scalability is the property of a system to handle a growing amount of work by adding resources to the system
- **overhead:** In a layered system, each layer may have a different view of the capacity and utilization of the underlying resources. Each layer considers what the layer below it do to be *overhead* in time and space, what the layers above it do to be *useful work*.
- **utilization:** The percentage of capacity used for a given workload
- **capacity:** Any consistent measure of the size or amount of a resource.

How can we improve performance?

- **Fast-path coding**
  - Split processing into two code paths
  - One optimized for common requests → fast path
  - One slow but comprehensive path for all other requests → slow path
  - Example: Caching



- **Batching**
  - Run multiple requests at once
  - Example: batch I/Os and use elevator algorithm
  - May improve latency and throughput
- **Dallying**
  - Wait until you accumulate some requests and then run them
  - Example: group commit
- **Concurrency**
  - Run multiple requests in different threads
  - May improve both throughput and latency, but must be careful with locking (overhead), correctness
  - Can be hidden under abstractions (e.g. MapReduce, transactions)
  - Example: different web requests run in different threads or even servers
- **Speculation**
  - Guess the next requests and run them in advance

- May overlap expensive operations, instead of waiting for their completion
- Example: prefetching

Communication

Learning goals
- approaches to design **communication abstractions**
  - transient vs. persistent
  - synchronous vs. asynchronous
- Design and implementation of **message-oriented middleware** (MOM)
- organizing systems using **BASE** methodology
- relationship BASE to eventual consistency and **CAP theorem** (it is impossible to provide **C**onsistency, **A**vailability and **P**artition Tolerance at the same time in distributed systems)
- alternative communication abstractions **data streams** and **multicast / gossip**

Partitioning
- use independent services to store different data
- Scalability and Availability improves but now coordination necessary
- employ a communication abstraction

Recall: Protocols and Layering in the Internet
- Layering
  - System broken into vertical hierarchy of protocols
  - Service provided by one layer based solely on service provided by layer below
- Internet model (here four layers)
  - **Application** (e.g. HTTP, DNS, Email..)
  - **Transport** (TCP, UDP)
  - **Network** (IP)
  - **Link** (Ethernet)

Recall: Layers in Hosts and Routers
- Link and network layers implemented everywhere
- End-to-end layer (i.e., transport and application) implemented only at hosts

Different Type of Communication
- Asynchronous: sender can immediately return after sending message without waiting for any acknowledgments from other components
- Synchronous
  - Synchronize at request submission (1)
  - Synchronize at request delivery (2)
  - Synchronize after being fully processed by recipient (3)
- **Transient** vs. **persistent** (temporal decoupling): in persistent communication the sender can send a message and go offline and the receiver can go online at any time and gets the message that was send to him. Sender and receiver can be independent in terms of operation time.

Examples:

- **Persistent** and **Asynchronous**: Email
- **Persistent** and **Synchronous**: Message-queuing systems (1), Online Chat (1) + (2)
- **Transient** and **Asynchronous**: UDP, Erlang
- **Transient** and **Synchronous**: Asynchronous RPC (2), RPC (3)

RPC
- Transparent and hidden communication
- Synchronous
- Transient

Message-Oriented
- Explicit communication SEND/RECEIVE of point-to-point messages
- Synchronous vs. Asynchronous
- Transient vs. Persistent

Message-Oriented Persistent Communication
- Queues make sender and receiver loosely-coupled
- Modes of execution of sender/receiver
  - both running
  - Sender running, Receiver passive
  - Sender passive, Receiver running
  - both passive

Queue Interface
- Put: Put message in queue
- Get: Remove first message from queue (blocking)
- Poll: Check for message and remove first (non-blocking)
- Notify: Handler that is called when message is added

Source / destination are decoupled by **queue names**

Multiple nodes (distributed system) inside message queuing system for high throughput communication

- **Relays:** store and forward messages
- **Brokers:** gateway to transform message formats

How to Employ Queues to Decouple System
Example Bank Transfer

- for scalability partition accounts onto different computers
- use message queue between two accounts
- for ensuring **atomicity** we need to use commit protocol (two-phase commit)

The CAP theorem
CAP Theorem: A scalable service cannot achieve all three properties simultaneously

- Consistency: (i.e. atomicity) client perceives set of operations occurred all at once
- Availability: every request must result in an intended response
- Partition tolerance: operations terminate, even if the network is partitioned

ACID
- Using 2PC (two phase commit) guarantees atomicity (C in CAP)
- If 2PC is used, a transaction is not guaranteed to complete in the case of network partition (either Abort or Blocked) → we choose Consistency over Availability
- If an application can benefit from choosing A over C we need different method (BASE). For example in a social network its more important to have availability.

