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

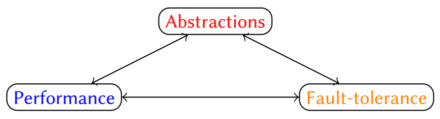
Fundamentals

- 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

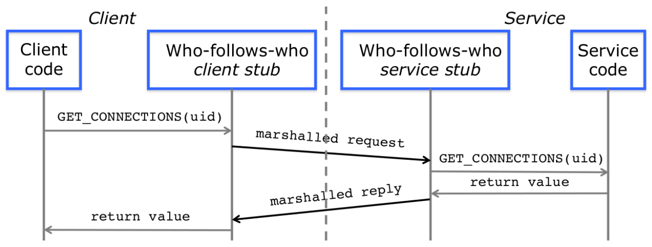
- Restrict communication to *message only*
- Client request / Service response (or reply)
- Conceptually client and service in different computers

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

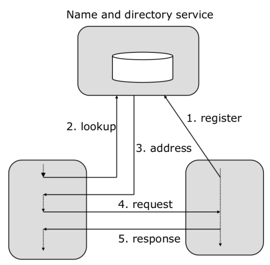
- **At-least-once**
  - Operation is *idempotent* (naturally occurs if side-effect free)
  - Stub just retries operation → failures can still occur!
  - Example: calculate SQRT
- **At-most-once**
  - Operation does have side-effects
  - Stub must ensure duplicate-free transmission
  - Example: transfer $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

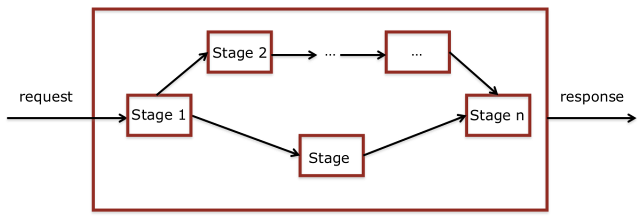
⇒ **How can we choose between I_1 and I_2?**

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

Concurrency Control

Transaction

- Reliable unit of work against memory abstraction

ACID Properties

- **Atomicity:** transactions are all-or-nothing
- **Consistency:** transaction takes database from one consistent state to another
- **Isolation:** Executes as if it were the only one in the systems (aka before-or-after atomicity)
- **Durability:** once transaction is done ("committed"), results are persistent in the database

The many faces of atomicity

- **Atomicity** is strong modularity mechanism!
  - Hides that one high-level actions is actually made of many sub-actions
- **Before-or-after** atomicity
  - == Isolation
  - Cannot have effects that would only arise by interleaving of parts of transactions
- **All-or-nothing** atomicity
  - == Atomicity (+ Durability)
  - cannot have partially executed transactions
  - Once executed and confirmed, transaction effects are visible and not forgotten

Goal of Concurrency Control

- Transactions should be executed so that it is *as though* they executed in some serial order
  - Also called **Isolation** or **Serializability** or **Before-or-after atomicity**
- Weaker variants also possible
  - Lower "degrees of isolation"

