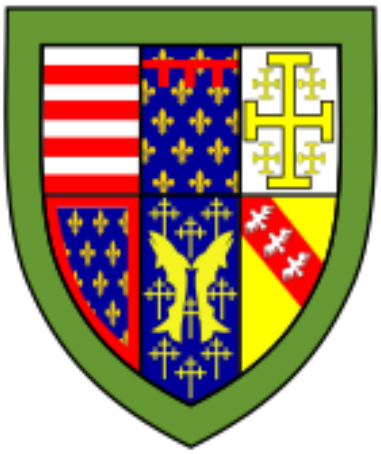
Queens' College Cambridge

Concurrent and Distributed Systems



Alistair O'Brien

Department of Computer Science

April 29, 2021

Contents

1 Concurrent Systems

- *Concurrency* is the interleaving of tasks in time to give the illusion of parallelism.
- Types of concurrency:
  - **Inter-process** concurrency:
    - Parallelism simulated by context switching processes
    - Independent memories, communication via IPC.
    - Added security due to process isolation.
  - **Intra-process** concurrency:
    - Parallelism simulated by context switching threads
    - Shared memory
    - Less overhead.
- *Parallelism* when two or more operations are executed simulatenously. Occurs with multi-processors (*concurrent hardware*).

Definition 1.0.1. (**Concurrent System**) A concurrent system is multi-threaded / process system with a shared address space (although processes have their own insolated address spaces).

1.1 Threading Models

- A process is defined as a program in execution. Each process has an address space to provide isolation. Each process has one or more threads, which have a `pc` and register file, etc.
- A thread is an independent sequential execution path through a program. As a thread executes, it changes **state** IMAGE Hence a thread's state is *non-deterministic*. A thread is a *unit of scheduling and execution*.

Definition 1.1.1. (**Kernel Processes and Threads**) At the kernel level, **kernel processes** contain one or more **kernel threads**, which share the process's resources (memory, file descriptors, etc). Kernel processes are said to be "heavyweight" units of kernel scheduling, since context switching is expensive (TLB flush, etc). Kernel threads are said to be "lightweight" units of kernel scheduling.

Definition 1.1.2. (**User Threads**) At the user level, a process can schedule multiple threads of execution, called **user threads**. They are "lower cost" than kernel threads.

- User threads are sometimes more useful than kernel threads e.g. web-server, 1 user thread per request.

Definition 1.1.3. (**Threading Model**) A threading model is the relation between kernel processes and threads and user threads.

- Relationship denotes `kernel : user`:
  - $1 : n$ User-level threading: 1 kernel process maps to n user threads.

    Advantages:
    * Lightweight creation, termination and context switch.
    * Application specific scheduling

    Disadvantages:
    * Difficult to handle blocking system calls / page faults
  - $1 : 1$ Kernel-level threading: 1 kernel thread maps to 1 user thread (e.g. Pthreads).

    Advantages:
    * Handles preemption, blocking system calls
    * Multi-processor safe

    Disadvantages:
    * Overhead due to OS
  - $m : n$ Hybrid threading: m kernel threads map to n user threads (e.g. Go).

1.2 Synchronization

- Progress of current thread depends on behavior of other threads. Types of thread interaction:
  - **Competition**: Two threads compete acquire some resources / execute instructions and only one thread at a time is allowed to use / execute them. (Mutual exclusion)
  - **Cooperation**: One thread can only progress after some event from another thread. (Synchronization)

1.2.1 The Mutual Exclusion Problem

Definition 1.2.1. (**Critical Section**) A part (or several parts) of a program that are required to be executed by a single thread at a time.

- Not a problem if the critical sections are **atomic** (indivisible).
- **Solution**: provide `enter` and `leave` procedures that are used to respectively enter and exit the critical section, ensuring that the critical section is only executed by a single thread at a time.
- Preventing simultaneous execution of critical sections is called **mutual exclusion**.
- Programming with mutual exclusion:
  - Make composite operations (that lead to inconsistent intermediate state) **atomic** using `enter` and `leave`.
  - Invariants are held before critical section has been entered. Invariants may be violated while in the critical section. Invariants are restored once thread leaves the critical section.

Lamport Bakery Algorithm

1.2.2 Race Conditions

- A **race condition** is an event where the outcome dependents on the relative ordering of execution of two or more threads.

- It occurs when there is missing: synchronization or mutual exclusion
- Very difficult to detect and leads to many errors.

1.2.3 Finite State Machines

-

1.3 Locks

- Locks are constructed for synchronization, mutual exclusion or both.
- There are two types of locks: spinning locks and blocking locks: IMAGE
- Spinning locks (busy-wait) wait until an event occurs: the lock is released, or the thread is preempted.
- Blocking locks block until an event occurs (using threading procedures e.g. `thread_block`).

1.3.1 Spin Locks

- A **spin lock** is implemented using a while loop checking for an event to occur:

```
void acquire(lock_t* l) {
    while (read_and_set(&l->value, 0) == 0); // spin
}
void release(lock_t* l) {
    l->value = 1;
}
```

- Acquiring a spin lock requires an **atomic** *read-and-set* operation, which sets `l->value` to `0` and returns it's previous value. (`0` = acquired, `1` = released).
- **Problem**: Processor is waiting until thread is preempted or lock is released (wasting time).

- **Solution**: Yield the thread to the scheduler (using `thread_yield`). (For a m multiprocessor environment, yield after m checks)

```
void acquire(lock_t* l) {
    while (read_and_set(&l->value, 0) == 0) thread_yield();
}
void release(lock_t* l) {
    l->value = 1;
}
```

- Could potentially leave to indefinite blocking (starvation)

1.3.2 Blocking Locks

- For a blocking lock, the acquiring thread performs a single check for an open lock and then blocks. The releasing thread has the responsibility of unblocking blocked acquirers (cooperation).
- So all blocking locks have:
  - A list of blocked acquirers.
  - Internal state to facilitate lock semantics.

1.3.3 Semaphores

- A semaphore `s` is a positive integer `value` (and a list of blocked acquirers) with two **atomic** operations:
  - `semaphore_down(&s)`: tests if `value > 0`, if so decrement it. If `value = 0`, block until another thread `up`s the semaphore.
  - `semaphore_up(&s)`: increments the value `value` of the semaphore, and unblocks a blocked acquirer (if any).

```
void semaphore_down(semaphore_t* s) {
  assert(s != null);
  intr_state_t old_state = intr_disable();
  while (s->value == 0) {
    list_push_tail(&s->waiters, thread_current());
    thread_block();
```

```
  }
  s->value--;
  intr_set_state(old_state);
}

void semaphore_up(semaphore_t* s) {
  assert(s != null);
  intr_state_t old_state = intr_disable();
  if (list_size(&s->waiters) != 0) {
    thread_unblock(list_pop_head(&s->waiters));
  }
  s->value++;
  intr_set_state(old_state);
}
```

- Mutual exclusion usage:

```
// Mutual Exclusion
void main(void) {
    semaphore_down(&s);
    // critical section
    semaphore_up(&s);
}
```

- In general: n threads of execution in critical section $\implies$ initial value to n.

- Synchronization usage:

```
// Synchronization
semaphore_init(&s, 0); // note initial 0 value

// Thread t1 // Thread t2
void main(void) { void main(void) {
    // action t1 // action t2
    semaphore_down(&s); semaphore_up(&s);
    // t2 complete }
    ...
}
```

- For multi-processor environment, we require a spin lock (instead of disabling interrupts) and **inter-processor interrupts** (IPIs) to wakeup a thread.

Dining Philosophers

1.3.4 Mutex Lock

- Solely used for mutual exclusion.
- Two kinds of mutex locks:
  - Recursive: lock owner can acquire the lock multiple times
  - Non-recursive: once acquired, the lock cannot be acquired again (until released)
- Mutual exclusion usage:

```
void main(void) {
    lock_acquire(&lock);
    // critical section
    lock_release(&lock);
}
```

- Implementation (using a binary semaphore):

```
void lock_init(lock_t* l) {
  assert(l != null);

  l->holder = null;
  semaphore_init(&l->semaphore, 1);
}

void lock_acquire(lock_t* l) {
  assert(l != null && !lock_held_by_current_thread(l))

  semaphore_down(&l->semaphore);
  l->holder = thread_current();
}
```

```
void lock_release(lock_t* l) {
  assert(l != null && lock_held_by_current_thread(l));

  l->holder = null;
  semaphore_up(&l->semaphore);
}
```

Producer-Consumer

1.3.5 Monitors and Condition Variables

- **Problem**: We wish to allow threads to have mutual exclusion and the ability to wait for a certain condition to be asserted.
- The critical sections with such a requirement are **conditional critical regions**:

```
shared int A, B, C;
region A, B {
    await(condition);
    // critical section with A and B and condition asserted.
}
```

- **Solution**: A **condition variable**; a queue of threads Q associated with a condition c, that allows threads to block until they're signaled by some other thread that the condition c has been asserted.

  Condition variables are always used in conjunction with a lock ℓ. They have three operations:

  - `wait(`c`, `ℓ`)`: Called by thread t that waits until c is asserted. Atomically: releases ℓ, moves t into waiting queue Q and blocks the thread. Once the thread is signaled (see below) and is resumed, the lock ℓ is acquired.
  - `signal(`c`)` called by a thread to indicate that c is asserted. Dequeues a thread t from Q and unblocks t.
  - `broadcast(`c`)` called by a thread to indicate c is asserted. But unblocks all threads stored in Q.

- **Monitors**: a construct that consists of a lock ℓ and a collection of condition variables $cv_1, \ldots, cv_n$. A single thread can be executing "within" a monitor.
- Condition variable types:
  - *Blocking Condition Variables*: The signaling thread t_s must block until the signaled thread t_S exists the monitor. (*Hoare-style*).

