Distributed Systems

Suchithra Ravi
snarkitten@gatech.edu

Last Updated: May 21, 2021

The future is already here, it is just not evenly distributed yet.

— *William Gibson, Not in Cyberpunk*

Preface

> *The editor looked at his clothes and asked, "Can you spell cat?". The boy looked at him and said, "Can you spell anthropomorphology?"*
>
> — A controversial lady, *Left as an exercise to the reader!*

These are my notes for CS7210, the Distributed Computing course taught at Georgia Tech. The intent of these notes is to allow for a review of concepts and act as a supplement for the lecture material.

These are ***not*** *official course materials; they were created throughout my time taking the course, so there cannot be any guarantees about the correctness of the content. You can refer to the official recommended textbooks such as Distributed Systems for Fun and Profit or Distributed Systems for a better, deeper look at the concepts covered here.*

If you encounter typos; incorrect, misleading, or poorly-worded information; or simply want to contribute a better explanation or extend a section, please contact me on Piazza, Slack or via email. 🙂

Here, I must take a moment to thank George Kudrayvtsev for creating this incredibly beautiful LaTeX template and being generous enough to share it (thus enabling my laziness to create one, but also inspiring me to make LaTeX notes at all). If you find the notes beautiful, the acclaim is all his, and if you want to fund his stellar (pun intended) LaTeX abilities, you may find his Venmo here: `@george_k_btw`.

Introduction

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"*
>
> — Leslie Lamport, *Somewhere*

1.1 Why study Distributed Systems?

Because they are everywhere!!

Examples of DS applications:

- OMSCS
- FANG: Facebook (social media), Amazon (online shopping), Netflix (streaming, CDN), Google (Cloud), etc.
- Enterprise systems (banking, SaaS) hosted on public, private clouds
- Telecommunication industry
- Newer domains: AR/VR, Self-driving cars
- HPC: Massive multi-core

1.2 What is a distributed system?

A system consisting of multiple independent components, that can fail in some way, intentionally or unintentionally, transiently or permanently, and interacting/collaborating in some manner to complete some common task.

1.2.1 Intuition

- These independent components do not share all information with each other.
- Interact via some form of message passing, and these messages allow components to influence each other's actions
- Messages are not perfectly reliable- can get delayed or lost, leading the intended receiver to act differently that might even lead the whole system to become unstable.
- **Independent** computing units must appear as a **single coherent** computing entity implies that these units work on a common task/goal.

> Definition 1.1: **Distributed System**
>
> A **Distributed System** is a collection of computing units that interact by exchanging messages via an interconnection network and appear to external users as a single coherent computing facility.

Note that the definition does not explicitly mention failures!!

1.3 Simple Model of a Distributed System

- **What we show**:
  - **Nodes:**
    - Receive message from channel
    - Take some time to act on it
    - Send message to one or more channels
  - **Messages:** Spend some time in the channel and are delivered zero or more times.
  - Time axis: Shows a series of messages and when they happened
- **What we abstract out**:
  - The actual underlying network: Communication channels between nodes, number of hops, etc.
  - Directionality of channels: Treat channels as unidirectional i.e. messages may actually use the same communication channels underneath, but treated as separate channels
  - Actual processing at the nodes: any processing manifests itself as a) Time delay at the node b) future messages to the rest of the system

- **What we can represent**: This model is general enough to represent very different types of systems
  - **Node failure**: Set time taken to process at node to ∞
  - **Unreliable communication**: Set number of times delivered to 0 if dropped, to >1 if retransmitted.
  - **Point-to-point vs Multi-cast**: Set appropriate number of channels to which the messages are sent.

1.3.1 More complex model of a Distributed System

Typically, we think of a DS as the set of processing actions that happen at the nodes and the state changes as a result of those actions. Can include this in our model by adding a state variable to each node. Note that we are still not representing the individual processing actions at each node. **Actions** are triggered in response to messages and the outcome of the action is a change in state at the node.

1.3.2 Importance of Model

Much of the research in DS is a combination of theoretical predictions based on building and analyzing models, as well as deployment and practical evaluation of real systems.

But, why use models at all? Because **Alternative is too complex**: Alternative is to build a prototype and test under all possible scenarios - would be hard to mimic the correct number and distribution of nodes.

How to choose a model Any model is defined by elements, rules, and assumptions or **model invariants**.

- ▶ **Model invariants**: Statements that are always true for a model E.g. "Every message is delivered on time" -> this assumes lossless message delivery and no network failure.

To choose a model- we have to ensure it is:

- **Accurate**: Represents the actual system being studied i.e. *some* problems can be studied using the model
- **Tractable**: Is *analysis of the current* problem possible using the model

In other words, while choosing a model, pay attention to:

- *What* problems can we study using this model
- Can we build and analyze solutions for *this* problem using the model.

A simple model may be sufficient: as long as we can adequately represent the current system we are studying and all the possible transitions in the system, we can use the model.

1.4 What is hard about Distributed Systems

- **Asynchrony**: Message latency can be zero or bounded or unpredictable or infinite. (Most systems have unpredictable or infinite)
- **Failures**: Failures can be a "failstop" (sudden stop) or transient or Byzantine (system is performing incorrect action). Also failures can be in the nodes or in the network links.
- **Consistency**: We want a single up-to-date copy of data and all nodes agree on this copy. But, to come to this conclusion, need to consider concurrency/ordering in which various events happened, is the data replicated/cached, etc.

All of these introduce various tradeoffs in the system design.

Alternate way to look at this: in terms of 8 fallacies (i.e. statements that aren't always true in a distributed system):

Assumption/Fallacy	How it is violated
Network is reliable	The network may be unreliable and messages can be delayed or lost.
Latency is zero	Real networks have non-zero (sometimes unbounded) latency for packet delivery.
Bandwidth is infinite	Real networks have finite bandwidth, so there is a limit to the number of simultaneous messages sent/received.
Network is secure	Naturally, network may be insecure and this can affect message passing in the form of bandwidth choking or even malicious manipulation of messages passed
Topology doesn't change	If some components fail or new components are added (or a failed component is repaired), the network topology changes. There may also be other changes to topology.
There is one administrator	In a large system, having a single administrator may be inefficient forcing systems to use consensus (and maybe no administrators!)
Transport cost is zero	Transport cost in terms of infrastructure/bandwidth, as well as energy is realistically non-zero.
Network is homogenous	Some parts of the networks may be slow or even shut down for various reasons, making the network non-homogenous.

We will have to build systems that will work despite these assumptions becoming false.

1.5 Properties of a Distributed System

- **Consistency**: The system gives correct answers always
- **Availability**: The system provides responses always
- **Partition Tolerance**: The system provides responses irrespective of failures or delays in nodes/network

Consistency also implies that the system should act as a single unit. For this, it should co-ordinate actions by multiple components in the presence of failures and/or concurrency.

Other desirable properties include:

- **Fault-Tolerance**: System should recover from component failures without performing wrong actions

- **High Availability**: System should restore operations and resume services even after components have failed
- **Recoverability**: Failed components can restart and rejoin system after failure has been resolved.
- **Scalability**: System should operate correctly even after some aspect of the system (including load/users) gets scaled to a larger size.
- **Predictable Performance**: System should maintain performance despite failures
- **Secure**: System should authenticate access to data, state maintained and services provided

1.5.1 Correctness

Intuition : If we gave the same set (or series) of inputs to a distributed system and to a single computing entity, the distributed system should give the same outputs as the single entity for it to be correct.

Problem is: each node in the DS actually receives a separate set of inputs. So, to act like a single for the single entity, the inputs should be delivered in the same order i.e. we are concerned not just about order of inputs ot a single node but *global* order of inputs across all nodes. Also, all participants should *agree* that this is the order in which events occurred.

Consistency Model: is a set of guarantees provided by the system about the ordering of events.

Different types of consistency models are possible:

- **Strict Consistency**: Guarantee that all events and changes in the system will have a single, uniform order and all parties agree upon this order.
  - ▶ Almost impossible to achieve: Impossible to guarantee all nodes will have the same notion of time.
- **Linearizable**: Transactions (i.e. Group of operations that read/modify some shared state) will not appear to be interleaved with other ongoing transactions i.e. the transactions will appear to be in linear order, even though individual operations are not.
- **Serializable**: The system guarantees that the outputs correspond to *some* ordering of the transactions, but this need not correspond to the real-time ordering itself. (this *could* have occurred on a single node, but *need not* have.)
  - ▶ All nodes in the system should still perceive the same ordering.

1.6 Brewer's CAP Theorem

This is actually a **conjecture**, not a theorem since it was never proven!

Theorem 1.1: CAP Theorem

A Distributed System can never simultaneously meet all 3 properties: Consistency, Availability and Partition Tolerance.

If a network partition occurs, you can have Consistency or Availability, but not both. Note: We **can** have both if there is no network partition (i.e. partition tolerance is not there).

Systems are often classified by the CAP tradeoffs they make (or guarantees they provide):

- **P+A**: Key-value stores like Cassandra, DynamoDB
- **P+C**: Megastore, MySQL Cluster

Despite what CAP theorem says, in practice, slow response = no response!
$\implies$ If a system is "available" but has high latency, it is not really available
$\implies$ Actual tradeoff is between **Latency** and **Consistency**

Theorem 1.2: PACELC

If there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?

Remote Procedure Call

2.1 Client-Server Architecture

Common architectural pattern for building distributed systems

- **Clients**: Some nodes in the system that send requests and/or data to the server nodes
  - ▶ Client has to find the address of the server it will contact
  - ▶ Client **may** need to establish connection with that server (depends on protocol)
  - ▶ Client has to copy "arguments" for its request from its memory to network packets
- **Servers**: Nodes that receive the request and "process" it
  - ▶ Processing may include accessing database to retrieve data
  - ▶ Copies data from server memory into netork packets to send to client
- Note here that the server and client *could* reside on the same machine

2.1.1 Challenges

- **Discovery and binding**: Client needs to find the server and establish connection
- **Identifying interface and parameter types**: Client needs to know the operation and the parameters required for it
- **Data representation**: Client and server should have previously agreed on how data will be represented
- **Explicit Data management necessary**: Data needs to be explicitly copied from the network packet buffer to server/client memory
- **Unpredictable Execution time**: Results may arrive at arbitrary time de-

pending on network/server delays.

- **Unknown Cause of failure**: Client cannot tell if failure in network or server (or even type of failure)

- ▶ All the problems above must be **explicitly** handled.

2.2 Role of RPC

- Address challenges above and simplify client-server interaction
- Hide complexity of distributed programming and make it similar to programming single node systems
    - ▶ Why? When these were developed, local programming was centered around procedures and procedure calls

To achieve its goals, RPC needs to provide the following:

- **Service registration mechanism**: Servers should be able to register the services they provide + clients should be able to find out that information
- **Connection establishment**: Support for any connections to be established
- **Interface specification**: Client should be able to find out what are the parameters required and results returned
- **Type specification**: Client should be able to find data type and order for parameters and results
- **Data management**: System should manage the data transfer from/to memory to/from network buffer
    - Also referred to as serialization (**marshalling**) and deserialization (unmarshalling)
    - Serialized bytestream has service descriptors (metadata) along with parameters/results (data)
- **Dealing with failures**: Support to deal with failures
    - Timeout and retry for transient failures
    - Timeout and give some error message for permanent failures

2.3 Architecture of RPC System

- **Topmost level: API**: Programming interface that clients and server applications use e.g. making RPC call

- **Second Level: Stubs**: RPC Call jumps into the stub layer. The stub layer knows about the procedure arguments/results -> Performs Data marshalling/unmarshalling
- **Lowest Level: RPC Runtime**: Connection management, sending/receiving data, failure management
- **Other Components**:
  - **Interface Definition Language (IDL)**: Used to create interface specification (servers describe services, arguments etc.)
  - **RPC Compiler**: Compiles IDL and spits out stub (and other code)
  - **Service registry**: Establishes rules on how servers would announce their services

How this is actually used :

1. Server developer develops app
2. Server developer provides specification in IDL
3. RPC Compiler compiles this spec
4. RPC Compiler generates stub code + skeleton server code
5. Implementation of the app service added to the skeleton server code
6. Service registered into the registry
7. Client writes client side code that calls the app
8. Client compiles this code with the code generated from RPC compiler
9. RPC Runtime takes care of everything else at runtime!!

2.4 Anatomy of an RPC Call

Assume server implements some operation ***op***(arg1, arg2) that the client cannot perform by itself i.e. To do op, client needs to send the op, arg1, arg2 to the server.

Goal of RPC: To make this work like a regular function call (do everything else under the hood).

Client Side :

1. Client makes function-like call res = op(arg1, arg2)
2. Program Counter jumps to the stub implementation of this function (instead

of the local memory address of the procedure implementation)

3. Stub creates a buffer with function descriptor and arguments required for the procedure in the format expected by server.
4. RPC runtime takes care of connection establishment, etc. required to send message
5. RPC runtime sends a message to the server with the buffer information

Server side :

1. Message received is given to the RPC Runtime which gives it to the stub
2. Stub unmarshals arguments and determines which function has to be called
3. Stub calls the actual local implementation of the function
4. Function is executed and returns results to Stub
5. Control comes back to the server side stub
6. Stub populates the message buffer with the result
7. Stub sends the message back to the client

Back on the client side :

1. Client stub unpacks the result
2. Client Stub quietly places it in the memory location expected
3. Control returns to *voila* the original calling function
4. Calling function can extract the result from the memory location *as if all of this just happened locally!*

2.5 Invocation Semantics of RPC Operation

RPC operations can be classified along different dimensions:

Classification based on control transfer

- **Synchronous** RPC operations: Blocking operation where client thread waits on RPC call to complete
  - ▶ Client makes RPC call -> waits for response
  - ▶ Calling thread cannot move forward till the RPC response is received.

  - Other threads might continue to run, even make other RPC calls, and wait for those!

- **Asynchronous** RPC operations: None-blocking operation where client makes RPC call but continues to do other operations.
  - ▶ Client makes the RPC call -> does other actions (not dependent on call) while waiting
  - ▶ All independent tasks completed -> client checks if rpc response available -> Processes result or waits
  - – Obvious advantage: Hides latency
  - ▶ Registering a **callback**: Client can choose to be notified when the response arrives and specify the things that need to be done when it does.

Classification based on guarantees about message delivery

A local procedure call executes *exactly once* and returns a result. Here, lack of response from procedure implies deadlock or failure. If the procedure hangs or crashes, we can always assume (correctly) that the procedure never executed, and the solution is to restart and redo the operation. In other words, *the caller of the procedure always knows whether the call was executed or not.*

In a distributed system, no guarantee that the server will respond. But lack of response does not imply failure. It might simply be because of network issues: request lost, response lost, etc. But it may also be a real server failure. Rerunning may have undesirable effects (if the command was previously executed, can corrupt the data-like append operation in Project2.)

Distributed Systems may give different types of guarantees about execution:

Invariant Definition 2.1: Exactly Once Execution

The ideal scenario: Commands executed exactly once (similar to local procedure calls)

- ▶ Client side: timeout and retransmit if no response
- ▶ Server Side: Distinguish between repeated requests, filter if needed
  - – may not be needed e.g. if add operation with both arguments specified on the call, redoing operation is ok.
- ▶ Dealing with persistent failure (e.g. server node failure)

Invariant Definition 2.2: At Most Once Execution

Slightly worse scenario: executed once or not at all

- Server Side: Distinguish between repeated requests, filter if needed
- Client side:
  - Timeout and retransmit if no response
  - Client knows the command may not have been executed

Invariant Definition 2.3: At Least Once Execution

Third option: Executed once or more (but never none)

- Client side: Timeout and retransmit if no response
- Server Side: No guarantee that dupicates will be eliminated

Other semantics also possible: e.g. if it works with multiple replicas, system may guarantee that calls will be replicated in all replicas or at least in one replica etc.

2.6 Examples of RPC Systems

- **sunRPC**: Original RPC developed by Sun in the 80's
- SOAP, CORBA: Older systems used in enterprise solutions
- Apache Thrift
- gRPC

Some of these RPC systems are specialized for specific contexts, e.g. high-speed (low latency), reliable network, embedded environment (optimizations to ensure small footprint), etc.

2.6.1 gRPC

Released by Google around 2016 - inspired by sunRPC.

- Relies on **protocol buffers** which allow us to describe interface, perform data serialization, etc.

- Interface specified in a .proto file, which is compiled to generate protoc file

(Refer to gRPC code example)

Time in Distributed Systems

> *People like us who believe in physics know that the distinction between past, present and future is only a stubbornly persistent illusion*
>
> — Albert Einstein, *Letter to Michael Besso*

3.1 The Time Problem

Physical Time: Time that we read off some "physical" clock.

3.1.1 Why do we need to measure time in DS?

In a single node, easy to determine sequence (i.e. ordering) of operations uniquely.

Why do we need to know the order?

- **Causality** Helps determine causality i.e. if one operation affected another -> needed for correctness, consistency.
- **Resource Allocation** Important for resource allocation, especially if we want to maintain some properties like fairness.
- **Garbage Collection** Useful to perform garbage collection. Can deduce if the results of an operation are no logner needed and free up resources.

