Cloud Computing and Distributed Systems
Raja Appuswamy

Lecture 1: Introduction to Cloud Computing

We live in a world of data. Every minute, 188M emails are sent, 390K apps are downloaded, etc.
Big Data is large pool of data that can be captured, communicated, aggregated, stored and analyzed.

More data leads to better accuracy.

How will we manage all this data?

- By ourselves: store, share, access, secure, etc.
- By someone else: pay for a management "service".

Transformation of IT: Innovation → Product (buy & the technology) → Service (on-demand services)
Requirements and Supporting Technologies:

- Connectivity to move data → Networked systems
- Interactivity for user-friendly interface → Web 2.0 & Human-Computer Interaction
- Reliability against failures → Dependable systems
- Acceptable performance → Parallel and distributed systems
- Developing new services → Programming languages
- Manageability for Big Data → Storage systems
- Pay-as-you-go → Utility computing and economics
- Scalability & elasticity to change needs → Virtualization

Cloud Computing is the delivery of computing as a *service* rather than a *product*, whereby *shared resources*, *software* and *information* are provided to computers and other devices as a *metric service* over a *network*.

Why? Pay-as-you-go, simplified, scale quickly, flexible, carbon footprint decreased.

Cloud Infrastructure

What is a *server*? Server is a computer that provides services to clients designed to process many requests.
What is a *rack*? Servers are grouped, placed & organized in racks (180 cm) which can hold up to 42 servers.
What is a *data center*? Facility used to house many computer systems and their components.
What is a *cloud*? A data center hardware and software offering computing resources & services.

Basic cloud service models:

- *SaaS* (Software-as-a-Service): no need to install and run an application on the computer, the software is delivered as a service over the internet and running on browsers (Gmail, google docs).
- *PaaS* (Platform-as-a-Service): a set of tools and APIs which allow users to create SaaS applications running on the provider's infrastructure (Google App Engine, Microsoft Azure).
- *IaaS* (Infrastructure-as-a-Service): providing computer infrastructure to access a standard Operating System environment and install & configure all the layers above it (Google Compute Engine).

Types of clouds: *public* (external: for on-demand resources), *private* (internal: for large enterprises) & *hybrid* (use local cloud and extend to public when needed).

Cloud Provider Leaders: Amazon, Microsoft, Google, IBM, Alibaba.

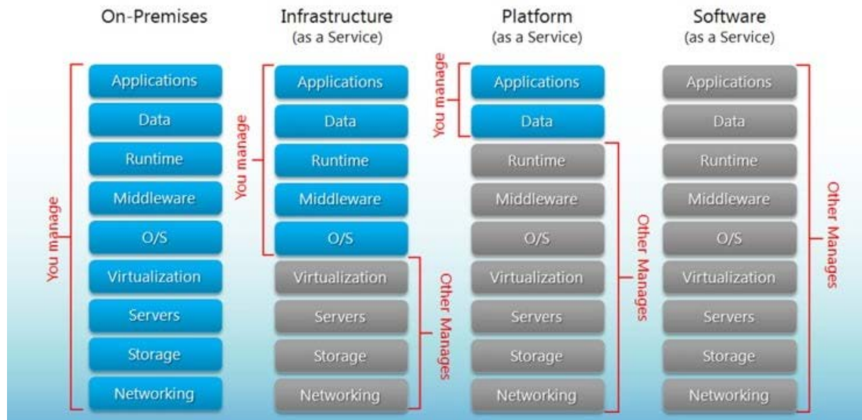


Figure 1 - Cloud Service Models and The Cloud Stack

Cloud Economics

There are three primary costs for using IT:

- Software cost (media + license)
- Support cost (support + updates)
- Management cost (Manpower + infrastructure)

Software Service Models

<table>
<tr><th>Model</th><th>Traditional</th><th>Open Source</th><th>Outsourcing</th><th>Hybrid</th><th>Hybrid+</th><th>SaaS</th></tr>
<tr><td>Software Cost</td><td>$4000 /user (one time)</td><td>$0 /user</td><td>$4000 /user (one time)</td><td>$ 4000 /user (one time)</td><td rowspan="3">$300 /user /month</td><td rowspan="3">< $100 /user /month</td></tr>
<tr><td>Support Cost</td><td>$800 /user /year</td><td>$1600 /user /year</td><td>$800 /user /year</td><td>$800 /user /year</td></tr>
<tr><td>Management Cost</td><td colspan="2">Up to 4x the cost of the software</td><td>< $1300 /user /month</td><td>$150 /user /month</td></tr>
<tr><td>Deployment Location</td><td colspan="2">Client side</td><td colspan="3">Client or Provider Side</td><td>Provider Side</td></tr>
</table>