    Advantages: Condition is asserted in t_S.

    Disadvantages: Complex to implement (signal may or may not context switch). Also must ensure invariants hold before signaling.
  - *Non-blocking Condition Variables*: The signaling thread doesn't block and the signaled thread is simply unblocked (and rescheduled at the scheduler's whim). (*Mesa-style*).

    Advantages: Simpler to implement (and reason about).

    Disadvantages: Condition c may not hold when signaled thread is scheduled:

    `while (!`c`) wait(`cv`, `ℓ`)`

1.4 Deadlock

- Concurrent systems must ensure:
  - **Safety** (mutual exclusion in critical sections)
  - **Progress**
- No progress: occurs when threads are waiting on each other, or if the threads make no progress.
- Also a desire for:
  - **No starvation** (single thread must make progress)
  - **Fairness**
  - **Minimality** (no unnecessary waiting / signaling)

1.4.1 Deadlock

- State where a group of threads are waiting for another group of threads (including itself) to take action.

Definition 1.4.1. (**Deadlock**) A set of threads $t_1, \ldots, t_n$ are deadlocked if the following conditions hold:

- **Mutual Exclusion**: Only a single thread t_i can use a resource R_j at a time.
- **Hold & Wait**: A thread t_i is holding the resources $R_a^i, \ldots, R_b^i$ is waiting to acquire resource R_c^j held by thread t_j.
- **No Preemption**: A resource R^i can only be released by the thread t_i holding it.
- **Circular Wait**: A cycle of thread requests exists: $t_1 \to t_2 \to \ldots \to t_1$

1.4.2 Deadlock Handling

Ignoring Deadlock

- Assume deadlock never occurs. The *Ostrich Algorithm*.
- Safe to use if deadlocks are *formally proven* to never occur.

Prevention

- **Idea**: Eliminate one (or more) conditions required for deadline $\implies$ deadlock can never occur.
- **Mutual Exclusion**: Not required for shared resources (unsafe :( ). Mutual exclusion must only hold for non-sharable resources.
- **Hold & Wait**: Require that when threads request resource, they don't hold any other resources. Require threads to request and allocate all it's resource at once. Results in low resource utilization and difficult to know maximal resource set (in advance).

- **No Preemption**: If thread t_i is holding resources (R_j^i) and requests R_k and cannot acquire R_k, then (R_j^i) are released. Preempted resources added to list of resources that t_i is waiting on. t_i is only scheduled again iff all old and new resources can be acquired.
- **Circular Wait**: Impose a partial ordering of all resources, requiring each thread t_i to acquire (R_j) in order.

Detection

- **Idea**: Dynamic detection of deadlock by monitoring resources.
- **Assumptions**:
  - Each thread t_i declares the maximal set of resources required.
  - A thread t_i will complete, terminate and free all it's resources once granted resources.
  - An allocation is granted, if deadlock **may** not occur.
- Allocation Graphs:
  - With n threads (t_i) with maximal resources $(R)^i_{max}$ and m **unique** resources (R_i).
  - Build a resource allocation graph G. If G contains a cycle $\implies$ deadlock. $O(\underbrace{m+n}_{V}+\underbrace{mn}_{E})$ time complexity.
  - If non-unique, then cycle $\not\implies$ deadlock. Split duplicate resources into unique ones (using isomorphisms), then run cycle detection. (**Difficult**)
- Banker's Algorithm:
  - n threads (t_i) and m distinct resource types (R_i).
  - Data structures:
    - **Available**$[0 : m-1]$, a m length vector indicating the number of available resources of each time.
    - **Max**, a $n \times m$ matrix of the maximal resource set of each thread. **Max**$[i][j] = k \implies t_i$ requires k instances of R_j.

    - ∗ **Allocation**, a $n \times m$ matrix of the number of resources of each type currently allocated to each thread.
    - ∗ **Need**, a $n \times m$ matrix s.t **Need** = **Max** − **Allocation**.
  - *Safety check* algorithm:
    1. Let **Work**$[0 : m-1]$ and **Finish**$[0 : n-1]$ be state vectors. Initialize **Work** = **Available** and **Finish** = $(false)$.
    2. Find index i s.t
       (a) **Finish**$[i] == true$
       (b) **Need**$[i] \leq$ **Work**
       If no such index i, goto 4.
    3. Set **Work** = **Work** + **Allocation**$[i]$ and **Finish**$[i] = true$. Goto 2.
    4. If $\forall i$.**Finish**$[i] == true$, then no deadlock. Otherwise, all threads t_i s.t **Finish**$[i] == false$ are deadlocked.
  - Similar *resource request* algorithm.
  - If we are deadlocked: **kill** a thread in the deadlock set, breaks the **no preemption** condition of deadlock.

1.4.3 Live-lock

Definition 1.4.2. (**Live-lock**) A set of threads $t_1, \ldots, t_n$ are live-locked when state of each thread is changing but no progress is made.

- An example might be: IMAGE The threads are changing state but no progress is being made.
- Difficult to **detect**.

1.4.4 Scheduling Problems

- **Problem**: Priority inversion, where high priority thread is indirectly preempted by low priority thread. Violates **priority** model.
- Scenario: t_1, t_2, t_3 with priority $p_1 > p_2 > p_3$. t_3 acquires lock ℓ. t_1 preempts t_3 and blocks for ℓ. t_2 is now scheduled (due to priorities), preventing t_3 from releasing ℓ, thus blocking t_1.

- **Solution**:
  - **Priority Inheritance**: When high priority thread t_h shares resource R with low priority thread t_ℓ, t_ℓ is assigned priority of t_h (temporarily).
  - **Random Boosting**: Ready threads holding locks are randomly boosted in priority until priority thread releases lock.
- **Problem**: Solutions are heuristic. Difficult to reason about. Only works for **locks**, which have a holder (consider semaphores, etc)

2 Transactions

2.1 Transactions

- **Idea**: Desire to compose atomic sub-operations, so-called **composite operations**.

Definition 2.1.1. (**Atomic**) An operation is said to be **atomic** (on persistent data) if and only if:

- The operation terminates normally, all its effects are made permanent (**durability**), otherwise, it has no effect.
- The operation is *traditionally* **atomic**, that is the operation is indivisible.

Atomic operations are crash-safe (simply restart the operation if a crash occurs).

Definition 2.1.2. (**Transaction**) A transaction is a (possible composite) atomic operation that transforms a system of objects from one consistent state to another.

- 5 states of a transaction:
  - **Active**: The transaction is executing
  - **Partially Committed**: The final operation of the transaction has executed. A transaction is defined to end with a **commit** operation.
  - **Failed**: The transaction is aborting. An exception / error has occurred and the transaction is rollbacking (undoing it's operations). Indicated by an **abort** operation.
  - **Aborted**: The transaction has rollbacked and the system's state is restored to the one prior to the start of execution.

  - **Committed**: Transaction has committed successfully.

  IMAGE

- Scope for concurrent transactions / parallelism of sub-operations.
- Transaction management (TM) systems must be:
  - **Crash-Resilient** to ensure **atomicity** and **durability**, must be able to rollback (undo) the operations of the transaction.
  - Must ensure the transaction schedule S is serializable. (Isolation)

2.1.1 ACID Properties

Definition 2.1.3. (**ACID**) Transactions are said to be ACID compliant if:

- **Atomicity**: All or none of the operations are performed (executed on the *persistent* data).
- **Consistency**: A transaction transforms the system from one consistent state to another. Preserves **invariants**
- **Isolation**: The effects of a transaction are not visible to other transactions until it is committed.
- **Durability**: The effects of a committed transaction persist (even if subsequent crashes occur).
- Atomicity & Durability: Deal with crash resilience:
  - **Atomicity**: Ensures there are no partially complete transactions
  - **Durability**: Ensures no committed transaction effects are lost in event of crashes
- Consistency & Isolation: Deal with concurrency control:
  - **Consistency**: Ensures interleaving of transaction operations maintain system invariants
  - **Isolation**: Ensures interleaving of transaction operations don't produce visible effects to other transactions

2.1.2 Serializability

- Consider transaction schedules S and their effects. A schedule is a sequence of operations (in order) of concurrent transactions.

Definition 2.1.4. (**Serial**) A transaction schedule S is *serial* if the transaction $T_i \in S$ are executed sequentially.

- Define an equivalence relation $\sim$ between transaction schedules. Two schedules S, T are equivalent, denoted $S \sim T$, if their effects are equivalent.

Definition 2.1.5. (**Serializable**) A transaction schedule S is said to be serializable if it is equivalent to a serial permutation of S.

- Consider arbitrary operations on some object O. We denote $O \leftarrow f(O_1, \ldots, O_n)$ to denote the operation f, where f updates O and reads from $O_1, \ldots, O_n$. The effect of f is updating O in some state $\mathcal{S}$.
- There are two termination operations **commit** and **abort** that have no effect on $\mathcal{S}$.

Definition 2.1.6. (**History**) A **history** of a transaction T consisting of operations $f_1, \ldots, f_n$ is a:

- Sequence of operations, defining some total order among the operations (in terms of execution). We denote a history H as $H = \langle f_1, \ldots, f_n \rangle$ where operation f_i precedes f_{i+1}.
- A function that represents a transition of states, that is $H = f_1 \circ \cdots \circ f_n$ where $\circ$ is the composition of operations.