3.1.2 Why is measuring time hard in DS?

Why can't we simply read the local clock at each node to determine the ordering of operations? For messages, whose clock should we read?

Option 1: Receiver-based Timing

Use the time at the receiver's end to maintain ordering. Problem: Because of network delays, messages may arrive out-of-order!

E.g., node n_3 receives message m_1 from n_1, then receives m_2 from n_2 and concludes that m_1 *happened before* m_2. But this need not be the correct order because:

- **There are no guarantees about network delays and packet losses** i.e. m_2 might have happened earlier, but the packet might have taken longer to arrive.
- **Different nodes may infer different orders** e.g. if the same messages are also sent to node n_4 that is, say, closer to n_2, it might receive message m_2 before m_1 and infer the exact opposite order as n_3!!
- **Some messages may never arrive!!** Messages can get lost forever (infinite delay), in which case no order can be inferred for those events!

So clearly, we cannot rely on the timestamp at the receiving node.

Option 2: Sender-based Timing

Stamp each message with **local timestamp** at the sender. Expect order to be unique and same as actual occurrence of events at the sender.

Not so fast!

Problem: **Clocks may ont be globally synchronized.** If there is a large enough skew between 2 senders, the ordering may be wrongly reported.

E.g. If clock at n_1 is running slightly ahead and reports a later timestamp than the clock at n_2, even though the message m_1 actually happened before m_2, the timestamps received by n_3 indicate the opposite, so the wrong order will be inferred.

Thus we can see measuring time is hard in Distributed Systems for the following reaons:

- **Consensus/Synchronicity**: All nodes need not agree on what the global time is
- **Network delays**: Message propagation need not take fixed (or predictable) amount of time. Network delays need not be constant or even consistent across nodes (i.e. some nodes may see greater delays in sending/receiving messages than other nodes)
- **Failures**: Both nodes and network connections can fail at any time.
- **Malicious nodes**: For instance, we can't trust all the timestamps we receive.

3.1.3 Logical Time

Physical clocks clearly unusable, but we still need some concept of time measurement. Solution: Logical clocks!

Logical clocks do not measure same kind of "time" as physical clocks. Instead, logical clocks generate timestamps which can be used to infer the real ordering and relationship of events in a distributed system. There are 3 types of logical clocks:

- ▶ Scalar Clocks (Lamport Clocks)
- ▶ Vector Clocks
- ▶ Matrix Clocks

3.2 Representing Time and Sequence

This section describes some common notation and terms we shall use later on.

- Processes represented by p_i generate events represented by e_i^k.
- **Happens before**: If a process p_i generates events $e_i^0, e_i^1, e_i^2 \ldots e_i^k, e_i^{k+1}, \ldots e_i^n$. Here each event e_i^k *happens before* e_i^{k+1}, represented by $e_i^k \rightarrow e_i^{k+1}$ Thus,

$$e_i^k \rightarrow e_i^{k+j} \; \forall j \geq 1$$

- **Process history**: ordered sequence of events in process p_i represented by H_i.

Note: We often focus on messaging events (sending or receiving), rather than internal events, since the impact of messaging events on other nodes is clear.

Definition 3.1: Happens Before Relationship

The **happens before** relationship is a relationship between 2 events defined by the following rules:

- At any node i, an internal event k *happens before* internal event $k+1$ i.e. $internal_i(k) \rightarrow internal_i(k+1)$
- At any node i, receiving message k *happens before* sending the next message i.e. $recv_i(m_k) \rightarrow send_i(m_{k+1})$
- Across two nodes i and j, sending of a message *happens before* receipt of the message elsewhere i.e. $send_i(m_k) \rightarrow recv_j(m_k)$

3.2.1 Time Diagrams

Alternate way to represent the sequence of events is using **Time Diagrams**.

- Processes are shown as horizontal lines
- Dots on the lines indicate different events
  - Messaging events are represented by arrows that connect the send event on one process with the receive event on another (with the arrowhead pointing towards the receiver)
  - Arrow-less dots represent internal events.

Definition 3.2: Concurrent Events

Two events may be such that there is NO *happens before* relationship between them. Such events are said to be **concurrent**.

E.g. message m_1 goes from node p_1 to p_2 and m_2 goes from p_3 to p_4. If these are completely unrelated processes, no relation between these events i.e. if $e_1 = send_1(m_1)$, $e_2 = send_3(m_2)$,

$$e_1 \not\to e_2 \text{ and } e_2 \not\to e_1 \implies e_1 \parallel e_2$$

.

- ▶ Either event could have actually happened before the other
- ▶ Swapping the order between the events has no impact on rest of the system
- ▶ Both events may still have happened before a different event e_3

$$e_1 \to e_3 \text{ and } e_2 \to e_3$$

3.3 Clock Consistency

Invariant Definition 3.3: Clock Consistency Condition

To be useful, a logical clock should timestamps that reflect the actual relationship between events. This is stated as the **Clock Consistency** condition.

Say logical clock C produces timestamps $C(e_i)$ for each event e_i.

- **Monotonicity** If 2 events are connected by a *happens before* relationship, their timestamps need to reflect that.

$$e_1 \to e_2 \implies C(e_1) < C(e_2)$$

Thismeans that a clock cannot produce the same timestamp repeatedly or timestamps decreasing in value.

- **No implication for concurrent events**:

$$e_1 \parallel e_2 \implies C(e_1)??C(e_2)$$

Strong Clock Consistency: Logical clocks that satisfy this property guarantee that we can determine the order of events uniquely by looking at their timestamps i.e. with stongly consistent clocks,

$$e_1 \to e_2 \iff C(e_1) < C(e_2)$$

Logical clocks are clocks used to map the history of events in a process to a partially ordered time domain T. Why partial? Because we cannot have a "complete" order if there are concurrent events.

A **clock function** C is defined as the set of rules that must be followed to produce proper, consistent timestamps.

To implement a logical clock, we need to decide upon

- A data structure to represent the timestamps
- Rules that will be followed to advance the time.

3.4 Lamport's Scalar Clock

Each node has its own implementation of the clock, which executes clock rules to produce timestamps. A node only knows the value of the clock that it computed, but all nodes see the clock as a single scalar value.

3.4.1 Clock Definition

- **Data Structure**: Each process p_i has its own clock C_i and the timestamp produced is a **scalar**
- **Rules to generate timestamps**: The rules can be derived by considering the rules for the *happens before* relationship.
  - For 2 internal events a and b where a *happens before* b, if $a = e_i^k$ and $b = e_i^{k+1}$,

$$e_i^k \to e_i^{k+1} \implies C(e_i^k) < C(e_i^{k+1})$$

. Therefore, p_i **increments** C_i **between successive events.**

$$C_i(b) = C_i(a) + 1$$

- For 2 related messaging events $a = send_i(m_k)$ and $b = recv_j(m_k)$ in processes p_i and p_j respectively,

  $$send_i(m_k) \to recv_j(m_k) \implies C_i(send_i(m_k)) < C_j(recv_j(m_k))$$

  But, $C_j(recv_j(m_k))$ has to be greater than any other internal event at p_j that happened before the arrival of the mesage m_k at p_j as well. Therefore,

  $$C_j(b) = max(C_i(a) + +, C_j)$$

3.4.2 Clock Correctness

Note that 3 types of time relationships are possible with this clock: (See lecture example)

1. One event *happens before* another
2. One event is concurrent with another
3. There is no clear order between 2 events. These will also be inferred to be concurrent!!

Note:

1. Last point above is not a problem for Lamport's clock because it only satisfies the clock consistency condition and not the strong consistency condition. (See below)
2. Even if we conclude $4@p_2 \to 3@p_1$ (based on their timestamps), it is ok because:
   - concurrency implies that swapping the order between them will not impact the rest of the system.
   - all observers in the system will make the same, uniform conclusion about the ordering.

Thus, Lamport's clock gives a mapping from all events in the system to a partially ordered list of timestamp events. (Partial because there is no order among concurrent events).

To establish total order, need some tie-breaking rule e.g. Events with same timestamp from 2 different processes can be ordered based on the process ID, i.e. $3@p_1 \to 3@p_2$. As long as everyone in the system uses same tie-breaking rule, the actual rule itself doesn't matter, since any consistent order between these events is acceptable.

Event counting

: If timestamps are always incremented by 1, the clock can be used to estimate the **minimum** number of events in the system that occurred before the current event, including events in all nodes, even nodes that have not previously communicated with this node in the past. Of course, this is a minimum estimate, so actual number of events in the system may be much larger.

Lamport's clock is consistent, but not strongly consistent!

$$e_1 \rightarrow e_2 \implies C(e_1) < C(e_2)$$
$$C(e_1) < C(e_2) \not\Longrightarrow e_1 \rightarrow e_2$$

Impact of consistency vs. strong consistency

- We cannot rely on Lamport's clock to get complete information about causality
- Consistency alone is sufficient for correctness, since ordering can be inferred everywhere that it is needed
- Lack of strong consistency leads to some loss of efficiency, because unrelated events may *appear* to be related and to require a specific ordering, which may be enforced by the system. E.g. 5@p_2 appears to be earlier than 6@p_3 though they are concurrent. System may try to enforce ordering even though it is unnecessary.

3.5 Vector Clock

Vector Clock is a vector of scalars with dimensionality of vector $\propto$ number of nodes in the system. ($\implies$ Size overhead: O(N) (unlike O(1) size of scalar clock)) More powerful than Scalar clock, but also a more complex clock.

3.5.1 Clock Definition

Basic idea:

- Each node maintains its own view of its own time as well as time at other nodes
- Each node has a copy of the vector of size n, and each element of the vector corresponds to the current nodes perception of time in the i'th node.

How it works: Say process p_i has clock vt_i

- If C_i is the i'th Lamport clock of process p_i,

$$vt_i[i] = C_i$$

- If C'_j is the p_i's perception/knowledge of the Lamport clock in process p_j,

$$vt_i[j] = C'_j$$

Rules to update the vector clock:

1. Before executing any event, update own clock:

$$vt_i[i] = vt_i[i] + d \text{ for } d > 0$$

2. Each message carries the vector clock vt of the sender at sending time. When a message is received, the receiver :
   (a) Update its knowledge of global time i.e. of all other processes

   $$vt_i[k] = max(vt_i[k], vt[k]) \text{ for } 1 \leq k \leq n$$

   (b) Execute rule 1 for itself i.e. update its own time

   (c) Deliver/process the message

▶ Rule 1 ensures that later events in the same process will always have a larger timestamp than older events

▶ Rule 2 ensures that before a message is processed, the recipient has the latest view of the system (either the same view as the sender, or a more advanced view). This means when the message is actually processed, its timestamp will be greater than all the events at the sender as well as at the recipient (since rule 1 is also executed!)

Rules to compare vector clocks:

- If all the elements in vector vt_1 are lesser than or equal to all the elements in vt_2,

$$vt_1 \leq vt_2 \text{ if } vt_1[i] \leq vt_2[i] \forall i$$

- If all the elements in vector vt_1 are lesser than all the elements in vt_2 AND at least one element in vt_1 is strictly lesser than vt_2,

$$vt_1 < vt_2 \text{ if } vt_1[i] \leq vt_2[i] \forall i \text{ AND } vt_1[k] < vt_2[k] \text{ for some k}$$

- If 2 timstamps are such that neither is strictly less than the other, they are considered to be concurrent.

$$vt_1 \parallel vt_2 \text{ if } !(vt_1 < vt_2) \text{ AND } !(vt_2 < vt_1)$$

Vector Clock is both consistent and Strongly consistent!

- Consistent because if one event occurs before another, the timestamp is definitely lesser.
- Strongly consistent because if the vector clock indicates concurrency of events, the events are guaranteed to be concurrent.

Thus, vector clocks provide a stronger consistency and higher efficiency (since we will no longer reorder unnecessary concurrent events (See consistency discussion)) at the cost of maintaining O(N) extra state i.e. additional clock state at each node, but also additional data sent with each message. Some of this can be reduced by using some compression techniques.

3.6 Matrix Clock

Assume process p_i has matrix clock mt_i.

- Extend vector clock to a matrix -> represent time as a matrix.
- The clock value at (i,i) for matrix clock of the i'th process holds its own scalar clock
$$mt_i[i, i] = C_i$$
- The clock values at row i of the i'th process hold its vector clock (similar to above)
$$mt_i[i] = vt_i$$
- The clock values in all other rows (say k) hold what process p_i thinks the vector clock of process p_k is.
$$mt_i[k] = vt'_k$$
- Every process maintains its view of every process' view of time (and not just its own view of every process' time)

Obviously more complicated, and more storage required. So what do we gain? Since each clock knows what other clocks think about everyone's view of time, **garbage collection** is possible.

E.g. we know that process 2 thinks that its own time is X and process 3's time is Y, if process 1's time is greater than both X and Y, any state concerning process 2 and 3 stored in 1 can be cleaned up. In other words, if $mt_i[k, j] > t$, that is- if "i's perception of what k thinks j's time is" is >t, then everything before t can be deleted.

$$\therefore min(mt_i[k, j]) > t \implies \text{ everything before t can be deleted!}$$

State in Distributed Systems

> *Everything is in a state of flux, including the status quo.*
>
> — Robert Byrne, *The 637 Best Things Anybody Ever Said*

4.1 The problem of State

Goal is to capture a correct snapshot of global state in the distributed system.

4.1.1 What is state?

Intuition: A distributed system is defined as a set of nodes connected by communication channels The state of this system is then the state of the nodes and the state of the channels. To execute a program, each node goes through a series of events, both internal and message send/receive events. Therefore, the state of the distributed system is defined by the sequence of these events. So, we also need to capture state transitions and their sequence.

Thus, state consists of:

- **Process State**: determined by all events that happened on that node until that point.
- **Channel State**: determined by the messages currently in-flight i.e. send events not paired with a corresponding receive event on another node.
- **State transitions**: Any event changes state of at least one of the components i.e. nodes or channels.
  - Internal events → modify state of single node.
  - Message send/receive events → modify state of 1 node + 1 channel.
- **Ordering of events** = sequence of state transitions.

- **Actual run**: Actual sequence of events (or state transitions) that happened in a system
- **Observed run**: Sequence of events that we observe.
  - IMPORTANT: This may not correspond to the actual run!
  - ∗ We may not observe one of the events that actually happened, or we may not know the strict ordering between 2 events

4.1.2 Cuts

- **Cut**: A curve that "cuts" the execution time in some manner so that events that occur before the cut are assumed to be completed.
  - In the case of a message in-flight, this means the send event was completed even if the receive wasn't.
- **Consistent Cut**: A snapshot of the system that divides the execution events so as to provide a *possible* ordering of events in the system.
  - Consistent cut need NOT correspond to a real situation that the system has actually been in, but it captures one *possible* situation that is "consistent" with the ordering of the events in the system.
- **Pre-recording events**: All events that happened before the time of taking the snapshot (or before the cut) in the system.
- **Post-recording events**: All events that happened after the time of taking the snapshot (or before the cut) in the system.

Intuition for cuts : Think of the cut as a line we draw on the **time diagram** representation of the distributed system.

- **Not a strict line** because we cannot guarantee that we can get a precise cut at the exact same time on all nodes. Our best guarantee is to get a cut that indicates the completion of certain events in each node.
- ***Any cut is NOT a consistent cut*** i.e. a general cut does not have to give a "correct" description of the system.
- **Can make deductions about events**:
  - **In-flight messages**: By looking at completed messages at any point, we can also deduce which messages are in-flight (i.e. send completed but receive was not completed.)
  - **Older events**: If a message receive event happens prior to the cut, the send event also definitely happened before the cut. Similarly, if event e_i^{k+1} occurs before the cut, event e_i^k definitely does.

- Note that: A cut that has, say a receive message event preceding it, but no corresponding send message event, would not be considered consistent.

4.1.3 Challenges in capturing state

Capturing state in a distributed system is hard for multiple reasons:

- **No instantaneous recording**
  - **No globally synchronized clock**: Cannot ever capture events at each node at exactly the same time.
  - **No centrally initiated snapshot**: For the same reason, cannot assume that a centralized node will be able to instantaneously initiate a snapshot of all the other nodes.
    - Random network delays imply that even if Node X initiates a snapshot at t_0, node Y and node Z may take the snapshot at times t_1 and t_2 with no guaranteed ordering between them.
- **Non-determinism**
  - **Non-deterministic Computation**: Because of concurrent events, at any point, we can have multiple possible next events → Makes ordering, consistency etc. hard even with a single, multithreaded system.
    - **Deterministic Computation** means that at any point int he execution, there is at most one event that can happen next.

4.2 System Model

For the remainder of this chapter, we will use the following model of a distributed system:

- System consists of processes p_i that communicate by exchanging messages via channels c_i
- Channel properties:
  - Directed: any channel carries messages in only one direction $\implies$ possible to have 2 processes p and q where p can send messages to q but not the other way around!
  - FIFO: messages are delivered in the order in which they are sent
  - Error-free: messages will not be corrupted

Though channel assumptions are not applicable to all systems, any system can be made to satisfy these properties by using TCP as the communication protocol (since TCP guarantees these properties).

