CS162
Operating Systems and Systems Programming
Lecture 24

Chord and Berkeley Data Analytics Stack (part I)

November 20th, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu

Outline

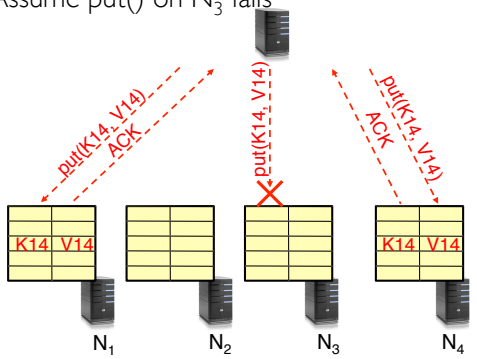
- Quorum consensus
- Chord
- Mesos

Quorum Consensus

- Improve put() and get() operation performance
- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
  - get() waits for responses from at least R replicas
  - W+R > N
- Why does it work?
  - There is at least one node that contains the update
- Why might you use W+R > N+1?

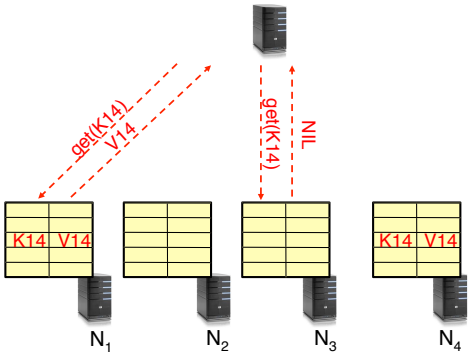
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N_1, N_3, N_4}
- Assume put() on N_3 fails



Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer



Outline

- Quorum consensus
- Chord
- Mesos

Scaling Up Directory – Challenges

- Directory contains a number of entries equal to number of (key, value) tuples in the system
- Can be tens or hundreds of billions of entries in the system!

Scaling Up Directory – Strawman Solution

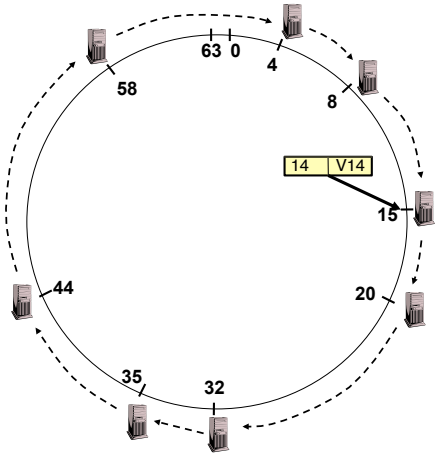
- Assume a system with m nodes
- Store $(key, value)$ on node $i = key \bmod M + 1$
- No need to keep any metadata!
- Challenge: what happened if you take away a node or add a new node?

Scaling Up Directory – Consistent Hashing

- Associate to each node a unique *id* in an *uni*-dimensional space $0..2^m-1$
  - Partition this space across *m* machines
  - Assume keys are in same uni-dimensional space
  - Each (Key, Value) is stored at the node with the smallest ID larger than Key

Key to Node Mapping Example

- m = 6 → ID space: 0..63
- Node 8 maps keys [5,8]
- Node 15 maps keys [9,15]
- Node 20 maps keys [16, 20]
- …
- Node 4 maps keys [59, 4]



Scaling Up Directory

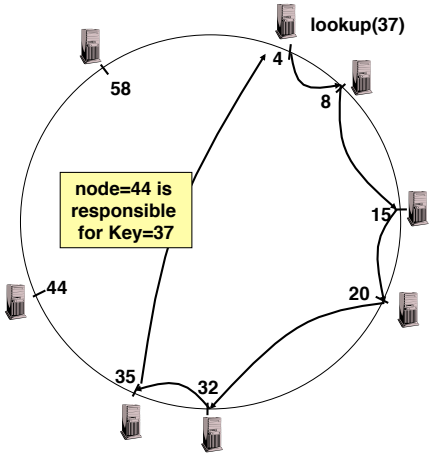
- With consistent hashing, directory contains only a number of entries equal to number of nodes
  - Much smaller than number of tuples
- Next challenge: every query still needs to contact the directory
- Solution: distributed directory (a.k.a. lookup) service:
  - Given a **key**, find the **node** storing value associated to the key
- Key idea: route request from node to node until reaching the node storing the request's key
- Key advantage: totally distributed
  - No point of failure; no hot spot

