# CS 7210 Distributed Computing Lecture Notes

Jie Wu, Jack

Spring 2023

# 1 Lesson 1: Introduction to Distributed Systems

## 1.1 What is a Distributed System

According to Lamport, a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable, where "you didn't even know" implies that there are multiple independent computers and they don't share all information with each other. Stated more formally a distributed system is a collection of computing units that interact by exchanging messages via an interconnecting network and appear to external users as a single coherent computing facility.

## 1.2 A Simple Model of a Distributed System

The simplest model of a distributed system consists of two or more nodes which are connected via unidirectional communication channels, and the nodes send and receive messages to and from others. The actions performed at each node is not crucial to be represented in order to characterize the node. They manifest themselves in some form of processing delay that is required to perform the actions. The only visible action to the distributed system is the message generated as a result of the action. This simple model can have the following implications

- taking the delay to infinity implies a node failure
- a message being delivered multiple times implies an unreliable communication channel
- a message being sent to multiple nodes simultaneously implies the existence of a multicast mechanism

## 1.3 A Slightly More Complex Model of a Distributed System

To account for the state change as a result of node action we can extend the above simple model by incorporating node state. The state of a node can change from one to another upon receipt of a message.

## 1.4 Importance of a Model

A model includes some system elements and rules. A model is also designed with some assumptions which translate to some model invariants that are always true for the model, e.g. the assumption that every message is delivered after some time implies that messages will not get lost or infinitely delayed, in other words we have lossless communication and the network will not fail.
A model is good if it is both accurate and tractable

**accurate:** the model can be used to accurately represent the problem

**tractable:** the model can be used to build and analyze a possible solution to the problem

The above simple model is a good model because

- it allows us to build basic algorithms considering simple scenarios, e.g. ensuring all nodes receiving all messages with the same information about the node's state
- that all messages will be delivered eventually can be realized by retransmission
- that no messages will be reordered can be enforced by using TCP
- no malicious actors implies that a node may crash but will never intentionally send incorrect messages, which can be enforced by adding cryptography to differentiate failures

## 1.5 What is Hard about Distributed Systems

**asynchrony:** for most real systems message deliveries are unpredictable and may even have infinite latency i.e. messages can be lost, thus a system design that assumes bounded latency will have limited applicability

**failure:** there are many different types of failures ranging from failstop, transient to Byzantine, moreover the failure may occur on an individual server or a process or a network link

**consistency:** we want to have a single and up-to-date copy of any data that is agreed on by all nodes, to achieve this we need to consider a lot of factors such as the concurrency and ordering of actions and the replication and caching of data

There are 8 fallacies about distributed systems which cannot be taken as an assumption for the design of any practical distributed system. They are

- transport cost is zero
- the latency is zero
- the bandwidth is infinite
- the network is reliable
- the network is secure
- the network topology doesn't change
- the network is homogeneous
- there is one single administrator

## 1.6 Properties of a Distributed System

We want a distributed system to always (high availability) give correct answers (consistency) regardless of failures or delay of a node or the network (tolerance to partitions). There are other desired properties of a distributed system listed in Google's distributed systems tutorial, which are

**fault-tolerant:** it can recover from component failures without performing incorrect actions

**highly available:** it can resume services even when some components have failed

**recoverable:** failed components can restart themselves and rejoin the system after the cause of failure has been repaired

**consistent:** it can coordinate actions by multiple components even in the presence of failure and different concurrency related ordering-related issues so that the system acts like a non-distributed system

**scalable:** it can operate correctly even as some aspect of the system is scaled to a larger size

**predictable performance:** the ability to provide desired responsiveness in a timely manner

**secure:** it authenticates access to data and services

## 1.7 Correctness

Having the same inputs refers to not only the values and/or parameters associated with the input events but also the timing and ordering of the input events. The guarantees we can make about the ordering that occurs in a distributed system is captured in the consistency model. In an ideal case we would like to make a guarantee that all events in the system have a single uniform order that is identical to the actual order in which these events occur in the real world. In addition we would like to make all nodes agree on this order. This is called the **strict consistency** and it is next to impossible to achieve in practice, because it is impossible for all nodes to have the same notion of time and the timing relationship of events. More practical models include

**linearizability:** reads and writes of shared states called transactions appear being executed in sequence instead of being interrupted or interleaved with some other transactions that are ongoing

**serializability:** all nodes agree on the ordering of the outputs due to some interleaving of the transactions that may not correspond to the real-time ordering of the individual transactions

## 1.8 The CAP Theorem

The theorem (actually hypothesis since it is not proved yet) states that a distributed system cannot meet all of the three desired properties

- if a network partition occurs (P), to guarantee consistency (C) the system may need to return errors or timeouts (lose A)
- if a network partition occurs (P), the system can continue providing responses (A) but some may return stale values (lose C)

Of course if there is no partition (P) the system can have both consistency (C) and availability (A). Systems are sometimes classified by the tradeoff of their design decisions made with respect to the consistency vs. availability, e.g.

**P + A:** Cassandra, DynamoDB

**P + C:** Megastore, MySQL

On the other hand there is an ever growing number of applications where a slow response is equivalent to no response. For such systems high latency is equivalent to no availability, and the tradeoff between availability and consistency becomes the tradeoff between latency and consistency. This led to the introduction of PACELC, which states as follows: 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).

# 2 Lesson 2: Primer on Remote Procedure Call

## 2.1 Client-Server Architecture

A common pattern for building distributed applications is that of client-server. A client send requests of data or processing to some server, the server retrieves the requested data or performs the requested processing and returns the result to the client. Protocols such as TCP requires that two endpoints establish a connection before they can send or receive messages. Other protocols such as UDP don't require an ahead of time connection. However even in that case a client needs to find and decide on the server that it will contact. This architecture also applies when the client and the server sit on the same machine, in which case the communication channel doesn't have to be based on a message-based network transport and can be based on shared memory.

## 2.2 Challenges in Client-Server

What is hard in client-server systems? The client needs to find and bind to the server. It needs to identify ahead of time the interface and parameter types for communication. The client and the server need to agree on the data representation. The operation parameters from the client and the result from the server need to be copied to and from network buffers so that they can be transmitted. The client needs to wait for the result potentially unknown amount of time. If the client suspects a failure it does not know what caused the failure. All these must be explicitly handled.

## 2.3 Role of RPC

As a high-level goal an remote procedure call or RPC system aims to hide the complexity of programming distributed systems, making it appear similar to local node programming. Procedures encapsulate some functionality and procedure calls pass arguments to procedures. The combination of these two observations led to the idea of remote procedure call or RPC.
More concretely the goals of RPC are to provide support

- for servers to register the services they provide and for clients to find out these services
- for connections between clients and servers
- for clients to find out the necessary service parameters and the expected results
- for clients and servers to agree on the data types of the parameters and results
- for data management including memory load and store, serialization (marshalling) and deserialization (unmarshalling)
- for dealing with failures e.g. retry for transient failures and error message for permanent failures

## 2.4 Architecture of an RPC System

In descending order of level the architecture of an RPC system consists of

**API:** for calling a remote procedure

**stubs:** part of the RPC runtime, instead of having the program counter jump to the address of the first instruction of the procedure, an RPC call results in a jump to the stub layer which contains the signature of the RPC and performs data marshalling

**RPC runtime:** responsible for connection management, sending and receiving data, dealing with failures etc.

The other components include

**interface definition language (IDL):** for interface specification, which is the language the server uses to describe the services it provides

**RPC compiler:** which automatically generates implementation code from the interface specification for operations called by the stub or RPC runtime

**service registry:** rules for announcing the services

The order the RPC system uses these components is that the server developer writes an specification of the RPC using the IDL → the specification is compiled by the RPC compiler → the compiler generates the implementation code → the server process is created by adding the implementation of the services and registering them with the registry, the client developer writes the client application referring to a remote operation using the specific interface and compiles the code together with the object files automatically generated by the RPC compiler, at runtime the RPC runtime takes care of everything else

## 2.5 Anatomy of an RPC Call

1. the client calls the `add` procedure
2. the program counter jumps to the client stub which builds a message
3. the message is sent by the client RPC runtime to the server
4. the message is received by the server OS and handed over to the server stub
5. the server stub unpacks the message and makes a call to the local `add` procedure
6. the server stub receives the result from the local procedure and packs it into a message
7. the message is sent by the server RPC runtime to the client
8. the message is received by the client OS and handed over to the client stub
9. the client stub unpacks the message and stores the result locally

## 2.6 Invocation Semantics of RPC Operations 1

We can distinguish RPC operations in multiple dimensions, one of which is

**synchronous:** the client makes a call and waits for the response

**asynchronous:** after making a call the client is free to do other things until the response arrives, if the client wants to be notified as soon as the response is available it can specify the operation to be performed when the response arrives, this is termed registering a **callback**

## 2.7 Invocation Semantics of RPC Operations 2

Another dimension that can distinguish RPC operations is based on the guarantees they make regarding the delivery of the RPC call

**exactly once semantics:** the client RPC runtime automatically performs retransmission upon time-out and the server must be able to eliminate duplicate requests, however this semantics cannot be met when there is a persistent failure

**at most once semantics:** the client transmits once and knows RPC calls may not be executed and can be programmed for that

**at least once semantics:** the client RPC runtime automatically performs retransmission upon time-out but the server does not guarantee that duplicate requests will be eliminated

## 2.8 Examples of RPC Systems

Sun RPC, SOAP, CORBA, Apache Thrift, gRPC

## 2.9 Examples of RPC Systems: gRPC

gRPC relies on protocol buffers to describe the interface and the data types. In gRPC the interface specification is stored in the .proto file which is compiled by the protoc compiler. Each element in the input/output data types is identified by its number.

# 3 Lesson 3: Time in Distributed Systems

## 3.1 Why Do We Need Time

We need time to determine ordering, which is needed to determine causality, perform scheduling, and determine progress measure for garbage collection.

## 3.2 Why is Measuring Time Hard in DS

The reasons are

- guaranteeing consistent availability of globally synchronized clocks is hard
- transmission time is not fixed for all messages and connections
- network delay would not be constant across nodes
- either nodes or the network may fail
- there may be malicious nodes in the network which send out false information

## 3.3 Logical Time

Logical clocks don't measure the real-world time, it simply is a generator for generating timestamps which advances in some manner and can be used to order events. The paper discusses three types of logical clocks: (1) scalar/Lamport's clock (2) vector clock (3) matrix clock.

## 3.4 Common Notations

The execution of a process $p_i$ is a sequence of events $e_i^0$, $e_i^1$, ..., $e_i^k$, $e_i^{k+1}$, ..., $e_i^n$. Each $e_i^k \rightarrow e_i^{k+j}$ $\forall j \geq 1$ where the arrow "$\rightarrow$" represents the "happens before" relationship. This ordered sequence of events is called the **history** $H_i$ of process $p_i$. We will focus on events related to sending and receiving messages, in particular we have $\text{send}_i(m) \rightarrow \text{recv}_j(m)$.
We can also use the *time diagram* of a distributed system to represent ordered sequences of events, in which dots are events and arrows point from send events to receive events, dots that don't have arrows connected are internal events.

## 3.5 Concurrent Events

If two events involve different, completely unrelated processes denoted by $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$, they are **concurrent events** denoted by $e_1 \parallel e_2$.

## 3.6 Logical Clock

For each event $e_1$ logical clock $\mathcal{C}$ produces a timestamp $\mathcal{C}(e_1)$. The timestamp must satisfy the following clock consistency condition

**monotonicity:** $e_1 \rightarrow e_2$ implies $\mathcal{C}(e_1) < \mathcal{C}(e_2)$, however $\mathcal{C}(e_1)$ and $\mathcal{C}(e_2)$ can be of any relationship for $e_1 \parallel e_2$

However the timestamp produced by a real clock satisfies the following strong clock consistency condition: $e_1 \rightarrow e_2$ implies $\mathcal{C}(e_1) < \mathcal{C}(e_2)$ and $\mathcal{C}(e_1) < \mathcal{C}(e_2)$ implies $e_1 \rightarrow e_2$.
For any event in a distributed system logical clock produces a timestamp, a value in time domain, and thereby maps an event history of a process to a partially ordered sequence in time domain. It is partially ordered because events may be concurrent. A clock function $\mathcal{C}$ also needs to define a set of rules to advance timestamps.