4.3 Finding a Consistent Cut

Goal of the algorithm :

- Capture snapshot of all distributed components i.e. Processes and channels
- Ensure the snapshot forms a consistent cut.

Example : (See figure in lecture) Consider 2 processes exchanging a sequence of messages. For the example, ignore all internal events and consider only message send/receives.

1. The processes start in initial states S_p^0, S_q^0. Say, we try to capture the snapshot of the system at some point, S_q^1.
2. We send a marker to the other node p (to capture a snapshot). Say the marker arrives at p at S_p^2. This point is after m3 has been sent by p.
3. To find hat happened to m3, send another marker message from p to q. Say this message arrives at S_q^3
4. Now, at q, we realize we already took a snapshot of this node. This snapshot indicates that m3 has not arrived at q yet.
5. This means, the message m3 must be in-flight.

Based on these steps, the recorded snapshot is :

- ▶ P is at S_p^2, Q is at S_q^1, Channel PQ has m_3 in flight, channel QP is empty.
- ▶ Recorded global state = (S_p^2, S_q^1), $(m_3, 0)$

Chandy-Lamport's Snapshot Algorithm

- **initiator node** is the node that will trigger the algorithm for capturing the state of the distributed system.
  - save its local state
  - send a marker token on all of the outgoing channels and all of the outgoing edges.
- All of the other nodes in the system will participate in this algorithm
  - On receiving the first marker on any incoming edge, they will save their state, they will mark the state of the incoming channel as empty
  - propagate a marker message on all outgoing edges
  - t resume execution, but they will also save incoming messages until a mark, on any of the channels until they see a marker arriving through that

particular channel.

  - When a marker does arrive on one of the incoming channels, then the process will mark the state of that channel. Such that all of these messages that were received since the process captured its state originally, all of these messages will be marked as messages that were in-flight.

This algorithm guarantees a consistent state (but not necessarily the state we actually went through previously or the state in which system is currently!)

4.3.1 Assumptions of the algorithm

- There are no failures and all messages arrive intact and only once. And we said that current technologies such as TCP/IP will allow this assumption to be true.
- The communication channel are unidirectional. And this really just means that we need to separately consider each direction of the point-to-point message exchanges among the processes.
- communication channel is FIFO ordered. TCP can give us the FIFO property
- does not interfere with the normal execution of the processes. Meaning that the markers, they don't stall or reorder the processing of the other messages. This is also in principle practically doable. You can just have a separate task or a separate thread that uses a separate socket and port number for the marker messages.
- And also it means that each process in the system records its local state and the state of all of its coming channels. assumption here that there are enough resources to be able to do this

4.3.2 Properties of state captured

- Captures a consistent state
- Doesn't necessarily tell us where the system exactly is in its execution at this specific point of time
- Observed global state is a permutation of the actual state or one of the other global states in this tree.

What does the last point mean? If we draw a tree of possible states of the system (i.e. start at initial state , then choose who sends the first message , etc. basically have multiple child branches for different concurrent next events), the algorithm results in one of the possible states in this tree.

If we execute the algorithm at one point in time, we can capture different sequences of states: E.g. we execute at S21, we can capture either (a) S10, S11, S21 for run that consists of events e11, e21, e12, or (b) S01, S11, S21 for run that consists of events e21, e11, e12 Obviously, both end in S21 and both correspond to consistent narratives of sequence of events so far.

4.4 Global State

4.4.1 Formal definition

If state recorded is S^*, sequence of computations done so far is seq, and the true initial and final states of the system are S_i and S_j.

- Recorded state S^* must be on *some* path from S_i to S_j. In other words, S^* is reachable from S_i and S_j is reachable from S^*
- There exists *some* sequence seq^*, which is a permutation of seq which we can take from S_i to S_j passing through S^*
- In this sequence, we know that S^* can be either the initial event or it happened after the initial event S_i i.e. $S^* = S_i$ or $S_i \rightarrow S^*$
- Similarly, S^* can either be the final event or it happened before the final event S_i i.e. $S^* = S_j$ or $S^* \rightarrow S_j$

Theorem: If we follow the Chandy-Lamport's algorithm, for an execution of the system (seq^*), which takes the system from an initial state (S_i) to a termination state (S_j), the recorded state (S^*) is such that it is reachable from the initial state and the termination state is reachable from the recorded state.

Again, recorded state need not be a real state that the system actually went through!

4.4.2 Benefits of Global State

The algorithm allows **Stable property detection** which allows us to do better garbage collection, and **Unstable property detection**, which can help detect correctness/consistency issues.

Stable property is a property of the system which once it becomes true, it remains true for the remainder of the execution of the system. E.g. Deadlock, Termination, Token Loss.

- If we know that some phase completed by S^*, then it definitely completed before the final state S_j, so we can do gc for it now.
- If a stable property is true in S^*, then it is definitely true in S_j (since it is valid for te rest of the execution).

- If a stable property is false in S^*, then it is definitely false in S_i (because if ti were true in S_i, it would be true for the rest of the execution).

Unstable property is one for which there is no guarantee that once it becomes true it remains true for forever. E.g. Buffer Overflow, Race condition, load spike, etc.

Since S^* might not have really occurred, if an unstable property is true in S^* it need not be true in the actual system or in the final state S_j (sine unstable properties are transient). If we observe an unstable property to be true in S^*, which is a *possible* state, then we know that this property *can* occur in the system for some valid sequence of actions (which is obviously a bad thing!) - like bug detection!

To summarize,

- For stable property y, if

$$y(S^*) = \text{ true } \implies y(S_j) = \text{ true}$$

- For unstable property y, if

$$y(S^*) = \text{ true } \implies y(S_j) \text{ could possibly be true}$$

Consensus in Distributed Systems

> *Science is really about individual experts reaching a consensus.*
>
> — Alan Stern, *Interview about Pluto's planet status*

5.1 What is Consensus?

Consensus is the ability of multiple distributed processes to reach agreement about something -maybe the value of some shared state of the system, some action to be taken or even the current timestamp, etc.

Why do we need it? Critical for making forward progress in a distributed system. Common scenario: Nodes need to agree upon the outcome of a transaction. E.g. In Project3, first process executes a command, responds to the client, then sends a backup request and dies. If the backup request is lost, the backup server never executes this and all future clients will see a different result i.e. one indicating this command was never executed. **Reaching a consensus makes it possible for the system to be correct.**

Challenges in reaching a consensus (Discussed earlier):

- Non-determinism
- Lack of global clock
- Network delays
- Failures
- Malicious behavior

The ability of a system to reach consensus implies 3 key properties:

- **Termination/Liveness**: Guarantee of forward progress i.e. process should either move forward to make a consensus or terminate i.e. the non-faulty processes eventually decide on a value.
- **Correctness/Agreement/Safety**: Guarantee that all processes decide on a single value (I mean, if we are calling it a *consensus*...)
- **Validity/Safety**: The value decided on must have been proposed by some of the processes (like, it can't be some arbitrary value pulled out of thin air!)

5.2 Theoretical Posibility of Consensus

5.2.1 System Model

We will consider the theoretical possibility of achieving consensus by studying the following model:

- **Asynchronous**: Messages may be reordered and delayed but not corrupted.
- **At-most one faulty process**
- **Failstop Model**: where the node simply stops working.
  - ▶ Indistinguishable from infinite message delay in the system, so we can ignore actual failure type/cause

Obviously, these are simplifying assumptions and real systems are more complex (and often break these assumptions).

Why is this model still useful :

- **If we prove possibility of consensus**: Can try to extend result to real complex systems through further investigation.
- **If we prove impossibility**: If impossible even in the simple model, obviously not possible for a complex model.

Some relevant terms :

- **Run**: Ordering of events in the system
- **Admissible Run**: Run with one faulty processor and all messages eventually delivered i.e. similar to the model defined above.
- **Deciding Run**: Run where some non-faulty processors reach a decision.
- **Totally correct consensus protocol**: Protocol where all admissible runs are also deciding runs i.e. for any ordering of messages where the model assumptions are met, the non-faulty processors will be able to reach a decision.

- **Univalent configuration**: System configuration in which the system can reach a single decision (single value). Obviously, this would be part of a deciding run.
- **Bivalent configuration** System configuration in which multiple (or at least 2) decisions are possible. This means, a consensus hasn't yet been reached, so this cannot be part of a deciding run.

5.2.2 FLP Theorem

Presented in the paper: "Impossibility of distributed consensus with one faulty process" by Michael Fischer, Nancy Lynch and Michael Patterson (F.L.P)

Theorem 5.1: FLP Theorem

FLP Theorem states that in a system with one fault, no consensus protocol can be totally correct.

Proof

Intuition : Under the model assumptions, if it is possible to identify a starting configuration and an admissible run where the system does NOT reach a deciding state (a.k.a ends up in a bivalent configuration) $\implies$ consensus cannot be reached in at least one configuration i.e. protocol is not *totally* correct.

0. Consider a system where nodes are capable of making one of two decisions: 0 or 1.
   - ▶ Assumptions from before hold: 1 faulty processor, messages are not corrupted, etc.
1. (Lemma 2 of paper) In a distributed system, there is at least one initial configuration where the final decision is not known already (i.e. where result is not known beforehand) $\implies$ there is at least one initial bivalent configuration.
2. There must be a single event or single message (whose delivery) will convert the bivalent system into a univalent system.
3. It is possible to delay the delivery of the message in point (2) so that it is never delivered and thus the system never transitions to a univalent state! $\implies$ An admissible non-deciding schedule does exist for this system!!

From these points, we can conclude that- for a system with 1 faulty node where messages can be delayed or reordered, there is always an initial bivalent state for which an admissible non-deciding schedule exists!

Deriving lemma 2 : Initial configuration depends on schedule of events- some might have a predetermined solution. But don't have to consider these because the definition of consensus is to choose a proposed (not predetermined) value.

- If we list all the configurations with their initial states, then any 2 configurations differ in the state of at least one process.
- If there is a fault in that differing process, the remaining processes form a bivalent configuration
  - The admissible run in both these configurations consist of the same values (i.e. initial states of the non-faulty processes).
  - Final state can have 2 different values (since initial state of faulty process is unknown or can't be deduced)
  - $\implies$ 2 admissible runs that can result in different states for the faulty process, so this is a non-deciding configuration.

Example: If config_0 refers to "p_0 has 0 as initial state, p_1 has 0 as inital state", while config_1 refers to "p_0 has 0 as initial state, p_1 has 1 as inital state", the two configurations differ in the initial state of p_1. So if p_1 is the faulty process, then we have 2 admissible runs, both has p_0 ith 0 as initial state, but they cannot decide what the initial state of p_1 is accurately, thus leading to a bivalent configuration.

5.2.3 Is Consensus Really Impossible?

Given that failures are inevitable and networks have delays, this is our best system model. So, can we never have a correct distributed system?

Don't give up yet!

Theorem above is based on certain assumptions i.e. Any system that satisfies these assumptions cannot have consensus. But, real world systems that change these asusmptions or system properties can!

This also means that the consensus protocols we shall see will not terminate if the specific conditions are not met (i.e. the assumptions are not broken.)

Consensus Protocols

6.1 Goals of Consensus Protocols

Continuing our discussion of consensus from the last chapter, goals of consensus protocols are:

- **Agreement**: Multiple parties should agree upon the same value
- **Validity**: Chosen value must be valid.
- **Safety**: The term **safety** refers to:
  - Only a value that was proposed is chosen
  - Only a single value is chosen and only a single chosen value is learnt
- **Liveness**: The term **liveness** refers to:
  - *Some* proposed value is chosen
  - *Any* chosen value is learnt

Liveness is related to forward progress i.e. the system should not be stuck in the process of choosing. Safety is related to correctness i.e. the system should choose correctly and consistently. From the Definition FLP Theorem, we know that we cannot have both safety and liveness.

3 types of nodes in a consensus protocol :

- **Proposers**: Nodes that propose a new value for the shared state
- **Acceptors**: Nodes that evaluate the proposals for the value and choose, often based on some kind of ordering (like timestamps)
- **Learners**: Nodes that need to access (read) the value.

6.2 2-Phase Commit (2PC)

2-Phase Commit(2PC) originated from database community.

- There is always a co-ordinator, who is assumed to not fail
- Phase 1- **Vote Collection Phase**: Co-ordinator proposes value, participants vote
- Phase 2- **Decision Phase**: Co-ordinator tallies the votes, makes the decision and communicates the decision (commit).
- **Blocking** protocol: blocks if there is a failure, $\implies$ does NOT guarantee liveness

6.3 3-Phase Commit (3PC)

3-Phase Commit(3PC) also originated from database community.

- Tries to solve the blocking problem above.
- Phase 1- **Prepare phase**: Votes are solicited
- Phase 2- **Pre-Commit Phase**: Decision communicated (might lock resources here)
- Phase 3- **Decision/Commit Phase**: Perform actual commit here
- **Non-blocking**: Timeout during phase 2 or 3 causes protocol to abort.
- Problem: Only works with fail-stop model- assumes that a failed node won't restart (or a timed-out message won't be eventually delivered)
- This means there will be **safety** issues on fail-restart.

6.4 Paxos

Fun Fact: **Original Paxos Paper**

The original Paxos paper was written by Leslie Lamport in 1990. However, the original paper was not accepted by the publication (*a lot of people who don't get humor on this plant, smh!* ☹)

Anyway the original paper used the metaphor to describe the following problem:

- Describes Paxos Parliament has members who pass decrees

- The members only work part-time
- They communicate by messages that may be delayed (or someone may not vote)
- Nobody is malicious

The paper then provided an algorithm with a set of rules they must follow to agree on a single decree (and multiple decrees i.e. Muti-Paxos)

Essentially, the paper described the consensus problem, provided a solution-an algorithm that is a state machine of rules/state transitions that must be followed to achieve consensus, and proof of correctness of the algorithm.

6.4.1 Basics of Paxos

This sections describes the workings of the **Paxos** consensus protocol.

Assumptions/System Model : Systems with asynchronous communication, non-Byzantine failures

- Agents operate at arbitrary speed
- Agents can have fail-stop failures (stop and restart)
- Agents have some persistent memory to recover information on restart
- Messages can take arbitrary times to reach
- Messages can be duplicated, lost or reordered
- Messages cannot be corrupted

Important underlying ideas :

- **State Machine Replication** Each node is a replica:
  - of the same state machine (or algorithm) that updates the state
  - following the same rules to update the state
- **Majority Quorum**: All decisions based on majority quorum
  - Each decision is based on majority quorum, so any 2 decisions will have some common members who agreed to it → even when some nodes fail, possible to disseminate the consensus decision
  - If some nodes fail and don't participate in a consensus round, when they restart, they will join a quorum that has at least 1 participant who knows about the past decisions

  - Helps tolerate fail-stop and fail-restart failures

- **Ordering**: Order among messages maintained using timestamps (so we can tolerate arbitrary message delays and message reordering)

6.4.2 Phases of Paxos

Protocol has 3 phases:

1. **Prepare Phase**: Node that initiates proposes an agreement round.
    - Proposal message timestamped with order number → Allows proposals to be distinguished/ordered
2. **Accept Phase**: Gather votes on whether agreement is possible and the value is agreed upon
    - Initiator gathers responses from the participants
    - Responses indicate whether they agree to participate, what value they agree to and timestamp of proposal being agreed to
3. **Learn Phase**: Agreed upon value learned by all
    - Starts once leading node gets enough nodes to agree to commit (quorum)
    - Initiator communicates the agreement to all participants with agreement round and agreed value
    - If initiator receives Acks from quorum of participants, it knows this round is complete

Note:

- Possible to learn in **2 rounds**
- Prepare and Accept form the write phase of the value; learn is the read phase

Why use proposal number :

- Proposal number being part of messages - ensures ordering
- Helps with dealing with fail-restart and delayed messages
- There may be multiple ongoing proposals at a time, helps to keep track of messages

Prepare Phase

0. Each round started by an initiator/leader/proposer
1. Proposer selects proposal number n and sends **prepare request** (with number

n) to acceptors

- n is totally ordered among all processes i.e. unique across all proposals and all processes
- ▶ No two processes can have same n
- ▶ Same process can never have 2 proposals with same n

2. The acceptors who receive the message respond with an **acceptor response**
    - If an acceptor receives a prepare request with n greater than any received so far, responds with promise not to accept any more proposals with value $< n$
    - If acceptor already received a proposal with a higher n (maybe initiator did not hear about that proposl number), it responds to prepare request with the higher value of n so initiator can update the proposal number

Note that:

1. There may be learner nodes that will eventually **read** the value of this variable but do not participate in the agreement process
2. Proposer node is simply one of the acceptor nodes (it is not necessarily a separate node).