Example

Consider two transactions (Xacts):

```
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```

- T1 transfers $100 from B's account to A's account
- T2 credits both accounts with 6% interest
- If submitted concurrently, net effect should be equivalent to Xacts running in some serial order
  - No guarantee that T1 "logically" occurs before T2 (or vice-versa) - but one of them is true

Different Solutions (Locking Protocols)

Solution 1

1. Get exclusive lock on entire database
2. Execute transaction
3. Release exclusive lock

- Transactions execute in *critical section*
- Serializability guaranteed because execution is serial!

Problems:

- no concurrency, only serial schedules

Solution 2

1. Get exclusive locks on *accessed* data items
2. execute transaction
3. release exclusive locks

- Greater concurrency

Problems:

- need to know objects a priori, least concurrency

Solution 3

1. Get exclusive locks on data items that are *modified*; get shared locks on data items that are only *read*
2. Execute transaction
3. Release all locks

- Greater concurrency
- Conservative Strict Two Phase Locking (CS2PL)

Problems:

- need to know objects a priori, least concurrency

Solution 4

1. Get exclusive locks on data items that are modified and get shared locks on data items that are read
2. Execute transaction and release locks on objects no longer needed *during execution*

- Greater concurrency
- Conservative Two Phase Locking (C2PL)

Problems:

- Cascading Aborts: assume T1 has locks on 1, 2, 3 and then starts to release locks 1, 2, which are immediately acquired by T2, but then T1 does not commit but aborts. now T2 also has to be aborted etc.
- need to know objects a priori, when to release locks

Solution 5

1. Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this *during execution* of transaction (as needed)
2. Release all locks

- Greater concurrency
- Strict Two Phase Locking (S2PL)

Problems:

- Deadlocks: assume T1 wants 1, 2, 3 and T2 wants 2, 3, 4 and that T1 acquires 1, 2 and T2 3, 4 now we have a deadlock.

Solution 6

1. Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this during execution of transaction (as needed)
2. Release locks on objects no longer needed during execution of transaction
3. Cannot acquire locks once any lock has been released (Hence two-phase (acquiring phase and releasing phase)

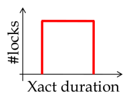
- Greater concurrency
- Two Phase Locking (2PL)

Problems:

- Cascading aborts and Deadlocks

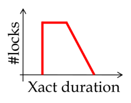
Summary of Alternatives

- Conservative Strict 2PL
  - no deadlocks, no cascading aborts
  - **But** need to know objects a priori, least concurrency



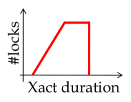
	S	X
S	Yes	No
X	No	No

- Conservative 2PL
  - no deadlocks, more concurrency than Conservative Strict 2PL
  - **But** need to know objects a priori, when to release locks, need to deal with cascading aborts



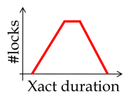
	S	X
S	Yes	No
X	No	No

- Strict 2PL
  - no cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - **But** deadlocks



	S	X
S	Yes	No
X	No	No

- 2PL
  - most concurrency, no need to know objects a priori
  - **But** need to know when to release locks, cascading aborts, deadlocks



Method of choice

- Strict 2PL
  - no cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - But deadlocks
- Reason for choice
  - Cannot know objects a priori, so no Conservative options → only if you would know something about application!
  - Thus only 2PL and Strict 2PL left
  - 2PL needs to know when to release locks (main problem), and has cascading aborts
  - Hence Strict 2PL
- Implication: Need to deal with deadlocks!

Lock Management

- Lock/unlock requests handled by lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- **Lock upgrade:** transaction that holds a shared lock can be upgraded to hold an exclusive lock

Dynamic Databases: Locking the objects that exist now in the database is not enough!

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not work correctly:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds **oldest** sailor (say, age = 71)
  - Next, T2 inserts a new sailor; rating = 1, age = 96
  - T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits
  - T1 now locks all pages containing sailor records with rating = 2, and finds **oldest** (say, age = 63)
- No consistent DB state where T1 is "correct"!

The Problem

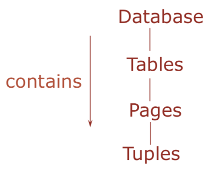
- T1 implicitly assumes that it has locked the set of all sailor records with rating = 1
  - assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. **(Index locking and predicate locking)**
- Example shows that correctness is guaranteed for locking on individual objects only if the set of objects is fixed!

Index Locking

- If data is accessed by an **index** on the rating field, T1 should **lock the index page** containing the data entries with rating = 1
  - if there are no records with rating = 1, T1 must lock the index page where such a data entry would be, if it existed!
- if there is **no suitable index**, T1 must **lock all pages**, and lock the file/table to prevent new pages from being added, to ensure that no new records with rating = 1 are added

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuple vs. pages vs. tables)
- Shouldn't have to decide!
- Data "containers" are nested



Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new **"intention" locks**
- before locking and item, Xact must set "intention locks" on all its ancestors
- for unlock, go from specific to general (i.e., bottom-up)
- **SIX mode:** like S & IX at the same time

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Schedules

- Consider a possible interleaving (schedule):

```
T1:   A=A+100,                        B=B-100
T2:               A=1.06*A, B=1.06*B
```

- The systems's view of the schedule

```
T1:   R(A),W(A),                       R(B),W(B)
T2:              R(A),W(A),R(B),W(B)
```

Scheduling Transactions

- **Serial schedule:** Schedule that does not interleave the actions of different transactions
- **Equivalent schedules:** For any database state
  - The effect (on the set of objects in the database) of executing the schedules is the same
  - the values read by transactions is the same in the schedules
    - Assume no knowledge of transaction logic
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions.

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads")