IMAGE

Definition 2.1.7. (**Equivalence**) Two transaction schedules S and T are said to be equivalent, denoted by $S \sim T$, if their histories H_S and H_T satisfy

$$\forall \text{ states } \mathcal{S}. H_S(\mathcal{S}) = H_T(\mathcal{S}).$$

Definition 2.1.8. (**Commutative**) Two operations f and g are said to be commutative, if

$$\forall \text{ states } \mathcal{S}. (f \circ g)(\mathcal{S}) = (g \circ f)(\mathcal{S}).$$

- **Non-commutative** operations are referred to as **conflicting**. e.g. (for read/write semantics):
  - $R(O)$ and $R(O)$ are commutative.
  - $R(O)$ and $W(O)$ are conflicting.
  - $W(O)$ and $W(O)$ are conflicting.
- A conflict enforces some ordering between the operations.
- A schedule S is **conflict**-serializable if it is equivalent to a serial permutation of S where only commutative operations are permuted.

Definition 2.1.9. (**Precedence Graph**) A precedence graph $G = (V, E)$ of a transaction schedule S is a graph whose vertices consist of transactions (T_i) in S where $(T_i, T_j) \in E$ if and only if there are conflicting operations $f_i \in T_i, f_j \in T_j$ s.t f_i precedes f_j in S.

IMAGE

Theorem 2.1.1. A transaction schedule S is (conflict-)serializable if the precedence graph G of S is acylic.

- Topological Sort / Depth-First Search can be used to determine whether a cycle exists in $\Theta(V + E)$ time. Potentially $O(V^2)$.

2.2 Concurrency Control Systems

- **Problem**: Constructing precedence graph is also complex and difficult.
- **Solution**: Provide non-strict isolation.

Definition 2.2.1. (**Strict and Non-Strict Isolation**) **Strict** isolation ensures that transaction schedules never experience no lost updates, ..., whereas **non-strict** isolation allows a transactions schedules to execute, despite potential issues.

- **Effects of Bad Schedules**:
  - **Lost Update**: T_1 updates O, but is later overwritten by T_2. A **write-write** conflict.

  - **Dirty Read**: T_1 reads O, but has been updated by uncommitted transaction T_2. A **write-read** conflict.
  - **Unrepeatable Reads**: T_1 reads O at t_0, T_2 (committed) updates O at t_1 and T_1 re-reads O at t_2, where $t_0 < t_1 < t_2$. A **write-after-read** conflict.

- Occur due to lack of isolation.
- Non-strict isolation increases concurrency but leads to:
  - **Effects of non-serializable** $\implies$ transactions must abort when problems are detected.
  - **Cascading Aborts**: Suppose S is a transaction schedule and $G = (V, E)$ is the precedence graph. If $T_i \to T_j$ and T_i aborts, then T_j must abort.

2.2.1 Rollback Implementations

- When **abort**, we must revert the state $\mathcal{S}'$ to the initial state prior to executing the transaction $\mathcal{S}$.
- Abort reasons: voluntary abort (due to **abort** operation), cascading abort, or deadlock.

Undo Implementation

- Consider schedule S with history H with an initial state $\mathcal{S}$. So we have a temporal ordering of operations $f_1, \ldots, f_n$.
- Assume operations (f_i) have inverse (f_i^{-1}).
- Suppose we abort transaction T_1 at operation f_i with state $\mathcal{S}_i$. Apply $f_i^{-1} \circ f_{i-1}^{-1} \circ \cdots \circ f_1^{-1}$ to $\mathcal{S}_i$ to compute $\mathcal{S}$.
- Redo operations of transactions $T_2, \ldots$.

IMAGE

- **Advantages**:

  - Can be optimized (due to commutativity of inverses)

- **Disadvantages**:
  - Not all operations have an inverse
  - Requires inverse operational semantics e.g. `undo (Credit acc x) = Debit acc x`

Copy Implementation

- Consider schedule S. Take a copy of the initial state $\mathcal{S}$ before executing S.
- On abort of T_1, revert the current state to $\mathcal{S}$.
- Redo operations of transactions $T_2, \ldots$
- **Advantages**:
  - Requires no programmer effort / system constraints on object semantics.
  - Copy implementation is simple and efficient for schedules where many operation would have to be undone.
- **Disadvantages**:
  - High overhead since initial state may be large
  - Copying might not even be required (not all transactions abort...)
- Optimize via **partial copying** (e.g. copy on write).

2.2.2 Two Phase Locking (2PL)

- Ensure serializability via locking (to improve concurrency, we use multi-read single-writer locks).
- Introduce two operations: **lock**($O.write$)/**lock**($O.read$), and **unlock**($O.write$)/**unlock**($O.read$)
- Read locks can also be upgraded to write locks, this is known as **lock conversion**.

Definition 2.2.2. (**Two Phase Locking**) Two phase locking (2PL) is a locking protocol that specifies once a transaction has released a lock it cannot acquire any additional locks.

- Results in two phases:
  - A **growing phase**, transactions acquire locks
  - A **shrinking phase**, transaction releases locks
- Operations may occur in either phase (providing correct locks are held).
- In general, locks are acquired as late as possible and released as early as possible, to maximize concurrency.

Theorem 2.2.1. A schedule S in which every transaction satisfies the 2PL protocol is conflict-serializable.

Proof. Assume that every transaction T_i in S conforms to 2PL. Assume S is not serializable, that is to say there exists a cycle in the precedence graph:

$$T_1 \to T_2 \to \ldots \to T_k \to T_1.$$

There exists an edge $T_i \to T_j$ in the precedence graph iff T_i releases a conflicting lock that T_j acquires. e.g. T_i : **unlock**$(A.read)$ followed by T_j: **lock**$(A.write)$. (See read/write semantics).

Tracing the cycle yields:

- T_1 locks and unlocks
- ...
- T_k locks and unlocks
- T_1 locks and unlocks

Hence T_1 has locked an object after it has released an object, hence it not a 2PL transaction. A contradiction!

□

- **Problems**:

  - Requires a lock manager using a lock table (hash table with key of lock id) where each entry is a list of transactions acquired the lock and a list of transactions waiting for the lock.
  - Risk of deadlock. However, deadlock detector (e.g. Banker's) can (safely) abort a single transaction to break the circular wait condition of deadlock.
  - **2PL** provides non-strict isolation, since operations can occur during shrinking phase:
    * Results in lost-updates, dirty-read and cascading aborts.

- **Strict 2PL**: A two-phase locking protocol where all locks are held by the transaction until it commits.
- Strict 2PL provides strict isolation, however, reduces concurrency and increases contention.

2.2.3 Timestamp Ordering (TSO)

- Each transaction T_i is associated with a unique timestamp t_i, at the *beginning* of the transaction.
- The timestamps determine a serialization ordering, that is $T_i \longrightarrow T_j \iff t_i < t_j$.
- Every object O stores the last transaction to read / write, denoted $R(O), W(O)$ respectively.
- In general: transaction T_i can access $O \iff t_i \geq V(O)$, where $V = \max(R(O), W(O))$.

```
T read(const object<T>& O, int t_i) {
    if (t_i < O.W) throw abort();
    O.R = MAX(t_i, O.R);
    T x = O.read();
    return x;
}
```

```
void write(const object<T>& O,T x, int t_i) {
    if(t_i < O.R) throw abort();
    if (t_i < O.W) return;
    O.W = t_i;
    O.write(x);
}
```

  - `read` and `write`: must be atomic.
  - `read`: prevents read-after-write conflicts.
  - `write`: prevent write-after-reads conflicts.

- **Advantages**:
  - Deadlock free.
  - Increases concurrency (since objects are locked only while operations are performed on them).

- **Problems** :
  - Requires rollback mechanism (see recovery)
  - Doesn't provide *strict isolation* $\implies$ cascading aborts. Strict isolation implementation requires locking on access granted to O (or two-phase commit. See ??).
  - Decides on a *priori* serializable schedule S for (T_i) s.t $t_1 < \ldots, <$ t_n and $S \sim T_1, \ldots, T_n$. Prevents other potential serializable schedules T .
  - Doesn't perform well under contention $\implies$ cascading aborts. May lead to *livelock*, a loop of abortions. (Consider two identical transactions in conflicting schedule.)

2.2.4 Optimistic Concurrency Control (OCC)

- 3 phases:
  - **Read and Execution**: Transaction T reads objects, producing *shadow copies*. The transaction T then executes on the shadow copies of data. (commit or abort).
  - **Validation**: On commit, a "validation test" is performed to determine whether the transaction is serializable.
  - **Write phase**: If T is validated, then transactions effects are applied to the state. This phase must be atomic.

- The transaction T_i is allocated 3 timestamps: $\text{start}(T_i), \text{validation}(T_i)$, denoting T_i entered it's execution, validation phases respectively and

finish(T_i), T_i finished it's write phase. Serialization determined by validation(T_i).

- Transaction T_j is validated iff for all unfinished transactions T_i with validation(T_i) < validation(T_j):
  - **Consistent Snapshot**: The shadow copies w/ timestamps $t_1, \ldots, t_k <$ validation(T_i) $\implies$ consistent snapshot at time t_k.
  - **Serializable**: The set of objects updated by T_i is disjoint to the set of items read by T_j (no-conflicts)

  Theorem 2.2.2. If $\forall T_i$.validation(T_i) < validation(T_j) and either:
  - finish(T_i) < start(T_j)
  - start(T_j) < finish(T_i) < validation(T_j) and the set of objects updated by T_i is disjoint to the set of items read by T_j

  then T_j is validated.

  Proof. Let T_i be arbitrary transaction s.t validation(T_i) < validation(T_j) We have two cases:
  - finish(T_i) < start(T_j): T_i is finished. Therefore consistent snapshot and serializable hold trivially.
  - start(T_j) < finish(T_i) < validation(T_j) and ...: Since start(T_j) < finish(T_i), we have consistent snapshot since $t_1, \ldots, t_k \leq$ start(T_j) < validation(T_i) < finish(T_i). We also have serializability since the set of objects updated by T_i is disjoint to the set of items read by T_j.