▶ IMPORTANT: Correctness in the case of failure scenarios is achieved through majority quorum for each step

Accept Phase

1. If a proposer hears back from majority of acceptors, it sends an **accept request** or **commit message** to each of them
    - Accept request contins value of proposal number n and proposed value v
    - v is the value of highest numbered proposal among the responses in prepare phase
    - If there were no values proposed, proposer chooses a value
2. If an acceptor receives an accept request for proposal n, it accepts the proposal, unless it has already responded to a higher number prepare request

Learn Phase

- Acceptor receiving a commit message implies the proposed value was accepted (or) Accepted value now becomes the Decided value.
- Decided value can be communicated to learners (or) learner read requests will be responded with decided value

- ▶ Learner also needs to get a majority quorum of responses from acceptors (since acceptor nodes themselves can fail)
- This means every acceptor has to send the decided value to the learner → obviously inefficient!
- Learners, in turn, might notify waiting clients, etc.

Distinguished Learners (*No, not you. Although, you* are *a distinguished learner too!* ☺)

- Choose a distinguished learner that receives accepted proposals from acceptors
- Once it receives proposals from a majority of acceptors, it informs the other learners
- Alternately, learners send read requests to the distinguish learner whenever they need (instead of being informed each time)
- ▶ This is a common idea in distributed systems: To separate the read nodes from write nodes
  - Ensures that the same nodes don't become bottleneck
  - Same nodes serving both end up slowing both reads and writes

Going one step further, can have multiple distinguished learners (at the cost of more communication)

How to deal with concurrent proposals : What if 2 nodes initiate a proposal at the same time

- Acceptors that receive proposal with lower number than the one they accepted will ignore it.
- ▶ Irrespective of who initiates the proposal first, the proposal with higher number will be served
  - If proposer 1 initiates proposal 1 first, but acceptors receive proposal 2 from proposer 2 earlier, they would have already responded ("accepted") proposal 2, so now proposal 1 is ignored
- Proposer 1 will eventually try later with a greater proposal number
- Ignoring lower proposal numbers ensures poposal 2 can actually get completed without interruption
- ▶ Once nodes agree on a value, this value should persist
  - If proposer 1 now re-initiates with poposal number 3:

  - Acceptors respond with the older agreed upon value and accepted proposal number
  - Proposer 1 will now commit the new proposal 3 with the already accepted value (from proposal 2)

- Important Note: iF proposal 3 from proposer 1 arrives before proposal 2 was committed, the proposal 2 would be discarded and newer proposal accepted instead.

6.4.3 Paxos vs. FLP

Paxos does NOT violate FLP because it is possible to reach consensus but does not guarantee forward progress (Safety but not Liveness)

- Possible liveness problem- if 2 proposers keep issuing proposals with increasing numbers before the previous one is accepted
  - In each case, no decision is reached and the previous proposal is discarded in favor of the new one
- Common workarounds:
  - Proposers use random delays before retrying with new value (so they don't always cancel each other)
  - Designate a "distinguished proposer" or leader
    - Need timeouts to deal with leader failure
- With these workarounds, liveness problem is **very unlikely, but not impossible!**. $\implies$ FLP still holds!

6.4.4 Paxos in Practice

Multi-Paxos

In real systems, obviously there is a series of values to be agreed upon. This is achieved through Multi-Paxos. (Part of the original protocol)

- Each simple single-decree Paxos is used to agree on an individual value
  - E.g. In Google Chubby, Paxos is used to agree on the ID of a node
- Multiple Paxos protocols executed to agree on values and order of operations

Problem: Protocol gets too complicated with too many in-flight exchanges.

Paxos Optimization

Also known as **ViewStamp Replication**

- Only one proposer allowed to submit winning proposals: Leader for the current "view"
    - ▶ Leader only valid for the current group of participants
    - – If nodes join or leave, need to elect new leader
- All values in system accepted/learned in the order that the leader proposes them i.e. leader is now responsible for the other nodes

▶ Obviously, important to accurately detect when "view" or leader needs to be changed.

Popular Implementations

The following are some implementations of Paxos - not proven to be equivalent (some are known to differ)

- DEC SRC (Known by Leslie Lamport himself)
- Google Chubby
- Zookeeper (Open-Source implemented by Yahoo)

6.5 RAFT

Why use another consensus protocol if Paxos already works? (*I mean, are we simply trying to ensure students have more to learn for the mid-term?*)

Real reason is: **Understandability** i.e. Is the protocol sufficiently simple for practical implementations can be written based on the specification? The specification is the one proven to work. Obviously, if an implementation does not follow the specification, the guarantees provided by Paxos won't work.

Additionally, is the implementation still correct after optimizations are added? Since even a single round is complex, Multi-Paxos is a lot more so

Solution: RAFT - Proposed by Ongaro and Ousterhout. Shown to be easier to understand by college students (*Isn't that what we* really *need?* 😜)

6.5.1 Phases of RAFT

1. Leader Election Phase: Any node can be a leader, but the one that receives most votes is selected to be one.
2. Log Replication Phase: Once leader elected, normal operation: Leader makes proposals for updates

▶ Every new leader election starts a new **term** (similar to "view" above.)

- Leader can be active for an arbitrary duration (or number of updates)
- Separating the leader election phase is good beacause:
  - Makes it easier to reason about the implementation of the system
  - Makes it easier to figure out which proposals should be the winning proposals.

Leader Election

1. All acceptor nodes exchange heartbeat messages wit leader
2. If any node doesn't receive heartbeat message, assume leader is dead (*Et tu Brute!*) → Start a new leader election phase
3. Servers applying to be candidates send messages with current term and index of most recent event in log
4. Message sent to all nodes, all nodes vote on who is the leader

Election satisfies the **Election Safety** property i.e. there can be at most one leader in any term. This is achieved by following these rules:

- **Rule 1**: Leader is elected by majority vote
  - New leader chosen implies start of new term
- **Rule 2**: Prevent outdated leaders from being elected
  - Servers only vote for a candidate that has a newer log (higher term number or same term number and longer log)
  - Losing candidates know they are outdated, but can also update their logs now
- **Rule 3**: If there is a tie,
  - If no candidate gets majority vote or there is a network partition
  - After a random delay, restart new election (random delay means not all leader candidates restart election simultaneously, so someone is likely to win)

Log Replication

0. Each node maintains a log of entries: Entry contains the operation performed and term number
1. Leader accepts requests for updates from clients → appends them to its log → pushes them to the logs of folowers
   - Leader node in an example is the one with the longest log (since that's

   how election works)

   - Followers may be up-to-date or might have fallen behind (if a node failed and restarted or lost network connection for sometime)
   - Can detect if a node is lagging behind when the node wakes up again (since the logs are numbered) → get log entries from leaders → replicate them

2. Leader pushes new log entry (+previous log entry) to folloers during heartbeat
3. Followers check if they have the previous log (implies they are up-to-speed), and send ack if they do
4. If leader receives majority quorum of acks, it can commit the log entry (and ack to client, if needed)
   - The operations are exchanged during heartbeat, so outdated followers can catch up

6.5.2 RAFT Correctness

Correctness : RAFT guarantees correctness (during log replication) through the following properties:

- **Leader Append-Only**: Log at the leader is append-only
- **Log Matching**: Since there is an append-only log at the leader, logs at any 2 nodes can be compared
  - ▶ If logs have the same index and term number, the current entry and all previous entries in the 2 logs are the same
  - – Logs being ordered helps to choose next value for the consensus protocol.
- Dealing with **inconsistency**: It is possible for a leader to die with some uncommitted portions in its log. But this is ok because:
  - – New leader forces all followers to use its log (Election strategy ensures new leader knows all about the committed logs)
    - ∗ Commit strategy ensures that majority of acceptors would have accepted the logs before it was committed.
    - ∗ New leader election guarantees that it is the one among them which knows the most committed logs!
  - – Any uncommitted logs that the new leader is unaware of may be discarded (clients will try again!)
  - – If new leader has uncommitted logs from older terms, it can push those to the followers! (If a client waits for a long time, the request *may* get committed!)

Safety : RAFT guarantees safety through the following properties :

- **Leader Completeness**: Once committed, log entry won't be overwritten.
- **State Machine Safety**: Once a log entry is applied in a node, no other node will apply a different log entry in the same slot.

In sum, RAFT guarantees nodes will choose a single value and the value will be agreed upon by all nodes through these properties:

1. Election Safety
2. Leader Append-Only
3. Log Matching
4. Leader Completeness
5. State Machine Safety

6.5.3 RAFT in Practice

Dealing with implementation difficulties :

- Problem: **Long log**
  - Logs may become very long simply due to duration of execution
  - Node may have fallen much behind everybody else (*like I am in this class. Jk. Jk.*) → Might end up having a long log to catch up
- Solution: **Garbage Collection** = Snapshot + Log Truncation
  - Periodically take snapshots (then we can discard older log entries)
  - When a node needs to recover, leader can send the snapshot directly during heartbeat instead of the entire log
    - ▶ May have to send a lot of data, but allows node to restart quicker

Like Paxos, RAFT is implemented in multiple real world systems.

Replication

7.1 What is Replication

7.1.1 Goals

Main goal is - Multiple systems providing the same service. To do this : Maintain same state at more than one location. - state can be files, chunks of files (file systems), databases (database server), application-level or OS-level state.

Advantages of replication:

- Fault Tolerance: If one node fails , the replica can provide the service instead.
- Availability/Disaster management- Can continue service during natural disasters by storing data in faraway server.
- Scalability: As load increases, one server might become bottleneck. Replica can help balance load.

7.1.2 Replication Models

Two main replication models:

- **Active Replication**: Each node is active and can accept requests.
  - For writes, must ensure the other nodes also completed operation
- **Stand-by Replication/Primary-Backup**: Only one replica is active at a time and accepts requests
  - Other replicas kept consistent so they can takeover when the active one fails

7.1.3 Replication Techniques

Two main techniques used to implement replication - choose either depending on the specific application scenario:

- **State Replication**: Execute operations on one replica, then communicate state updates i.e. **data**) to the others.
  - Advantage: No need to execute multiple times
  - Disadvantage:
    - State might be very large to copy over
    - Might be hard to figure out which parts of the state got updated by the operation.
- **Replicated State Machine**: Execute each operation on all replicas i.e. communicate **operations/log** to the others.
  - Works when the operation is deterministic i.e. both executions produce the same result
  - Advantage: Operations are much smallr to transmit than data
  - Disadvantage: Need to ensure both produce same result and are deterministic.

7.1.4 Replication and Consensus

Obviously, in all methods above, need to ensure the replication happened correctly i.e. whatever is communicated (data or log) must be present in all replicas and reflect the same value => replicas need to reach a consensus on what the final value is! Can use some consensus protocol for this:

- Primary-Backup: Primary can be the leader
- Active Replication: Each replica can be leader for its own operations
  - Not necessary though. Can assign specific replica to be the leader always and simplify design.

Factors affecting ordering/consistency of updates :

- Consistency model used
- Granularity of updates: Individual operations updated or transactions?

Cost of consensus : Proportional to number of nodes

- For 1 primary, 1 replica - need minimum 2 rounds of messages for consensus, say Round Trip time = T
- More replicas, O(N) slowdown. Need RTT >2T because:
  - Primary needs to wait for more responses

  - Each replica needs to send/receive more messages
  - More nodes are now executing the operation

This means performance doesn't scale as well.

7.1.5 How to choose replication method

- **Workload**: Read-intensive or write-intensive? Distribution of reads and writes over time? Are they issued to change shared state or isolated in different parts of the system state.
- **System Configuration**: Number of nodes, Network properties, failure rates, etc.
- **Consistency Requirements**

7.2 Chain Replication

Definition 7.1: **Chain Replication**

Consider a scenario with 3 replicas: R_1, R_2, R_3. In each "chain", the first replica (say, R_1) is called *head* and last replica (say, R_3) is called *tail*.

- Write Requests always sent to the head replica
- Head replica replicates write requests only to the *next* replica in the chain.
- Each replica propagates the write request to next replica in the chain.
- Tail acknowledges write by sending Ack to the head replica.
- Read requests always sent to the tail replica

Why is this a good idea:

- Still perform the same number of writes, but R_1 only handles one set of replication requests, so no longer as much of a bottleneck.
- Read requests served by tail, so guaranteed to see latest committed data
  - If read served by intermediate node, might see a write result which is not yet completed on a later replica in the chain or tail node. If any of these fail, the write will be discarded as incomplete, but now a client saw that data!

7.2.1 Pros and Cons

Pros:

- **Leader Scalability**: Adding replicas doesn't affect number of messages seen by leader.
- **Higher Write Throughput**: Fewer messages processed.
  - Can use **pipelining**- write can be processed by one replica while it is being sent to the next.
- **Strong Consistency possible**: Reads are guaranteed to return only successfully committed writes

Cons:

- Cannot use for **read-intensive workloads**
- **Inefficient**: Obviously intermiediate nodes underutilized

7.3 CRAQ

Definition 7.2: **CRAQ**

Chain Replication with Apportioned Queries or CRAQ tries to improve chain replication to be used for read-heavy workloads:

- **Chain Replication**: Similar replication pattern, with writes still handled by head replica
- **Apportioned**: Reads divided among the different replicas
- **Queries**: Refers to reads

How can this method maintain correctness?

- When no write in progress, has single copy of data - any node can respond to read requests
- When a write is completed at a replica: Maintain both old and new data at each replica.
  - Can respond with old value when 2 values present: Until write is Ack'ed assume that write is incomplete.
  - Can check with the tail: if tail responds with new value, then write is considered committed.

- Once write is acknowledged, discard old value and consider new value as committed.

7.3.1 CRAQ Performance comparison with Chain Replication

Experimental Setup :

- Compare read throughput since CRAQ was developed to improve that over chain replication
- Use either 3 or 7 replicas for CRAQ
- Issues 0 to 100 writes/s at head
- Obviously, reads are sent to tail in chain replication, and distributed in CRAQ
- Plot reads/s against writes/s

Results : CRAQ consistently delivers better read throughput

- With 3 replicas:
  - Upto 3x read throughput at low write loads
  - About 2x read throughputs even at high write loads
- With 7 replicas:
  - Upto 7x read throughput at low write loads
  - Approx >5x read throughputs even at high write loads
- As write throughput increases, CRAQ replicas have to maintain 2 copies, check with tail etc., which is why read throughput drops as writes increases

Fault Tolerance

> *And oftentimes excusing of a fault*
> *Doth make the fault the worse by the excuse*
>
> — William Shakespeare, *King John*

8.1 Basics of Failures

From fault to Failure :

1. A failure first starts with a **fault**.
    - Fault can be in hardware or software.
    - System may function correctly even with fault until the fault is **activated**.
2. Activated fault leads to an **error**
3. Error propagates through the system as it executes and leads to a **failure**

Types of faults :

- **Transient**: Manifest once then disappear
- **Intermittent**: Manifest occassionally
- **Permanent**: Once activated, the fault persists until it is removed

Types of failures :

- **Fail-Stop**: One or more components stop working
- **Omission**: Some actions are missing i.e. Components fail to send or receive some messages

- **Timing**: Affected system components may not meet timing requirements (i.e. this might cause delays).
  - Can lead to failures if system relies on retry or reconfiguration on timeout.
- **Byzantine**: Arbitrary failures i.e. system continues to function but produces incorrect results
  - Can be because of malicious nodes or some software bugs, etc.

8.1.1 How to deal with failures

:

- **Avoidance**: Idealy, avoid all failures by detecting early (before failure happens) and taking corretcive action.
  - Problem: Tends to be too expensive
  - Also prediction may be hard - so this is impractical
- **Detection** :Detect that a failure *has* occurred:
  - Common method: Heartbeat mechanism (or ping) to check if nodes are responsive.
  - Heartbeat can detect fail-stop but not Byzantine (i.e. if a node is behaving incorrectly)
  - Error Correction Codes: Can be used to detect incorrect execution.
- **Removal**: Once detected, we would want to remove the root cause of failure and revert system to the last clean state
  - Rollback: Take system to a point before the fault manifested. (If fault is transient, maybe we won't hit it again)
  - Rollback may not be possible if there are some external actions by the system
- **Recovery**: System should be able to recover from failure, i.e. detect, remove root cause, revert to last clean state and resume clean execution.
  - A system that can do these is considered fault-tolerant

8.2 Rollback-Recovery

Basic idea : When a fault is detected, rollback to a previous state (where system was known to be correct), then re-execute with fault removed. Rollback includes:

- Rolling back effect of any messages transmitted after fault was detected

- Rollback any updates made to system state

How to implement this :

- What state to roll back to?
  - From Consistent Cuts, we know system has to roll back to a "consistent" state (from a time before the fault occurred.)
  - ▶ System need not roll back to an actual state that ever happened (consistent state doesn't mean it was a real state, just a state that is consistent with the operations in the system)
- How far back should we take the consistent cut from? (There may be many consistent cut points before the current point, which one to choose?)
  - Try to find by progressively rolling back changes in the system to find consistent cuts
  - ▶ This can potentially rollback to the beginning and all data can be lost!!
  - Instead choose one of the 2 methods: (See section below)
    - Checkpointing
    - Logging

Granularity of operation may vary:

- **Transparent/Full-System**: Do not require any application level modification i.e. transparent to the application
  - Rollback-Recovery system needs to track every individual message send/receive (and their ordering), every state update (and its success/ordering).
  - Obviously, large overhead.
- **Transaction-level**: Application modified to use transactional APIs (system will ensure transactions are executed atomically).
  - Executing distributed transactions atomically is a whole other problem (see Chapter 9)
- **Application-specific**: System-level might track too many things. Application has better insight into when/what needs to be saved/checkpointed.
  - Example, in HPC domain, massive number of operations/state updates.
  - Only useful when the performance degradation caused by saving too much state is a real concern. Typically, this might be overkill.