## 3.7 Lamport's Scalar Clock

A Lamport's scalar clock $\mathcal{C}_i$ associated with process $p_i$ produces scalar timestamp values. Each node has its own implementation of the clock, and knows only the value of the timestamps it produces i.e. each node has its own view of current time. The rule of advancing the timestamp is that

- for two events $a$ and $b$ in the same process, $a \rightarrow b$ implies $\mathcal{C}_i(a) < \mathcal{C}_i(b)$, thus a process has to increment the value of its logical clock each time it generates a new event
- for the message sending event $a$ and receiving event $b$ from $p_i$ to $p_j$, $a \rightarrow b$ implies $\mathcal{C}_i(a) < \mathcal{C}_j(b)$, thus $\mathcal{C}_j(b) = \max(T_m, \mathcal{C}_j)$ where $T_m = \mathcal{C}_i(a)$ i.e. $\mathcal{C}_j(b)$ must be greater than not only the message sending timestamp but also all the timestamps used by process $p_j$ to enforce the first rule

Lamport clock is only consistent not strongly consistent. It can be used to establish partial ordering but not causality, because there can be concurrent events and Lamport clock imposes no rule on them. To produce a total order we need to introduce additional tie breaking rule for concurrent events e.g. based

on the order of process IDs. Not supporting strong consistency does not render the clock incorrect. It only means there is some loss of efficiency, because additional ordering effort is required if we need to order some unrelated events.
Lamport clock has some other useful properties, e.g. it can be used to estimate the number of events, because if the timestamp increment is always 1 then local clock produces the minimum number of preceding events in the process.

## 3.8 Vector Clock

The timestamp produced by a vector clock is a vector $vt_i$ with the $i$th element $vt_i[i] = p_i$'s Lamport clock $\mathcal{C}_i$ and the $j$th element $vt_i[j] = p_i$'s knowledge about $p_j$'s Lamport clock $\mathcal{C}_j$. As with scalar clock each node maintains its own view of current time. A vector clock advances its timestamp according to the following two rules:

**R1 (events on the same process):** before executing an event $\mathcal{C}_i$ updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \qquad d > 0$$

**R2 (events on different processes):** each message $m$ is piggybacked with the vector clock $vt$ of the sender process at sending time, on the receipt of such a message $(m, vt)$, process $p_i$ executes the following sequence of actions:

1. update each element of its vector timestamp to be the maximum of the value of that element and the value of the same element in the vector timestamp piggybacked in the message, i.e. $vt_i[k] := \max(vt_i[k], vt[k])\ \forall 1 \leq k \leq n$
2. execute R1
3. deliver the message $m$

The vector timestamp comparison rule is as follows:

- $vt_1 \leq vt_2$ if and only if $vt_1[i] \leq vt_2[i]\ \forall i$
- $vt_1 < vt_2$ if and only if $vt_1 \leq vt_2$ and $\exists k$ such that $vt_1[k] < vt_2[k] \Rightarrow$ consistency condition uses this relationship
- $vt_1 \parallel vt_2$ if and only if neither $vt_1 < vt_2$ nor $vt_2 < vt_1$ is true

Like Lamport clock vector clock satisfies the consistency condition $a \rightarrow b$ implies $vt(a) < vt(b)$. But it also satisfies the strong consistency condition $vt(a) \parallel vt(b)$ implies $a \parallel b$. The efficiency is achieved at the expense of larger clock size $O(N)$ where $N$ is the number of nodes in the system.

## 3.9 Matrix Clock

The timestamp produced by a matrix clock is a matrix $mt_i$ with the $(i, i)$th element $mt_i[i, i] = p_i$'s Lamport clock $\mathcal{C}_i$ hence the diagonal of $mt_i$ is simply the vector clock of $p_i$, and the $(k, l)$th element $mt_i[k, l] = p_i$'s knowledge about $p_k$'s knowledge about $p_l$'s Lamport clock $\mathcal{C}_l$ thus each process maintains its view about every other process's view of the global time. The additional complexity allows garbage collection, because if $\min(mt_i[k, l]) > t$ then every process knows that all processes have advanced beyond time $t$.

# 4 Lesson 4: State in Distributed Systems

## 4.1 Global State, Snapshots, and Other Terminology

The state of a distributed system is a collection of the states of its nodes and the states of the communication channels that connect them. Each message sending or receiving event affects one node and one channel, e.g. a message receiving event reduces one inflight message in the channel and increases one message received by the destination node. Each state transition corresponds to an event, and a sequence of events constitute a **run**. The sequence may correspond to the events that indeed happened in the system which we call an *actual run*. However it is possible that some event that we are unaware of happened in which case the ordering may not be the real one and the run is an *observed run*.
A **consistent cut** of a time diagram of a distributed system corresponds to a snapshot of the execution which may not correspond to an actual run. All events prior to a consistent cut are called *prerecording* events and all events after are called *postrecording* events. One example of inconsistent cuts is that a message receiving is a prerecording event while the corresponding message sending is a postrecording event.

## 4.2 Challenges about State in Distributed Systems

The reason why determining a global state of a distributed system is hard is first and the foremost that we don't have a globally synchronized clock. In addition because of the random network delay we can't assume that nodes can instantaneously tell each other to capture their states. Furthermore multiple events can take place concurrently which makes the distributed system a non-deterministic system.

## 4.3 System Model

For the remainder of this lesson we will work on the following model of distributed systems—there are processes that exchange messages via directed, FIFO, and error-free channels

**directed:** the channel from $p$ to $q$ is separated from the channel from $q$ to $p$

**FIFO:** messages are delivered to the destination in the same order in which they are sent

**error-free:** there will be no corruption on the messages

These assumptions can be fairly well met by a TCP/IP channel.

## 4.4 Finding a Consistent Cut: Algorithm in Action

$\Rightarrow$ To determine the global state I suggest treating the first message marker arrow as a consistent cut and examine the prerecording states for the nodes right before the cut: for $q$ it is $S_q^1$ and for $p$ it is $S_p^2$. As for the channel states since $S_p^1$ precedes $S_p^2$ the message $m_1$ has been delivered and hence there is no message in the $q$-to-$p$ channel. On the other hand $S_p^2$ indicates that the message $m_3$ has been sent, yet because $S_q^1$ precedes $S_q^3$ $m_3$ has not been received by $q$. Therefore there is one message $m_3$ in the $p$-to-$q$ channel.

## 4.5 Snapshot Algorithm

One node is the initiator node which triggers the algorithm for capturing the global state of the system. It first saves its local state and then sends marker on all outgoing channels. All other nodes participate in the algorithm according to the following rules—upon receiving the first marker on any incoming channel they

- save their local states, mark the state of the incoming channel as empty, and propagate the markers on all outgoing channels
- resume normal execution, but also save incoming messages until a marker arrives through the channel, all these messages will be marked as messages inflight, the pseudocode is

```
if p has not recorded its local state
    record its local state
    channel c state = empty
else
    channel c state = message received on channel c
```

## 4.6 Global State

The above snapshot algorithm is the Chandy-Lamport algorithm for finding global snapshots in a distributed system. The state captured by this algorithm may not correspond to any real global state the system currently is in or has been in before, but it is a consistent global state and corresponds to a possible global state that can be derived from permutation on other possible global states including the real global state (permutation is OK as long as it does not break the casual relation).

## 4.7 Properties of a Global State

If the state recorded is $S^*$, the sequence of distributed computations done by the system is $seq$, the true initial and final states of the system are $S_i$ and $S_f$, then $S^*$ is a **consistent global state** if

- $S^*$ is reachable from $S_i$ and $S_f$ is reachable from $S^*$
- there exists a sequence of distributed computations $seq^*$ which is a permutation of $seq$ such that
  1. either $S^* = S_i$ or $S_i$ occurs before $S^*$ in $seq^*$
  2. either $S^* = S_f$ or $S^*$ occurs before $S_f$ in $seq^*$

## 4.8 Benefits of Global State: Evaluate Stable Properties

Even if the system wasn't ever in that state, a possible global state allows us to evaluate stable properties of a system. A property is called **stable** if and only if if it becomes true in a state $S$ it remains true for all states reachable from $S$ such as deadlock, termination, token loss. Formally since $S^*$ is reachable from $S_i$ and $S_f$ is reachable from $S^*$

- if a stable property is true in $S^*$, it is also true in $S_f$
- if a stable property is false in $S^*$, it is also false in $S_i$

## 4.9 Definite vs. Possible State

An **unstable** property is a property for which there is no guarantee that once it becomes true it remains true forever such as buffer overflow, load spike, race condition. If we observe that an unstable property is true in some possible global state $S^*$ then this property can possibly be true under some condition in the system, which suggests prevention precautions.

# 5 Lesson 5: Consensus in Distributed Systems

## 5.1 What is Consensus

A **consensus** is the ability of multiple distributed processes to reach an agreement on something such as a value, an action, a timestamp etc. The ability to reach a consensus is a key ingredient in making sure that a distributed system is going to behave correctly. There are a number of factors that make it hard for nodes to reach a consensus such as non-determinism, lack of global clock, uncertain network delays, malicious behavior etc.
The ability of a distributed system to reach a consensus implies three key properties

**liveness/termination:** all non-faulty processes eventually decide on a value

**safety/correctness:** all processes decide on a single value

**validity/safety:** the value that is decided on must have been proposed by some process in the system

## 5.2 Preliminaries: System Model

The system model used by the FLP theorem has the following characteristics

- we have an asynchronous system i.e. messages may be reordered and delayed but not corrupted
- there is at most one faulty process
- the only possible type of failure is fail-stop—a node fails by stop working, which is indistinguishable from an infinite message delay

If consensus is not possible in this simple system, then it will not be possible in a system where messages can be corrupted, more than one process can be faulty, or Byzantine failure is possible.

## 5.3 Preliminaries: Definitions

**admissible run:** a run with one faulty process and all messages eventually delivered

**deciding run:** an admissible run where some non-faulty processes reach a decision

**totally correct consensus protocol:** a protocol where all admissible runs are also deciding runs

**univalent configuration:** a configuration in which the system can reach a single value (deciding)

**bivalent configuration:** a configuration in which at least two decisions are possible (non-deciding)

## 5.4 FLP Theorem

The theorem states that in a system with one fault, no consensus protocol can be totally correct. The idea of the proof is as follows: consider a system with asynchronous communication, one faulty process, and fail-stop model, is it possible to identify a starting configuration and an admissible run where the system will not reach a deciding state, in other words can we have at least one admissible run which is not a deciding run (e.g. the system remains in a bivalent configuration)?

## 5.5 Proof in a Nutshell

Start with a sysetm where nodes are capable of making one or two decisions 0 or 1, one faulty node is possible and messages may be delayed and reordered. The three arguments are

1. (Lemma 2) based on such a simple system model there exists some initial configuration for which the final decision is not predetermined but depends on the schedule of events, hence there must exist an initial bivalent configuration
   this can be proved by contradiction—assume that all initial states have predetermined outcomes, describe these configurations by the 0 or 1 of their initial states, we are guaranteed to find two configurations which differ in the state value at one single process, so that if that process is faulty then the admissible runs which get executed in these two configurations are the same since the faulty process would not contribute to the runs, in such a scenario starting from the same initial configuration we can reach either 0 or 1 i.e. the final decision is not predetermined
   also if the system only has predetermined states, then it is trivial to prove that it won't be able to reach consensus, because consensus requires that the agreed-upon value is proposed by one of the processes

2. there exists a single event such as a message exchange that makes the system transition from a bivalent state to a univalent state

3. it is possible to delay or reorder messages in such a way that the message that triggers the bivalent-to-univalent transition is sufficiently delayed beyond the run of the admissible schedule, which means that the system will never transition from a bivalent undecisive state to a univalent decisive state, therefore an admissible non-deciding run can exist

## 5.6 Is Consensus Really Impossible

There exist a few consensus protocols such as 2-phase commit, 3-phase commit, Paxos, Raft. They do not contradict with the FLP theorem. Instead they change some of the assumptions or properties of the system model used by the theorem.

# 6 Lesson 6: Replication

## 6.1 Goal of Replication

With replication the system maintains the same state at more than one location. The state can be files, database, application level state or operation system level execution state associated with virtual machines. Having the state available at more than one location allows multiple nodes to provide the same service.
There are multiple reasons why replication is useful, one is fault tolerance which ensures that the same service can continue to be delivered even when there is a permanent or transient failure. For example a common technique that enterprises use is to replicate a VM to a faraway data center as part of the disaster recovery mechanism. Another reason for replication is to improve scalability. For example by replicating files on multiple machines requests can be distributed.

## 6.2 Replication Models

There are two main replication models

**active replication:** each node is active and can handle requests, a replica that receives an update request must ensure that the update is appropriately replicated to all other replicas

**standby/primary-backup replication:** only one replica is active at a time and serves requests which we call the *primary*, and all other replicas are on standby, when the active replica fails some other replica must intervene and take over as the primary, the primary must ensure that the update is appropriately replicated to all standby replicas

## 6.3 Replication Techniques

There are two main replication techniques

**state replication:** update operation is executed on one replica and then the modified state is replicated to other replicas

- **pro:** no need to re-execute the same update multiple times
- **con:** state may be large thus costly to replicate or hard to identify when updates are spread out all over the replicas

**replicated state machine:** update operation is replicated to and executed on all replicas to produce the same modified state, this requires deterministic operation

- **pro:** operation logs are much smaller to replicate than states
- **con:** the same update must be executed multiple times and is only applicable to deterministic executions

Both techniques can be used with either active or primary-backup replications.