Lecture 2: Fundamentals of Cloud Computing

Economic Fundamentals

Utility Pricing: should you go for public cloud if the unit price is higher than a home-grown solution?
Calculation:

- $L(t)$: load (demand for resources) where $0 < t < T$.
- $P = \max L(t)$: Peak Load
- $A = \text{avg}\, L(t)$: Average Load
- B: Baseline (owned) unit cost; B_T: Total Baseline Cost
- C: Cloud Unit Cost; C_T: Total Cloud Cost
- $U = C/B$: Utility Premium

$$B_T = P \times B \times T$$
$$C_T = \int C \times L(t)dt = A \times C \times T = A \times U \times B \times T$$

Cloud is cheaper than owning when $C_T < B_T \rightarrow A \times U \times B \times T < P \times B \times T \rightarrow \boldsymbol{U < P/A}$
→ When Utility Premium is less than Peak to Average load ratio, Cloud Computing is beneficial.
→ Utility Pricing is good when demand varies over time (start-up or seasonal business).

Multiplexing: combining many infrastructures (e.g. one built for peak requirements and one built for average use) into a bigger one.

→ Higher utilization and lower cost
→ Reduce unserved requests

A measure of smoothness: let's introduce a coefficient of load variation $C_v = \sigma/|\mu|$: the larger the mean for a given sd, the smoother the load curve is.
→ for workloads with low smoothness (high C_v), fixed asset servicing in bad.

Case study: let $X_1, \dots, X_n$ be n independent jobs (random variables) with identical sd and positive mean.
Let $X = X_1 + \dots + X_n$ (multiplexing), so $\mathbb{E}[X] = n\mu$ and $\mathbb{V}[X] = n\sigma^2$, finally $C_v(X) = \frac{\sqrt{n}\sigma}{n\mu} = \frac{\sigma}{\sqrt{n}\mu} = \frac{C_v(X_i)}{\sqrt{n}}$
→ We obtain a smoother aggregate load with multiplexing.
Best case (negative correlation): consider jobs X and $1 - X$. Multiplexing: $Y = X + 1 - X = 1$. $C_v(Y) = 0$: optimal smoothness, best CPU utilization.

Takeaway lesson: cloud provider of any size can be profitable!

Infrastructure Fundamentals

Sharing resources: how to share a physical computer among multiple applications?

VMs

Virtual machine: "a fully protected and isolated copy of the underlying physical machine's hardware."
Virtual Machine Monitor: "a thin layer of software between the hardware and the OS, virtualizing and managing all hardware resources."
Two types of hypervisors:

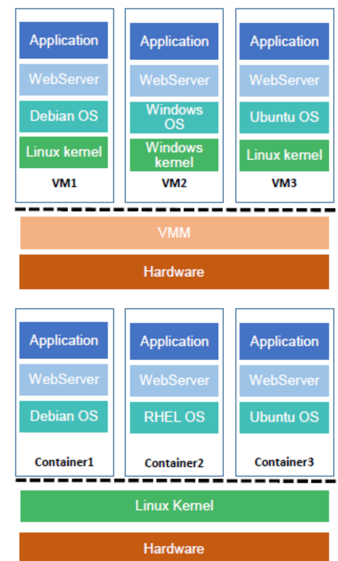
- VMM runs directly on top of the physical hardware and performs scheduling and resource allocation.
- VMM built on top of a host OS. The OS provides resource allocation to each guest OS.

Virtualization and the Cloud: virtualization is the enabler of IaaS
The cloud provider leases to users VM Instances (i.e. computer infrastructure) using virtualization technology
The user has access to a standard OS environment and can install & configure all the layers above it.

Containers

Linux containers is an OS-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. It provides *cgroups* functionality to limit and prioritize resource usages among different groups.

Docker was developed in 2013 to enable packing application and all dependencies as a container images after development to ensure that it runs similar on test and production systems.