The future sections discuss transparent systems, but the discussion can be generalized to transactions or even application-initiated checkpoints.

Definition 8.1: Checkpointing

Checkpointing is the process of saving the application state periodically so that the system can revert to the checkpoint on failure.

- During normal operation: Periodically ave state of application/node -> Flush checkpoint to disk or persistent storage
- On failure: do any repair activities -> restore checkpoint from persistent storage -> restart system
  - On hardware failure- might repair/replace parts (or) simply start a different node to replace failed node.

- Advantage: Can restart instantaneously after restore.
- Disadvantage: At each checkpoint, lot of I/O to save full system state. (this is where application specific)
  - Can reduce with application-specific checkpoint to save only necessary data
  - Can track delta changes since the last checkpoint and only save those

Definition 8.2: Logging

Logging is the capturing of information about operations performed so that the operations can be repeated to redeem the latest state on failure.

- Basic idea is to log information about operations performed i.e. change in different state variables
- 2 styles possible:
  - **UNDO**: Store original value of changed variables - can use if we want to "undo" operations one by one
  - **REDO**: Store new value of changed variables - systems is "rolled back" to original application state, then "redo" operations.
- Obviously, log has to be written into persistent storage

- Advantages:
  - Not storing whole system state, so lesser I/O to persistent storage (not much storage required)
  - I/O time happens while application executing, so good to have lesser I/O
- Disadvantages:

  - Recovery takes much longer
  - With REDO log, even regular application operations expensive - need to look through log to find most recent value of dependent parameters

Combining logging and checkpointing

- System can periodically perform checkpoint
- Between checkpoints, use logging to save updates (earlier logs can be discarded).
  - Message send/receive also considered updates here.
- Advantages:
  - Limit duration of recovery i.e. System doesn't have to go back to beginning, just needs to return to most recent checkpoint with consistent cut.
  - Limit space and bandwidth usage
- Disadvantage: Need to detect a checkpoint that is a stable consistent cut.

8.3 Checkpointing

System model for the next few sections:

- Fixed number of processors that may interact with outside world
- Processors communicate among each other only via messages
- Network is non-paritionable, but other assumptions vary (FIFO or not? Reliable communication or not? Remember we can achieve these using TCP)
- Number of tolerated failures may vary depending on the protocol

8.3.1 Uncoordinated Checkpointing

- Processes take checkpoints independently (See figure)
- On failure, the recovery line needs to be computed and process reverts to it.
  - If the failing process rolls back to a particular checkpoint, others have to roll back to consistent checkpoints
    - ∗ E.g. if P3 rolls back to a time before was m3 sent from P2, P2 should roll back to a point before the send message m3 event. This process is continued till all processes are consistent.
  - To achieve this, need to store dependency information (or track which messages were sent, so we can decide if we found a valid recovery line or not)

- Disadvantages:
  - **Domino Effect**: Since the checkpoints are uncoordinated, when we roll back to a point for P3, it may force us to go to an earlier point for P2, which itself might force us to move to an earlier point for P1, etc. i.e. we might rollback a lot more than we need and maybe even to the beginning!
  - **Useless checkpoints**: (like in the example above where many had to be discarded) Many checkpoints are useless as they can never be part of a globally consistent state
  - **Multiple checkpoints per process**: Since we need to keep rolling back till we find a consistent state, need to store multiple checkpoints for each process
    - ▶ Particularly bad since many of these checkpoints might also be useless - so extra storage AND wasteful storage.
  - **Garbage Collection**: To clean up these extra (and maybe useless) checkpoints, need to run gc - but this may be complex and time-consuming.

8.3.2 Coordinated Checkpointing

- Processes co-ordinate when they take the checkpoint so that the checkpoint results in a consistent state
- Advantages:
  - No longer need a dependency graph to calculate the recovery line: most recent checkpoint is valid!
  - No domino effect: All checkpoints taken are relevant checkpoints!
  - This means, need only one checkpoint per process
  - No garbage collection needed
- Disadvantages: The co-ordination itself
  - **Delay in initiator message** E.g. If the initiator message is received in P3 after message m_3 arrived, but in P2 before message m_3 was sent?
    - ▶ Synchronous system: Can simply take checkpoints every T units of time.
    - ▶ If message delivery is reliable and bounded: can come up with a round-robin sort-of scheme
  - **Unnecessary checkpoints**: Nodes may be forced to take checkpoints even though no changes since the previous checkpoint on that node!

8.3.3 Communication-Induced Checkpoints

Use a consensus protocol to decide whether to take the checkpoint now or not.

Different approaches possible:

- Blocking approach: Nodes do not process any other message while this is going on
  - Initiator starts a 2PC or other protocol to start the consensus process
- Non-blocking approach: Global snapshot Algorithm
  - Relies on special marker messages, needs the network to be FIFO
  - To avoid FIFO, piggyback marker message on a regular message
  - To capture state of nodes not communicating with anyone else, periodic independent checkpoints are taken
  - If node receives message with marker, first take snapshot, then process actual message. (All nodes will snapshot before this message was sent/received).

8.4 Logging

Logging saves storage but needs more complex recovery (i.e. saves i/o, needs more compute!) Again, need logs to capture information such that a consistent cut can be obtained. Should not lead to **orphaned events** i.e. receive is in the log, but send isn't.

Different approaches to achieve this:

- **Pessimistic Logging**: Each process logs *everything* to storage before events propagated. (Basically send message is logged before actually sending)
  - Obviously high overhead- write to persistent memory is slow- plus this is in critical path. Can improve slightly by using faster persistent memories.
- **Optimistic Logging**: Assume that the log will be always persistent before the failure + allow effects to be reversed
  - This assumption is hard to meet- need to track dependencies
  - Have to identify incomplete operations and remove their effects during recovery
  - Any operations that have external effects (e.g. robot movement), the operation has to be delayed until system can capture the information (and ensure this operation will not have to be reversed later.)

- **Causality Tracking**: This is what we really need
  - Operate optimistically when there are no dependencies
  - Capture dependencies so that causally related events are deterministically recorded
  - No risk of delaying external events indefinitely (causality tracking itself depends on some message exchange- once the causality messages are all delivered, safe to execute)

8.5 Which Method to Use?

Right choice depends on multiple factors:

- **Workload type**: How often data updated, size of updates, are most updates shared data, is fast recovery important etc.
- **Failure types**: Types of possible failures and how they can be recovered
- **System configuration**: Cost/overhead of communication vs storage, system scale, etc. (These may change over time)

Changes from when the paper was written (see table from paper):

- Pessimistic logging considered inefficient because writes to persistent memory are slow, but this is becoming faster with time
- Coordinated checkpointing (default for HPC)- favorable for most features, but delays operations - requires system-wide coordination. This might become too much overhead depending on the application (or network costs in the future)

Distributed Transactions

> *Those Are Not Transactions (Cassandra 2.0)*
> *- Dave Scherer, Blog*
>
> — Martin Kleppmann, *Blog, Designing Data-Intensive Applications*

9.1 Transactions and Distributed Transactions

A **transaction** is a group of operations that need to be applied together in an indivisible manner. Transactions are usually expected to be done with **ACID** properties i.e. Atomicity, Consistency, Isolation and Durability.

- **Atomicity**: Either all the operations in the transaction are applied or none of them.
  - Cannot execute one transaction with inputs that are results of a partially applied, different, transaction.
- **Consistency**: Described differently in different contexts. Typically, later transactions should see the effects of other transactions committed in the past.
- **Isolation**: Concurrent transactions leave the database in a state as if they could be obtained if those transactions were executed in some order (Similar to serializability)
- **Durability**: Once a transaction is committed, it stays committed even after system failure. (Should write to persistent storage)

Example: Consider a transaction Tx defined by 2 operations A and B that update variables a and b. Assume 2 clients executing this transaction as C_1 and C_2 on the database.

- It is ok to have both C_1 and C_2 completed or neither.
- It is not ok to have situations where C1 completed A but not B. C_2 cannot see

a value of a from C_1 that was the result of A, without seeing the updated value of b (i.e. result of operation B).

Thus, only 2 possible outputs for a transaction: either it is committed (permanently visible) or aborted (all intermediate updates are lost).

Transactions are useful for:

- **Concurrency control**: If they complete atomically in isolation, then we can come up with a proposed ordering among the different transactions to get a consistent output.
- **Fault Tolerance**: We may not want to save partial states from the transactions (e.g. debit shows in one bank account but the corresponding credit doesn't show up for the bank transfer!)

Definition 9.1: **Distributed Transaction**

Distributed transaction is similar to a regular transaction but executed across multiple nodes. Still need to guarantee ACID, but now across multiple nodes.

Common solution: Assign a leader/initiator that initiates transactions, executes consensus protocols across a) multiple participants of a single distributed transaction b) across multiple transactions.

9.2 Google Spanner

Google, Facebook etc. deal with millions of user requests by building an underlying data management layer which is geographically distrbiuted across the world. Spanne is:

- Global Data management layer used by Google for Ads, Play, etc.
- Also available as a Cloud DB service.
- Allows applications to interact with data through SQL queries (but unlike MySQl, offers much higher scalability.)

How does it work:

- Data Stored globally at geographically separated zones (say US, Brazil, Russia, etc.)
- At each location, the database is **sharded** (partitioned) across 1000s of servers.
- At each location, data is also replicated across multiple sites (say datacenters in different cities in the US) to provide fault tolerance and availability

9.2.1 Spanner Stack

Spanner is made of a stak of multiple components:

1. Bottom Layer: Persistent Storage - Distributed File System (like GFS, Colossus, etc.)
    - ▶ Ensures data is written to persistent storage, replicated, written out to disk, etc.
    - Data organized as files (extension of GFS)
    - GFS: Optimized to serve reads and appends (deletes are rare)
2. Next Layer: Data Model/Tablet- Exposes underlying files to applications in a specific data model
    - ▶ Applications don't have to explicitly think about searching through files - presented data as a unit
    - Used a version of key-value store BigTable
    - BigTable tablets modified to modify logical grouping of data:
        - Timestamps (similar to version number) to help with transaction ordering
        - Support operations that modify not just single key or range of keys but more like SQL queries on different types of related data.
    - Resultant layer called MegaStore that exports a view of the data store similar to a relational database
        - Each tablet unit has related database entries grouped based on a set of keys
        - Tablets replicated using Replicated State Machine kept consistent using Paxos
        - Locks and 2-phase locking (similar to 2-Phase Commit): to manage concurrency for transactions across replicas
            - ▶ Lock co-ordinator additionally checks if all participants can grant the lock
    - For transactions across replica sets, use distributed transactions with 2-Phase Commit

9.2.2 Consistency Requirements for read operations

- To get a truly consistent view, will have to block all incoming writes and read data → Obviously expensive
- Instead, take a snapshot of data at a sufficiently recent moment, then use the

data to create the view (like Newsfeed/Timeline)

- Problem: Even distributed snapshot spans multiple replicas, datacenters, and geographic locations! → Also expensive, maybe more than even locking.
- "Approximate" Consistency may be insufficient: 2 seemingly unrelated operations might in fact be related in real life (think of the unfriending your boss and posting about your job search example!)

▶ Goal 1: **External Consistency**: Operations appear to be consistent to th eoutside world (or) order of events in real world = order in which events appeared on global clock

▶ Goal 2: **Strict** Serializability If transaction1 is assumed to be completed by the outside world before transaction2 started, transaction2 MUST see the results of transaction1.

9.3 True Time

DEFINITION 9.2: **True Time**

True Time is not real time, but an uncertainty interval around real time. Allows different operations:

- TT.now(): Returns a time interval [start time, end time] - values of what earliest and latest time it could be at the given moment
- TT.after(t): Returns True if time t has *definitely* passed
- TT.before(t): Returns True if time t has *definitely* not arrived yet

Essentially provides a way to reason about as well as limit the uncertainty (ϵ) in measuring time. The time is represented as a window $t \pm \epsilon$ i.e. when we read time t, it can actually be $[t - \epsilon, t + \epsilon]$

How to measure this uncertainty :

- By periodic probing of a master clock in datacenters
- Use both GPS and atomic clock
- Compute $2.\epsilon$ based on the probing period and probe latency
- This ϵ essentially gives how "off" the local time can be from global time.

How to use True Time to order transactions Goal: Assign timestamps to transactions such that we can be sure of their order of execution

1. Acquire locks required for transaction
2. Pick a timestamp for the transaction (this is where we need True Time and can't simply read off the clock)
   - Get current reading of clock s' = TT_now().latest
   - Instead of taking time t, the "latest" above gives $t + \epsilon$, i.e. the latest time it can be now if we account for clock uncertainty
   - Since we pick the "latest" value of current time, any transaction before this would have picked an earlier value, and is guaranteed to be ordered correctly
3. Pick a timestamp to release locks
   - Wait until TT_now().earliest $>$s
   - Instead of taking time t, the "earliest" above gives $t - \epsilon$, i.e. the earliest time it can be now if we account for clock uncertainty
   - Essentially waiting for this window to start everywhere i.e. if the "earliest" time now is itself greater than the s we pick, the actual time everywhere in the system is guaranteed to be greater.
4. Actually release the locks

Essentially even if it is a tiny operation, we wait for the window $2.\epsilon$ to commit it, so that we can be sure that

- All operations in the system before time s were completed
- Any operation after the current one will have timestamp $>$ s

9.3.1 Ordering Write Transactions

Note that in the sequence above we are using **pessimistic locking**: we obtain all locks *before* we start transaction → prevents conflicts.

Alternative is to use: **optimistic locking** or **Optimistic Concurrency Control**(OCC): let the transaction execute and acquie lock only when necessary (Can improve throughput) → Leads to lots of conflicts, especially with long transactions. Here since write needs to be replicated as well, transactions include network delay of communicating to replicas so is a long transaction.

Goal: Ensure write completes in all replicas, replicas have a consensus on final value, and transaction gets timestamped with unique value.

How participants ensure transaction completed across all their replicas :

1. Acquire locks

2. Pick s (See previous section)
3. Start consensus algorithm
4. Achieve consensus
5. Wait until time is greater than uncertainty window (See above)
6. Then actually commit transaction i.e. release locks
7. Leader releases locks then notifies other replicas.

For transactions that span multiple replica sets :

- 2-Phase Commit to ensure all participants in the transaction complete it before considering the transaction as committed.
- Each participant itself needs to ensure the transaction is replicated across all replicas -> See steps above

Thus, the actual steps are:

1. Co-ordinator starts transaction i.e. acquires locks + notifies participants
2. Participants acquire locks and compute their individual s value
3. Each participant logs their operations (this is part of the 2-Phase Commit Protocol)
4. Once logging and operation completed (Prepare Phase complete!), notify co-ordinator that operation done and send s value
5. Compute overall s value (largest value of s so that when commit happens everyone is sure to have passed that time)
6. Wait out the time required (until TT.now().earliest $>$ computed s)
7. Co-ordinator completes transaction (i.e. releases locks) → notifies participants to release locks as well

Example: Assume $\epsilon = 4$ here.

- Say we start a transaction (the post deletion) at local time 6 or (6-4, 6+4) = (2, 10).
- Say we communicate to replica which sees local time = 8. So the time window here is (4, 12).
- Participant notifies the co-ordinator, which finalizes the timestamp as $s = 8$.
- Once the transaction completes, both nodes wait until the timestamp window has earliest $>$8 (say, $s = 13$, so window is (9, 17)) before actually committing.

The next transaction can't start until this time, when its local time is **guaranteed** to at least have a value 9. The next transaction might pick an $s = 15$ when it actually acquires a lock, ensuring the ordering.

9.3.2 Ordering Read Transactions

Two types of read transactions: 1) Read Now 2) Read recent

For read now transactions : These still need to be externally ordered

- Leader determines safe timestamp (using the same steps above i.e. determine timestamp t such that the TT.now().earliest > s)
  - Leader is the Paxos leader for transactions in same replica set
  - Leader is transaction co-ordinator for cross-replica-set transactions

"Safe" timestamp implies a timestamp greater than all current writes in progress $\implies$ **Ongoing prepared, but not committed writes can delay reads** (We want the read to happen after all ongoing writes are committed, so ongoing writes can delay reads!)