## 6.4 Replication and Consensus

We need consensus to ensure that each update is properly reflected on all replicas. For primary-backup replication the primary obviously is the leader/coordinator; for active replication any replica can be designated as the leader. The ordering and visibility of updates depend on the consistency model used and the granularity of the update executions (individual or grouped).

## 6.5 Chain Replication

Regardless of what consensus protocol we use there are at least two rounds of message exchange among the leader and the participants. This means that as we add more replicas the response time will increase with the number of replicas, not only because the response has to wait for more round trip times from other replicas to be completed but also because each replica that handles the request needs to send and receive more messages which reduces the capacity for other processing. Hence the cost of consensus is proportional to the number of nodes.
One solution to address this scalability problem is chain replication. In chain replication the first node is called the head and the last the tail. Write requests are always sent to the head. When the head receives some write request it only replicates the request to the next replica in the chain. Each node in the chain will in turn update only its subsequent node until the tail is reached. When the write request reaches the tail it acknowledges the write. Read requests are always served by the tail to guarantee that the read request always see the latest committed update. One reason for not allowing intermediate nodes to serve read requests is that a write may not reach the tail to be committed due to the failure of the tail etc. and be discarded, and serving read requests from intermediate nodes may expose the read to such an uncommitted write.
The pro and con of chain replication are

**pro:** there are three

**great leader scalability:** as the head only updates the next replica in chain

**high write throughput:** through pipelining—a second write can be accepted while the first is pushed through the chain

**strong consistency:** reads can be guaranteed to return only committed writes

**con:** there are two

**poor read-heavy scalability:** as only the tail serves read requests

**inefficiency:** intermediate nodes serve neither reads nor writes thus are underutilized

## 6.6 CRAQ

CRAQ stands for chain replication with apportioned queries. It builds on the original chain replication and makes the following modifications

- reads are divided among the nodes in the chain
- keep multiple versions of the data at each node—the new uncommitted value is marked and the old committed value is not discarded
- before the new value is committed intermediate nodes serve the old committed value, after the new value is committed intermediate nodes serve the new value and discard the old value, this ensures sequential consistency

## 6.7 Summary

The design choice depends on

**workload:** whether it is read-intensive or write-intensive

**system configuration:** number of nodes, failure rate, network bandwidth, round-trip time

**consistency requirement:** failure type, fault-tolerance method

# 7 Lesson 7: Fault Tolerance

## 7.1 Some Taxonomy

The difference between fault, error, and failure is

**fault:** faulty hardware or software bug, the system can function correctly if the fault is not activated

**error:** once a fault is activated it leads to error—incorrect behavior, incorrect information etc.

**failure:** when the error propagates through the system it causes failure

We can classify faults by duration

**transient:** occur once and then disappear

**intermittent:** recur occasionally

**permanent:** persist until the fault is removed

Faults can lead to many different types of failure

**fail-stop:** fail silently and stop working

**timing:** behave outside of timing expectations leading to timeouts

**omission:** fail to send all messages or fail to receive all messages

**Byzantine:** arbitrary behavior e.g. malicious

Options to manage failure are

**avoidance:** consider all possible outcomes, rely on predictability thus impractical

**detection:** using heartbeat signals or error correction codes

**removal:** roll back to the point of execution before the failure, possible for transient failure but impossible if real world is impacted

**recovery:** correct execution despite the occurrence of failure, such systems are called **fault tolerant**

## 7.2 Rollback-Recovery Idea

The basic idea of rollback-recovery based fault tolerance is that, when a failure is detected the system rolls back to a previous state which we know is correct, and then continues from that point to re-execute the operations with the fault removed. By rollback we mean rolling back (1) any effects of exchanged messages and (2) any updates to internal states. The state that the system rolls back to may not be an actual state that the system has ever been in during its previous execution. It just needs to correspond to some consistent cut i.e. some possible global state. We can potentially roll back all the way to the beginning but then a lot of work would be lost. Hence instead we rely on two basic mechanisms to define a state to roll back to—checkpoint based and log based.
The granularity at which fault-tolerant systems operate differs

**transparent/full-system:** implemented at the system level, rollback mainly concerns about effects of exchanged messages

- **pro:** require no modification to applications
- **con:** very high overheads

**transaction-based:** use special transactional APIs which group sets of related operations, the system will then ensure that the transactions are performed atomically, the transaction itself may be distributed and may be performed by more than one node

- **pro:** overhead reduced to groups of related operations
- **con:** rely on transactional API and require modification to applications

**application-specific:** e.g. HPC

- **pro:** applications know best what state is needed for recovery
- **con:** limited applicability

In the remainder of this lesson we will focus on the transparent method.

## 7.3 Basic Mechanisms

A checkpoint corresponds to saving the state of a process to some persistent storage. The pro and con of checkpoint-based rollback-recovery is that

**pro:** it can restart instantaneously

**con:** the amount of I/O on checkpoint is significant, which can be improved via application-specific checkpoint or delta compression etc.

In a log-based rollback-recovery only the log information about the operations executed is saved. The log information may include the original values of the affected variables. We call this *undo log*. The log information may include the new values instead and the log is used during recovery to redo the changes. We call this *redo log*. The pro and con of log-based rollback-recovery is that

**pro:** much smaller amount of I/O on log

**con:** recovery takes longer, as we need to examine the log and figure out the updates to undo or redo etc. in addition even regular operations may take longer since they have to look through the log to find out the most recent values of dependents for execution

The two systems can be combined so that each node periodically performs checkpointing and between checkpoints it logs updates. Note that sending a message is also considered an update thus is also an operation that needs to be logged. The pro and con of this combined mechanism is that

**pro:** reduce the duration of recovery, reduce required storage

**con:** must detect consistent cut

## 7.4 Checkpointing Approaches

There can be several ways for processes to decide when to take a checkpoint: (1) uncoordinated (2) coordinated (3) communication-induced. We will consider a model with

- a fixed number of processes
- communication between them is carried out only through messages
- processes interact with the outside world
- network is non-partitionable, but other assumptions vary across different protocols such as FIFO, reliable communication channel etc.
- number of tolerable failures vary across protocols

## 7.5 Uncoordinated Checkpointing

In this approach processes take checkpoints independently, which means that we need to construct a consistent state upon failure, in particular any message sent after the checkpoint is considered lost and we need to remove all of its effects from the system for calculating the recovery line. For this purpose we need to store dependency information associated with all messages. Uncoordinated checkpointing suffers from a very serious **domino effect**, which is that a consistent cut is not reached until we roll back to the beginning of the execution.
Other than the domino effect there are other issues associated with uncoordinated checkpointing

**useless checkpoints:** many checkpoints will never become part of a consistent cut thus the effort of saving them is wasted

**multiple checkpoints:** since we don't know how far we need to roll back to reach a consistent cut, each process must maintain multiple checkpoints instead of just the most recent snapshot

**garbage collection:** to cope with the storage requirement for excessive checkpoints we have to run garbage collection periodically to clean up obsolete checkpoints

## 7.6 Coordinated Checkpointing

In coordinated checkpointing processes coordinate when they take a checkpoint to ensure that the checkpoints taken are part of a consistent state. The benefits are

- recovery no longer requires maintaining a dependency graph to calculate the recovery line
- there is no longer the danger of domino effect
- one single checkpoint per process is sufficient
- garbage collection is trivial as all earlier checkpoints can be discarded

The challenges of coordinated checkpointing are

- how to coordinate among various processes to agree upon when to taken a checkpoint
- there is no synchronous clock to guarantee the consistency of checkpoints
- message delivery is not reliable nor in bounded time, which makes coordination through message exchange nontrivial
- having an initiator sending checkpointing requests to all other nodes may lead to many processes having to create a checkpoint even when there is no relevant change in their states, therefore the problem of taking unnecessary checkpoints is not completely eliminated via coordination

## 7.7 Communication-Induced Checkpoints

One way to ensure that the nodes properly coordinate is to use a consensus protocol such as 2-phase commit. Here the consensus the nodes are trying to reach is whether they should take a snapshot at a particular time. This coordination is blocking in the sense that when a process receives a checkpointing request it blocks all underlying applications, and no message should be processed from the moment the coordination is initiated until it completes.
Non-blocking alternative exists such as the Chandy-Lamport snapshot algorithm in Lesson 4. The FIFO requirement of the algorithm can be removed by piggybacking the marker message in a message. To make sure that we have recent checkpoints of the nodes that may have not communicated with others for some time, periodic independent checkpoints are encouraged. In between such independent checkpoints each node monitors incoming messages, and if they contain a marker a checkpoint is taken. In fact a snapshot is also captured just before processing the message that carries a marker so as to prevent the scenario in which a message sending is a postrecording event whereas the corresponding message receiving is a prerecording event.

## 7.8 Logging

Compared to checkpointing logging saves the amount of I/O but requires more complex recovery, because the log needs to be used to first roll back and then re-execute operations. Moreover logs must be created such that they specify deterministically a valid execution of the system and they don't lead to any orphaned event. There are several approaches to achieve this

**pessimistic logging:** each process should log everything to persistent storage before allowing events to propagate and commit, which however incurs high overhead

**optimistic logging:** assume that the log will be persistent before failure and it is possible to remove the effects of an operation in case it needs to be aborted, which cannot be guaranteed thus requires additional mechanism to track dependencies, moreover for any operation that will lead to some externally visible events which cannot simply be undone, the output of such an operation has to be delayed until the system is sure that the operation has been persisted and will not be aborted

**causality tracking:** operate optimistically whenever there are no dependencies, ensure causally related events are deterministically recorded, and guarantee the property of not having orphaned events in a way similar to pessimistic logging, since causality tracking relies on message exchange we need to ensure that there are no inflight messages carrying causality related information

## 7.9 Which Method to Use

The design choice of fault tolerance solution depends on

**workload characteristics:** how often the data is updated? how big is the update? how many nodes are involved in a single application level transaction? is fast recovery important?

**failure characteristics:** what kinds of failure that need to be tolerated? what is the failure rate?

**system characteristics:** cost and overhead of communication vs. storage, system scale

Note that thanks to technology advancement some of the assumptions of the paper no longer hold today since we now have faster and more reliable networks and faster persistent storage. On the other hand coordinated checkpointing, which is the default strategy of HPC and is among the most favorable across most of the metrics, are now facing the challenge of global coordination overhead across continents.

# 8 Lesson 8: Paxos and Friends

## 8.1 Goal of Consensus Protocol

Reaching a consensus in a distributed system means all processes can agree on the value of a shared state and the value they agree on is proposed by some process and thus is valid. To do so we need to consider different types of role for the processes

**proposer:** propose value for the shared state

**acceptor:** evaluate proposals and decide which proposed value to choose

**learner:** access and read the chosen value for the shared state

A consensus protocol must guarantee safety and liveness

**safety:** only a value that has been proposed is chosen, only a single value is chosen, only the chosen value is learned by the learners

**liveness:** some proposed value will ultimately be chosen and the chosen value will ultimately be learned

## 8.2 2PC and 3PC

Two protocols originating from the database community are examples of consensus protocols. They are 2-phase commit (2PC) and 3-phase commit (3PC).
In the 2-phase commit protocol there is always a coordinator which is assumed to never fail. The two phases are

**vote collection phase:** the coordinator proposes a value, sends out a vote request to all participants, and collects votes from them

**decision phase:** the coordinator makes a decision, sends it out to all participants, and collects acknowledgment from them, upon which it commits the decision

The 2PC protocol blocks if there are failures thus it does not guarantee liveness.
The 3-phase protocol addresses this blocking problem by adding a pre-prepare phase at the beginning followed by the actual prepare phase when the votes are solicited and then the decision phase. If a node fails this protocol won't block all nodes thus it guarantees liveness. However it assumes fail-stop mode—if a node fails it won't restart, or if a message is delayed beyond the timeout it will not be ultimately delivered. It does not guarantee safety if the assumption fails.