```
T1:  R(A), W(A),                R(B), W(B), Abort
T2:              R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts)

```
T1:  R(A),                       R(A), W(A), C
T2:            R(A), W(A), C
```

- Overwriting Uncommitted Data (WW Conflicts)

```
T1:  W(A),                W(B), C
T2:         W(A), W(B), C
```

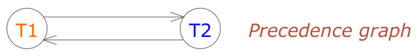
Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
  - involve the same actions of the same transactions
  - every pair of conflicting actions is ordered the same way
  - two actions are called **conflicting** if they access the same object and one of them is a write
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Example: Schedule not conflict serializable

- the cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

```
T1:     R(A), W(A),                              R(B), W(B)
T2:                 R(A), W(A), R(B), W(B)
```



Precedence Graph

- **Precedence graph:** One node per Xact; edge from T_i to T_j if operation in T_j conflicts with earlier operation in T_i
- **Theorem:** Schedule is conflict serializable if and only if its precedence graph is acyclic
- Strict 2PL only results in conflict serializable schedules
  - Precedence graph is always acyclic

View serializability

- Schedules S1 and S2 are **view equivalent** if:
  - T_i reads initial value of A in S1, then T_i also reads initial value of A in S2
  - if T_i reads a value of A written by T_j in S1, then T_i also reads value of A written by T_j in S2
  - If T_i writes final value of A in S1, then T_i also writes final value of A in S2

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other

- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps
- Lower timestamps get higher priority, i.e., older transactions get prioritized
- Assume T_i wants a lock that T_j holds. Two policies are possible:
  - Wait-Die: If T_i has higher priority, T_i waits for T_j; otherwise T_i aborts
  - Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Deadlock Detection

- Create a **waits-for graph**:
  - Nodes are transactions
  - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph

The Problems with Locking

- Locking is a pessimistic approach in which conflicts are prevented
- Disadvantages:
  - Lock management overhead
  - Deadlock detection/resolution necessary
  - Lock contention for heavily used objects
- We must devise a way to **enforce serializability**, without destroying concurrency
- Two approaches:
  - prevent violations → locking
  - fix violations → aborts

Optimistic CC: Kung-Robinson Model

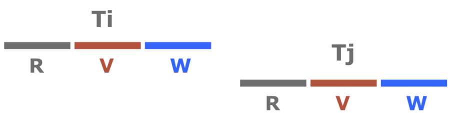
- optimistic approach based on aborts
- Xacts have three phases
- **READ:** Xacts read from the database, but make changes to private copies of the objects
- **VALIDATE:** check for conflicts
- **WRITE:** make local copies of changes public

Validation

- Test conditions that are **sufficient** to ensure that no conflict occurred
- Each Xact is assigned a numeric id
  - simply use a **timestamp**
- Xact ids assigned at end of READ phase, just before validation begins
- ReadSet(T_i): Set of objects read by Xact T_i
- WriteSet(T_i): Set of objects modified by T_i

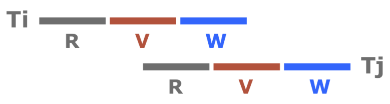
Test 1

- For all i and j such that $T_i < T_j$, check that T_i completes before T_j begins



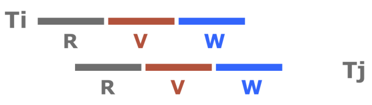
Test 2

- For all i and j such that $T_i < T_j$, check that:
  - T_i completes before T_j begins its Write phase
  - WriteSet(T_i) ∩ ReadSet(T_j) is empty



Test 3

- For all i and j such that $T_i < T_j$, check that:
  - T_i completes Read phase before T_j does
  - WriteSet(T_j) ∩ ReadSet(T_J) is empty
  - WriteSet(T_i) ∩ WriteSet(T_j) is empty



Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact
  - must create and destroy these sets as needed
- Must check for conflicts during validation, and must make validated writes "global"
  - Critical section can reduce concurrency
  - Scheme for making writes global can reduce clustering of objects
- Optimistic CC restarts Xacts that fail validation
  - work done so far is wasted; requires clean-up
- Still, optimistic techniques widely used in software transactional memory (STM), main-memory databases