Containers vs. VMs:

- Abstraction levels
  - Hypervisors work at hardware abstraction level
  - Containers work at OS abstraction level
- Density
  - VMs need $O(GB)$ vs. containers need $O(MB)$
  - Can pack many more containers per server
- Elasticity
  - Containers are easy to scale up
  - Everything in Google is containerized
- Software development lifecycle
  - On containers, it's easy to build, test and deploy software without worrying about portability

Serverless functions

Run user handlers in response to events (web request, db update)
Pay per function invocation (pay-as-you-go, no charge for idle time between calls)
Share server pool between customers (any worker can execute any handler)

Lecture 3: Programming models & runtime: MapReduce

To increase power, supercomputers are made of distributed/parallel computers (1000s processors, low latency)

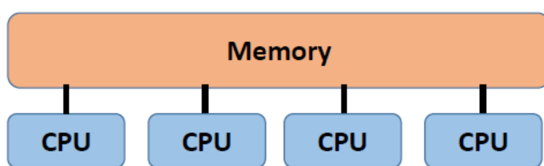
How do we program supercomputers?

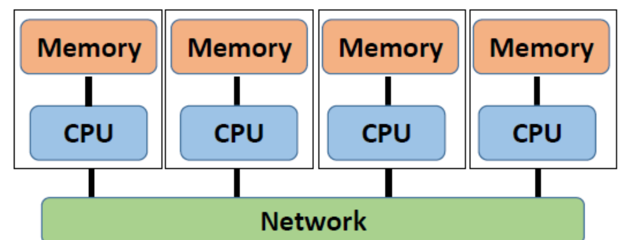
Shared memory parallelism:

- Cache coherent: store made by one CPU is visible to load by other CPU
- Shared memory: any CPU can access any memory location

Message passing:

- No cache coherent, no shared memory
- Need to communicate across machines by sending messages using a Message Passing Interface





MPI Communicators: communication domain: a set of process allowed to communicate with each others. Communicators are used as arguments to all messages transfer MPI routines.

MPI_Comm_size: the number of processes

MPI_Comm_rank: the label of calling process $\in \{0, \ldots, size\}$

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,int dest,
     int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Example