## 8.3 Paxos Made Simple

Paxos is designed for systems with asynchronous communication and non-Byzantine process failures. The agents in the algorithm operate at arbitrary speed, may fail by stopping and may restart. Each participant has some persistent memory to record information after restart. Messages can take arbitrarily long to be delivered, can be duplicated, lost, or reordered, but cannot be corrupted. The underpinning ideas of Paxos are

**state machine replication:** each node is a replica of the same state machine following the same rules

**majority quorum:** each decision is based on a majority quorum, two quorums are guaranteed to have intersecting members, thus consensus decision can be safely disseminated even when some node failed and did not participate in a consensus round, because after restart it can catch up from the intersecting member which has learned about the past agreements, this is necessary to tolerate fail-stop and fail-restart failures

**ordering:** everything is timestamped so that it can be ordered, this is necessary to tolerate arbitrary message delays

## 8.4 Paxos Made Simple: Phases

The protocol has three phases

**prepare:** a proposer makes a proposal to the rest of participants for reaching a consensus, the propose message is timestamped by the order number of the proposal so that newer proposals can be distinguished from the old ones

**accept:** the proposer gathers votes from the participants, which communicate through their votes whether they are willing to commit to agreeing on the proposal with the same proposal number, if a node has already agreed to some value then the vote will include the value the node has committed to agreeing on together with the timestamp of the proposal they agreed on

**learn:** once the proposer has received a majority quorum of commitment in the $n$th round, it communicates the agreed upon value along with the round number of the proposal to the participants, if the proposer receives an acceptance from a quorum number of participants it knows that the agreement has been reached

It is possible to learn in two message rounds. The proposal number part of the messages not only allows the participants to order proposals but also solves the problem of fail-restart and delayed messages.

## 8.5 Paxos: Prepare Phase

**proposer:** each consensus round in Paxos is driven by an initiator called proposer, the proposer selects a proposal number $n$ and sends the *prepare request* with the number $n$ to a majority of acceptors, the number $n$ is a member of a set that is totally ordered over all processes, i.e. no two processes use the same $n$ and a process never uses $n$ twice, any acceptor can become a proposer

**acceptor:** if an acceptor receives a prepare request with $n$ which is greater than that of any prepare request it has accepted, then it responds with the promise not to accept any more proposals with number $< n$; if it has accepted a higher number proposal then it responds with that number so that the proposal has to use an even higher number for future proposals

**learner:** there are other nodes called learners which do not participate in the above mentioned proposal agreement, they will learn the agreement some time in the future by contacting a sufficient number of acceptors

Paxos is designed to work correctly even when nodes fail and then restart to rejoin the system. To achieve correctness every decision involves a confirmation from a majority quorum—$N/2 + 1$ if there are $N$ nodes in the system.

## 8.6 Paxos: Accept Phase

**proposer:** once a proposer hears back commitment from a majority of acceptors, it sends an *accept request* to each of those acceptors containing a value and the proposal number $n$, if based on the responses the proposer learns that there was a value that was already agreed upon it will communicate this value, otherwise it is free to choose whichever value it wants to propose

**acceptor:** if an acceptor receives an accept request for a proposal numbered $n$, it accepts the proposal unless it has already responded to a prepare request with a number $> n$

## 8.7 Paxos: Learn Phase

The prepare and accept phase constitute the write side of the protocol while the learn phase constitute the read side of the protocol.

**acceptor:** when an acceptor receives a *commit message* it knows the accepted value has become the decided value for the proposal number $n$, since to deal with fail-restart the read has to be done by a majority quorum as well and each acceptor needs to send the decided value to a learner

**learner:** it is inefficient for each acceptor to send the decided value to each learner, a better way is to designate a distinguished learner to receive the decided value from all acceptors, once the distinguished learner receives *decision message* from a majority of the acceptors it can inform the other learners of the decided value, greater reliability can be achieved at the expense of greater communication complexity by designating a set rather than one distinguished learners

The separation of nodes involved in reads and writes is a common technique in distributed systems, because if all nodes participate in both reads and writes we end up with slowing down both reads and writes.

## 8.8 Corner Cases

There can be two proposers proposing at the same time. The way a proposer chooses a proposal ID is that it combines some local counter with a node identifier, and this is how the protocol ensures that the two proposals will have different proposal IDs even though they are proposed at the same time. Note that even though the newer proposal may be accepted after the old one, the decided value of the old proposal remains in record.

## 8.9 Paxos vs. FLP

Paxos does not contradict with FLP because it does not guarantee liveness—two proposers can keep issuing a sequence of proposals with increasing proposal numbers, one overthrowing the acceptance of the other consecutively. Workarounds include random delays in re-proposing, designating a distinguished proposer with timeout check for failure detection. All these make it unlikely not to achieve liveness but they do not make it impossible. Thus FLP still holds.

## 8.10 Multi-Paxos

Paxos allows the system to agree on a single value. In practice program execution requires a sequence of updates. Multiple Paxos protocols or multi-Paxos need to be executed for agreeing on the order and value of the sequence of updates. In multi-Paxos each single-decree Paxos is used for agreeing on an individual value. There are a number of optimizations that address the increased complexity of multi-Paxos, one of which is to separate the stages of the execution termed "views" where only one proposer is allowed to propose for a particular view, and all values in that view get accepted and learned per Paxos. This proposer leader is valid only for currently active group of participants. If nodes join or fail it may be necessary to re-elect a new proposer leader. There is also a mechanism to detect whether the view also needs to be changed. This protocol is very similar to another consensus protocol known as viewstamp replication or VR.

## 8.11 Paxos in Practice

The first known implementation is Lamport's DEC SRC. More recent famous variants are Google Chubby service and Zookeeper atomic broadcast (ZAB) in Zookeeper

## 8.12 RAFT

The main motivation behind looking for additional consensus protocol even though Paxos is proven to be practical is understandability. We want a protocol that is sufficiently simple so that practical implementation can be accomplished per specification. However for Paxos there is a lot of complexity even in a single agreement, which gets trickier as one adds optimization and uses multi-Paxos.

## 8.13 RAFT Overview

Unlike Paxos RAFT has a distinct **leader election** phase. Like in Paxos any node can be a candidate for a leader but only the node which receives the most votes is elected and the rest are followers. After a leader is elected the normal operation phase starts which is that of **log replication**. During the log

replication phase the leader makes proposals for updates and replicates this information among the followers. Each time a new leader is elected RAFT refers to it as a new **term**. A leader can be active for an arbitrary duration and arbitrary number of updates. By explicitly separating the leader election phase from the log replication phase RAFT makes it easier to reason about the behavior of the system.

## 8.14 RAFT Leader Election

All followers exchange heartbeat messages with the leader to know that the leader is active. If any follower times out i.e. does not receive a heartbeat message from the leader for some time, it will assume the leader has failed and it will start a leader election phase. Any server that is applying to be a candidate for a leader will send a message to all nodes, which includes information about its current term number and the index of the most recent event in its log. All nodes in the system have the right to vote and a leader is elected based on majority votes.
RAFT follows a set of rules which guarantee the following election safety property

**property #1 at most one leader per term:** there is at most one leader in any term

Its leader election rules are:

1. a leader is elected by majority votes
2. outdated leaders are prevented from being elected, which is enforced by the rule that a server can only vote a candidate when that candidate has a newer log, newer in the sense that the candidate has a higher term number or the same term number but a longer log, through this vote the losing candidates will learn that their log is outdated and have the necessary information to catch up
3. when there is a tie in the election, to ensure liveness RAFT adopts random timeout for each server similar to Paxos to restart the election phase

## 8.15 RAFT Log Replication

Each log entry contains the operation and the term number and is identified by the log index. The leader appends update requests from clients to the end of its log and then pushes the new logs to followers. The information that is exchanged during heartbeats allows outdated followers to catch up. Failed and restarted nodes can easily retrieve the missing logs from the leader to catch up. To replicate log entries the following steps are performed:

1. the leader pushes the new log entry along with its previous logs to followers during heartbeats
2. each follower checks whether it has the previous logs, and if yes accepts the new log entry and sends acknowledgment to the leader
3. after the leader receives acknowledgment from a majority of followers, it commits the new log entry and notifies the client

In this way the protocol will guarantee that there is a consensus on whether the update is successfully applied, which value corresponds to the update, and how to order updates from clients. The following two properties guarantee the correctness of the execution during log replication:

**property #2 append-only update:** the log of the leader is append-only

**property #3 log matching:** if two entries in different logs share the same index and term number, then this entry and all the previous entries are the same

It is possible to have a leader failure before committing some log entries, in which case the new leader forces all other nodes to use its log, and the uncommitted logs that the new leader is unaware of may then be discarded. This is safe because the leader election rule guarantees that the new leader knows all the committed logs. Receiving no acknowledgment from the new leader the client then knows that the update request is not successful. If a newly elected leader has some uncommitted log entries from a previous term when it was a leader, once it becomes the leader again it has a chance to commit the uncommitted logs in the new term. Thus if the client waits long enough it may receive the acknowledgment.
One drawback of this mechanism is that the resulted log may be very long, and some nodes may have fallen quite behind and would have a long log to replay after rejoin. The common way to deal with it is to periodically truncate the log by taking a snapshot of the nodes committed system state. Creating a state snapshot makes it possible to discard old log entries and to garbage collect them, and the snapshot can be sent by the leader to the nodes trying to catch up to avoid re-executing the long sequence of logs.

### 8.16 RAFT Safety

RAFT guarantees safety by having the following properties:

**property #4 leader completeness:** once committed a log entry cannot be overwritten

**property #5 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 of the log

## 9 Lesson 9: Distributed Transactions

### 9.1 What are Distributed Transactions

A **transaction** is a group of operations that needs to be applied together in an indivisible manner. Ideally transactions need to be done with ACID properties—atomicity, consistency, isolation, durability. In addition transactions are not allowed to execute with inputs that see some but not all of the updates made in another transactions. This is enforced by requiring that transactions are executed atomically in isolation so that updates made within transactions cannot be partially visible to the rest of the system. Transactions are useful for concurrency control in which transactions are used to access shared state, and fault tolerance for which transactions ensure that either all of the updates are saved or none. A distributed transaction is just like a regular transaction except that it is executed across multiple nodes because the state that needs to be updated resides on multiple nodes. We still need to ensure the ACID properties and atomicity isolation across not only participants of a single distributed transaction but also multiple distributed transactions. A common solution is to rely on a coordinator or leader that initiates the transaction and executes a consensus protocol such as 2PC or Paxos or RAFT.

### 9.2 Spanner Brief

Spanner is a global data management player developed and used by Google. It has also been made available as a cloud database service. With Spanner the application data is stored globally and distributed across different geographical locations. At each location the data that belongs to a data store will be partitioned across thousands of servers, which is termed *sharded* across servers. At each geographical location the data will be also replicated across multiple sites.

## 9.3 Spanner Stack

The layers from bottom to top are

**distributed persistent storage:** named Colossus, which is a distributed file system based on GFS optimized for reads and appends because deletes are rare

**data model:** named tablet, which is a versioned/timestamped key-value store based on BigTable and exports the underlying blocks of files from Colossus, different from BigTable tablets include a timestamp similar to a version number for transaction ordering, in addition a Megastore layer is added on top of BigTable to export a view of the distributed store that matches a relational database so as to support SQL-type queries

**replicated state machine:** per tablet, each tablet corresponds to a group of related database tables, each table is replicated and the replicas are maintained consistent using Paxos, which is sufficient as long as the transaction concerns data in the replicas
2-phase locking similar to 2PC (to grant a lock instead of to commit a value) is used for concurrency control to support simultaneous accesses to the same replica set, 2PC is used to support cross-replica set transactions

## 9.4 Consistency Requirements

Blocking all incoming writes for a consistent view is expensive. Taking a consistent snapshot across all distributed replicas may be even more expensive. However we should not give up consistent reads. Because what actually matters is external consistency—the order in which updates are applied to the system must match the order in which the corresponding events appear externally in the real world. In terms of transaction if transaction #1 is completed before the start of transaction #2 then transaction #2 should see all the updates performed by transaction #1. This property is equivalent to having strict realizability, but without a globally synchronized clock we cannot guarantee strict realizability in a distributed system.

## 9.5 True Time

Instead of using a single value to represent a moment of time, Google's True Time represents time using an uncertainty interval around the (uncertain) real time. The interval provides a method to guarantee that a particular moment of time has elapsed or has not arrived yet. True Time is implemented by periodically probing master clock servers in datacenters, which include both GPS clock and atomic clock. The probing period and probing latency determine the width of the uncertainty interval denoted by $2\epsilon$—twice of the average uncertainty. Thus to ensure the atomicity isolation of transactions we can pick a timestamp $s >$ TT.now().latest() to ensure that the transaction starts after the lock is acquired and a timestamp $s <$ TT.now().earliest() to ensure that the transaction is completed before the lock is released. This constitutes a *commit wait*, that is a transaction cannot immediately commit.

## 9.6 Ordering Write Transactions with TT Timestamps

There are two locking strategies

**pessimistic locking:** acquire all locks up front in anticipation of lots of conflicts

**optimistic locking:** lock up only when a transaction needs to access another object, rollbacks may be necessary when a conflict surfaces which is more likely with long transactions

Since a write transaction has to be replicated to all replicas, the transaction is long with high probability of conflict. Hence Spanner adopts pessimistic locking for write transactions.
For write transactions that span multiple replica sets, Spanner uses 2PC to ensure that the transaction will commit only after all participants in the transaction perform their share of the updates. Logging starts later than the TT.now().latest() of all participants' lock acquisition. Then each participant updates what they are supposed to perform. After receiving the update completion notification from all participants, logging is done and the prepare phase is completed. At this point the coordinator can also compute a timestamp for the transaction taking into account the commit wait, after which it can commit the write and notify the participants the outcome of the transaction, which in turn release the lock.