For read recent transactions : Need *some* valid recent value, not necessarily latest

- Simply need some recent valid value
- Without TrueTime would take a consistent cut using a distributed cut algorithm, use that to get value
- Since timestamps are guaranteed to be ordered here, can simply choose snapshot at a recent timestamp, read from the snapshot! (*Ez pz! Thank you, True Time!*)

9.3.3 TrueTime alternatives

TrueTime achieved by using GPS + atomic clocks resulting in a few ms of uncertainty (5-7 ms) $\implies$ a transaction can be delayed by this value (few ms) to ensure ordering.

- **NTP Protocol**: 100ms delay. Obviously, impractical.
  - Can be useful if we ignore external consistency for some transactions i.e. Choose to use NTP only when external consistency is necessary (and ignore for others)
- **CockroachDB**: Exposes a linearizability flag in transactions to applications
  - ▶ Applications explicitly choose which transactions need external consistency

Note that if only some transactions are strictly ordered, we end up with multiple concurrent transactions in the system. For these transactions to be serializable:

- Optimistic Concurrency Control
- **Snapshot Isolation** or **Multi-version Concurrency Control**: To provide Atomicity and Isolation
  - Sequence of snapshots is serializable
    - To keep transactions serializable - drawing a directed line among snapshots produced should result in a graph with a cycle. If there a cycle might be formed, abort that transaction and restart it.
  - Transactions read from snapshot version of the distributed state

TrueTime achieves linearizability, but these methods achieve serializability.

9.4 AWS Aurora

Different solution to the same problem - used by AWS.

- **Primary-Replica** Architecture: Single primary node that processes Read-Writes, Replicas perform Read-Only
- Design for **Availability**: Guarantees service even if an entire zone is lost
  - Servers in many zones - each zone independently managed (and maybe geo-distributed)
  - Data replicated to 3 zones, in each zone replicated to 2 nodes.
  - Uses voting quorum to provide read/write consistency (i.e. majority of zones agree on a value)
  - ▶ Need enough replicas outside a zone to reach a quorum

I/O Amplification - Since 6 replicas lots of I/O for each transaction

- **Mirrored MySQL**: With traditional architecture, for each transaction, primary sends log (operations performed)+ data + metadata.
- **Aurora**: Avoid this by using only **log replication** i.e. Send only log(operations) + some metadata
  - ▶ Why does this work? The underlying storage layer (unlike MySQL database) is also a distributed file store
  - This layer guarantees efficient access/storage of distributed data between primary and replicas

  - Sending log (operations) alone is sufficient for the replica to get the correct data and perform consistent read operations

Note: This method might sometimes result in reads of stale write data unless some additional locking is used.

Consistency in Distributed Data Stores

We have previously discussed consistency; we now continue our discussion, but specific to the context of distributed data stores (represented as key-value stores).

Why is Consistency important Unlike in Spanner, most systems cannot assume the availability of something like TrueTime, but still need to offer consistency guarantees.

In the absence of specific actions, it is easy to break consistency in distributed systems. Example:

1. Bob first says "Sally is sick" at time t_0 and "Sally is well" at time t_1.
2. Alice sees both updates and says "Great news!" at time t_2.
3. Carol only sees Bob's first update and then Alice's update, so she is confused!

→ Obviously, this won't happen if the message boards were a single centralized database, but this is quite likely in a distributed system if Bob's second message is delayed or lost.

Why is consistency hard

- **Replicated state**: Usually same data present in multiple replicas, so we should it is possible for some replicas to have stale data. Need to ensure this data is not used.
- **Caching**: Most DS use cashing to improve performance, but the cache is now another copy of the data that can go stale and must be updated correctly.
- **Distributed State**: Need to propagate writes to distributed state in the correct order and offer ordering guarantees.
- **Failures**: Further complicated by the occurrence of failures (should not lose data if a node fails, or have stale data in one node, etc.)

10.1 Consistency Models

Guaranteeing consistency obviously comes with a cost to availability and performance. (Remember CAP Theorem)

To deal with this, we define consistency models. A consistency model is a contract between the system and the user. Here, the system makes a guarantee about the ordering of the updates, and how these will become visible to ongoing read operations.

Some examples of consistency models include:

- Strong Consistency guarantees that the **real** order of execution of events will be visible to all.
  - Linearizability is a related concept where the transactions may appear to be in linear order, even if individual operations are not.
- **Sequential Consistency** guarantees that all the events will be visible to all participants in the system, in the **same** order, but this order need not correspond to the *real* order of ocurrence of those events.
  - guarantees that the transactions appear in the **same** order to all participants in the system, but this order need not correspond to the *real* order of occurrence of the transactions.
- **Causal Consistency** is a model that guarantees that the ordering will be enforced only for events that are related to each other by the happens before relationship. There are no guarantees of ordering for concurrent events.
  - If event A *happens before* event B, all participants in the system will see A before B. But if there is no clear relationship between A and B, i.e. they are *concurrent*, then participants may see them in any order.
- **Eventual Consistency** is a model commonly used in practice where if the faults or failures are not permanent, the model guarantees that all writes will become *eventually* visible. Note that this means it is possible to have periods where the view of the data store is not up to date or consistent.

→ In general, consistency models can be placed on a spectrum where models trade-off weaker consistency for greater availability.
Some efforts to quantify this trade-off use the number of delayed updates at a node and the number of out of order updates that may be visible in the system (but aren't yet committed).

Definition 10.1: **Key-Value Stores**

Key-value stores are data stores, where each piece of data is uniquely identified by some key, and the value associated with that key can be some arbitrary data blob of some size. E.g. A simple hashmap.

Basic operations on a key value store are:

- **Put**: Writes a value for the associated key
- **Get**: Reads or returns the value associated with a key from the data store

Other operations that *may* be supported are:

- **Scan or Range Query**: Returns all values with keys in a given range
- **Transactions**: Operations involving multiple keys E.g. Multi-Get, Multi-Put etc.

It may also store more information such as timestamps, relationships between elements etc. which allows it to support even more complex operations.

10.2 Look-Aside Cache

Definition 10.2: **Look-Aside Cache**

Look-aside cache is a cache that does not sit on the *front* of the data store, but on the *side*. This means that, the client application that accesses the cache will also explicitly access the database when needed, instead of the cache itself routing requests to the next layer of the memory hierarchy.

10.2.1 Look-Aside Cache Read Operation

1. Client sends request to the cache
2. If the cache has the data, it returns the data (Cache hit).
3. If the data is not present (cache miss), then the miss is returned to the client.
4. Client performs explicit database lookup.
   - ▶ The database lookup may be a more complex operation than the cache lookup since the database interface may be more complex than the simple "get" query.
5. Once the database responds with the query result, client performs explicit set

operation to add the data to the cache.

Obviously, this is still a demand-filled cache. Note that, in the original paper, the database store used was a SQL DB stored on hard disks.

10.2.2 Look-Aside Cache Update Operation

There are different design options for how to perform the update, e.g. client could mark data invalid in cache, or updated the new value. But then if client crashes or is delayed or corrupted, it may lead to inconsistent state.

Actual solution: Delete the cached entry from memcached and update the database.

- Good because ***deletes are idempotent w.r.t correctness***. Worst case scenario with unnecessary deletes is an **ineffective** cache, but not an incorrect one!
- Deleted entries can be reused for new inserts.
- Cache also has LRU mechanism to discard least recently used updates to make space for new ones.
- ▶ Since database updates are performed before the delete, no actions to be done after delete is completed.

10.3 Memcached

Memcached is a simple and popular key-value store officially designed and released by Facebook and it was presented at the NSDI conference in 2013. An open source version is also available.

The paper describes:

- How Memcache is used across
  - varied deployment contexts:
    - Within a cluster (col-located ina single datacenter)
    - Across multiple clusters (still in the same datacenter)
    - Across geo-distributed datacenters.
  - to serve different clients:
    - External clients or users
    - Different facebook applications that perform operations on the data.
  - with data from different types of databases in the storage tier: (Memcached serves as an in-memory cache for the objects retrieved from storage)

        - ∗ MySQL database
        - ∗ Cluster of databases across which the data is sharded

- Different mechanisms used to provide consistency across these varied use cases and the trade-off that apply.

Why was a cache needed

- **Nature of Workload**:
    - Read-heavy
    - Reads not uniformly distributed: Some items accessed more frequently than others called **hotter items**.
    - Hot items follow spatial and temporal locality.
- Data needs to be persisted into persistent storage, but persistent memory accesses are slower
- ▶ A smaller in-memory cache with hot/recent items stored in it can improve access speed.
    - Note that not all items can be stored in this cache, only a subset is stored.

Thus, Memcached is a simple in-memory object store, and the cache components of its architecture are called Memcache.

10.3.1 Features of Memcache

- ▶ Memcache only consists of clean, read-only data!
- Memcache is a **non-authoritative cache** i.e. the database (not Memcache) is the final authority on the correct value.
- These operations are exposed to the client, so if client needs stricter guarantees, client can perform these operations themselves.
    - ▶ Note that typically the cache has to provide the same semantics as the database to the client since the client can access either. But here, because the client explicitly accesses the caching tier, the 2 interfaces can be completely different.

It is possible to make Memcache persistent by backing it up with persistent non-volatile memory.

10.3.2 Mechanisms in Memcached

A potential consistency problem in memcache :

1. Say there is a key k0 whose initial value is A and it is not present initially in the Memcache.
2. Server0 tries to read this value:
   - Database responds with A
   - Server0 sends a Set(A) to the memcache to set the value to A.
3. A different process server2 sets the value of the key to B in the data store.
4. Server1 tries to read this value:
   - Database responds with B
   - Server0 sends a Set(B) to the memcache to set the value to A.
5. Both server updates arrive out-of-order at the memcache (this is possible because of network delays)
   - First the value is set to B in the cache.
   - Then the new value is set to A.

→ Obviously, here, the final state is incorrect.

To resolve this, memcache uses **Leases**. A lease is a token issued by the cache to provide greater control on when/how it serves data on read operations.

- It is issued on a miss.
- It can detect concurrent writes, so can be used to enforce some ordering on the writes.
- For the thundering herd problem, by controlling how many leases are issued at a given point of time, memcache can control the number of accesses served, and thus number of accesses that will proceed to the database.
- Can also be used to avoid serving stale values.

Thundering Herd Problem

The **thundering herd problem** occurs when a large number of processes or threads waiting for an event are awoken when that event occurs, but only one process is able to handle the event. When the processes wake up, they will each try to handle the event, but only one will win. All processes will compete for resources, possibly freezing the computer (or server), until the herd is calmed down again!

10.3.3 Scaling Memcache

This is essential because ultimately Memcache has finite capacity, and we may need to increase the cache capacity to serve larger loads.

Within a single cluster

This is done by adding more MC instances:

- Data is **sharded** so that each instance is responsible for a subset of keys.
  - Shard boundaries can be adjusted i.e. can configure how many keys the instance is responsible for.
- Routing decisions offloaded to the client i.e. client decides which instance to send the request to.
  - **mcrouter** component encapsulates routing functionality and state in the client.

→ These decisions allow Memcache design to stay simple while still being scalable.

Across multiple clusters

Single cluster memcache instances can have bottlenecks:

- **From client perspective**: Number of memcache instances it can route across efficiently, is limited.
- **From a shard perspective**: If the shard holds hot content, may be a bottleneck on number of requests served.
- **Failures**: As system grows bigger, some component's gonna fail!

Solution: Use multiple clusters. organize the memcache instances into multiple separate memcache cluster, where each cluster holds a smaller number of memcached instances. Here, the same data is held on the corresponding instances on different clusters.

- **Scaling to more requests in general**: Since multiple clusters can serve requests, the total number of requests served is higher.
- **Serving hot content**: Can handle more requests for hot content because the same hot content will now be present on multiple instances on different clusters.
- **Failure Domains**: Since there are multiple clusters, we now have multiple failure domains. A failure only impacts a single cluster at a time and others will continue to serve requests.

Equivalent to having multiple MC caches, so now we need to take of consistency among the different instances!

- ▶ **DB-driven invalidations**: DB initiates invalidations whenever the data is updated in the DB commit order .
- Note that this is necessary for multiple MC cache instances - can no longer avoid it like in the single MC case where client drove the maintaining of consistency.
- It is possible for the memcache data to be stale, but the updates will not be reordered.
- Note that the storage tier itself can be a cluster with many nodes.

Geo-distribued clusters

- Data and services are distributed geographically across many sites. $\implies$ there are multiple memcache cluster groups at different locations.
- Problem: Obviously, network latency beteen locations is very large, so a single **centralized** cluster collecting updates from different locations or driving invalidations is impractical.
- Solution: MC expects data to be replicated across locations by the storage layer.
- Problem: Obviously, now more copies of data that need to be maintained consistently.
- Solution: Memcache specifies protocol on operations to be done on a write when there are geo-distributed datacenters.
  - Distinguishes between a master database site and replica databases.

Protocol on writes for geo-distributed data centers :

1. Client web server wants to perform an update.
2. Client sets a marker in the local memcache for the corresponding entry
   - Marker tells MC that the value is involved in remote operation.
   - This helps to ensure proper ordering among concurrent operations.
3. Actual write performed against the remote master database.
4. Local memcache isn't updated yet. Instead its value is deleted.
   - ▶ Here, the local database may not yet have the new value, so cannot update the cache yet.
5. Replication logic for geo-distributed database replicates the update to all replicas (including local one).
6. After database is updated, marker is deleted.

    - ▶ Until marker is deleted, reads for this entry only served from remote master database (not from local cache or db).
  - Obviously, goal is to reduce this cross-datacenter traffic (i.e. try to replicate as quickly as possible).

10.4 Causal+ Consistency

Why Causal consistency is not enough : Previously, we saw some examples of when operations that appear unrelated need to be ordered correctly to avoid confusing end users. E.g. The "Sally is sick" example and and the job search example in Chapter 9. These problems still exist with causal consistency since in this consistency model, the data accesses for these operations still happen on different servers and appear concurrent! To address these, Wyatt Lloyd (now at Princeton), introduced the new model called Causal+ Consistency or **COPS**.

System described in COPS: Similar to the one in Memcached.

- Key-value based object store on one datacenter, which itself has multiple servers.
- Data sharded across the individual servers in the datacenter.
- data store geographically distributed to other locations as well, so data needs to be replicated across multiple locations.

How COPS works

1. Client performs get operation to read data (served from the local instance).
2. Unlike Memcached, client library captures information about the read.
3. Say there is a later put operation on this instance.
4. Operation sent to other replicas/clusters along with ordering metadata.
   - Actual client library issues a **put after operation** with all the meta data about the dependencies for this put operation has with other (apparently unrelated) values read previously by this client.
   - Local key-value store logs this data and communicates this information during replication.
5. During replication, remote object store only performs update after dependencies have been met. Update becomes visible only after this.

There are newer consistency models that define consistency in a manner more useful from the client's perspective, while continuing to offer high performance and scalability.

Peer-to-Peer and Mobility

In this chapter, we discuss some aspects of the messaging layer on top of which distributed systems are built.

11.1 Communication Support assumed so far

- **Point-to-point Messaging**: So far, assumed that all messages are sent from one node to another.
  - ▶ Even when same message sent to multiple nodes, sent via multiple point-to-point messages.
- **Application-level namespace**: Namespace used to specify different nodes at the application/service level. e.g. primary node, node that stores the shards, etc.
- **Network-level namespace**: Namespace used by network protocols to deliver messages, such as IP addresses or link level addresses.
- **Mapping**: Need an intermediate metadata service to map the two namespaces

What is this mapping layer

- As described above, provides mapping between application namespace and network-level namespace
- Can think of this as an overlay network to which the application layer describes the nodes/endpoints it needs to talk to and the mapping layer maps these "addresses" to the exact network level addresses
- **How can it find this mapping?** As part of the control plane of the distributed service, when different processes are created and launched, the network address is recorded and the information about this mapping is distributed to all nodes.
- The mapping layer **must** allow for these these mappings to change dynamically. can also be more easily reconfigured, for instance, if we need to scale it to a larger number of nodes.

- **Why should we deal with change**: In a DS, change is inevitable - there may be failures or reconfigurations when we scale the system.
  - Nodes may fail.
  - If we scale the system, we may need to reconfigure.
  - If workload changes dynamically, may need to reconfigure as well.
  - ▶ Need to have a way to send messages to the correct network address despite these changes.
- **Challenges in dealing with changes in mapping:** The goal is to ensure correct address can be determined quickly after a change. But this can be hard:
  - **Scale**: Might not be able to send this new address information to all nodes quickly in a large system.
  - **Geo-distribution**: Network latency across different domains also makes transmitting this information difficult.
  - ▶ These are particularly hard if system is very dynamic and changes happen frequently.
  - **Failures**: Should be able to both deal with failures (ensure messages are no longer sent to the failed node), as well as propagate information despite the failures.
  - **Decentralization or Multiple administrative domains**: Nodes may belong to different administrative domains, then we may not always know where to send the new config information to.

11.2 Interconnect Support

The interconnect network consists of the network device hardware (Network interface cards), drivers and protocol stack used by the communication services. So far, we have not assumed any specific properties for the interconnect network. However, often the network itself may provide additional capabilities. These are usually called **collective operations** and are operations involving (typically > 2) devices in the network. Collective operations manage m x n communication patterns, synchronization primitives, etc. The support for these operations needs to be provided by both the hardware and the software stack, especially to be scalable. E.g. in HPC, separate networks are provided by the hardware to support tree-combining algorithm for barrier operations.