  So we are done. □

- **Advantages**:
  - No rollback / abort explicit implementation required.
  - Decides on a serializable schedule S after transaction performed $\implies$ admits many serializable schedules that TSO prevents.
- **Disadvantages**:

  - Requires shadow copies $\implies$ large memory requirement per transaction.
  - Poor performance under contention $\implies$ performs lots of work on stale data.
  - Long transactions operating on "data hot-spots" may be repeatedly aborted, resulting in starvation.

2.3 Recovery

- **Assumption**: Crash-stop model. If the node crashes, all in-memory state is lost.
- **Problems**: Recovery systems deal with:
  - *Node crashes*: In memory state is lost. Certain transactions must be aborted / redone.
  - *Media failure*: A read-write head crash / other disk failure destorys all (or part) of non-volatile storage.
  - *Transaction failure*: Logical errors that prevent the transaction committing / Node errors that terminate the transaction (e.g. deadlock).
- Note that the OS implements block-based non-volatile storage using *buffers* (Q_i), containing blocks (B_j) residing in main memory:
  - `read_disk`(Q_i, B_j) reads block B_j into buffer Q_i.
  - `write_disk`(Q_i, B_j) writes block B_j from Q_i to disk.

  using an atomic single-sector write operation.

  Let B_O be the block containing object O and Q_O be the buffer containing block B_O. The following procedures are used to read and write objects from Q_O to main memory:

  - `read_buffer`(O, Q_O, B_O). Returns the value of object O stored in buffer Q_O, block B_O. `read_disk` procedure must be executed before accessing O.

  - `write_buffer(`O, x, Q_O, B_O`)`. Writes the value x to object O stored in buffer Q_O, block B_O. `write_disk(`Q_O, B_O`)` is executed at the OS's whim.

- Two main methods: **log-based recovery** and **shadow paging**.

2.3.1 Shadow Paging

- Persistent storage is split into blocks. Each block has a corresponding logical unit, a *page*. Pages are mapped to blocks using a *pagetable* p (stored in persistent memory) a mapping from pages (p_i) to blocks (B_j).

- Each transaction T_i has a *shadow pagetable* p_i, copied from p (stored in main memory)

  On `write_buffer(`O, x, Q_O, B_O`)`: Let p^O be the page mapping to B_O. A new block B'_O is allocated, copied from B_O and read into buffer $Q_{O'}$. The operation `write_buffer(`O, x, $Q_{O'}$, $B_{O'}$`)` is performed.

  The shadow pagetable is updated s.t $p_i[p^O] \leftarrow B_{O'}$.

- On **commit**, the relevant buffers (Q_i) are flushed by performing `write_disk`. The shadow pagetable p_i is then written to disk.

- There is a special disk address A_{atp} storing the address A_p of the current pagetable. The pagetables are swapped atomically by swapping the addresses of the current pagetable A_p and the shadow pagetable A_{p_i} in A_{atp}.

- **Advantages**:

  - Simple and fast recovery: load the pagetable with address $*A_{atp}$.
  - No requirement for undo / redo operations.

- **Disadvantages**:

  - Fragmentation of data $\implies$ more blocks must be loaded to perform operations.
  - Periodic freeing of unused blocks.
  - Committing requires writing multiple blocks back to disk, **slow**.

2.3.2 Log-based Recovery

- A write-ahead log $\mathcal{L}$ is stored on persistent storage. The log $\mathcal{L}$ consists of a sequence of:
  - $\langle T_i, \text{BEGIN} \rangle$ record: Recorded when a transaction T_i starts.
  - $\langle T_i, O, x_{old}, x_{new} \rangle$ record: Recorded *before* T_i executes $W(O, x_{new})$
  - $\langle T_i, (\text{COMMIT} \mid \text{ABORT}) \rangle$ record: Recorded when a transaction T_i commits (or aborts).

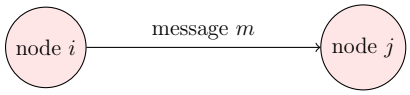
  Updates are split into:
  - Writing to the write-ahead log $\mathcal{L}$
  - Writing updates to persistent storage.
- Redo and undo operations:
  - Traverse the log $\mathcal{L}$ from the end.
  - undo(T_i) operation:
    - On $\langle T_i, O, x_{old}, x_{new} \rangle$, write x_{old} to O and add the record $\langle T_i, \text{UNDO}, O, x_{old} \rangle$ to $\mathcal{L}$.
    - On $\langle T_i, \text{BEGIN} \rangle$, append $\langle T_i, \text{ABORT} \rangle$ to $\mathcal{L}$ and halt.
  - redo(T_i) operation:
    - On $\langle T_i, O, x_{old}, x_{new} \rangle$, write x_{new} to O.
    - On $\langle T_i, \text{UNDO}, O, x_{old} \rangle$, write x_{old} to O.
    - On $\langle T_i, \text{BEGIN} \rangle$, halt.
- On **commit**, the transaction log is flushed (using `write_disk` procedure). A transaction is said to be committed when it's commit log record is written to disk.
- **Problem**: When $\mathcal{L}$ is large, recovery is slow, since the entire log is traversed; and performs unnecessary redo operations.
- **Solution**: Checkpointing:
  - Periodically write a $\langle \text{CHECKPOINT}, \{T_1, \ldots, T_k\} \rangle$ record to $\mathcal{L}$, where (T_i) is the set of *active* transactions. Then flush log record to disk (using `write_disk`).

  - Flush all "dirty" objects.
  - Atomically write the address of the CHECKPOINT record to a special disk address $A_{checkpoint}$.

- **Recovery** algorithm with checkpointing:
  - Find the last checkpoint record $\langle \text{CHECKPOINT}, \{T_1, \ldots, T_k\} \rangle$ using $A_{checkpoint}$.
  - Initialize the *undo set* $U \leftarrow \{T_1, \ldots, T_k\}$ and *redo set* $R \leftarrow \emptyset$.
  - Traverse $\mathcal{L}$ from the checkpoint record:
    * On $\langle T_i, \text{BEGIN} \rangle$: $U \leftarrow U \cup \{T_i\}$
    * On $\langle T_i, (\text{COMMIT} \mid \text{ABORT}) \rangle$: $U \leftarrow U \setminus \{T_i\}$ and $R \leftarrow R \cup \{T_i\}$
  - Perform undo(T_i) for all $T_i \in U$ and redo(T_i) for all $T_i \in R$.

3 Distributed Systems

Definition 3.0.1. (**Distributed Systems**) A distributed system is a collection of *nodes* $(n_i, n_j, \ldots)$ with some shared communication channel C allowing transmission of messages m.



Definition 3.0.2. (**Latency and Bandwidth**) *Latency* L is defined as the time delay between the events send(m) and recieve(m).

Bandwidth is defined as maximum data transfer rate of the medium.

- The communication channel c has an intrinsic latency and bandwidth. e.g. Hard drives in a van has a high intrinsic latency (e.g. 1 day) and high bandwidth (e.g. 1 Gb/s)
- **Problems**:
  - Communications may fail
  - Nodes may crash
  - These behaviors are nondeterministic

3.0.1 Remote Procedure Call

- Nodes are split into *clients* and *services*.
- Remote procedure call (RPC) is a when a procedure is executed in a different address space (on another node). The communication required for this is performed by the *stub*, provided by the RPC framework.
- In an Object Oriented context, RPC is known as *Remote Method Invocation*.

- RPC trace:

  1. The client executes the stub with arguments $x_1, \ldots, x_n$ The stub marshals the arguments $x_1, \ldots, x_n$ into a message m_1.
  2. The service unmarshals the arguments from m_1 and executes the remote procedure, yielding the result x. The result is then marshalled into message m_2, m_2 is sent by the service.
  3. The stub unmarshalled the result from m_2, returning it to the client.

  IMAGE

- The *stub* and remote procedure have identical types. Ideally *location transparency* is implemented by RPC. (Rarely done in practice...)

Definition 3.0.3. (**Location Transparency**) A system that hides where a resource is located implements *location transparency*.

- Common examples of RPC: REST-API (*web services*) with following RESTful principles:
  - Communication is stateless (each request / message is independent)
  - Resources are represented by URLs
  - Object operations CRUD correspond to HTTP method types. e.g. (READ $\cong$ GET)
- Service interfaces implemented by JSON / *Interface Definition Language* (that generate stubs) e.g. gRPC (*Protocol Buffers*).

3.1 Two Generals Problems

3.1.1 The Classical Two Generals Problem

- Consider two armies, led by two generals preparing to attack a city. Generals can communicate using *messengers* that may be captured by the city. The attack is successfully iff both attack at the same time. IMAGE

- Example of consensus with a fair-loss link. See ??
- No deterministic protocol with a finite number of messages.

  Proof. Suppose we have a deterministic protocol. There exists a sequences: $m_1, \ldots, m_k$ and $\overline{m_1}, \ldots, \overline{m_\ell}$ where m_i and $\overline{m_j}$ were and were not delivered respectively.

  Consider m_k send by general G_i to G_j. Suppose m_k wasn't received by G_j, then G_j wouldn't attack. However, from the viewpoint of G_i, G_i cannot determine whether m_k is delivered or not.

  Since the protocol is deterministic G_i must attack anyway. A contradiction! Since G_i is attacking, and G_j isn't. □

- This models network behavior (fair-loss link).