## 9.7 Read Transactions

There are two types of read transactions

**read now transactions:** reads that need to return the current value in the system, it needs a Paxos leader to determine a safe timestamp if the state is distributed across multiple nodes in a single replica set or a transaction coordinator if the state is distributed across multiple replica sets, by determining a safe timestamp based on True Time, the now value is going to be delayed due to the prepared but not yet committed transactions

**read at specific timestamp:** much easier because all transactions are timestamped reflecting correct external ordering (if transactions are not timestamped then we need to run some consistent cut algorithm), we can just take a snapshot and then read the value we need by matching up their versions based on the timestamp

## 9.8 No True Time

True Time is possible through the GPS and atomic clocks and fast network, with which a few ms uncertainty is possible. In contrast the NTP protocol would introduce hundreds of ms uncertainty, which renders the commit wait intolerable and the application unless. Thus having True Time available is expensive but essential for external consistency.
On the other hand people do recognize that in many cases external consistency is not really important, thus having a globally synchronized clock like True Time many not be necessary in these cases. One example system is CockroachDB which is intended to give the benefits of Spanner for systems that don't have the capability to implement True Time, and provides explicit option for linearizability prompting users to consider whether or when external consistency is needed.
The situations when transactions are allowed to execute concurrently require some optimistic locking mechanism and optimistic concurrent control, optimistic in the sense that it allows the operations to execute concurrently as opposed to expecting them to acquire all locks up front. But we still need to ensure that transaction atomicity is preserved and isolation maintained, as this is what ensures that transactions are serializable. Note that being serializable doesn't necessarily mean they they will be serializable in the same order as what is externally visible. One popular technique to achieve this is snapshot isolation (SI) which is also termed multi-version concurrency control (MVCC). To guarantee correctness it ensures that transactions read from a snapshot, and the order in which the snapshots are used and the order in which they depend on one another forms a serializable sequence, i.e. directed lines among the snapshots that a transaction depends on or produces should not form a cycle.

## 9.9 Another DDB Example: AWS Aurora

Different from Spanner AWS Aurora follows a primary replica architecture—there is a single primary node that can handle reads and writes while all the other nodes only serve read traffic. By keeping writes localized the system doesn't have to perform distributed transactions. One motivation behind this design is its goal to guarantee availability under a large number of failures. Instead of the common triplication Aurora first replicates data to each zone, and in each zone it further replicates the data to two locations, with which Aurora can guarantee correctness even if an entire zone fails and it does so by relying on quorum. However as a result there are 6 instead of 3 replicas which leads to a huge amount of *I/O amplification*. I/O amplification means that each time we are trying to perform a write operation there are so many additional write operations that need to be performed. To address this issue Aurora uses only log replication—only the information about the operations that the primary performed along with relevant metadata are communicated from the primary to the other replicas. This is possible in part because there is already a distributed shared storage among all replicas. The observation is that if the storage layer is already made efficient for distributed accesses, then we don't need to replicate the same functionality at the level of distributed transactions, although in this way the data may correspond to slightly stale writes.

# 10 Lesson 10: Consistency in Distributed Data Stores

## 10.1 Why is Consistency Important and Hard

There is a tradeoff between consistency and availability—in general weaker consistency models can provide greater availability. There exist a number of consistency models. The representative ones in descending strength of consistency or ascending level of availiabity are

**strong consistency/linearizability:** it guarantees that the real ordering of updates will be visible

**sequential consistency:** it guarantees that a single ordering of updates will be visible, which may match the real one

**causal consistency:** it only guarantees ordering for updates which are causally related or per happens before, there is no guarantee for concurrent events

**eventual consistency:** it is possible to have periods when the view of data is not consistent, however as long as partitions or failures are not permanent it guarantees that all writes will eventually become visible

When we say a system supports some consistency model, we mean that the system makes a guarantee about the ordering of the updates and how they will become visible to ongoing reads.

## 10.2 Key-Value Store

Key-value stores are data stores in which each piece of data is uniquely identified by some key and the value associated with that key can be of arbitrary type and size. The basic operations on key-value stores are `put` which writes value for the associated key and `get` which reads the value associated with a key. Key-value stores can also support other operations for example `scan` or range query which returns all values with keys within a given range. Additional more powerful models may be layered above this basic key-value model.

## 10.3 Memcached

One simple and popular key-value store is Memcached developed by Facebook, which was subsequently replaced by Tao to optimize graph-based operations. The different contexts where Memcached is used include (1) within a cluster (2) across clusters (3) geographically distributed. In these contexts Memcached serves as an in-memory cache for objects that are retrieved from storage and hence the name Memcached.

## 10.4 Look-Aside Cache Design

The design of Memcached as a cache for data stores follows a look-aside cache design. Many large scale internet applications are very read-heavy and reads are served over large data sets. In addition it is likely that reads are not uniformly distributed and there are some "hot" data that are more popular. Temporal locality is also common i.e. more recent data tend to be hotter than old ones. All of these features make the workload a good candidate to benefit from a small in-memory cache.
The design of Memcached makes two decisions

- serving as a simple in-memory key-value object cache to make Memcached tier as simple as possible for additional efficiency
- look-aside = instead of sitting in front of the database it sits on the side, a client will explicitly perform a `get` request from this cache, in case of cache miss instead of being automatically routed to the lower levels of the hierarchy the miss is returned to the client which explicitly performs a database lookup
  demand-filled = once the data is retrieved from the database the client explicitly issues a `set` operation to add this value to Memcached

Instead of updating which relies on the client not to crash or be delayed or be corrupted, when an update needs to be performed Memcached simply delete the cached entries and update the database, as deletion is idempotent respect to the correctness of the database and makes room for new inserts. As such Memcached only consists of clean unmodified read-only data. To cope with limited capacity Memcached uses LRU as its replacement mechanism.
Note that Memcached is only intended to optimize performance, only the database is considered persistent thus has the definite view of the data. Therefore applications which require strong guarantee need to bypass Memcached and directly perform database operations. Nonetheless it is possible for Memcached to provide some guarantees. One straightforward way is to back it with persistent non-volatile memory. By explicitly separating cache tier from storage tier, it is possible to associate different guarantees with the two tiers and achieve optimization in the cache tier.

## 10.5 Mechanisms in Memcached

One problem with look-aside cache is that the `set` operation triggered by two consecutive updates on the same data may be reordered due to network latency. The solution Memcached uses to prevent this is a mechanism called **lease**, which is a token issued by the cache upon cache miss. It can be used to enforce ordering on concurrent writes. Lease can also be used to address thundering herds by controlling how many leases are issued at a given time.
To expand the capacity of Memcached we need to scale horizontally by adding more Memcached instances. To take advantage of this expanded capacity the key-value space needs to be sharded and the shard boundaries can be adjusted. To make it simple Memcached relegates the routing decision to clients through an encapsulating *mcrouter*.
A single Memcached cluster still has bottleneck. From the perspective of clients one limitation is how

many Memcached instances can be routed efficiently. From the perspective of individual shards the limitation is how many requests can be served for hot content. Thus instead of having one huge Memcached cluster it makes sense to organize Memcached instances into multiple independent Memcached clusters, so that hot content can be served from multiple clusters and one failure will only impact one cluster but not the others. To enforce content consistency across multiple clusters Memcached uses an invalidation based mechanism for database updates. But with multiple clusters it becomes necessary for the invalidations to be centralized for ordering guarantees. Invalidations are pushed to the appropriate Memcached instances in an order that corresponds to the database commit order.
However given that the clusters are geographically located it is not realistic to expect a single sharded cluster will collect all updates from all locations and respond to cache misses at each location. It needs to be combined with replication at the storage layer because database will be eventually replicated at each location, for which we will distinguish master database from replica database. When a client requests an update it first sets a marker in the local Memcached with the entry associated to the value to be updated indicating that it is involved in a remote operation. Then the actual write is performed against the remote master database. At the same time the value in the local Memcached is deleted because the local database may not have the updated value. Some time later the database replication kicks in which will replicate the update to all replicas including the local one. When the local database is safely updated the marker indicating the value is involved in a remote operation is removed. Until that time the updated value can only be served by the remote master database.

### 10.6 Causal+ Consistency

Causal consistency is inadequate because in causal consistency the system observes data accesses in order to determine causality, but often updates access different servers and thus appear as if they are concurrent, i.e. causal dependencies are not visible except when updates are localized in one single server. To address this issue clusters of order-preserving servers system or COPS was proposed to guarantee a so-called causal+ consistency. COPS targets systems such as Memcached. In COPS when a client performs the `get` operation on the local data store, it also captures the information that this data was read. Later on when a client needs to perform a `put` operation it includes not only the `put` operation and the value to be updated but also some ordering metadata. The client library of COPS then issues a `put_after` operation to provide all the metadata about the dependencies that this `put` operation has with other values that have been read previously by this client as well as information about their timestamp or version. The local key-value store is going to log this data and communicate this information during replication. Thus when an update gets replicated the information received by remote nodes includes the key and value to be updated along with all dependencies if any. With the dependencies the remote node can perform dependency check and blocks the update until all of its dependencies become visible.

## 11 Lesson 11: Peer-to-Peer & Mobility

### 11.1 Communication Support Assumed So Far

At the application level or even at the level of the distributed services, to determine where a message should be sent we use some namespace identifier such as the primary node of a replica or the node that stores a shard of the object that corresponds to a certain key range. This namespace needs to be mapped to the namespace used at the network level such as the IP address or network path in a LAN. Hence there is some intermediate layer to support this mapping and we can describe it as an overlay network. This mapping is recorded when processes are created and distributed to all nodes, and it needs to be updated upon change due to failure or expansion or workload balancing etc. which is

challenging for very dynamic and large systems because we can't assume the remapping can be quickly distributed and we need to consider the overhead associated with the distribution. Besides scalability, geographical distribution and failure, the fact that the nodes may belong to different administrators i.e. decentralization further complicates the remapping distribution.

## 11.2 Interconnect Support

Interconnect support refers to the network device hardware and drivers used by the communication services such as network interface card or NIC. With this support network can do much more than sending messages. At the most basic level there typically is support for broadcast (one to all the rest) or multi_cast (one to some), which are examples of what we refer to as *collective operations*. More advanced networks support other types of collective operations such as gather and all_reduce which are communication operations where a node waits to receive messages from all nodes in the communication group. Collective operations can also serve as synchronization primitives e.g. barrier which is a binary gather operation where all nodes need to ensure that everyone has reached the barrier point before they can proceed. Interconnect hardware can also include support for atomic operations such as compare_and_swap or CAS. Some of the recent interconnects include support for timestamps to be generated by the NIC itself, which enables remote direct memory access (RDMA) bypassing CPU since we don't have to rely on the host CPU to generate those timestamps. Another important feature is support for direct cache injection (DDIO) which allows direct binding of data payload in the cache. All these types of functionalities require some additional support for scalable implementation. In some cases this may be as extensive as a separate dedicated network e.g. tree combining algorithm for barrier implementation in CS6210.

## 11.3 Peer-to-Peer Systems

Peer-to-Peer systems take a completely opposite view of a distributed system where we cannot make any assumptions about the hardware capabilities supported by the interconnect. This is legitimate because to rely on some advanced capabilities of the interconnect we need to assume that all nodes in the distributed system are outfitted with the same infrastructure, which may be true for tightly coupled data center clusters but unlikely for wide-area internet network. Because for WAN the only assumption we can safely make is the presence of IPs of the internet protocol. Thus for WAN the question of how to manage mapping from application-level identifiers to network-level identifiers remains open, which is even trickier when the WAN nodes are not managed by the same administrator. As the name suggests in a peer-to-peer system all nodes are peers meaning no one is more important than another. One characteristic of such systems is that the only assumption we can make is that peers can use their IP addresses to communicate among them. Note that peer-to-peer system can also be applied to data centers and same organization distributed systems.

## 11.4 Connectivity in P2P

There are three ways for peers to find out how to connect to each other

**centralized registry:** there is a centralized entity that has the information about what data is stored in which peer, i e.g. Napster

**pro:** it is possible to find out which peer should be contacted within a single round trip time

**con:** the centralized registry is a single point of failure, limits scalability, and relies on the trust from peers

**flood- or gossip-based protocols:** each peer broadcasts the information about the content it stores or the requests that it can serve, e.g. Gnutella and bitcoin

**pro:** requires no assumption on centralized authority or trust

**con:** there is no bound on lookup time

**distributed hash table (DHT):** provides structured routing trees based on DHT, e.g. Chord, Kademlia for ethereum

**pro:** permits decentralized index with which many nodes can be involved in providing information about how the data or services are distributed among peers, the structure of DHT provides some guarantees about the lookup time bound

Among the three the third approach is the most popular one.
Note that the concepts of peer-to-peer systems are still relevant to distributed systems in general including distributed data centers, e.g. DHT is a common building block of AWS DynamoDB.