BASE
- **Basically-Available:** only components affected by failure become unavailable, not whole system
- **Soft-State:** a component's state may be out-of-date, and events may be lost without affecting availability
- **Eventually Consistent:** under no further updates and no failures, partitions converge to consistent state

A BASE Scenario
- Users buy and sell items
- simple transaction for item exchange

Now we can **Decouple Item Exchange with Queues**, by having one component taking care of transactions and the other one of users and updates to them. The following issued arise

- Tolerance to loss
- Idempotence
- Order: order can be implemented by a last transaction pointer in the receiver, would be to restrictive on queue implementation to ensure a order

Other Types of Communication Abstractions
- **Stream-Oriented**
  - Continuous vs. discrete
  - Asynchronous vs. Synchronous vs. Isochronous
  - Simple vs. complex
- **Multicast**
  - SEND/RECEIVE over groups
  - Application-level multicast vs. gossip

Gossip
- Epidemic protocols
  - No central coordinator
  - nodes with new information are infected, and try to spread information
- Anti-entropy approach
  - Each node communicates with random node
  - Round: every node does the above
  - Pull vs. push vs. both
  - Spreading update form single to all nodes takes O(log(N))
- **Gossiping**
  - more similar to real world analogy
  - If P is updated it tells random Q
  - if Q already knows, P can lose interest with some percentage
  - at some threshold P stops telling random nodes
  - Not guaranteed that all nodes get infected by update

Data Processing: Basic Concepts and External Sorting

Different Cost Models for Algorithms

- **RAM model**
  - Every basic operation takes constant time
  - Memory access, simple additions, does not matter
- **I/O model**
  - Transfer data from disk in large blocks / pages
  - Count number of I/Os performed
  - **Assumption:** I/Os dominate total cost, any I/O as good as another one → not always true
- **More sophisticated cost models**
  - Create cost function which mixes CPU, memory access, I/O costs
  - Differentiate types of access patterns (sequential, random, semi-random, etc)
  - Complexity can grow very high, very quickly

Example for first external memory algorithm: Sorting

Why sorting?

- Important in data processing (relational queries)
- Used for eliminating duplicates (SELECT DISTINCT)
- Bulk loading B+ trees (need to first sort leaf level pages)
- Data requested in sorted order
- Some join algorithms use sorting
- Some MapReduce implementations use sorting to group keys for reducers

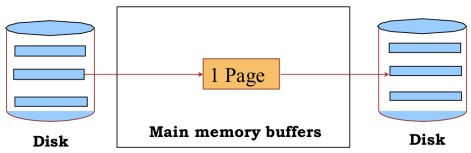
Example: Sorting

If we want to sort 1 TB of data with 1 GB of RAM we cannot rely on normal in-memory sorting implementation using for example QuickSort.

Idea: 2-Way External Merge Sort

2-Way External Merge Sort: Phase 1

- Based on merge sort (now with two phases)
- Read one page at a time from disk
- Sort it in memory (e.g. QuickSort)
- Write it to disk as one temporary file (called "run")
  - Given an input with N pages, Phase 1 produces N runs
- Only one buffer page used



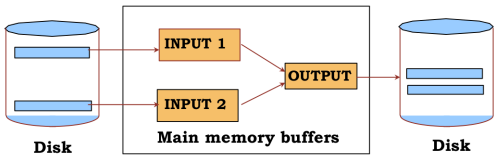
Phase 1: Example

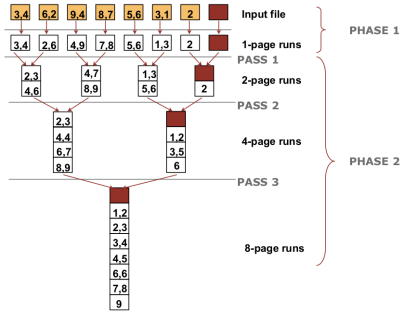
- Assuming the input file has N data pages of size M
- The cost of Phase 1 is:
  - in terms of number or I/Os: 2N
  - in terms of computational steps: $O(NM\log(M))$

2-Way External Merge Sort: Phase 2

Make multiple passes to merge runs

- Pass 1: Merge two runs of length 1 (page)
- Pass 2: Merge two runs of length 2 (pages)
- ... until 1 run of length N
- Three buffer pages used





2-Way External Merge Sort: Example

2-Way External Merge Sort: Analysis

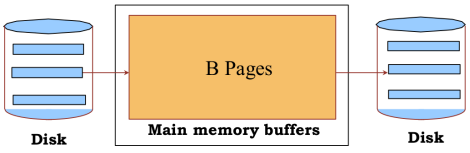
- Total I/O cost for sorting file with N pages
- Cost of Phase 1 = 2 N
- Number of passes in Phase 2 = $\lceil \log_2 N \rceil$
- Cost of each pass in Phase 2 = 2 N
- Cost of Phase 2 = $2N \cdot \lceil \log_2 N \rceil$
- Total cost = $2N(\lceil \log_2 N \rceil + 1)$

Can we do better?