3.1.2 The Byzantine Generals Problem

- Consider n armies, led by n generals preparing to attack a city. Generals can communicate using *messengers* that *cannot* be captured by the city. However, t generals may be *traitors* The attack is successfully iff both attack at the same time. IMAGE
- No protocol with $n < 3t + 1$ generals.
- This models node behavior (Byzantine behavior).

3.2 System Models

- A system model is defined as a node model, a network model and a timing model.

3.2.1 Network Models

- Assume bidirectional **point-to-point** communication link ℓ between n_i and n_j with the following link model:
  - **Reliable** link: A message m is received if and only if it is sent. Messages may be reordered.

  - **Fair-loss** link: A message m may be lost, duplicated or reordered. With the assumption that there exists a finite number n of retries for m to be delivered.
  - **Arbitrary** link: A malicious adversary may interfere with messages. (eavesdrop, modify, drop, spoof, replay)

- Possible to convert an arbitrary link to a reliable link (using TCP + TLS), provided the network is partitioned for some timeout time T (TCP retry-timeout) and adversary doesn't indefinitely block messages.

Definition 3.2.1. (**Network Partition**) A network $N = \{n_1, \ldots, n_m\}$ with links $L = \{(n_i, n_j), \ldots\}$ is partitioned s.t there exists disjoint $N_1, N_2 \subseteq N$ with $N = N_1 \cup N_2$ and

$$\{(n_i, n_j) \in L : n_i \in N_1 \wedge n_j \in N_2\} = \emptyset,$$

for some finite amount of time t.

3.2.2 Node Models

- Nodes execute with one of the following models:
  - **Crash-stop**: A node n is faulty iff it crashes (at any moment). After crashing, the node stops executing (forever). *Note that the node may rejoin under a new identity.* Models unrecoverable hardware faults.
  - **Crash-recovery**: A node n may crash at any moment, losing all in-memory state. May resume execution later.
  - **Byzantine**: A node n is faulty iff it deviates from the algorithm. Faulty nodes may do anything; including crashing / malicious behavior.
- A non-faulty node is said to be *correct*.

3.2.3 Timing Models

- Timing models:

  - **Synchronous**: Message latency L has a known upper bound. Nodes execute algorithms at a known speed.
  - **Partially synchronous**: The system is asynchronous from some finite time t (unknown), but otherwise, synchronous.
  - **Asynchronous**: Messages can be delayed arbitrarily. Nodes can pause execution arbitrarily. No timing guarantees.

- Some problems are impossible to solve asynchronously. So partially synchronous is used as a compromise.

3.2.4 Faults and Failure

- **Idea**: Measure *availability* of a service $\implies$ measured by percentage of non-faulty nodes at a given time.

Definition 3.2.2. (**Faults and Failure**) A faulty node is defined by the *node model*. A system *fails* (a *failure*) if the service cannot be provided due to *faulty nodes*.

- **Fault Tolerant**: Achieve high *availability* and avoid a *single point of failure*
- **Fault Detectors**: Algorithms that detects whether a node is faulty. Implemented via *health checks* w/ *timeouts*.
- **Problem**: Fault detectors cannot distinguish between a faulty node and a network partition / delayed message:
  - Only possible in a *synchronous, crash-stop and reliable links* system model.
  - Correctness of fault detectors in *partially synchronous systems* is asymptotic.

3.3 Time

3.3.1 Physical Time

Definition 3.3.1. (**Physical Clock**) A physical clock is a clock that uses a analogue technique to determine time.

- **Quartz Clocks**:
  - Quartz crystal resonates via *piezoelectric effect* at fundamental frequent f_0: IMAGE
  - **Problem**: Imperfections / environmental factors $\implies$ inaccuracy / error in expected frequency of the clock.

  Definition 3.3.2. (**Drift**) The clock *drift* δ is the rate of error between the expected frequency f_0 and the actual frequency f. Measured in ppm (parts per million).

  - $\delta > 0 \implies$ clock is running fast. $\delta < 0 \implies$ clock is running slow.

- **Atomic Clocks**:
  - Uses characteristic energy state transition in electrons (Caesium-133) to obtain frequency.
  - Improved accuracy. **Very expensive**.

- **GPS**:
  - Satellites w/ atomic clocks broadcast time + location.
  - Receiver computes distance Δs (from current + satellite location) to $\Delta t = c\Delta s$. Time is $t = \Delta t +$ satellite time.
  - Additional complexity added due to atmospheric effects

- **GMT** (Greenwich Mean Time):
  - Defined by 12:00 when the sun is south of Greenwich observatory.
  - $\implies$ Greenwich meridian.

- **Problem**: 2 non-consistent definitions of time (astronomy (GMT) and quantum (Atomic clocks)).

- **Solution**:
  - UTC (Coordinated Universal Time) is TAI (International Atomic Time) w/ corrections to be consistent w/ GMT.

  - Corrections via *leap seconds*
  - UTC Offsets = *time zones / daylight savings*.

- **Leap Seconds**:
  - UTC introduces *leap seconds* due to astromical reasons (e.g. variations in Earths rotation)
  - At 23:59:59 UTC (at the end of a month):
    * **Negative Leap Second**: Immediately jump to 00:00:00 UTC.
    * **No Effect**: After 1 second, jump to 00:00:00 UTC.
    * **Positive Leap Second**: After 1 second: 23:59:60 UTC. After 1 second: 00:00:00 UTC.

    Computed in *advance*.

Time Stamps

- A representation of a particular point in time.
- **Representation**:
  - **Unix Time**: Number of seconds, since 01-01-1970 00:00:00 UTC (*ignoring leap seconds*)
  - **ISO 8601**: `year-month-dayThour:minute:second+offset` w/ UTC `offset`.

  Problem: Conversion between representation requires:
  - Gregorian calendar (leap years)
  - and future / past leap seconds
- **Solution 1**: Ignore leap seconds.
  Problem: Ignoring leap seconds $\implies$ issues w/ time:

  a_1 `= System.currentTimeMillis(); // unix time`
  a_2 `= System.currentTimeMillis(); // (500ms apart)`
  a_2 `-` a_1`; // if during +ve leap second` $\implies$ `0/-ve`

- **Solution 2**: *Smear* the leap second (distribute over multiple hours) by adjusting expected frequency (to produce drift).

3.3.2 Synchronization

- **Problem**: Clock drift $\implies$ errors (over time) / clock skew

Definition 3.3.3. (**Clock Skew**) The difference θ between 2 clocks at given point in time.

- **Solution**: Periodically request time from a more accurate clock (e.g. GPS / atomic) to synchronize.
- **Network Time Protocol** (NTP):
  - Divide clocks in hierarchy (*strata*):
    - **Strata** 0: Atomic Clock / GPS Receiver
    - **Strata** $n+1$: Server synched w/ device from strata n
  - Client requests time from several NTP to estimate random variable Θ and compute mean.
  - Estimating Θ: IMAGE

$$\begin{aligned}
\delta &= (t_4 - t_1) - (t_3 - t_2) = L_1 + L_2 && \text{Round-trip network delay} \\
\Theta &= (t_3 + L_2) - t_4 && \text{Skew} \\
\hat{\theta} &= t_3 + \frac{\delta}{2} - t_4 && \text{Estimated Skew} \\
\varepsilon &= \left|\hat{\theta} - \Theta\right| = \frac{1}{2}|L_1 - L_2| && \text{Error}
\end{aligned}$$

  - Correcting clock drift:
    - $|\hat{\theta}| < 125$ ms $\implies$ **slew** clock (modify drift δ)
    - $|\hat{\theta}| < 1000$ ms $\implies$ **step** clock. Instantly reset clock to estimated timestamp: $t_4 + \hat{\theta}$.
    - $|\hat{\theta}| \geq 1000$ ms $\implies$ **panic** (rely on administrator).
- **Problem**: NTP stepping + leap seconds $\implies$ time-of-day clocks result in error.

Definition 3.3.4. (**Montonic Clocks**) A physic clock that produces monotonically increasing timestamps (not influenced by NTP stepping / leap seconds).

Time of Day	Monotonic
Unix/ISO timestamps	Number of seconds for arbitrary point
Subject to leap seconds / NTP stepping	Monotonically increasing
Compared (synced) across nodes	Compared internally
`System.currentTimeMillis()`	`System.nanoTime()`
`clock_gettime(CLOCK_REALTIME)`	`clock_gettime(CLOCK_MONOTONIC)`

3.3.3 Causality, Happens-Before and Logical Clocks

- **Problem**: Reliable networks allow *re-ordering* $\implies$ comparable timestamps *between nodes* to determine order.
- **Problem**: Synchronized clocks by NTP has error ε $\implies$ synched time-of-day clocks are not *causally consistent*.

  Solution: Happens-before relation formalizes order

Definition 3.3.5. (**Happens-before Relation**) Let M be the set of messages. Let $E = \bigcup_{m \in M} \{\text{send}(m), \text{recieve}(m)\}$ be set of events. Let $N(e)$ be the node n where event e occurred.
Let $\sqsubset_n : E \nrightarrow E$ be the strict partial order of events occurring on node n.
The happens-before relation $\longrightarrow : E \nrightarrow E$ is inductively defined by:

$$\frac{N(a) = N(b) = n \qquad a \sqsubset_n b}{a \longrightarrow b}$$

$$\frac{a = \text{send}(m) \qquad b = \text{recieve}(m)}{a \longrightarrow b}$$

$$\frac{a \longrightarrow b \qquad b \longrightarrow c}{a \longrightarrow c}$$

Theorem 3.3.1. $(\longrightarrow, E)$ is a strict partial order.