## 11.5 Distributed Hash Table (DHT)

The main element of a DHT is a hash function, which takes some input (bytes, strings, objects etc.) and produces some unique and condensed signature of that input. The hash function guarantees that it will produce the same output from the same input (the sense of pseudo in pseudo random), and that it will randomize how the input gets mapped to some sufficiently large but simpler space of the output identifiers (potentially integers). Every participant in the system would use the same hash function, and the mapping can be adjusted depending on node failures, changes in the load, or scaling.

## 11.6 Chord

In Chord the DHT is represented as a ring of all the numbers from 0 to $N-1$, where $N$ is sufficiently large that no two peers are mapped to the same node. The keys for the objects that need to be looked up and the IP addresses of the peers are mapped to the same namespace. To insert information in Chord first the hash is computed on the data key and then the corresponding node is updated. If a data key hashes to a value for which there exists no node, then the node corresponding to the first successor value is updated, which may not be the immediate successor thus additional trials may be needed before a successor can be reached. This suggests an $O(N)$ number of trials in general case.
To improve the lookup time Chord maintains so-called *finger tables*. Each node maintains information about the known nodes that serve a particular key range. The entries in a finger table are organized as follows: at each node $n$ the $i$th finger entry has information about the key range starting from $n+2i$ that corresponds to a range of $2i$ elements. By using finger tables the lookup time can be reduced to $O(\log N)$ in most cases. When a new node joins it finds where it needs to be entered and for which node it is a successor or predecessor in the ring. Then it takes over the data that was previously stored at the corresponding nodes for which it now needs to be responsible. To assist with the process of updating the finger tables particularly when nodes fail, Chord also maintains some information about the successors of some nodes in the finger table. The combination of all these make it possible to speed up lookup. Although there are still some pathological cases where the system may fail to provide the result in $\log N$ time, the paper proved some probabilistic guarantees for the $O(\log N)$ lookup performance.

## 11.7 Hierarchical Systems

Besides peer-to-peer systems another way to achieve scalability in the communication layer is to rely on hierarchical system design. The design choice has implication on the tradeoff between the cost of

communication and the cost of maintaining the states, and depends on many properties of the system such as failure probability, mobility, number and type of nodes, communication pattern, locality etc. To address all these factors it makes more sense to adopt a hybrid approach or a hierarchical solution where we adopt different design choices for different levels of the hierarchy.

## 11.8 Heterogeneous Systems: A Mobile Network Example

In a mobile network there are two types of nodes

**mobile support station (MSS):** stationary, interconnected with high-speed wired network, power availability is not a concern

**mobile host (MH):** belong to a cell associated with a MSS, connected to MSS via low-speed mobile network, mobile, battery energy is limited

The goal is fast lookup of a MH with low communication overhead and computation overhead. The heterogeneity of nodes mandates the heterogeneity of algorithm to achieve this goal.

## 11.9 Alternative Algorithms

The paper describes several metrics for choosing among alternative algorithms—search/lookup cost, insert/remove node cost, impact of an update. The analysis considers the fact that the cost of wireless communication is much high than the cost of wired communication $C_{\text{wireless}} \gg C_{\text{wired}}$, and the number of MSSs is much smaller than the number of MHs $N_{\text{MSS}} \ll N_{\text{MH}}$. Much of the algorithm is related to executing token passing to give each MH a chance to make a call. Some comparison examples are

- the cost of point-to-point communication $C_{\text{comm}} = 2 \times$ the cost of wireless communication $C_{\text{wireless}}$ + the cost of search for the MH $C_{\text{search}}$, for the latter we can have

  **algorithm 1:** all MHs form a logical ring, $C_{\text{search}} \sim O(N_{\text{MH}}, C_{\text{wireless}})$

  **algorithm 2:** all MSSs form a logical ring and each MSS knows about MHs in its cell, $C_{\text{search}} \sim O(N_{\text{MSS}}, C_{\text{fixed}})$

  since $C_{\text{wireless}} \gg C_{\text{wired}}$ and $N_{\text{MSS}} \ll C_{\text{MH}}$, $C_{\text{search}}$ of algorithm 2 $\ll C_{\text{search}}$ of algorithm 1

- for MH moves we can have

  **algorithm 1:** no update on MH move, only when MH needs to be reached original MSS searches for new MSS on demand, $C_{\text{update}} \sim O(C_{\text{fixed}} + C_{\text{search}})$

  **algorithm 2:** new MSS informs original MSS each time a new MH joins, thus update is needed each time a MH moves, $C_{\text{update}} \sim O(\#\text{moves} \times C_{\text{fixed}})$

  which algorithm is better depends on the lookup time over the wired network and how often MHs move relative to how frequently they are involved in the communication

# 12 Lesson 12: Distributed Data Analytics

## 12.1 Data Processing at Scale

There are at least three techniques

**data parallel:** divide and conquer, inputs are divided and each subset is assigned to a different node, commonly used in traditional scientific computing, it assumes that there is a way to partition the data for good workload balancing, which is difficult if the processing is input dependent
in general there is data dependency which needs to be communicated via explicit messages, thus it is more realistic to expect local computation phases are followed by global communication phases, barriers can serve as a good transition between these two phases

**pipelining:** divide work into smaller tasks and have each node specialize in one of them, and stream data in chunks through the task pipeline, this increases throughput

**model parallelism:** when the processing depends on application states, we can divide the states of the application across nodes so that each node processes a subset of the states, input is passed to all nodes while output is combined from all nodes, dependencies need to be handled via explicit message exchange

## 12.2 MapReduce Brief

The processing is divided into two phases

**map:** input = a set of key-value pairs which is divided into smaller chunks, each chunk is passed to a worker process, output = an intermediate new key-value pair

**reduce:** input = the intermediate output from the map function, output = a final reducer combining output from all intermediate reducers

This process is orchestrated by a master process which determines the work assignment for the map and reduce function.
Translated to the commonly used word count example it is

**map:** input = a file with (filename, content) key-value pair, output = an intermediate file with (word, count) key-value pair

**reduce:** input = intermediate output from the map function, output = list of words with total counts

The MapReduce model involves all the three data processing techniques that is mentioned above

**data parallel:** divide input into chunks and assign each chunk to a mapper

**pipelining:** data is first processed by mappers and then by reducers

**model parallelism:** final output is aggregated from that of individual reducers

The MapReduce model can also be viewed as a dataflow model because the execution of the MapReduce pipeline is determined by the flow of the data. A reducer cannot start executing unless the output from the mappers is ready.
Other examples of MapReduce applications include URL access frequency, reverse weblink graph for page ranking, inverted word index for document search etc.

## 12.3 Design Decisions of MapReduce

A system which implements the MapReduce model needs to make several design decisions concerning several dimensions:

**master data structure:** how many states per worker to track progress, how to organize them, how frequent to update them

**locality:** how to schedule mapper and reducer on each node, placement of intermediate output

**task granularity:** finer granularity offers more flexibility but requires more extensive management operations vs. coarser granularity reduces the management overhead at the expense of flexibility

**fault tolerance:** standby replication for master vs. failure or straggler detection and re-execution trigger for workers

**semantics in the presence of failure:** performance—is the slowdown acceptable? vs. importance of consistency and complete result—still useful even when some portion of the data is not included?

**backup task:** speculative backup in parallel

## 12.4 Limitations of MapReduce

MapReduce depends on having persistent I/O meaning all intermediate data have to be made persistent which essentially serve as checkpoints. The problem with this fault-tolerance mechanism is that

- serialization to and from persistent storage may be costly
- a worker can serve both as a mapper and a reducer, thus we don't have a lot of control over what different input that need to be read by the worker, moreover there is both remote access and data movement that adds to the overhead

MapReduce may lead to data amplification as well. Because often executions are iterative, intermediate data may be read multiple times. What is worse the size of intermediate data may be much larger than the input.
Finally given such a large scale we may only afford commodity storage devices.

## 12.5 Spark

The motivation of developing Spark is to address the I/O overhead of MapReduce. Spark solves it by allowing in-memory data sharing, the benefit of which is two-fold—DRAM is much faster than HDD or even SSD, and if data is in memory then the serialization cost can be avoided.
Spark also uses a different fault tolerance mechanism.

## 12.6 Resilient Distributed Datasets (RDDs)

Spark achieves in-memory data sharing by relying on its RDD abstraction. An RDD is a read-only i.e. immutable collection of records which can be partitioned in different machines. RDDs are created via deterministic and lazy transformations

**deterministic:** a transformation always produces the same output given the same input

**lazy:** a transformation does not get executed until the result is needed

RDDs are used via **actions** such as `count`, `collect`, and `save`. RDDs are also aware of their lineage so that they can be reproduced from their parent RDDs. Spark also provides users with APIs to allow explicit control of persistence and partitioning for specific RDDs.

## 12.7 RDDs through Example

An RDD includes (1) actual data (2) lineage information including parent RDDs and transformations (3) metadata on partitions, partitioning scheme, and dependencies.

## 12.8 RDD Transformations

RDD transformations can be grouped into

**narrow dependencies:** the elements of child RDDs are in 1-to-1 correspondence with those of parent RDDs, e.g. map, filter, and union

**wide dependencies:** the elements of child RDDs are in many-to-many correspondence with those of parent RDDs, e.g. groupByKey

Spark actions are executed as a DAG of stages so as to create as many narrow dependencies as possible. That way we can achieve greater parallelism and limit I/O contention. This also allows us to maximize the benefit of locality during scheduling.

## 12.9 Did Spark Achieve Its Goal

Spark behaves like a distributed shared memory-like runtime, and we just need to track the updates and persist the lineage. Note that we log only the coarse-grained transformations applied to all elements of an RDD instead of the elements themselves. The pro and con of this mechanism are

**pro:** there are three

- less data to persist in execution critical path
- less slow storage I/O, if lucky we may need to read the data only once
- dependency tracking offers more control over locality

**con:** longer recovery time, which can be reduced by selective re-execution only for affected partitions

Spark is best for batch workloads for which coarse update granularity is achievable and high throughput is desired.

# 13 Lesson 13: Support for Datacenter-Based Distributed Computing

## 13.1 Datacenter Trends

Moore's law advocates leveraging the economy of scale with commodity components to achieve performance and scale. But the trend deviates from Moore's law further and further as time elapses due to practical limitations. In response a number of techniques have emerged such as multi-core and multi-threading. One very promising approach is that of specialization and heterogeneity, which relies on specialized components to provide advantages for some aspect of the workload, e.g. GPU and TPU for AI workloads and new persistent memory and storage classes. In a datacenter the mix of workloads may change over time. As such to provide greater flexibility one recent trend is disaggregation, which means that different types of resources can be independently added. Examples of such technologies, in part motivated by the limitation of the traditional x86 + DRAM + Ethernet commodity hardware, are

- high-speed interconnects with support for RDMA which enables shared memory access across distinct nodes (Mellanox)
- programmable interconnects with integrated CPU which allows moving common tasks such as Paxos to the network (Mellanox, Intel)
- persistent memory—memory that is both byte addressable like DRAM and persistent like storage (Micron, Intel)
- specialized accelerators such as GPU and TPU (Google, NVidia)
- disaggregation which allows more network-attached resources such as network-attached memory
- software packaging and distribution from VMs to containers and microservices (Docker, Kubernetes)

In this lesson we will focus on RDMA, persistent memory, disaggregation, and containers.

## 13.2 What is RDMA

RDMA stands for remote direct memory access. As the name suggests a network with RDMA capabilities will make it possible to bypass remote CPU when accessing data on a remote node, which is supported by the network adapters or NICs as well as the endpoint protocols. This provides higher bandwidth and lower latency than commodity Ethernet networks at the expense of higher cost per port.
RDMA was formalized through the virtual interconnect architecture (VIA) early 2000s and Mellanox emerged as a leader in InfiniBand interconnect. RDMA is also supported by RoCE (RDMA over converged Ethernet), iWARP (internet Wide Area RDMA Protocol) etc.
There are two main forms of RDMA

**two-sided RDMA:** more similar to traditional send/receive operations, the two endpoint CPUs are involved in the communication, but RDMA allows protocol processing to be performed on NICs which avoids CPU load thus is faster

**one-sided RDMA:** only one endpoint CPU is involved in the communication which initiates reads and writes, the NIC of the other endpoint ensures the remote memory access for read and write requests by pinning the relevant data in memory, recent direct cache injection technology such as Intel's DDIO makes it possible to push the updates directly into the LLC

## 13.3 RDMA-Specialized RPC

Given there are two main forms of RDMA, the obvious question is which one should be chosen when implementing RPCs for datacenter services

**one-sided RDMA:** reduce CPU utilization, but may take multiple round trips because an RPC often requires some invocation of a remote service thus may still need CPU execution

**two-sided RDMA:** remote service can be completed in a single round trip, also allow simple integration with faster network

There are additional opportunities to further optimize the RPC implementation by taking advantage of common features in RDMA fabric such as InfiniBand

- InfiniBand supports connection-less protocol, which is preferred because maintaining a connection implies that there is some state associated with each endpoint which imposes limitation on the number of clients the service can scale to
- for the destination NIC to be able to write to memory there needs to be some preregistered pool of memory which may lead to load balancing issues, a feature called shared receive queues allows this data to be aggregated across different connections, which improves scalability even though the indirection implies some loss of performance for unloaded keys
- small datagram packet allows performance specialization on small RPCs, small in the sense of arguments

One implementation of RPC that combines some of these features is FaSST (fast, simple, scalable RPC).