Snapshot Isolation

- Often databases implement properties that are **weaker** than serializability
- **Snapshot isolation**
  - **Snapshots:** Transactions see snapshot as of beginning of their execution
  - **First Committer Wins:** Conflicting writes to same item lead to aborts
- May lead to **write skew**
  - Database must have at least one doctor on call
  - Two doctors on call concurrently examine snapshot and see exactly each other on call
  - Doctors update their own records to being on leave
    - No writ-write conflicts: different records!
  - after commits, database has no doctors on call

Experimental Design

Techniques to Evaluate Performance

- Three main techniques
  - Analytical Modeling
  - Simulation
  - Experimentation

Analytical Modeling

- Get intuition about system performance
  - Without actually implementing it!
- Simple model for virtual memory system with paging:

$$\text{AverageLatency} = \text{HitRatio} * \text{Latency}_{\text{Hit}} + (1 - \text{HitRatio}) * \text{Latency}_{\text{Miss}}$$

- With high hit ratio (say, >95%), average time can be pretty close to main memory
  - Some requests still require going to disk, of course, and take full disk latency blow
- How can we know the hit ratio?

Simulation

- Study properties of hard-to-model process, e.g., locality of workloads vs. hit ratio in cache
- Configure model with known parameters
  - In our example, $\text{Latency}_{\text{Hit}}$ and $\text{Latency}_{\text{Miss}}$
- Simulate behavior of system to get HitRatio

Simulation

- Pros
  - Effort may be smaller than full-blown implementation
  - Allows you to simulate "impossible" or hard-to-experiment-with scenarios → 10Ks of machines, next-generation flash disk not on the market yet

- Cons
  - Estimating parameters
  - Validating models and approximations
  - Choosing workloads

Choosing Workloads & Datasets

- Synthetic workloads & datasets
  - Example: use Zipf distribution to generate workload of page accesses (Zipf distribution: most frequent word occurs twice as often as 2. most frequent and 3 times as often as 3. most frequent etc.)
- Real workloads & datasets
  - Example: Take **trace** of page requests from real application
  - replay trace on your simulator
- Combinations also possible
  - Use real dataset but generate accesses using a distribution
- Issue: How can you tell if workload is representative?

Experimentation

- **Implement** real system (or prototype)
- **Measure** how it behaves with experiments
  - most respected method
  - but also requires most effort
- **Profile** system to determine where time goes

Simple Factor Experimentation

- Understanding multiple influences
- Vary one factor at a time, keep others fixed
- Example: Skew of workload and size of cache
  - Skew = 0.5, vary cache size from 1MB to 1GB
  - Cache size = 500MB, vary skew from 0 to 1
- Care required: Parameters may **influence** each other!

Benchmarking

- **Micro-benchmarks**
  - Measure a specific variable or piece of code, e.g., memory and disk latencies in small experiment to calibrate simulation model
- **Application-level benchmark**
  - whole application designed to stress certain types of systems
  - **SPEC** benchmarks for compute-intensive apps, web servers, file systems, and many others
  - **TPC** benchmarks for databases

Necessary Care with Executing Experiments

- Select **event counts**
  - Number of pages/chunks read
  - Number of clock cycles elapsed $\longrightarrow$ wall-clock time
- But control for **overhead** of event counting itself!
- **Sampling / monitoring**
  - e.g., I/O via iostat/vmstat
- "**Statistics** can prove anything?!"
  - Number of measurements
  - Mean and variance
  - confidence intervals
  - dealing with outliers
  - Setup matters!

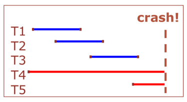
Comparing Alternatives

- Two systems, with throughput-oriented measurements R_2 and R_1
  - Both systems travel same distance D, i.e., do same work but take different time
  - $R_2 = D/T_2$; $R_1 = D/T_1$
- Speedup
  - $S_{2,1} = R_2/R_1 = T_1/T_2$
- Relative change
  - $\Delta_{2,1} = (R_2 - R_1)/R_1 = S_{2,1} - 1$
- Example statements
  - System 2 is 1.4 times faster than System 1
  - System 2 is 40% faster than System 1

Implementing All-or-Nothing Atomicity

- Atomicity
  - Transactions may abort ("Rollback")
- Durability
  - What if system stops running? (Causes?)