Definition 3.3.6. (**Concurrent**) Events a and b are concurrent, denoted $a \| b$, iff

$$a \| b \iff a \not\longrightarrow b \wedge b \not\longrightarrow a.$$

- Happens-before $\implies$ causal reasoning, defined by Relativity (limited by speed of light)

Definition 3.3.7. (**Causal Order**) Let $\prec$: $E \longrightarrow\!\!\!\!\!+\ E$ is a strict total ordering on E. $\prec$ is a *causal order* iff

$$\forall a, b \in E. a \longrightarrow b \implies a \prec b$$

- **Problem**: Computing $\longrightarrow$ is slow.
- **Solution**: Logical clocks $T : E \to \mathcal{D}$ that are *causally consistent*, defining a causal order: $T(a) < T(b)$.

3.3.4 Lamport Clocks

- **Idea**: Maintain *integer* count t of # of events occurred on node n
- **Lamport Clock**:
  - Each node n maintains Lamport clock L, and increments L on every *local* event e. $L(e)$ is value of L after incremented on local event e.
  - On sending message m: attach $L(\text{send}(m))$.
  - On recipient of message m w/ clock $L(\text{send}(m))$:

$$L(\text{recieve}(m)) = \max\{L, L(\text{send}(m))\} + 1$$

```
t ← 0; // on initialization

on_local_event() {
    // increment counter if local event occurs
    t++;
}

on_send(m) {
    t++ // increment (local event)
    send (t, m) via link ℓ
}

on_receive(t′, m) {
    t ← max(t, t′) + 1;
    deliver m to application
}
```

Theorem 3.3.2. For all $a, b \in E$:

(i) $a \longrightarrow b \implies L(a) < L(b)$. Hence causally consistent.

(ii) $L(a) < L(b) \not\implies a \longrightarrow b$

There exists $a, b \in E$ s.t $L(a) = L(b)$ for $a \neq b$

- **Problem**: $L(e)$ is *not a total ordering*.

Definition 3.3.8. (**Lamport Timestamp**) The pair $(L(e), N(e))$ is the Lamport timestamp of the event $e \in E$.

- Lamport timestamps *uniquely identify* events.

Definition 3.3.9. (**Lamport Total Order**) The Lamport total order $\prec$: $E \longrightarrow E$ is defined as the lexical ordering on Lamport timestamps:

$$a \prec b \iff (L(a) < L(b)) \vee (L(a) = L(b) \wedge N(a) < N(b))$$

Theorem 3.3.3. The Lamport total order $\prec$ is a *causal order*.

- **Problem**: Random tie for $a \| b$ (either $a \prec b$ or $b \prec a$)

3.3.5 Vector Clocks

- **Idea**: Maintain a *array* of integers, on for each node.
- **Vector Clock**:
  - Assume k nodes in system $N = (n_1, \ldots, n_k)$. Each node maintains a Vector clock $T = \langle t_1, \ldots, t_k \rangle$ where t_i is the Lamport clock of node n_i.
  - $V(e)$ is the vector clock (*timestamp*) of event e. $V(e)$ represents the set of events $\{a \in E : a \longrightarrow e\}$ (e + causal dependencies)
  - On sending message m: attach $V(\text{send}(m))$.
  - On recipient of message m w/ timestamp $V(\text{send}(m))$, merge timestamps.

```
T ← (0, ..., 0); // on initialization on node n_i

// increment counter if local event occurs
on_local_event() {
    T[i]++;
}

// increment counter for local event (send)
on_send(m) {
    T[i]++;
    send (T, m) via link ℓ
}

// update entire clock w/ lamport timestamps (indirect causal dependencies)
// and then increment for local event (receive)
on_receive(T', m) {
    T ← (max(T[j], T'[j]) : 1 ≤ j ≤ k);
    T[i]++;
    deliver m to application
}
```

Definition 3.3.10. (**Vector Clock Ordering**) We define the following equivalence relation $=$, partial order $\leq, <$ and concurrency relation on vector timestamps:

(i) $T = T' \iff \forall 1 \leq i \leq k.T[i] = T'[i]$

(ii) $T \leq T' \iff \forall 1 \leq i \leq k.T[i] \leq T'[i]$

(iii) $T < T' \iff T \leq T' \wedge T \neq T'$

(iv) $T \| T' \iff T \not\leq T' \wedge T' \not\leq T$

Theorem 3.3.4. For all $a, b \in E$:

1. $V(a) \leq V(b) \iff \{e \in E : e \longrightarrow a\} \subseteq \{e \in E : e \longrightarrow b\}$
2. $V(a) < V(b) \iff a \longrightarrow b$
3. $V(a) = V(b) \iff a = b$
4. $V(a) \| V(b) \iff a \| b$

3.4 Broadcast Protocols

- **Idea**: Generalize **unicast** (point-to-point) messaging to **multicast** (broadcasting to a group of nodes N)
- System model: partially asynchronous w/ reliable link (or best-effort w/ retransmit)

  IMAGE
- Note group N may be dynamic (grow / shrink). Nodes also broadcast to themselves.
- Broadcast protocols in 2 layers:
  - Convert best-effort broadcast into *reliable broadcast* via retransmission
  - Enforce ordering on reliable broadcast (requires buffer of messages)

3.4.1 Reliable Broadcast

- Broadcasting node sends message directly to every other node (assumes complete network)
- Retransmission + deduplication is applied to *best-effort links*
- **Problem**: Node crashes during broadcast $\implies$ partial broadcast
- **Solutions**:
  - **Eagar Reliable Broadcast**: On receiving message, forward it to every directly connect node.
    Inefficient (results in high traffic / congestion)
  - **Gossip Protocols**: Forward message to a small fixed number nodes, randomly chosen.
    Doesn't guarantee delivery (however, may be optimized s.t probability of not receiving is minimal)
- **Notation**:

- $N(m)$ is the node that broadcasts m.
- broadcast$(m) \in E$, event of broadcasting message m.
- $T(m)$ is the times (a vector) that m arrives at nodes $n_1, \ldots, n_k$. $T_i(m)$ is $T(m)[i]$

3.4.2 FIFO Broadcast

Definition 3.4.1. (**FIFO Broadcast**) A FIFO broadcast algorithm satisfies:

$$\forall m_1, m_2 \in M.$$
$$N(m_1) = N(m_2) \wedge \text{broadcast}(m_1) \longrightarrow \text{broadcast}(m_2) \implies T(m_1) < T(m_2)$$

- Ensures messages sent by the same node arrive in FIFO order. Messages sent by *different* nodes may be delivered in any order.

```
buffer ← ∅ // buffer of messages received
sequence_number ← 0 // current sequence number

delivered ← (0, ..., 0) // vector of received sequence numbers for all nodes

// sending m: tag m w/ node index i + sequence number
on_broadcast(m) {
    send (i, sequence_number, m) via reliable broadcast
    sequence_number++;
}

// receiving message m from j w/ sequence number and message m
on_receive(j, sequence_number', m) {
    buffer ← buffer ∪ {(j, sequence_number', m)};
    while ((j, delivered[j], m') ∈ buffer) {
        buffer ← buffer \ {(j, delivered[j], m')}
        deliver m' to application
        delivered[j]++;
    }
}
```

3.4.3 Causal Broadcast

Definition 3.4.2. (**Causal Broadcast**) A Causal broadcast algorithm satisfies:

$$\forall m_1, m_2 \in M.$$
$$\text{broadcast}(m_1) \longrightarrow \text{broadcast}(m_2) \implies T(m_1) < T(m_2)$$

- Concurrent messages can be delivered in any order

```
buffer ← ∅ // buffer of messages received
sequence_number ← 0 // current sequence number

// vector of received sequence numbers
// for all nodes (vector clock)
delivered ← (0, ..., 0)


// sending m:
on_broadcast(m) {
    T = delivered; T[i]++;
    send (i, T, m) via reliable broadcast;
    sequence_number++;
}

// receiving message m from j w/ vector T and message m
on_receive(j, T, m) {
    buffer ← buffer ∪ {(j, T, m)};
    while (j, T', m') ∈ buffer && T' < delivered) {
        buffer ← buffer \ {(j, T', m')}
        deliver m' to application;
        delivered[j]++;
    }
}
```

3.4.4 Total Order Broadcast

Definition 3.4.3. (**Total Order Broadcast**) A Total order broadcast al-

gorithm satisfies:

$$\forall m_1, m_2 \in M.$$
$$(\exists n \in N. T_n(m_1) < T_n(m_2)) \implies T(m_1) < T(m_2)$$

- Precise ordering of messages isn't defined e.g. m_4 arrives before m_3 on a given node.
- **Implementations**:
  - **Single Leader**: A node is designated as leader (sequencer). To broadcast message, send it to the leader, then FIFO broadcast from leader.
    Problem: Sequencer crashes $\implies$ consensus for new sequencer
  - **Lamport Clocks**: Attach Lamport timestamp to every message. Deliver messages in total order.
    Problem: Requires FIFO links to determine minimum timestamp.

  Not-fault tolerant. A single crash $\implies$ errors. (See supervision work)

3.5 Replication

- **Problem**: Fault tolerant distributed systems require maintaining copies of same data, or *replicas*, on multiple nodes.
- Cannot apply RAID (Redundant Array of Independent Disks) solution, since RAID relies on a *controller*.
- **Problem**: Managing updates (since immutable copies require one-time copies). Network issues $\implies$ repeated updates!
- **Solution 1**: Deduplicate requests (when retransmitted due to lost `ACK`s), Requires request log, stored in *stable* (non-volatile) storage.