## 13.4 What if Memory is Persistent

For the longest time we have differentiated memory and storage

**memory:** volatile, byte addressable, small capacity, fast

**storage:** persistent, block addressable, large capacity, slow

Persistent memory combines the advantages of the two—byte addressable and low latency like DRAM and persistent and scalable like storage. A simplest design is to add battery to DRAM. Persistent memory is made commercially available by Intel's Optane memory.
The options of RPC implementation for accessing and updating persistent memory are

- for a singe write we can let RDMA push the data to LLC and then we are done
- for a persistent write (pwrite) placing the data to LLC is not sufficient because cache is not persistent, we need to ensure that the destination NIC pushes the data to the persistent memory controller, however persistent data operations require flush to persistent memory and persistent RDMA operations are much more costly then persistent write operations, from this standpoint the use of persistent to some extent removes the advantage of RDMA over send/recv RPCs

## 13.5 Disaggregation

Optimal server configuration obviously depends on the type of workloads. But workloads evolve for which the inflexible monolithic server configurations are not suitable, because we cannot elastically scale individual resource components and trying to design for the worst case will lead to imbalances and resource inefficiency. The solution to this problem is to adopt disaggregation. With resource disaggregation data centers can be built from pools of different resources and having them network attached and independently scaled. In addition we now have more availability of devices that can be network attached via e.g. programmable NICs.

## 13.6 Systems Software in Disaggregated Systems

Traditional operating system stacks include different subsystems each responsible for different types of resources. If the resources are not disaggregated the same should apply to the corresponding OS components.

## 13.7 LegoOS Approach

One proposal of how to achieve a disaggregated operating system is LegoOS. In traditional operating system design the operating system is responsible for all hardware resources and uses a network to communicate with external operating systems. 10 years ago a multi-kernel architecture was proposed in which different operating system kernels are responsible for some subset of hardware resources so that they can be specialized to deal with hardware heterogeneity or scaling. Each kernel communicates with each other as though this is a distributed system, but they still sit on a single monolithic server. LegoOS is based on a so-called split-kernel architecture, where each type of resources is managed not by a complete operating system but by a monitor just for that particular resource type. All these monitors interact with network messages across a non-coherent (no cache is attached and maintained coherent) network architecture. In this way the functionality of the entire operating system kernel is split and distributed across the different types of resources.

## 13.8 Disaggregating CPU and Memory with LegoOS

After disaggregating the DRAM memory from the server and making it reachable via network, memory access will rely on the MMU and TLB to determine the appropriate physical address, thus it makes sense to have them co-located with the DRAM so that load and store instructions from CPUs can continue using virtual addresses. From the operating system perspective they constitute a virtual memory subsystem and all levels of cache become virtual cache. However this means that any memory access due to cache miss will now go through network, and network is still slower than local memory bus. To hide this latency the authors proposed to add extended cache to the processor component which they call ExCache. ExCache can be managed exclusively by software or with some hardware assistance and is inclusive and accessed via virtual addresses.

## 13.9 LegoOS Select Experimental Result

The authors built a prototype based on emulation because they did not have a full system of different disaggregated types. They used a collection of monolithic servers with some resources ignored (e.g. the machine that implements memory resource uses all available DRAM memory but only a small number of CPU resources for running just kernel-space functionality for the memory monitor), and existing Linux kernel modules to implement the monitors. All of these were connected via RDMA network and communicated via fast RPC over RDMA. ExCache was implemented with DRAM and fully managed by software.

# 14 Lesson 14: Datacenter-Based Distributed Management

## 14.1 Management Stack in Datacenters

Datacenters typically host both general purpose server components and specialized configurations for certain workloads. Multi-tenancy is common and they may have different requirements (e.g. batch jobs emphasize throughput vs. long-running services emphasize responsiveness). Each application may have multiple processes which may differ in

**end-to-end metrics:** some are more latency-sensitive while others are more throughput-intensive

**resource requirements:** some are compute-intensive demanding accelerators while others are data-intensive facing tradeoff between data access speed and data capacity

All these requirements need to be translated into resource allocation decisions. Given the many interdependent tasks in an application it also requires orchestration. The ultimate goal is to meet the service-level objective or SLO which is specified in contractual service-level agreement or SLA.

## 14.2 Datacenter Management at Scale

The Borg resource manager developed by Google forms the basis for the nowadays most widely used containerized datacenter orchestrator Kubernetes. In Borg a **cell** is a collection of machines which is the unit of management in Borg. The machines in a cell belong to a single cluster. A **cluster** lives inside a single datacenter building. A **site** consists of multiple buildings. An application job is represented by the actual tasks that need to be scheduled. The responsibility of Borg is to perform resource allocation and scheduling decisions for these tasks. When a task is submitted provided that it is accepted by the system it becomes pending and will be periodically scheduled for running, and upon completion it enters the dead state. A pending task can be sent straight to the dead state upon failure, kill, or loss, from which they exit the scheduler system.

## 14.3 Overview of Borg Operations

**Borgmaster:** the brain of the Borg system, one per cell, handle all client requests via RPC, write jobs to the pending queue, assign tasks to machines, monitor the state of all machines in the cell and maintain the state of the entire cell in memory

**task scheduler:** read tasks from the pending queue, run feasibility check to determine on which machine a task can be run and further score the feasible machines to quantify the fit, submit machine assignment to Borgmaster (only analysis not scheduling)

**Borglet:** local Borg agent on every machine, start/stop tasks, restart tasks upon failure, tasks are run inside containers and Borglet can manipulate container properties, report machine state to Borgmaster

Borgmaster polls the Borglet for machine info and cell state update, if a Borglet does not respond Borgmaster marks the machine down and reassigns its tasks to other machines

## 14.4 Achieving Scalability

**Borgmaster reliability:** to ensure reliability Borgmaster is replicated 5 times, one replica serves the master determined using a Chubby lock (Google's version of ZooKeeper), only master can mutate the state of the cell and it also serves as the leader for writing to the Paxos store, each replica also saves the state of the cell to a Paxos based store

**scheduling heuristic:** scheduling heuristic is designed to both limit the impact of failure and ensure forward progress, which include (1) automatically rescheduling evicted tasks, (2) spreading tasks across failure domains to reduce correlated failures, (3) avoiding repeating task-machine pairing that leads to some failure, (4) limiting the rate of task disruption and the number of tasks from a job that can be simultaneously down to ensure that there is some forward progress, (5) decoupling task assignment from scheduling to open more asynchrony opportunities, (6) optimizing scoring of machine-task pairs (7) avoid segregation of production and non-production workloads

**communication efficiency:** Borgmaster uses separate threads for read-only RPCs and talking to Borglets, and uses link shards to summarize information collected from Borglets

Further optimization include

**score caching:** scores for task assignment are cached until machine or task properties change (small changes are ignored)

**equivalence class:** group tasks with identical requirements, scores are computed once per equivalence class, Borg tasks have an application class—latency-sensitive requiring priority treatment vs. batch which can tolerate low quality service

**relaxed randomization:** task scheduler examines machines in a random order for serendipity until it has found enough feasible machines to score

Resources may be classified as

**compressible:** can be reclaimed from a task by decreasing its quality of service without killing it, e.g. CPU cycles, disk I/O bandwidth

**non-compressible:** cannot be reclaimed without killing the task, e.g. memory, disk space

### 14.5 Experimental Results

One decision made by Borg is to share the underlying resources by different types of workloads (high priority services vs. low priority batch jobs). This pooling decision enables Borg to execute the same workload with fewer machines, because the resource reclamation allows Borg to find some resources for low-priority jobs and ready to be given back as soon as high-priority jobs require.

## 15 Lesson 15: Distributed Machine Learning

### 15.1 Distributed Machine Learning

Machine learning is both compute-intensive and data-intensive. A lot of the data for machine learning are generated far from datacenters at the edges of the network. What is worse many machine learning applications are global in scale, which rely on data generated across globally distributed sensors. Data movement in such a scale will be very expensive.

### 15.2 Distributed Machine Learning Approaches

The simplest model is to aggregate all data to a central location, train the model, and distribute the model to all locations, which would incur huge data movement cost and be limited by data sovereignty (moving data out of border may be restricted by local laws). An alternative is a federated system in which data is evaluated locally and periodically these locally trained models are aggregated. Based on these centrally collected updates an overall update is computed and disseminated to all locations. One example of this type of systems is Gaia. Similar goals as Gaia are part of the federated distributed learning model developed by Google. Both Gaia and federated learning leverage the functionalities similar to the parameter server approach. In the parameter server model some machines in a data-center are designated as workers while others are parameter servers. The training data are distributed across all workers and the model parameters are distributed across all parameter servers. The learning process is in an iterative manner—workers work on a set of parameters, compute the gradients and parameter updates, and then communicate the updates to its local parameter server. The parameter servers aggregate these updates, synchronize among each other, and update the model, which is then disseminated to all workers.

## 15.3 Geo-Distributed ML

The naive approach is to distribute workers and parameter servers over geo-distributed datacenters and let them communicate via WAN. But it incurs significant slowdown due to slow yet significant WAN traffic.

## 15.4 Leverage Approximation

The key idea of Gaia is to decouple the synchronization of the model within a datacenter from the synchronization of the model among datacenters. It means that within a datacenter the workers and parameter servers will interact in the same way as in the parameter server model. However across datacenters parameter servers will be out of sync and they will synchronize only infrequently. This approach makes sense because machine learning is imprecise as there is always model error and it is able to function correctly on approximately correct model copy. Gaia leverages this approximate computing to relax the consistency requirement. As to what and when to synchronize across datacenters the authors discovered that $> 95\%$ updates are less than $1\%$ significant. Hence another key idea of Gaia is to communicate across datacenters only the significant updates.

## 15.5 Gaia: An Approximate Synchronous Parallel System

To implement the above two ideas Gaia adopts a new synchronization model called **approximate synchronous parallel** or ASP. There are three mechanisms to support ASP

**significance filter:** determine what are significant updates, Gaia does it by providing to users an API to specify a significance function to compute the level of significance, when a significant update is determined it is used to create an ASP selective barrier

**ASP selective barrier:** due to the slow WAN connection even significant updates may take a long time to be disseminated, to ensure that parameter servers are updated at least for the significant updates Gaia introduces ASP selective barrier to stop workers, during a remote sync some index specifying the information about the updates that will come (through data queue) will be sent first (through control queue) so that the remote datacenter knows to wait

**mirror clock:** to prevent some datacenters from becoming too stale datacenters exchange clock information (all communications will be tagged with the local clock), which can be used to estimate the staleness and to determine whether a datacenter needs to slow down

With these mechanisms Gaia achieves performance close to the LAN-speed system for geo-distributed machine learning.

## 15.6 Tradeoffs of Using Global Model

A global model is not always needed because

- there is a lot of locality and personalization in data pattern, which can be better served by more tailored models which are smaller and more efficient
- due to the complexity of training and the potential for overfitting with large models, building a good global model is much more difficult from the algorithmic perspective
- even with the optimizations implemented in Gaia data transfer cost is still high

One extreme alternative to using global model is isolated learning. As the name suggests in isolated learning each node builds custom model in isolation. However it suffers from

**loss of accuracy:** there may be insufficient data in isolation

**loss of efficiency:** there is no sharing of data which prevents learning the same pattern in multiple locations, no sharing of computation so that we have to re-learn the same thing in each location, this is particularly wasteful if there is some trend in the input data propagating from one location to another

## 15.7 Collaborative Learning with Cartel

To address these problems of isolated learning collaborative learning was proposed and a prototype system called Cartel was developed. Cartel has different goals from the other systems:

- maintain small customized models at each node
- when there is change in the environment or variation in the workload pattern, find a node/peer in the system where similar pattern has been observed before, and then transfer the knowledge from that peer i.e. perform model update
- provide a jump start to adapt the model at one node to changes it observed by making it possible to find the right peer node and perform knowledge transfer

Cartel relies on a centralized metadata service (centralized registry or DHT) to aggregate the metadata about the learning happening at different nodes periodically. Each node dynamically evaluates the performance of its learning. When it detects a drift i.e. an accuracy drop it contacts the metadata server, which provides some information about a good peer that can help with the model update. Once such a peer is identified the exchange of parameters takes place for knowledge transfer.
Evaluation shows that the benefits of collaborative approach include

- adapt more quickly to changes in workload than isolated systems
- reduce data transfer cost significantly than centralized systems
- enable use of smaller models leading to faster training than centralized systems

## 15.8 Beyond Geo-Distributed Training

Training is only the first step in the machine learning pipeline, which is followed by serving, hyperparameter tuning, streaming, simulation, and featurization. The Ray system developed by Berkeley Ray lab integrates all these functionalities into a unified framework, which improves the efficiency in the end-to-end distributed process.