- Desired behavior after system restart
  - T1, T2, T3 should be durable
  - T4, T5 should be aborted (effects not seen)



Assumptions

- Concurrency control is in effect
  - Strict 2PL, in particular
- Updates are happening "in place"
  - i.e. data is overwritten on (deleted from) memory using READ / WRITE interface
  - we will use two-level memory with buffer and disk
- Types of failures
  - Crash
  - Media failure
- Always fail-stop! (components notify that they have crashed)

Volatile vs. Nonvolatile vs.Stable Storage

- Volatile Storage
  - Lost in the event of a crash
  - Example: main memory
- Nonvolatile Storage
  - Not lost on crash, but lost on media failure
- Stable Storage
  - Never lost
  - How do you implement this one? (Replication only lower chance of failure)

Surviving Crashes: How to handle the Buffer Pool?

- **Force** every write to disk?
  - Poor response time
  - But provides durability
- **Steal** buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial (?)	
No Force		Desired

More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - to steal a frame F: current page in F (say P) is written to disk; some Xact holds a lock on P
    - What if the Xact with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P)
- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications

	No Steal	Steal
Force	No Redo No Undo	No Redo Undo
No Force	Redo No Undo	Redo Undo

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a **log**
  - Sequential writes to log (put it on a separate disk)
  - Minimal info (diff) written to log, so multiple updates fit in a single log page
- **Log:** an ordered list of REDO/UNDO actions
  - logical vs. physical logging
  - example physical log record contains:
    <XID, pageID, offset, length, old data, new data>
  - good compromise is physiological logging

Write-Ahead Logging (WAL)

- Golden Rule: Never modify the only copy!
- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update **before** the corresponding data page gets to disk
  2. Must write all log records for a Xact **before commit**
- #1 guarantees Atomicity
- #2 guarantees Durability

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?

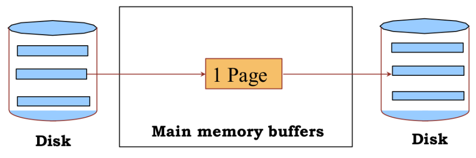
- Important in data processing (relational queries)
- Used for eliminating duplicates (SELECT DISTINCT)
- Bulk loading B+ trees (need to first sort leaf level pages)
- Data requested in sorted order
- Some join algorithms use sorting
- Some MapReduce implementations use sorting to group keys for reducers

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.

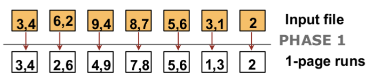
Idea: 2-Way External Merge Sort

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

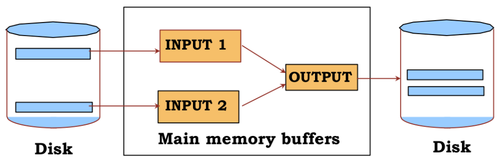


- Assuming the input file has N data pages of size M
- The cost of Phase 1 is:
  - in terms of number or I/Os: 2N
  - in terms of computational steps: $O(NM\log(M))$

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

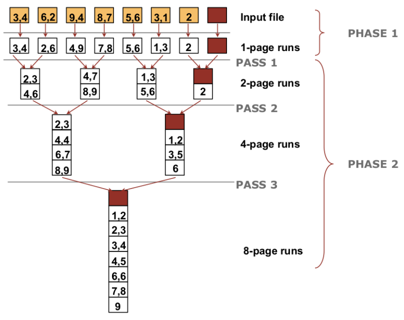
Multi-Way External Merge Sort Phase 2

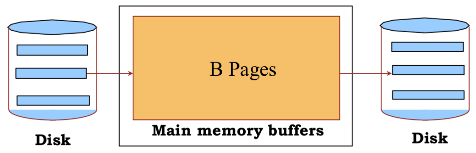
- Phase 2: Make multiple passes to merge runs
  - Pass 1: Produce runs of length $B(B-1)$ pages
  - Pass 2: Produce runs of length $B(B-1)^2$ pages

    ⋮
  - Pass P: Produce runs of Length $B(B-1)^P$ pages

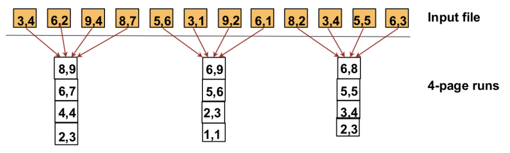
Multi-Way External Merge Sort: Analysis

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