Definition 3.5.1. (**Idempotent**) A function $f : A \to A$ is idempotent iff

$$\forall x \in A. f(x) = f(f(x))$$

$(or\ f = f \circ f)$

- **Solution 2**: Implement request operations as idempotent operations $\implies$ repeated requests have no effect on state (no deduplication)
- Solutions define different: **retry behaviors**
  - **at-most-once** semantics: send request, don't retry, update may not be applied.
  - **at-least-once** semantics: retry until acknowledged. may repeat requests.
  - **exactly-once** semantics: retry + idempotence or deduplication
- **Problem**: Systems w/ different idempotent *request types* (e.g. `add, delete`) $\implies$ interleaving may not be idempotent.
- **Solution**: Add logical timestamps to each request and instead of removing records, we add a `delete` flag (a *tombstone*).

  A protocol is periodically ran between replicas (anti-entropy) to resolve differences to reach a consistent state (using ordering on timestamps)
- **Problem**: Resolving **concurrent updates**
- **Solutions**:
  - **Last Writer Wins** (LLW): Use a total order on timestamps (e.g. *Lamport timestamps*) and store the *latest* ($\implies$ **data loss** on concurrent updates)
  - **Multi-value register**: Use a partial order on timestamps (e.g. *Vector clocks*) and preserve all concurrent updates as a set ($\implies$ **no data loss**. Storing clocks + sets of values is expensive)

3.5.1 Quorums

- **Problem**: Increasing number of replicas n $\implies$ increases number of faulty nodes

  Let $0 \leq p \leq 1$ is the probability of a node being faulty (assumed to be independent of other nodes). Let N be the random variable modelling the number of faulty nodes. $N \sim B(k, p)$.

$$P(N = n) = \binom{k}{n} p^n (1-p)^{k-n}$$

$P(N = k)$ decreases exponentially w/ $k \implies$ achieving fault tolerance w/ replicas requires a *quorum*.

- Majority **quorums**:
  - A majority quorum of nodes $N = \{n_1, \ldots, n_k\}$ is a subset $N' \subseteq N$ s.t $|N'| > \frac{k}{2}$ (based on Two Generals w/ Fair-loss)
  - A majority quorum N' will agree on consensus for the request response (using timestamps to determine correct response)
  - **Idea**: split replicas into **read** N_r and **write** N_w quorums s.t $N_r \cap N_w \neq \emptyset \implies$ RAW consistency.
- **Idea**: Synchronizing replicas may either use: **anti-entropy** protocol or *read repair*: if a **client** receives an older copy from a given replica, then it updates the replica w/ the latest received copy.

3.5.2 Broadcasts

- **Idea**: Implement replication (broadcasting state updates) using broadcasting algorithms. Quorums rely on **best-effort** broadcasts.
- **State Machine Replication**:
  - Replicas are *state machines* w/ deterministic updates.
  - Use a FIFO-total order broadcast algorithm to deliver requests:

    ```
    on_send_request(r) {
        send r via FIFO-total order broadcast;
    }

    on_receive_request(r) {
        // received from FIFO-total order broadcast algorithm (middleware)
        update state using r and deterministic procedures;
    }
    ```

    Requests delivered in the same order on all nodes $\implies$ consistent states. Known as *active replication* (independent state updates)
  - **Problems**: FIFO-total order must be **fault-tolerant** and requests *cannot* be processed immediately after receiving (due to broadcast buffering).

  - *Leader based* database replication:
    1. All transactions are executed on a leader replica L.
    2. Order of transaction commits $\implies$ total-order.
    3. Total order broadcast the commit updates to follower replicas

    Passive replication (dependent state updates, L only executes transactions) (Based on total-order broadcast implementation w/ leader)

- **Idea**: Use weaker broadcast algorithms $\implies$ introduces more requirements on state updates

Broadcast Algorithm	State Update Function Requirements
Total-order	Determinism
Causal	Determinism, *commutative* **concurrent** updates
Reliable	Determinism, commutative updates
Best-Effort	Determinism, commutative, idempotent, tolerate message loss

3.6 Consensus

- **Problem**: Leader implementation of total-order $\implies$ **single pointer of failure**.
  Nodes must elect a new leader via **manual intervention** (*failover*, inefficient for unexpected failure) or via **consensus algorithms**

Definition 3.6.1. (**Consensus**) Consensus is a set of nodes coming to an agreement about a value.

- Equivalent to total order broadcast: agreeing on the *next message* to deliver.
- System model: Fair-loss, partially synchronous, crash-recovery

Theorem 3.6.1. (**FLP Result**) There does not exist a *deterministic consensus algorithm* that is guaranteed to terminate in an asynchronous crash-stop system model.

- **Solution**: Use a *non-deterministic consensus algorithm* to avoid FLP restrictions. Byzantine consensus w/ quorums (see ??) also exists.
- Consensus algorithms prove **safety** properties (correctness). Timing properties e.g. *liveness* depends on *timing model*.
- Consensus algorithms:
  - Examples: **Raft, Paxos, Multi-Paxos, Zab**
  - Consists of *electing a new leader* when the current leader is faulty / unavailable (due to partition). Detected by **fault detectors** (health checks w/ timeouts)
  - **Problem**: Two leaders elected at the same time. Known as "*split brain*".
  - **Solution**:
    - Ensure ≤ 1 leader per **term**.
    - Term is incremented everytime a new election is started (when the leader is unavailable). Each node can only vote **once** per term.
    - Requires a **majority quorum** of nodes to elect a new leader.
  - **Problem**: Two leaders elected in *different terms*. Occurs due to incorrectness of **fault detectors** in *partially synchronous* networks.
  - **Solution**: When determining value for term t, leader must receive majority quorum acknowledgement *before* reaching consensus $\implies$ outdated leaders always discover if a new term exists + old leaders cannot make decisions.

3.6.1 Raft Algorithm

TODO

3.7 Transactions and Consistency

- Different definitions for *consistency* e.g. RAW consistency, Replication consistency, Transaction consistency $\implies$ **consistency models**.

3.7.1 Two-Phase Commit

- **Problem**: Executing a *distributed transaction* on *sharded data* (transactions that read / write on multiple data, w/ potential replication). Requires *atomicity* (all nodes commit or all nodes abort) $\implies$ *atomic commit*.
- **Note**: *atomic commit* differs from consensus:

Consensus	Atomic commit
≥ 1 nodes propose a value	*All* nodes (executing T) vote whether to commit / abort.
Any of the proposed values is used	Must commit if all nodes vote to commit. Otherwise abort.
Provided majority quorum, tolerant to faulty nodes.	Must abort if participating node is faulty

- *Two Phase Commit* (2PC) algorithm:

1. Client broadcasts transactions to each (participating) replica. Replicas execute transactions independently.
2. Client (manually) commits, sending request to the *leader* (or *transaction coordinator*)

(P1) 3. Coordinate sends **prepare** request to each replica

(P1) 4. Replicas respond w/ message indicating whether replica is able to commit (*checks disk / consistency constraints*)

(P2) 5. Coordinate collects responses and **decides** whether to commit. Abort if at least one node is faulty (fails to respond) or responses w/ abort.

(P2) 6. Coordinate sends commit / roll back message (depending on decision) to replicas.

IMAGE

- **Problem**: Coordinator $\implies$ **single point of failure** $\implies$ in-doubt / partially committed transactions (if coordinator fails during making / broadcast decision)
- **Solution**: fault-tolerant 2PC using *consensus* (total-order broadcast).

```
on_transaction(T) {
    commit_votes[T] ← ∅; // list of replicas that have voted
    replicas[T] ← ∅; // list of replicas participating in T
    decided[T] ← false; // whether current node (replica) has decided to
                        // commit / abort
}

// commit transaction T participating nodes R
// sent by client.
on_receive_commit(T, R) {
    for (r ∈ R) {
        send (Prepare, T, R) to r via link ℓ;
    }
}

// on receiving (Prepare, T, R) on replicate n_i
on_receive_prepare(T, R) {
    replicas[T] ← R;
    ok ← determine whether (node) current replica can commit;
    send (Vote, T, i, ok) via total order broadcast to replicas[T];
}

// on receiving (Vote, T, node id, ok). Delivered by the total order
// broadcast middleware
//
// total order broadcast prevents race condition introduced by fault detector
// since each node receives the votes in the same order ⟹ makes the same
// decisions.
on_receive_vote(T, j, ok) {
    // if voted, or not participating in T or already decided then ignore
    if (j ∈ commit_votes[T] ∨ j ∉ replicas[T] ∨ decided[T])
        return;
```

```
        if (ok) {
            commit_votes[T] ← commit_voted[T] ∪{j};
            if (commit_votes[T] = replicas[T]) {
                decided[T] ← true;
                commit(T);
            }
        } else {
            decided[T] ← true;
            abort(T);
        }
    }

    // fault detector, suspects node j is faulty
    on_suspected_faulty_node(j) {
        for (∀ transaction T. j ∈ replicas[T]) {
            send (Vote, T, j, false) via total order broadcast to replicas[T];
        }
    }
```

3.7.2 Linearizability

- **Problem**: Atomic commitment is not sufficient for *concurrent modification* of replicas $\implies$ *linearizability*

Definition 3.7.1. (**Linearizability**) (*Informally*), a set of operations are *linearizable* if :

(i) every operation takes effect **atomically** at some point during it's lifetime (after it started, before it finished)

(ii) all operations *behave* (despite many replicas) as if executed on a single copy of data

- Linearizability $\implies$ every operation returns "latest" value.
- **Note**:
  - serializability $\neq$ linearizability
  - timing of linearizability is defined by *real time / physical causality* $\neq$ happens-before relation (defined on events)