Chord: Distributed Lookup (Directory) Service

- Key design decision
  - Decouple correctness from efficiency
- Properties
  - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
  - Guarantees that a tuple is found in O(log(*M*)) steps
- Many other lookup services: CAN, Tapestry, Pastry, Kademlia, …

Lookup

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
- E.g., node=4 lookups for node responsible for Key=37



Stabilization Procedure

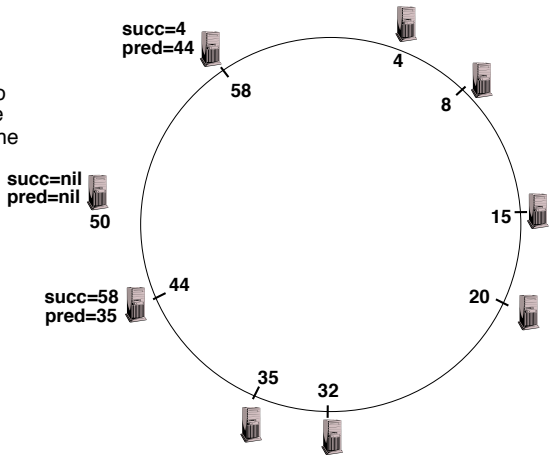
- Periodic operation performed by each node n to maintain its successor when new nodes join the system