Collective operations typically provided by all networks:

- **Broadcast**: Sending message from one node to all the nodes

- **Multicast**: Sending message from one node to some of the nodes in the system.

Additional collectives provided by some networks:

- **Gather**: One node receives messages from all of the nodes in that communication group and aggregates the data
- **All-reduce**: One node receives messages from all of the nodes in that communication group and performs some reduce operation on the received data. Final result is available in all the nodes.
- **Barrier**: This is ≡ to binary gather. Here, all nodes must reach the barrier point before they continue to the next phase of execution.
- **Atomic operations like Compare-and-Swap**: Requires hardware support
- **Timers**: Packets may be stamped with timestamps generated by the network hardware itself. This can denote real time without depending on CPU to generate those timestamps.
    - ▶ Especially useful for RDMA

Advanced features provided by some networks:

- **Remote Direct memory Access (RDMA)**: This allows packets to be transmitted from one node to another directly without engaging the CPU.
- **Direct Cache injection (DDIO)**: Directly injecting data into caches of the nodes. This is useful because we save a memory access, so it is faster.

However, note that we cannot always make assumptions about the capabilities of the interconnect:

- Usually high-end capabilities only available in a tightly-coupled (single) datacenter system, but not the internet wide-area network at large, where the only reliable thing is the use of IP protocol (**everything else** can change!)
- Multiple datacenters may not all have the same kind of infrastructure
- Nodes may belong to different administrative domains.

11.3 Peer to Peer Systems

There are many popular examples of peer-to-peer systems used for sharing music, videos, etc. (E.g. Bittorrent, Tor, BitCoin, Ethereum, etc.). The characteristics of these systems are:

- All nodes in the system are peers - and they collaborate with each other to achieve the common goal (as the case may be).

- Only reliable assumption is that they can use IP addresses.
- Cannot make any assumptions about network topology.

Studying peer-to-peer sytsems is useful because these assumptions can be applied to datacenters within same organization, or the machines in the same datacenter.

11.4 Connectivity in P2P

How do peer nodes in a P2P system find each other's addresses:

11.4.1 Approach 1: Centralized entity

- Assume the existence of a centralized registry where everyone can register their addresses with (the registry knows which data resides where, which process runs where, etc.)
- Example: Napster: peer to peer service where information about which songs were available through which peers was maintained centrally
- Advantage: Can compute which peer has to be computed within 1 RTT
- Disadvantages:
  - The central entity is a single point of failure
  - It can become a performance bottleneck → limits scalability.
  - All nodes need to trust that entity

11.4.2 Approach 2: Flood or Gossip based protocols

- No centralized entity
- Each peer broadcasts information about the content it stores (or requests it can serve, etc.), so *eventually* everyone can identify the right target.
- Advantage: Makes no assumption about trusted, centralized entity,
- Disadvantage: No upper bound on time taken to find the location of the peer
- Example: Gnutella, Bitcoin

11.4.3 Approach 3: Distributed Hash Table

- Allows for a decentralized index where many nodes can provide information about how the data or service is distributed among peers.

- Structure of DHT such that it can make some guarantees on upper bound of lookups required to find the information (art least probabilistically)
- Example: Chord, Pastry, Tapestry, Kademlia (used in Ethereum), Amazon DynamoDB (uses DHT as a building block) etc.

11.5 Distributed Hash Table (DHT)

A **Distributed Hash Table** or **DHT** is a table used to translate the namespace of input (name of a file, string, part of a file, etc.) to a simpler namespace, such as an integer using a hash function.

DEFINITION 11.1: **Hash Functions**

Hash functions are functions that take an input (file, name of a file, part of a file etc.) and produce a unique, condensed form of the original object as the output.

- Always produces same output from same input.
- Typically reductive: output usually a *condensed* form of the input
- Inputs mapped to a sufficiently large set of output values.
- Guarantees the output will be sufficiently randomized.

Hash functions are usually referred to as pseudorandom functions.

Example usage in distributed systems: All clients use the same hash function to produce signature from the input (name of a file or string etc.) The output is then used as identifier for the node (in which the data resides).

Example of how DHT is used: Say we have a system where data is stored on multiple nodes. We may use a DHT to determine where a particular data item is stored. So, hash function would take in input as filename and give a number 0 to n-1 (for number of nodes) as output.

- File name given as input to hash table
- Say hash function produces integer values in a specific range (0 to n-1 above)
- All clients can use the same hash function to compute this integer.
- Each integer can be statically mapped to the IP address of one of the nodes in the system.
    - Number ID space much larger than actual number of nodes.

- Number to node IP mapping can be changed when needed (Unused keys can be mapped to new nodes.)
-

Distributed Machine Learning

In this chapter, we discuss system support for distributed machine learning, specifically in the context of geo-distributed systems (as opposed to data center systems). Most of the discussions will be in the context of training, rather than inference.

The key to the success of ML and AI in recent years is the ability to build robust models from vast amounts of data, a process which is both data intensive as well as compute intensive.

- Has led to datacenters being equipped with massive systems as well as components specific to accelerating ML - GPUs, TPUs, accelerators, etc.
- Much of the data needed come from the edges of the network
- All this data needs to be moved from edges to backend data centers.
- Particularly challenging when data is generated on globally distributed sensors and environments.
- Often need to move this data to multiple geo-distributed data centers (not just the local one) → obviously expensive.

12.1 Distributed Machine Learning Approaches

To deal with this, we can have multiple approaches:

1. **Centralized Approach**:
    - How it works:
        (a) Collect data from different locations
        (b) Move data to single centralized place for analysis
        (c) Build model in centralized location.
        (d) Distribute models back to global locations
        (e) Use model i.e. Perform inference in those locations
    - Disadvantage:

    - **Slow**: Tremendous data movement (53x slower than having all the data locally)
    - **Data sovereignity**: Data movement may be governed by different laws and may have to be moved across countries.

2. **Federated System Approach**:
    - How this works:
        - Here, data is evaluated locally, and some learning happens locally.
        - These locally computed models are then aggregated periodically (Parameters are collected centrally)
        - Based on the centrally collected updates, a general update is computed,
        - General update is disseminated across all locations.
    - Example: Gaia (published in ATC 2017)
        - Designed primarily for data centers
        - Leverage the approach of parameter servers.
        - Similar solutions: federated distributed learning model by Google.
    - How **parameter servers** work:
        - System deployed in a data center- in a cluster with many machines- some designated as workers and others as parameter servers.
        - Training data distributed across all workers; model parameters need to be distributed to all servers
        - Learning performed iteratively:
            (a) Workers get some set of parameters and compute model updates or gradients.
            (b) Updated parameters communicated to the servers
            (c) Servers aggregate this information and synchronize amongst themselves.
            (d) Server determine final model updates for this iteration and communicate to workers.
            (e) Repeat loop until model converges (difference in update to parameters beteen 2 iterations below a threshold.)

3. **Isolated Learning Approach**: **isolated Learning** is to perform learning independently at each location using the data from that location.
    - Each node builds its own custom model in isolation.
    - Model is truly tailor-made.

- Problems:
- - There may be insufficient data at that single location → learning process may not converge
  - Even with sufficient data, may be suboptimal since some patterns may be present at multiple locations - have to relearn the same information
  - Particularly wasteful if some trends in data propagate over time from one location to others.

12.2 Geo-Distributed ML with Gaia

We would now like to extend parameter servers to geo-distributed data sources (multiple datacenters)

Why naive extension of the approach doesn't work

- Works functionally but causes too much of a slowdown (20 times) compared to when all data is in the same datacenter.
- Slowdown is due to characteristics of wide-area-network connectivity.
- Authors studied slowdown by on AWS instances in 11 EC2 regions.
- Slowdown most significant when the WAN connectivity performance of the region is the poorest.
- Even when the WAN connectivity performance is good, there is a 3-4x slowdown.

Leverage Approximation: How this was solved in Gaia

- Key idea: decouple synchronization of the model within the data center from synchronization of the model across data centers
- Essentially, keep the functionality within datacenter same as before and synchronize servers and workers within datacenter regularly.
- Across datacenters, synchronize infrequently (periodically).

▶ **How can this work?** it works because ML is already imprecise/approximate. Some level of error among different locations can be tolerated without sacrificing correctness!!

When to synchronize among datacenters?

- Key observation by authors: 95-97% of updates led to $<$ 1% of change in model!

  - Plot percentage change in model(x) with number of updates that didn't result in a change greater than x% (y)
- Key idea: Communicate only the significant 3-5% updates across datacenters.

12.2.1 ASP

As described above, Gaia relies on a new synchronization model that they call approximate, synchronous, parallel or **ASP**.

Mechanisms required to implement ASP:

- **Significance filter**: Filter out updates based on siginficance. This is done as follows:
  - Gaia exposes an API to allow programmers to specify what's significant for their case.
  - System dynamically computes significance of updates based on this function and filters out insignificant ones.
- **ASP Barrier**: Sometimes, synchronizing significant updates alone still takes time, but need to ensure this is done synchronously. To achieve this Gaia implements a barrier.
  - During a remote sync, an index with some information about the pending updates sent to remote datacenter.
  - This index indicates to the datacenter to wait for the pending update before proceeding.
- **Exchange clock information**: Need to ensure one datacenter doesn't lag behind or become too stale because of slow WAN.
  - Datacenters exchange clock information.
  - Can use this information to estimate staleness and round trip times
  - Also use it to determine if one datacenter needs to slow down its parameter servers to allow other datacenters to catch up and for overall system to be more in sync.

How this works overall

1. Within a data center, workers communicate with local parameter servers as usual.
2. Updates are aggregated within the datacenter.
3. Significance filter applied to updates.

4. When a significant update is determined, this information is used to create information for the ASP selective barrier.
5. ASP barrier communicated to remote datacenters.
    - Note that this is control information sent via separate control queue, so not blocked by the pending data to be transmitted to remote datacenter!
6. All communication tagged with local clock and these clock values used to determine when the learning process needs to be slowed down.

12.2.2 Results from the paper

Consider one experiment described in the paper.

Experiment details:

- Performed with 11 EC2 servers (different AWS regions)
- Datacenters in Virginia vs California; Singapore vs San Paolo.
- Baseline case: ML performed over LAN (within same datacenter)

Results

- Significant drop in performance when WAN is used compared to LAN.
- Using parameter server simply across geo-distributed datacenters much worse than performing ML in a single datacenter.
- Gap between LAN vs WAN much lesser when the datacenters are closer together or connected by a better WAN.
- Difference in Gaia performance compared to LAN is minimal.

$\implies$ Gaia allows ML at geo-distributed datacenters to be performed at speeds similar to ML in a single datacenter!!

12.3 Collaborative Learning

12.3.1 Tradeoffs of Using Global Model

- Both Gaia and parameter servers try to create a single global/unified model to use across entire system.
- Global model not always needed
- Locality in data trends and patterns at different locations - personalization possible.

- Can better serve these applications using a small, more tailored model - also more efficient.
- Also, building good global model harder from algorithmic perspective - leads to overfitting, less accuracy, etc., especially when data tends to display different properties at different locations.
- Even with optimizations in Gaia and federated learning, data transfer costs large and hard to justify.

12.3.2 Collaborative Learning with Cartel

Solution: New approach - **Collaborative Learning**

Goals for this new approach:

- Allow each node to benefit from small, customized models, but also, transfer knowledge when needed.
- When there are changes in environment or variations in workload pattern, look for nodes where similar patterns have occurred and transfer knowledge from those (perform model update).
- System level support to jumpstart the peer-finding process and to enable performing the right type of knowledge transfer.

What is Cartel:

- **Cartel** is a prototype system for collaborative learning
- Developed by Ada's group in collaboration with Nokia Bell Labs and published in Cloud COmputing Symposium 2019.

Advanatages of Cartel:

- Particularly good for systems that have localized trends that that propagate over time to other locations.
- More lightweight models compared to centralized approaches
- Less data transfer time and lower training time compared to centralized approaches
- Better model accuracy than learning in isolation.
- Can be deployed in vatious types of situations e.g. at cellular base stations of mobile networks - might need to learn to configure h/w and s/w of mobile network.

Basic operation: Cartel relies on a centralized component: **metadata service** to aggregate metadata about learning processes at different nodes.

1. Each node receives some number of requests
2. Each node performs local learning process using its local storage
3. **Drift Detection** Each node dynamically evaluates the quality of the learning and when it detects a drift (model accuracy drops for some sort of change, it contacts metadata server.
4. Metadata server, during regular operation, accumulates some metadata about classes observed at each location, accuracies at these locations, etc.
   - ▶ Small amount of data compared to data transfer in centralized approaches.
5. Using metadata information, the service finds a good peer node (called good **logical neighbor**)
6. **Knowledge Transfer**: Once peer is found, parameters are exchanged between these nodes.

Experimental setup

- Compare results to 2 extreme baselines: centralized approach , isolated approach.
- Different workloads i.e. different patterns of variation in geographical locations.
- Different metrics:
  - How quickly the model at a single location adapts to changes
  - How much data transfer is required to facilitate learning (for learning to converge)
  - Size of the resulting model
  - How long it takes to learn about other models or to perform inference

Results

- **Adapts quickly to changes in workload:** When shift occurs, converges 8x faster compared to isolated learning.
- **Reduces total data transfer costs:** Upto 1500x less data transferred compared to centralized approaches.
- **Produces smaller models:** 3x more lightweight models than global models leading to 5.7x faster training.

12.4 Other stages of ML Pipeline

ML Pipeline has other stages:

1. Training
2. Model Serving : Model used to serve queries
3. Hyperparameter Tuning
4. Steaming
5. Simulation
6. Featurization

Some stages related to creating and optimizing models, others related to data delivery or execution of the distributed tensor manipulations.

- Ray Lab in Berkeley developed a system called **Ray** that integrates all these types of functionalities in a single unified framework.
- Opens up many efficiencies in the end-to-end process : Currently each stage performed using different technology, so different types of systems need to interact and coordinate to exchange data between various stages → room for optimization.
- Ray Paper from OSDI 2018.

Byzantine Fault Tolerance

Consensus algorithms previously discussed always assumed that all nodes bevae properly unless they fail (i.e. stop working). But this need not be the case in reality. In this chapter, we look at failures where nodes behave incorrectly, and how to achieve consensus in such scenarios.

13.1 Byzantine Failure and Byzantine Generals

Byzantine failures are failures where nodes in a distributed system continue executing but start sending incorrect messages, for malicious or arbitrary reasons. The most general definition of the failure is that the faulty participant presents different symptoms to different observers.

Definition 13.1: **Byzantine Generals Problem**

The Byzantine Generals Problem was first described by Leslie Lamport in a paper published in 1982. It raises the question of how to reach a consensus, (i.e. a single correct decision) in the presence of Byzantine failures in the system.

In the original problem, a group of generals need to agree on whether to attack some location or retreat (so they need to reach consensus). Underlying assumption is that to succeed, the generals to all attack together or retreat entirely (a half-baked attack will likely fail!) The problem refers to the problem of achieving the right conclusion despite the presence of faulty generals (or messengers).

Details of original Problem

- The generals can only communicate with each other using messages sent via messengers.
    - ▶ Each general receives messages from everyone else about what their opinion is

    - a decision can be reached by considering all the received messages (similar to a consensus protocol)

- **Neither the generals nor the messengers can be trusted!**
    - A general may have become corrupt, so may send different messages to different generals
    - A messenger may have been compromised so the messages arrive corrupted.
- When a general receives faulty messages, they may reach the incorrect conclusion.
    - For example, if another general appears to sometimes agree to attack but sometimes agree to retreat, or rather tells the attackers they will attack and tells the retreaters they will retreat, then the remaining generals would act counterproductively.

Typically Byzantine faults are caused by malicious players or by nodes that are internally inconsistent (somehow the node lost some information, so for one query it responds one way and the next query it responds differently)

Note that, a node appearing to be active to one node while appearing dead to another (maybe due to partitioning) is typically not considered Byzantine failure, but a node that responds with two different values for the same Paxos query by two different .nodes is.

Goal here is to reach consensus with safety, liveness, correctness, etc. while tolerating up to F failures despite byzantine behaviors.

1. **Corrupt Messages**: Use cryptographic methods to authenticate communication and verify that the messages have not been tampered with.
2. **Untrustworthy participants**: Increase the number of participants in the system to tolerate Byzantine failures as well.

Note that FLP still holds here: the protocol can guarantee safety but to really guarantee liveness as well, need to add requirement that that messages will be delivered with some bounded delay.

13.2 Practical Byzantine Fault Tolerance: pBFT

pBFT is an algorithm proposed by Miguel Castro and Barbara Liskov from MIT and was presented at OSDI'99 for practically achieving Byzantine fault tolerance. Though there were other algorithms that also proved the (3f+1) nodes requirement, pBFT was the first solution (at the time of presenting) capable of processing large number of operations per second.