- Focus on **client-observable** behavior (leaves implementation abstracted)
- **Problem**: Quorum read / writes are not linearizability (see lectures for example).
- **Solution**: Read-repair $\implies$ *ABD algorithm*. Note: allows concurrent `set`s to overwrite each other (e.g. via LWWs resolution)
- **Idea**: Implement `cas` operation, atomic compare and swap using Quorum w/ *ABD* algorithm.
- **Problem**: Replicas see operations in different orders $\implies$ therefore different results of `cas` operation. So `cas` is not linearizable.
- **Solution**: Total-order broadcast defines order of operations on *all nodes* $\implies$ linearizable.

```
// Client request handling methods
send_get_request(x) {
    send (GetOperation, x) via total order broadcast;
    wait for response;
}

send_cas_request(x, old, new) {
    send (CasOperation, x, old, new) via total order broadcast;
    wait for response;
}

// Operations, delivered by total order broadcast middleware
on_receive_get_operation(x) {
    return response w/ local_state[x] as result;
}

on_receive_cas_operation(x, old, new) {
    success ← false;
    if (local_state[x] = old) {
        local_state[x] ← new;
        success ← true;
    }
    return response w/ success as result;
```

```
}
```

Advantages	Disadvantages
Distributed systems behave (wrt client view) as non-distributed systems	Performance (total order broadcast $\implies$ lots of message buffering and messages exchanged for consensus)
Simple programming model	Scalability issues. Suffers from the leader bottleneck
	Availability (if quorum is faulty $\implies$ livelock)

- **Problem**: Issues w/ linearizability $\implies$ weaker consistency models (w/ trade-offs)

3.7.3 Eventual Consistency

Theorem 3.7.1. (**CAP Theorem**) A system can either be strongly **Consistent** (linearizable) or **Available** in the presence of a network **Partition** (CAP)

- CAP Theorem $\implies$:
  - Linearizable consistency: Nodes cannot respond to requests if cannot communicate w/ a quorum.
  - Availability: Allow replicas to respond (even if cannot communicate w/ quorum) $\implies$ availability (but not linearizability). Known as *optimistic replication* (allows replicas to process operations on local state $\implies$ operating on stale data).

  Definition 3.7.2. (**Eventual Consistency**) Eventual consistency is defined as the property: if no new updates are made to an object, *eventually* all reads will return the latest value.

- Eventual consistency is applied to *optimistic replication*

- **Problem**: Eventual consistency is *weak*. e.g. premise may never be true / no guarantee on upperbound of *eventually*
- **Solution**: *Strong eventual consistency*.

  Definition 3.7.3. (**Strong Eventual Consistency**) Strong eventual consistency is defined as:

  - **Eventual delivery**: Every update made to one non-faulty replica is eventually processed by every non-faulty replica (guarenteed by *Eventual consistency*)
  - **Convergence**: Any 2 replicas that have processed the same set of updates are in the same state (regardless of order).

- Properties of eventual consistency models:
  - Doesn't require waiting for quorum / network communication
  - Causal broadcasts (or weaker algorithms) may be used
  - Requires **conflict resolution** for concurrent updates

TABLE

3.8 Concurrency Control

- **Idea**: Linearizability w/ multi-value register may be used for replicated collaboration software (e.g. google docs, etc)
- **Problem**: Inefficient, every read/write operation would require a quorum.
- **Solution**: Provide strong eventual consistency w/ optimistic replication.
- **Problem**: Reconciling concurrent updates.
- **Solutions**:
  - Conflict-free replicated data types
  - Operational Transformations

3.8.1 Conflict-free Replicated Data Types

Definition 3.8.1. (**Conflict-Free Replicated Data Type**) A CRDT is a replicated object that is accessed via the interface of a abstract datatype e.g. a set / tree

- *Operation*-based **map** CRDTs:
  - **Eventual Delivery**: Reliable broadcast $\implies$ every operation is eventually delivered to all replicas
  - **Convergence**: Applying an operation is *commutative*

  Implemented w/ Lamport timestamps.

```
// set of values, consisting of key k, value v and timestamp t
values ← ∅;

// client code for sending operations
on_send_read(k) {
    send (Read, k) via reliable broadcast;
}

on_send_set(k, v) {
    t ← new_timestamp(); // globally unique timestamp e.g. Lamport
    send (Set, t, k, v) via reliable broadcast;
}

// Reads a value w/ key k. Assumption: k has a unique value
on_receive_read(k) {
    if (∃t, v.(t, k, v) ∈ values) {
        return v;
    } else {
        return null;
    }
}

// Sets the mapping k ↦ v (requested at time t).
// Delivered by reliable broadcast middleware. Implements LLW
// approach (latest timestamp takes precedence)
```

```
on_receive_set(t, k, v) {
    previous ← {(t′, k′, v′) ∈ values : k′ = k};

    // if value doesn't exist *or* is the
    // latest version of the value, then set
    if (previous = ∅ ∨ ∀(t′, k′, v′) ∈ values.t′ < t) {
        values ← (values \ previous) ∪{(t, k, v)}
    }
}
```

- *State*-based **map** CRDTs:
  - **Idea**: Broadcast the state (set of values) instead of the operations.
  - Define a merge operator $\sqcup : \Sigma_s^2 \to \Sigma_s$:

    $$s_1 \sqcup s2 = \{(t, k, v) \in (s_1 \cup s_2) : \nexists (t', k', v') \in (s_1 \cup s_2).k' = k \wedge t' > t\}$$

    Informally, union of the two states w/out key duplicates (LWW approach).
  - $\sqcup$ is commutative, associative and idempotent.

Advantage	Disadvantages
Tolerates lost / duplicates messages once states converge Doesn't necessarily use broadcast. May use quorums w/ anti-entropy protocol	Increased message size (compared to operation-based map CRDTs)

```
...

on_send_set(k, v) {
    t ← new_timestamp();
    values ← {(t′, k′, v′) ∈ values : k′ ≠ k} ∪ {(t, k, v)}
    send (Set, values) via reliable broadcast;
}
```

```
...

on_receive_set(values′) {
    values ← values ⊔ values′;
}
```

- **Problem**: Operation-based **map** CRDTs not sufficient for text editing. *State* s consists as an array of characters. Operations:
  - Insert character c at a given index i
  - Delete a character at an index ??

  Operations *shift* indexes $\implies$ w/ network partition, indexes are *stale* (due to shifts of other operations)
- **Solutions**: Unique identifiers / operational transformations.
- *Text editing* CRDTs:
  - To identify positions in state s, attach *unique identifiers* i to each character c_i.
  - c_i between positions w/ identifiers j, k has identifier $i = (j + k)/2$ s.t $i, j, k \in \mathbb{Q}_{0 \leq 1}$
  - Operations:
    - Insert: Add character c_i to array, then sort by positions i.
    - Delete: Delete character w/ position i (added by node n), simply removed from array.

```
// returns element at index in chars
element_at(chars, index) {
    ...
}

// Initial characters. Each character consists of
// position i, node identifier n_j (node that adds c), and character c
chars ← {(0, None, ⊢), (1, None, ⊣)};

on_receive_read(index) {
```

```
    (_, _, c) ← element_at(chars, index + 1);
    return c;
}

on_send_insert(index, c) {
    (j, _, _) ← element_at(chars, index);
    (k, _, _) ← element_at(chars, index + 1);
    send (Insert, (j + k) / 2, n_j, c) via causal broadcast;
}

on_receive_insert(i, n_l, c) {
    chars ← chars ∪ {(i, n_l, c)};
}

on_send_delete(index) {
    (i, n_l, _) ← element_at(chars, index + 1);
    send (Delete, i, n_l) via causal broadcast;
}

on_receive_delete(i, n_l) {
    chars ← {(i', n_l', c') ∈ chars : i' ≠ i ∨ n_l' ≠ n_l};
}
```

3.8.2 Operational Transformations

- **Idea**: CRDTs could be implemented by computing a *transformation* on two concurrent operations $\implies$ resultant operation.

Definition 3.8.2. (**Operational Transformation**) An operational transformation T is a transformation on 2 concurrent operations resulting in a conflict-free operation.

- Generalizes to n concurrent operations (via folding)
- **Example** w/ Text-Editing CRDTs: $T(o_1, o_2)$ is the transformed operation of concurrent local operation o_1 and remote concurrent operation o_2. Requires track of operation history.

3.8.3 Google Spanner

- Example of concurrent system.
- Google spanner is a distributed **database system** w/ millions of nodes, petabytes of data, distributed worldwide.
- Strong consistency properties:
  - **Serializable** transactions
  - **Linearizable** reads and writes
  - **Atomic commits** of transactions (across shards)
- Uses following algorithms:
  - State machine replication (w/ **Paxos**) within shards
  - **Two-phase locking** for serializability
  - **Two-phase commit** for cross-shard atomicity.
- **Problem**: **Two-phase locking** for *read-only* transactions is disruptive for large # of objects.
- **Solution**: Transactions read from a *consistent snapshot* (like OCC), w/ *consistent* = causally consistent.
- **Multi-version Concurrency Control** (MVCC):
  - A form of OCC (Optimistic concurrency control)
  - Each read-write transaction T_w had a **commit timestamp** t_w $\implies$ updated values are tagged w/ t_w (old values still remain, until commit)
  - Each read-only transaction T_r has a **snapshot timestamp** t_r $\implies$ T_r ignores values w/ $t_w > t_r$ (uses old values)
- **Problem**: Cannot use physical clocks (not consistent w/ causality) or logical clocks (linearizability dependents on real-time ordering), and a leader distributing timestamps $\implies$ not sufficiently distributed.
- **Solution**: *TrueTime*, a system of physical clocks that return an *uncertainty interval* δ:

- Synchronization / physical clocks have *errors* (drift, skew, NTP synchronization, etc) $\implies$ may define an *uncertainty period* δ.
- Uncertainty intervals used to ensure events avoid overlapping intervals (by waiting). DIAGRAM