# 16 Lesson 16: Byzantine Fault Tolerance & Blockchain

## 16.1 Byzantine Failure and Byzantine Generals

Byzantine failures are failures where failed nodes continue participating in the distributed system but send incorrect messages for either malicious or arbitrary reasons.
Byzantine generals problem is that Byzantine generals each with their own armys are trying to decide whether they can attack a city or should retreat. They don't have an easy way to communicate. Consensus may be reachable via Paxos. But both the generals and the messengers can be corrupted. It raises the question of whether it is possible to reach a correct consensus even in the presence of this corruption.

## 16.2 Byzantine Fault Tolerance

Our goal is to reach consensus in an asynchronous network with the desired safety, liveness, and validity property which can tolerate up to $f$ failures, and even in the presence of Byzantine behavior. The main idea for achieving this goal is three-fold

**against corrupt messages:** use cryptographic methods to authenticate communication and messages

**against corrupt participants:** increase the number of participants, to tolerate up to $f$ failures it is proven that there needs to be at least $3f + 1$ nodes

**against corrupt leader:** additional checks among participants for leader message consistency

Combining the above three techniques we can guarantee safety. But to ensure liveness we need the additional requirement that messages will eventually be delivered with bounded delay (eventual synchrony).

## 16.3 Practical Byzantine Fault Tolerance: pBFT

All these ideas come together in pBFT, an algorithm for practically achieving Byzantine fault tolerance. pBFT considers a system of replicated services, in which the client needs a guarantee that the servers will reach a consensus when replicating the client's updates, or they will provide responses such that the client based on the majority of these responses will be able to determine the correct response. One of the replicated servers is a leader, the rest are backup. The primary node determines the current view of the system and the primary node may change over time. Each replica maintains consistent information about service state, message log, and current view. Communication integrity is guaranteed cryptographically through the use of public keys, message digests etc.
To see why we need $3f+1$ nodes to tolerate $f$ faults, suppose there are $N$ nodes. To tolerate $f$ faults we need to decide via a quorum among $N - f$ nodes. In the worst case the $f$ missing nodes are just those that are delayed rather than faulty and all of the $f$ Byzantine nodes are included in the set of $N - f$ nodes. Then we need to have more non-faulty response among these $N - f$ nodes, i.e. $(N - f) - f > f$. Therefore $N > 3f$.

## 16.4 pBFT Algorithm

Since even the leader can be corrupted the client sends the request to all servers. Once more than $f+1$ responses are received that provide the same information the client then knows that it has the correct response.
The request processing protocol executes in three phases

**pre-prepare:** when the request is received based on the current view the primary picks a sequence number, computes the digest of the message, and multicast a *pre-prepare request* to all backup replicas, it also stores this request in its log
backups accept the pre-prepare request if (1) the signature $\sigma$ and digest $d$ are cryptographically correct (2) the view $v$ matches the current view they are aware of (3) the sequence number $n$ is new and lies between two watermarks $h$ and $H$ to prevent the faulty primary from sending out a large sequence number to block any other request

**prepare:** a replica multicasts a *prepare message* if it accepts the pre-prepare request and adds this prepare message together with the pre-prepare request to its log, it then waits for $2f$ matching prepare messages with the same $v$, $n$, and $d$ upon which the $prepared(m, v, n, i)$ predicate is set to be true marking the completion of the prepare phase

**commit:** a replica multicasts a *commit message* and adds this commit message to its log, it then waits for $2f$ matching commits and evaluates a *committed-local*$(m, v, n, i)$ predicate, which is true if *prepared*$(m, v, n, i)$ is true and $2f + 1$ matching commits have been seen (including its own), setting *committed-local*$(m, v, n, i)$ predicate to be true marks the completion of the commit phase and it executes and sends a response to the client

The paper also discusses the details of garbage collection, view change due to corrupted primary, using timeout to guarantee liveness, as well as optimizations such as reducing the number of messages and overlap in message processing. Based on pBFT the authors developed a distributed Byzantine-fault tolerant file services called BFS.

## 16.5 Byzantine Consensus vs. Blockchain

An underlying technology enabling blockchain is that of a distributed ledger, which is a timestamped sequence of records that is replicated across distributed machines in a consistent agreed upon manner. Each node agrees on the order and content of the ledger entries regardless of failures. It is unique and cannot be altered by some participants which must be achieved without using centralized clearinghouse for reaching an agreement.
pBFT allows us to achieve decentralized consensus in an unreliable network even in the presence of Byzantine failures. However the key requirement of having $N = 3f + 1$ nodes in order to tolerate up to $f$ faults makes it unsuitable for implementing blockchain. Because the number of participants is unknown in advance and an attacker can create many faulty instances of themselves. Moreover even it is able to put a bound on $N$, the number messages required by pBFT scales with $O(N^3)$ which is quite costly. Instead the technical elements that formed the distributed ledger mechanism of blockchain combine ideas from Byzantine consensus as in pBFT with a few other ideas:

- not everyone is able to participate in maintaining the ledger, which is achieved by having the participants solve cryptographically challenging puzzles and present a proof of work (mining in the blockchain ecosystem)
- an incentive structure is built into the blockchain design to reward good behavior via cyptocurrencies so that participants have more to gain if they behave correctly compared to if they behave maliciously, in particular miners receive cryptocurrency if they can create a new block in the system and they can collect fees for transactions included in the block

The combination of these factors makes it possible to probabilistically establish a much lower requirement for the number of nodes needed to reach an agreement and the number of messages needed for an update. Interestingly the famous white paper by Satoshi Nakamoto that introduces the bitcoin concept does not explicitly mention Byzantine fault tolerance.

# 17 Lesson 17: Edge Computing & IoT

## 17.1 Tiers in Computing

Traditional we have end-user applications and devices (tier 3) interact with services deployed in remote cloud data centers (tier 1). This is true both for traditional client devices and new application classes such as smartphones and AR/VR devices, the latter creating huge demand for bandwidth and requiring low latency, which cannot be met with current service deployments based on remote clouds. In response a new tier of infrastructure emerges outside of datacenters in the access regions of the network closer to end-user devices (tier 2), such as vehicular and luggable micro-datacenters.
There is another emerging tier based on very low power devices (tier 4), which encapsulate basic sensing

capabilities and connectivity, and even some computational capabilities that allow them to perform certain data processing or act as actuators. Many of these do not have a reliable source of energy and are accessible only intermittently, which present new challenges beyond traditional embedded systems. The differences among these four tiers with respect to device capabilities, usage models, and application requirements are so stark that this trend presents a need to change the design assumptions of distributed systems.

## 17.2 Why Edge Computing

The annual virtual network index published by Cisco shows a continual growth in the number of devices, traffic volume, and demand for wireless bandwidth. The demand is in part driven by the emergence of new bandwidth-intensive and latency-sensitive workloads such as high-definition videos, AR & VR, smart city & automation. In addition providing connectivity requires capacity forecast and planning to determine where and how much infrastructure should be deployed. The pandemic has shifted this almost overnight, and this trend is likely to stay beyond pandemic to some extent given that many companies have adopted remote work on permanent basis.

## 17.3 Closing the Latency/Bandwidth Gap

One way to meet the increasing demand for bandwidth is to adopt new technology such as 5G, which however is costly to deploy and takes years to have meaningful penetration. In fact due to cost the rollout of new technology is always slow. On the other hand the commoditization of the communication infrastructure creates the opportunity to tap into the computational and storage resources at the edge of networks to migrate some workloads from a remote cloud closer to the end-users. We refer to this trend as **edge computing**, which was initially proposed for mobile networks and then extended to the more general multi-access edge computing (MEC). What this approach really does is it trades one type of resources for another—we use the computational resources that are available closer to the end-user to satisfy its demand for connectivity.

## 17.4 Is Edge Computing New

Over half of the internet traffic is served by the content delivery network or CDN which aims at the same goal of providing better connectivity with lower latency thereby reducing the backhaul bandwidth demand. However there are only thousands of CDNs globally compared to hundreds and thousands of cellular tower in US alone. The latter is a substantially larger distribution of edge computing endpoints. To leverage this infrastructure mobile network operators partner with traditional cloud providers with the goal of transforming this new compute tier into the next cloud frontier.

## 17.5 Edge Computing Drivers

The fundamental drivers behind edge computing include speed of light (speed limits the distance of service delivery), energy (demand by data traffic), regulatory constrain/data sovereignty (mandates localized data processing).
A number of killer apps are emerging as potential drivers for MEC e.g. HD video, AR & VR. Interestingly 5G is both an enabler and a driver of MEC. Some of the processing and signaling requirements of 5G precisely rely on the presence of distributed computing at the edge of the network.
In terms of bandwidth as a driver there is a general asymmetric upload/download speed for various types of connections and countries, which is a problem since many of the use cases mentioned above have similar or even higher demand for upload bandwidth.

## 17.6 Distributed Edge Computing

A number of characteristics make the considerations of the edge environment different from the assumptions used when designing a cloud

**scale and geo-distribution:** although many of the components of datacenters are tightly coupled, thus if we take the same protocols that we use in datacenters and deploy them at the edge we may find out that they are chatty—lots of timeout messages and heartbeat messages

**elasticity:** datacenters are designed with the assumption that there are more (unlimited) nearby resources vs. edge is not elastic—we can't run it on another CPU at another edge, as such it is much important for edge computing to trade resources for reduced service level

**mobility, device churn, reliability:** datacenters are designed with very reliable technologies which is not at all the case for edges, as a result in edge computing the degree of churn, the types of devices, and the mobility can be order of magnitude more significant than in datacenters, and any fault tolerance solution for datacenters will end up with much high overheads or even ineffective

**heterogeneity:** the datacenter designs discussed so far in this course all make some assumption about some symmetry existing among nodes, which is not applicable to edge settings

**localization, contextualization:** there is a greater degree of local properties for edge computing vs. the states and processing taking place across different servers of datacenters are much more homogeneous

**decentralization:** in edge computing it is much more common to expect that not all resources are operated by a single provider

**lack of physical security:** there is a significant difference in the security assumptions at the edge, which are largely ignored for the cloud

## 17.7 IoT and Distributed Transactions

Using redo or undo log to ensure consistent execution doesn't make sense when it comes to activations of the physical environment, the physical state of the actuator may not be consistent with the application level state. As such distributed transactions cannot help. The problems are fundamentally attributed to three cases of dependencies

- when an actuation action depends on the sensing variable, the actuation should not be triggered if the sensed value does not satisfy the predicate
- the dependency of updates to the application state on the sensed value
- the dependencies among the updates to the application state and the actuation actions, both the application level state and the actuation are either both performed or neither takes place

## 17.8 Transactuations

Transactuations are a high-level abstraction and a programming model, which is specified by

**application logic:**

**sensing policy:** can be expressed in terms of the hard values required for some of the sensors specified in the sensor list, it can also include a time window that specifies the time when these sensors should be read, it can specify whether all or some or any of these requirements should be met

**actuating policy:** expresses the dependencies among the updates to the application state and the actuation of the physical devices, it can specify whether to allow all or some or any of the actuationa to be succeeded

The programming model also allows users to specify the desired behavior of transaction success or failure (e.g. commit or abort). As a result the model can provide some guarantees regarding the atomic durability of the actuations that take place in the environment. It also provides enough information so that one can schedule different updates and when to perform them so as to avoid concurrency bugs. The two concepts key in expressing transactuations and then building a runtime to enforce them are

**sensing invariant:** transactuation executes only when the staleness of its sensor reads is bounded as per specified in the sensing policy (time window, how many failed is acceptable), e.g. at least one $CO_2$ sensor was read within the last 5 mins

**actuation invariant:** transactuation commits its application state only when enough actuations have succeeded as per specified in the actuating policy (how many failed is acceptable), e.g. at least one alarm should be turned on

The resulted runtime then has sufficient information to

- insert checks at appropriate places for the different invariants and for the policies that they need to enforce
- determine when to start (according to sensing policy) or when to commit a transaction (speculative commit to find a serializable order vs. final commit according to actuating policy)
- ensure that a serializable ordering among concurrent transactions is enforced