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
```

Deadlock: MPI_Send is blocking that means the program will not continue until the communication is completed.

How to avoid? Break circular wait (odd processes send message and even processes receive)

Other functions: MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce (receive an array as input and apply a function such as sum to each element then send the output to the root process), MPI_Allreduce (same as previous but send the output to all processes).

Parallelization Challenges

How to deal with failures?

- Checkpoints: periodically write out state of all processes and restore when failure occurs. But significant data might be written between two checkpoints.

Load balancing: how to split data across workers to keep all machines busy?

Synchronization: how do workers access a shared resource?

→ Express a problem declaratively (describe what to do not how to do, e.g. SQL)

→ Express a problem functionally

- Map: takes a function $f: x$ as argument and apply it to each elements of a list. In what order? Any one we want even in parallel.
- Fold: takes a function $g: x, y$ and an initial value as arguments. g is applied to each element of the list and the output of g in the previous step. e.g. $g^{(i)} = (g^{(i-1)}, l[i])$ and $g^{(0)} = g(v_0, l[0])$

Google's MapReduce

Key-value pairs (integers, floats, strings or any data structure) are the basic data structure in MapReduce

→ Mapper: applied to every key-value input in order to generate intermediate key-value pairs

$$map: (k_1, v_1) \rightarrow [(k_2, v_2)]$$

→ Reducer: applied to the values associated with the same intermediate key to generate output key-value pairs

$$reduce: (k_2, [v_2]) \rightarrow [(k_2, v_3)]$$

NB: There is an intermediate step (shuffle and sort) that produces pairs $(k_2, [v_2])$

Steps:

1. Files are split into $16 - 64MB$ pieces
2. Master picks workers and assign mappers and reducers
3. Map worker reads input split, calls map function, buffers map output to memory
4. Periodically flush in-memory data to disk & master is informed of disk location
5. Master notifies reduce worker of the location, reduce worker reads map output, sorts data
6. Reduce worker iterate over sorted data, passes values associated to unique key to reduce function

Combiner function: pre-aggregate function at the mapper level to decrease size of intermediate data.

MapReduce benefits:	MapReduce drawbacks:
• Large-scale computations • Scales well • Easy to program	• Does not fit small data (high overheads) • Does not fit to small updates to Big Data • Does not fit to unpredictable reads

Lecture 4: Programming models & runtime: Memory Hierarchy & Spark

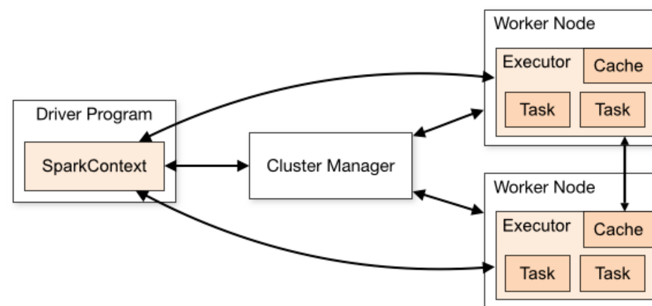
MapReduce: simple programming model for building distributed applications to process vast amounts of data.
Hadoop: makes MapReduce broadly available.

MapReduce is entirely disk-based: input and output sit on HDFS.
Example: k-means algo (each iteration reads and writes data from disk-based HDFS: this is bad!)
HDFS read → Map (assign sample to centroid) → Network Shuffle → Reduce (new centroids) → HDFS write
The 5-minute rule: if data is accessed more than once within 5 minutes, cache it in memory.

Solution? Spark. Exploit memory by caching data to enable fast data sharing.

Spark Fundamentals

- Spark Context: entry point to Spark API (holds config information) → applications are isolated
- RDD: immutable, partitioned collection of objects
- Transformations: define new RDD based on existing one (textFile(), filter(), map(), groupByKey())
- Actions: return values (count(), take(n), collect())



RDD & Spark
Data id RDD is not processed until an action is performed (an action triggers computation)

- DAG: The Scheduler splits the Spark RDD into stages based on various transformation applied.
- RDD can be cached for faster reuse
- RDD can be automatically rebuilt on failure

RDD Summary

- Resilient: recover from node failure
- Distributed: partitioned across the cluster
- Dataset: RDD created from a file

Spark Summary

- Easy to use (Java, Python, Scala)
- Generality: for different workloads (batch, streaming, ML)
- Performance: low latency (caching)
- Fault-tolerance: immutability of RDD can be used for precomputation

Lecture 6: Cloud Data Management: Distributed File Systems

Relational operations over MapReduce

	Map	Reduce
Selection	For each tuple t, check C Emit a key/value pair (t, t)	Identity reducer
Projection	For each tuple t, construct t' by eliminating useless columns Emit a key/value pair (t', t')	For each t' key, fetch t' Emit a key/value pair (t', t')
Union	For each tuple t in S/R emit (t, t)	For each t key there will be one or two values: always emit (t, t)
Intersection	For each tuple t in S/R emit (t, t)	If t key has value $[t, t]$, emit (t, t) Otherwise, emit $(t, NULL)$
Difference	Emit pairs (t, R) and (t, S)	If value $[R]$, emit (t, t) If $[R, S]$, $[S, R]$, $[S]$ emit $(t, NULL)$
Natural Join	Emit $\big(b, (R, a)\big)$ and $\big(b, (S, c)\big)$	For each key b, emit $(b, [(a_1, b, c_1), \dots, (a_n, b, c_n)])$

RDBMS vs. MapReduce: designed to meet different requirements

- Relational DB
  - Guarantees ACID properties
  - Optimized for random access and scans
- MapReduce
  - Latency-insensitive
  - Focus on faults during query rather than recovery after update

Networked File System: let authorized users to access their files from any computer