- The cost depends on the #passes
- #passes depends on
  - fan-in during the merge phase
  - the number of runs produced by phase 1

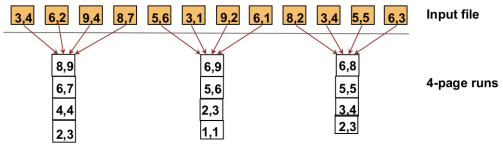
Multi-Way External Merge Sort

- Phase 1: Read B pages at a time, sort B pages in main memory, and write out B pages
- **Length of each run = B pages**
- Assuming N input pages, number of runs = N/B
- Cost of Phase 1 = 2N



2-Way External Merge Sort: Example

Number of buffer pages B = 4



Multi-Way External Merge Sort Phase 2

- Phase 2: Make multiple passes to merge runs
  - Pass 1: Produce runs of length $B(B-1)$ pages
  - Pass 2: Produce runs of length $B(B-1)^2$ pages
  - ⋮
  - Pass P: Produce runs of Length $B(B-1)^P$ pages

Multi-Way External Merge Sort: Analysis

Total I/O cost for sorting file with N pages

- Cost of Phase 1 = $2N$
- If number of passes in Phase 2 is P then: $B(B-1)^P = N$
- P = $\lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost of each pass = $2N$
- Cost of Phase 2 = $2N \cdot \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Total cost = $2N \cdot (\lceil \log_{B-1} \lceil N/B \rceil \rceil + 1)$
- compared to $2N(\lceil \log_2 N \rceil + 1)$

Can we produce runs larger than the RAM size?

Tournament sort (a.k.a "heapsort", "replacement selection sort")

- use 1 buffer page as the input buffer and 1 buffer page as the output buffer
- Maintain 2 Heap structures in the remaining B-2 pages (H1, H2)
- Read B-2 pages of records and insert them into H1
- While there are still records left
  - get the "min" record m from H1 and send it to the output buffer
  - If H1 is empty then start a new run and put all the records in H2 to H1
  - read another record r from the input buffer
  - if r < m then put it in H2, otherwise put it in H1
- Finish the current run by outputting all the records in H1
- If there is something left in H2, output them as a new run

More on Heapsort

- Fact: average length of a run in heapsort is 2(B-2)
  - B-2 pages are used for the heaps
  - (B-2) + 1/2 (B-2) + 1/4 (B-2) + 1/8 (B-2) + . . . ≈ 2(B-2)
- Worst-Case:
  - min length of a run is
  - this arises if all elements are smaller than elements in H1
- Best-Case:
  - max length of a run is entire size of input
  - this arises if all elements are already sorted (close to sorted)
- QuickSort is faster, but longer and fewer runs often mean fewer passes!

External Merge Sort: Possible Optimizations

- In Phase 2 read/write blocks of pages instead of single page
- Double buffering: to reduce I/O wait time, *prefetch* into "shadow block"

Key Points (External Sorting)

- When data is much too big to fit in memory our "normal" best algorithms might not be the best
- External sorting
  - sorting with two (or more) disks
  - use merge sort (2-phase)
- Optimizations
  - Utilize memory to the fullest
  - use heap/replacement sort to reduce #runs
  - Read sequences of pages from disk
  - keep disks "busy"

System Design Principles

Design principles applicable to many areas

- **Adopt sweeping simplifications**
  So you can see what you are doing.
- **Avoid excessive generality**
  If it is good for everything it is good for nothing
- **Avoid rarely used components**
  Deterioration and corruption accumulate unnoticed - until next use.
- **Be explicit**
  Get all of the assumptions out on the table
- **Decouple modules with indirection**
  Indirection supports replaceability
- **End-to-end argument**
  The application knows best
- **Escalating complexity principle**
  Adding a feature increases complexity out of proportion
- **Incommensurate scaling rule**
  Changing a parameter by a factor of ten requires a new design
- **Keep digging principle**
  Complex systems fail for complex reasons
- **Law of diminishing returns**
  The more one improves some measure of goodness, the more effort the next improvement will require
- **Open design principle**
  Let anyone comment on the design; you need all the help you can get
- **Principle of least astonishment**
  People are part of the system. Choose interfaces that match the user's experience, expectations, and mental models
- **Robustness principle**
  Be tolerant of inputs, strict on outputs
- **Safety margin principle**
  Keep track of the distance to the edge of the cliff or you may fall over the edge
- **Unyielding foundations rule**
  It is easier to change a module than to change the modularity

Common Faults, Misc

- **safety vs. security:** when a question asks about safety of a system it means it robustness, how well it can tolerate faults etc. and not about security with adversary etc.
- when drawing schedule (no overlapping operations!)
- over engineering in design question / misunderstanding the question