Model Assumptions/Goals

- The client interacts with a group of servers that implement a replicated service.
- Clients need guarantee that servers reach consensus when replicating client's updates or will respond such that client can determine correct response using majority voting among received responses.
- One of the replicated servers is a leader; rest are backups
- An arbitrary set of upto f servers may fail
- The primary (or leader node) determines the current view of the system.
- In the view, identity of primary node may change over time. (e.g. if primary itself fails)
- Each replica maintains consistent information about server state, messages exchanged and current view (including who the primary is).
- All communication is secure

▶ Messages may be secured cryptographically through use of public key infrastructure, message digests, etc.

Why do we need (3f+1) nodes

- If we have n nodes in the system and need to tolerate f faults, we have (N - f) active nodes.
- Now if we simply decide based on (N - f) nodes, it is possible f of those nodes were simply delayed, but a different f nodes had failed with Byzantine failures.
- This means we actually only have (N -f -f) reliable nodes.
- To reach a decision (majority voting), a majority of these reliable nodes must be > f.

$$N - f - f > f$$

$$\implies N > 3f$$

$$\implies \text{At a minimum, } N = 3f + 1$$

13.3 pBFT Algorithm

Client side:

1. Client makes a request to at least one server (say, the primary), and it ultimately receives a response.

- ▶ Since even leader can be corrupt, the client sends request to all servers and receives multiple responses.

2. Once >f+1 responses are received, client concludes that it has the correct response.

Server side:

1. Once received at the primary, request is processed in 3 phases.
2. **Pre-prepare Phase**:
   - **What the primary does:** Primary needs to verify the message and send pre-prepare requests piggybacked on regular messages.
     (a) Picks and sequence number
     (b) computes the digest of the message
     (c) multicast a pre-prepare request to all of the backup replicas.
   - **What the replicas do:** Each of the replicas needs to check whether they can accept the pre-prepare request.
     (a) Checks that signature and message digest are cryptographically correct.
     (b) Stores this message in the log
       - If the log is full, the response to this message will be delayed or blocked.
     (c) Message received corresponds to the current view that they are aware of
     (d) Sequence number included in the message is new
     (e) Sequence number received lies between two watermarks (based on some maximum number of inflight operations in the system.)
       - ▶ Ensures that a faulty primary doesn't just start sending messages with a large sequence number so as to block other requests from getting replicated.
3. **Prepare Phase**: If a pre-prepared message is accepted, then the replica enters the prepare phase.
   (a) Multicasts a prepare message to all other nodes.
   (b) Logs that this message has been sent
   (c) Each replica then waits for 2f matching prepare messages to be received from other replicas.
   (d) Waits for the predicate $\textit{prepared}(m, v, n, i)$ to be true.

- Log has a pre-prepare request AND 2f matching responses have been received with same v, n, d.

4. **Commit Phase**: Once >2f valid responses are received, a replica enters the commit phase.
   (a) It sends a commit message to all other replicas
   (b) Logs that commit message was sent.
   (c) Waits for the predicate $committed - local(m, v, n, i)$ to be true.
     - $prepare(m, v, n, i)$ is true AND 2f commits (other than its own) have been received with same v, n, d.

   (d) Once the request is committed, it can be executed and a response can be sent to the client.

Some additional details in the paper

- **Special cases**:
  - **Log Garbage collection**: When log is full and needs to be cleaned up.
  - **View Change**: Problems with primary and view changed.
  - **Liveness**: use of timeouts to guarantee liveness.
- **Performance Optimizations**:
  - To reduce number of messages
  - To reduce overlap in message processing
- Example system: Byzantine-fault tolerant Distributed File service implemented with pBFT

13.4 Byzantine Consensus vs. Blockchain

Some aspects of blockchain have parallels with pBFT. For instance, **Distributed Ledger** is the underlying technology enabling many blockchain solutions.

- Distributed Ledger is a timestamp sequence of records replicated across distributed machines in a consistent manner. (similar to replicated logs in Paxos and pBFT)
- Each node agrees on the precise order and content of ledger entries, regardless of failures.
- The ledger encodes the execution of a series of updates or transactions and their entire history.

- The ledger must be unique and unchanged even if some participants in the system try to make changes or to create an alternative view of the history.
- And it must achieve that without introducing some centralized clearinghouse for reaching agreements.

Can we used to pBFT for Blockchain?

- **Why is it possible**:
  - pBFT allows us to achieve consensus in a decentralized manner.
  - (Unlike Paxos,) Can achieve consensus while dealing with Byzantine failures and unreliable networks.
- **Why is it a bad idea**:
  - pBFT has a relationship between number of faulty nodes and number of total nodes in the system.
    - These values may not been known *a priori*.
    - An attacker can create many faulty instances of themselves, so this number is not useful.
  - Even if we could determine N, number of messages exchanged are cubic with respect to number of participants $\implies$ Costly!

How distributed ledger actrually works Distributed ledgers use some important ideas to a) probabilitically reduce number of participants required for consensus, and b) reduce number of messages required for a safe chain update (and forward progress), thus reducing the computational power and energy required

- **Proof of Work**: Participants are required to solve cryptographically challenging problems/puzzles and to present proof of work.
- These puzzles need to be solved within the shortest amount of time.
- **Incentive structure**: Good behavior rewarded via cryptocurrencies.
- Encourages participants to behave correctly (discourages malicious or Byzantine behavior)
- Miners receive cryptocurrency when creating a new block in the system,
- Miners also collect fees from transactors whose transactions are included in the values of those blocks.

Trivia: *The original BitCoin paper depends on multiple popular technologies, but not explicitly on solutions for Byzantine fault tolerance.*

The details of specific distributed ledger solutions vary with respect to a number of features or design goals regarding performance, trust assumptions, etc. For instance, obvious differences exist among **permissionless** (fully decentralized) solutions vs. **permissioned** solutions with a subset of trusted parties.

Edge Computing and the Internet of Things(IoT)

In this chapter, we look at some recent trends around new types of components and infrastructure in the computing landscape, and how these change distributed systems assumptions and designs. We specifically look at edge computing and the Internet of Things(IoT).

14.1 Edge Computing?

Tiers in Computing

1. **Cloud Data Centers** Traditionally, these supplied data to the end-user through geo-distributed datacenters. (Usually done to deal with latency requirements). However, with newer technologies, the physical limitations on latency imply that newer requirements cannot be met by supplying from these alone.

2. **Cloudlets** Newer data centers closer to the edge, to meet these requirements better. Include micro-datacenters in enterprise locations, restaurant chains, vehicles, etc.

3. **End-user devices** Traditionally, end-user applications and devices interact with services deployed in remote cloud data centers. Newer devices on the edge (AR/VR applications, autonomous driving, etc.) have created newer demands for latency/bandwidth.

4. **Very low-end, low power devices**: Encapsulate basic sensing/actuating devices - may not have reliable energy sources, and may have newer challenges.

Newer tiers have vastly different requirements necessitating the use of newer approaches and technologies to serve these.

Why edge computing Consider the Cisco report Virtual Network Index that tracks network traffic volume data:

- Recent years and projected increase in both number of connected devices and amount of raw data.

- Also projected increase in demand for wireless connectivity.
- Increase in demand primarily due to:
  - Emergence of new types of workloads that require more bandwidth, better latency guarantees, etc.
  - Post-pandemic: Many companies have adopted remote work on a more permanent basis. So, now connectivity is not just for games or entertainment,
- ▶ Post-pandemic workload distribution: More in residential areas than in traditional hotspots.

14.1.1 Closing the Latency/Bandwidth Gap

Given that there is increasingly greater demand for network resources, how can we close this gap?

- **New technologies**: Could potentially rely on new technologies like 5G, etc. But this is not good enough because:
  - Might take a long time to deploy these new technologies
  - Even after deployment, older technologies will continue to be around.
  - Usually, deployment also comes in with higher cost.
- **Open Source technologies**: Cost factor can be addressed to some extent by using open source software stacks and commodity hardware.
  - Number of open source efforts that provide:
    - ∗ hardware specifications
    - ∗ open source implementations of mobile network stacks, including radio stacks and netork core stack
    - ∗ prototypes of mobile network system prototypes that allow federated models of participants to interact in the network.
  - E.g. academic project presented at NSDI to address connectivity gaps in South-east Asia.
- ▶ **Moving to the edge**: More promising to tap into the compute and storage resources at the edges of the network.

Originally called **Mobile Edge Computing** or **MEC**, where mobile refers to the edge infrastructure in the mobile networks, MEC has later been extended to refer to the access theory of the network and now stands for Multi-access Edge Computing. The basic idea here is to trade one type of resource for another i.e. instead of finding ways to provide greater connectivity, we use computational resources closer to the edge to satisfy the connectivity demand.

Edge computing is often broadly defined and can refer to infrastructure in different form factors integrated in the end to end communication paths -in cellular towers or in specific locations, etc.

Edge Computing vs CDNs Edge Computing is not necessarily new. Even in 2017, >50% of the internet traffic was served through CDNs, and this number is rapidly growing. Since CDNs try to deploy servers at different locations closer to the end-user to offer better connectivity and reduce backhaul bandwidth, they are similar to edge computing in some sense. However, the primary differences are:

- **Scale**: CDNs are concerned with deployment of infrastructure (on the order of 1000's of locations globally). Something like mobile infrastructure needs 2 orders of magnitude more infrastructure points (Can derive this from FCC data registering devices on antennas)
- **Ownership**: CDNs are typically owned by the CDN providers (not end-users!)

▶ Mobile network operators are now partnering with cloud providers to tap into these edge resources to provide a new "cloud".

Note that, mobile networks are not the only solution for the edge tier. A report called **State of the Edge** provides a taxonomy of the different types of edge locations, including wireless/cellular access points, aggregation points, new devices like drones, cars, cameras etc.

14.1.2 Edge Computing Drivers

The fundamental drivers behind edge computing are:

- **Speed of Light**: poses fundamental limitations on latency and forces us to deploy services closer to the end-user.
- **Increase in data traffic**: creates demand for bandwidth
- **Energy Demand**: caused by increased network capacities and power consumption, poses limitations on how much data can be driven into a datacenter.
- **Regulations**: Data sovereignity laws, privacy laws etc. mandating data to be processed within national boundaries.
- **Newer applications**: such as mentioned above- drones, autonomous cars etc.
- **5G**: is not only an enabler of edge computing, but also a driver! E.g. latency requirements of sub-10-20 ms
- **Letency as a driver**: Keynote given by Pablo Rodriguez classifies different

use cases in different latency bands. Number of these requirements could be met over wired infrastructure, but not through wireless hop, thus necessitating a new tier.

- **Bandwidth as a driver**: In recent paper, Microsoft researchers discuss available data rates in different countries. Apart from large differences between countries/technologies, there is also large difference in upload vs. download bandwidths, and future demands for higher uplink bandwidths is also a driver

Edge Computing is expected to have a wide variety of applications from food industry (Chick Fil-A) to agriculture.

14.2 Distributed Edge Computing

So, how is distributed edge computing different from traditional distributed systems we have been seeing thus far? Or, why can't we simply use the same technologies as before:

- **Scale**: While data centers may have extremely large numbers of components, those are more tightly coupled. These protocols are too *chatty* to be used on edge devices and can introduce too many overheads.
- **Edge is not elastic**: Datacenter systems assume that on failures, resources are replaced by other nearby resources (compute, storage, etc.) This is not true for edge devices - if we are unable to reach one edge device, cannot run the same service on another!
  - This implies we need to provide a lower service level rather than complete availability.
- **Device Churn**: Obviously datacenters don't have nearly as much device churn as edge computing. This can cause a decrease in service reliability.
- **Mobility**: Edge devices tend to move much more (e.g. movile devices may move between base stations, etc.)
- Fault-tolerant solutions based on datacenters will cause a lot more overheads if used in edge computing.
- **Variability**: Much greater heterogeneity in the types of devices used, with large variations in compute and network capabilities, thus performance is also quite variable. (Cannot assume symmetry, unlike with datacenters)
- **Localization**: Datacenter actions tend to be more homogeneous, wheread there is greater localization in actions at edge devices (such as with access patterns, network infrastructure, etc.)
- **Decentralization**: More common to find resources operated by multiple providers.

- **Security assumptions**: Security is a much higher priority since with the variability of devices, cannot assume the same level of security.

These factors need to be considered to build solutions for distributed edge computing.

14.3 IoT and Distributed Transactions

Consider **IoT** as an example scenario to look at how distributed systems concepts need to be modified for new infrastructure tiers. In IoT, typically, gateways closer to the user (at home, work, hospitals, etc.) sense the environment, then feed this data to backend services deployed in the cloud that process the data and may provide updates to (or trigger actions at) actuator devices closer to the user.

Example: Intrusion detection application Consider an example of an intrusion detection system, that uses a motion detection device (may be a camera), that triggers an alarm if unusual activity is detected. The motion detection device feeds camera data to the backend service, which performs some processing actions and then triggers the alarm if needed. It is necessary for the service to avoid redundant actions, e.g. not trigger the alarm if it was already triggered.

This might involve 2 different state variables: alarms.strobe that triggers the alarm and state.alarmActive that shows that alarm is active. The update of these variables needs to be performed atomically.

The classic distributed system solution for these would be to use transactions and an undo/redo log. However, "undo"ing the operation in a physical environment does not make sense (*How do we even "undo" an alarm ringing?*), so we need to ensure there are no inconsistent states during execution. For example, if the alarmActive state is set but the alarmStrobe action was lost in the network, the alarm never rings, but it shows as already activated, so the system doesn't trigger the alarm again! (Also, the application state here would be inconsistent with the physical state of the system.)

While this is a simple example, there are other possible scenarios that expose the underlying problems of directly using datacenter-based approaches for IoT. *Thus, we need different solutions!*

The problems above are attributed to 3 types of dependencies:

1. **Sensing → Actuating**: Here, an actuation action is triggered based on a sensing variable. The actuation should not be triggered if the sensed value does not satisfy the required predicate.
2. **Sensing → App State Update**: Here, the application state is updated based on a sensed value.
3. **Actuating → App State Update**: Here, the application state is updated

based on actuatoin actions. (Example above where the state should indicate alarm was already triggered.)

In all these cases, the correct predicates should be checked before the dependent operation is performed to retain consistency.

14.4 Transactuations

To deal with these problems, a new concept is proposed called transactuation. Transactuations are a high level abstraction and a programming model, where the transaction is specified by:

1. **applicationLogic**
2. **sensingPolicy**:
   (a) The sensingPolicy can be expressed as a set of hard-coded values required for some of the sensors **sensorList**.
   (b) Can also include **timeWindow** that specifies the time when these values should be read and when the actions should take place
   (c) Can also specify whether all requirements need to be met or only some of them.
3. **actuatingPolicy**: Expresses the dependencies among updates to the application state and actuation of physical devices.
   (a) specify whether the updates are allowed when any or some or none of the device actuations have succeeded.
   (b) programmers can also specify the desired behavior of transactions on success (commit) or failure (abort)

This programming model allows us to describe the operations in the system as well as make some guarantees about the atomic durability of the actuations in the environment. It also provides sufficient information to schedule updates and thus avoid concurrency bugs.

Two important aspects of expressing transactuations are:

1. **Sensing Invariants**: specifies when transactuations can execute with respect to some property of the sensor values
   - Time upto which sensor values can be used - this can bound the staleness of the sensor values read.
   - How many failed sensors are there allowed to be in the environment, etc.
2. **Actuating Invariants**: Specifies when a transactuation commits its applica-

tion state updates.

  - Guarantee that a sufficient percentage of the actuations have already succeeded as per the specified actuation policy. E.g. at least one alarm must be turned on before the alarmState is updated.

These policies and invariants can be compiled to generate a runtime that can insert checks at the appropriate places to enforce these, decide whether a transaction can be committed, a new transaction can be started, etc., thus leading to Serializability among concurrent transactions.

- Transaction execution will start when the sensing policies are satisfied
- Ordering of the device actuations will be determined so as to avoid any rollbacks
- Only after the final commit is performed by the actuation policy, the internal state dependent on the actuations will be determined.

14.4.1 Evaluation of Transactuations

The paper also evaluates if the transactuations are useful at all.

Experimental Setup

- Evaluated with different applications
- Applications from different categories: convenience, energy efficiency, safety, security.
- Done by modifying original application implementation to add consistency cecks and then comparing with the re-implementation of the application using the transactuation programming system

Metrics used for this are:

- **Lines of Code**: Programmability metric
- **Performance impact on failure-free execution**: Do the runtime checks significantly slow down the execution
- **Correct behavior during failures**
- **Generalizability**: Done by testing on diverse IoT applications.

Results :

- **More compact implementation**: 2-3 times fewer lines of code (depends on sensing and actuation policies used)

- **Runtime performance impact**: Introduces average 50% overhead (In some cases, much lesser. Depends on the invariant checks being enforced)
- **Correctness**: Ensures all consistency requirements.

We may conclude that the overheads seen are acceptable given the programmability and correctness guarantees.

Index of Terms