RCP: a request-response protocol to communicate between a client and a server.

Marshaling: converting to local representation (e.g. little-endian, big-endian)

RPC challenges:

- Communication failures: delayed/lost messages, connection reset
- Machine failures: server/client

Naïve FS design: use RPC to forward every FS operation to the server.

\+ Same behavior as if both programs were running on the same local filesystem

− Latency accessing the server is higher than accessing local machine

How to avoid going to the server for everything? **Client-Side Caching**. What to cache?

- Read-only file and directory data
- Data written by the client machine
- Data written by other machines

→ Problem 1: Consistency problems!

→ close-to-open consistency: always ask server before `open()` (loose a bit of performance for consistency)

→ Problem 2: failures (data in memory not in disk, lost messages, client crashes and client cache loss)
→ use unique ID that will be used once (e.g. delete(1337f00f) instead of delete(foo)),
→ after closing a file, .close() does not return until modified blocks are received by the server

Andrew File System: to have a consistent namespace for files across computers.
Aggressive caching: AFS caches to disk in addition to RAM
→ Lower server load than NFS (more files cached), but may be slower
→ For both, server is a bottleneck: reads/writes hit it at least once each file use

Google File System: tradeoff between consistency, performance and scalability
Workload characteristics: files are huge, most file updates are appends, high bandwidth
The client reads metadata from master but data from chunk servers to avoid the single-master bottleneck
Chunk size: 64 MB (vs. 512B – 8KB for normal FS) → less load on server.
Operations:

- Read:
- Update (write/append): for consistency, updates to each chunk are ordered in the same way for all replicas

Lecture 7: Consistency Models

Replication is the process of maintaining the data at multiple computers to:

- Improve performance (the client accesses nearest copy of data)
- Increase the availability of data
- Secure against malicious attacks

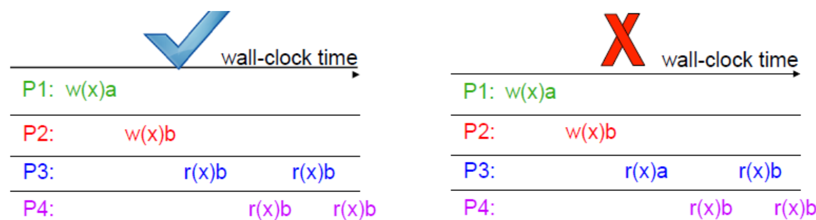
Challenge: keep the data **consistent**, because different processes can read/write different copies!
Distributed **S**hared **M**emory: each machine can access a common address space and has a local copy:

- Read: from local copy
- Write: send update msg to each host

Problem? A machine can receive data in different order due to network latency: unexpected behavior.

Model 1: Strict Consistency: each operation is stamped with a global wall-clock time

1. Each read gets the latest written value
2. All operations at one CPU are executed in the timestamp order



Unfortunately, time between instructions << speed of light
→ Clock Synchronization.

Cristian's Time Sync

Process p requests time from server at t and sets its clock to $t + T_{round}/2$

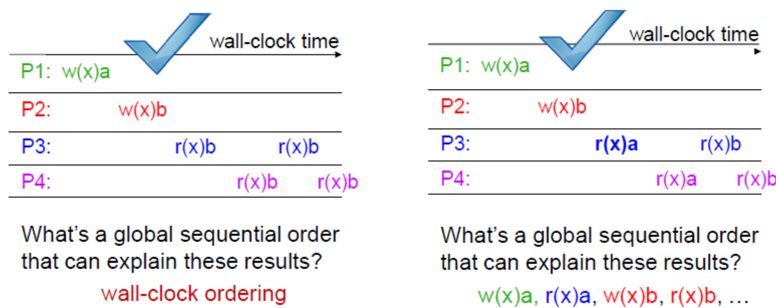
The Barkley Algo:

1. The time daemon asks slaves for their clock values
2. It gets answers
3. The time daemon tells everyone how to adjust clock based on an average value

→ strict consistency is tough to implement efficiently since clock are never exactly synchronized.

Model 2: Sequential Consistency: doesn't assume real time

1. Each machine's operations appear in order
2. All machines see results according to total order



Easier to implement then strict consistency because no notion of real time.

What matters is not the real time but the order.

Happens-before relationship: $e \to e'$ (we say e happens before e') is defined as:

- Local ordering $e \to e'$ if $e \to_i e'$ for any process i
- Messages: $\text{send(m)} \to \text{receive(m)}$
- Transitivity: $e \to e''$ if $e \to e'$ and $e' \to e''$

We say that e is concurrent to e' ($e||e'$) if neither $e \to e'$ nor $e' \to e$

Lamport Logical Clock: maintain local clock for each process and increment it for each local event. For a receive-a-message event set the clock to max between previous local event and received event + 1.

Vector Clock: each process maintains a vector $\texttt{VC}$ of length #processes and $\texttt{VC[i]}$ is the clock of p_i

Model 3: Casual Consistency: all processes agree on the order of the causally related operations.

Lecture 8: Atomic Commitment

Update different copies of object O (saved in different nodes) in the same order (consistency).

All nodes must commit or abort TXNs (atomicity).

Agreement: nodes act in the same way (fault-tolerance).

Failure Models: synchronous (bounded delay of machine/network) and asynchronous (arbitrary delay)

Atomic commitment problem: participants have constraints on values they will agree on.
Consensus problem: participants can accept any value, they have just to agree.

One-phase commit: a coordinator broadcasts the commit to participants and waits until all reply
Two-phase commit:

1. Call each participant and ask for availability (voting)
2. If all available, recall for commit (committing)
3. Else abort.

Recovery: log the state-changes
2PC is safe but not live: blocking protocol: if the protocol fails, participants will never solve their TXNs.

3PC: split commit/abort phase into 2 phases:

1. Communicate the outcome to others
2. Let them commit only after everyone knows the outcome

→ Doesn't block: always make progress by timeout
Disadvantages: long latency to complete TXNs, not totally safe: e.g. in network partitions (no communication between participants).