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
     succ = x;      // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
     pred = n';     // if n' is better predecessor, update
```

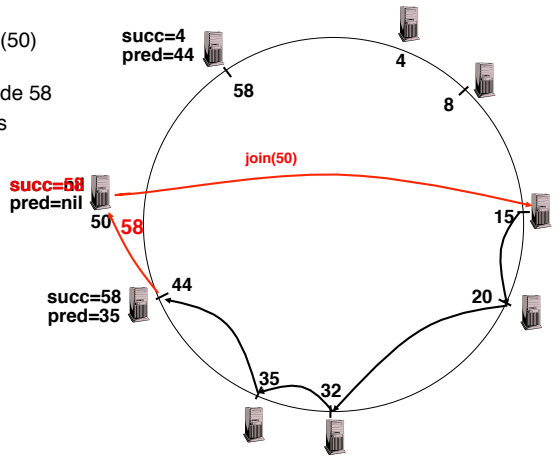

Joining Operation

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15



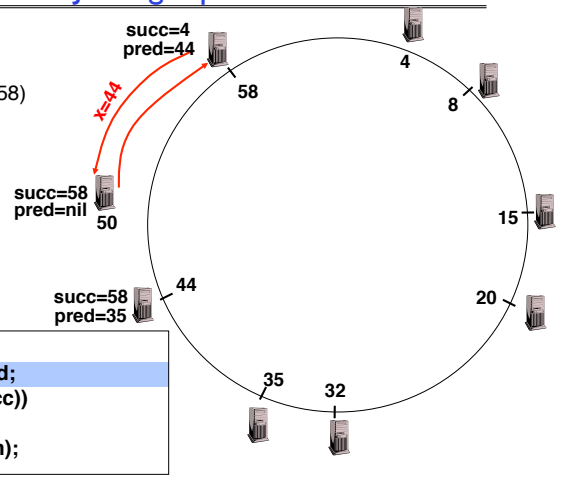
Joining Operation

- n=50 sends join(50) to node 15
- n=44 returns node 58
- n=50 updates its successor to 58



Joining Operation

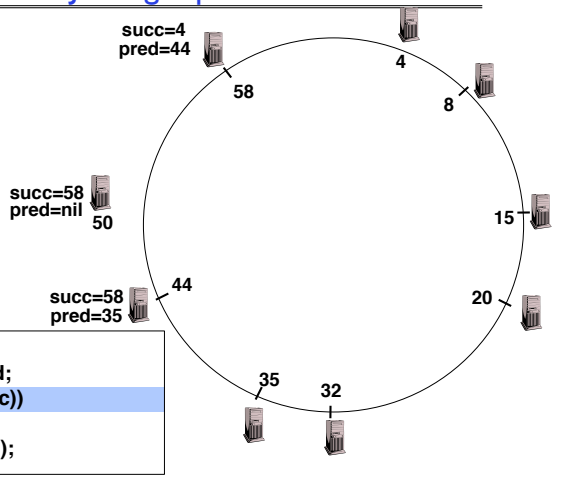
- n=50 executes stabilize()
- n's successor (58) returns x = 44




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

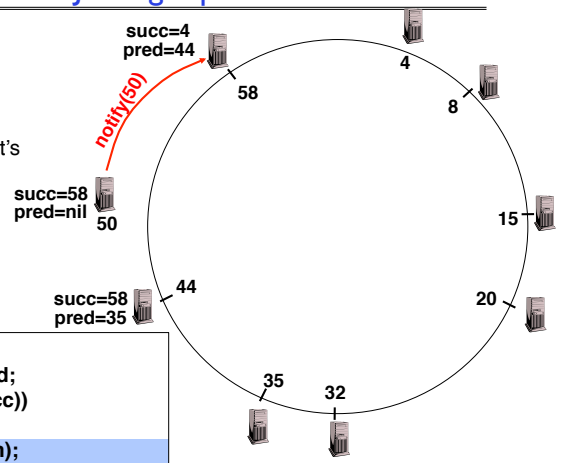
- n=50 executes stabilize()
  - x = 44
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

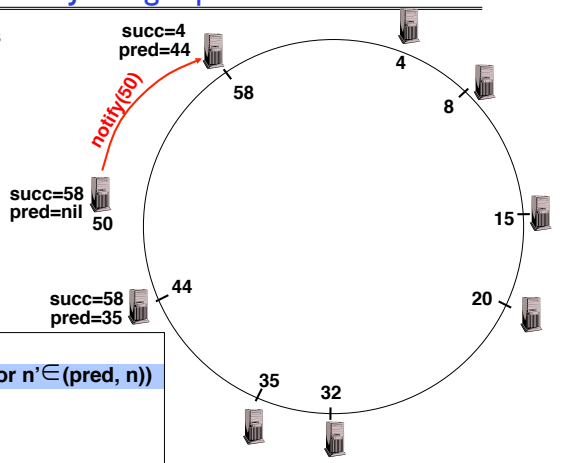
- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it's successor (58) notify(50)




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

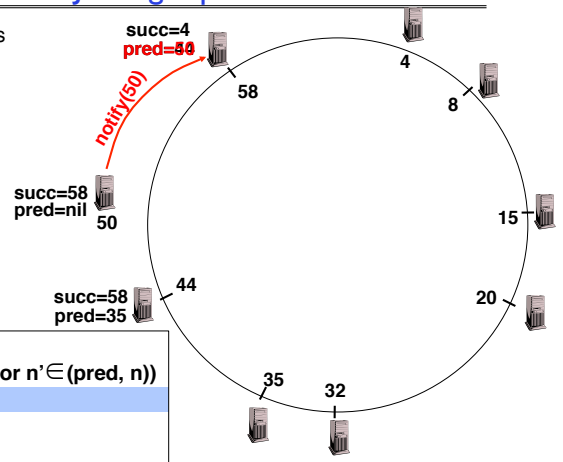
- n=58 processes notify(50)
  - pred = 44
  - n' = 50




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```

Joining Operation

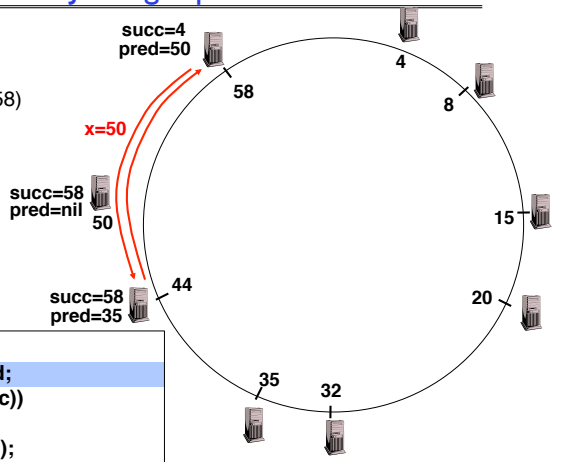
- n=58 processes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```


Joining Operation

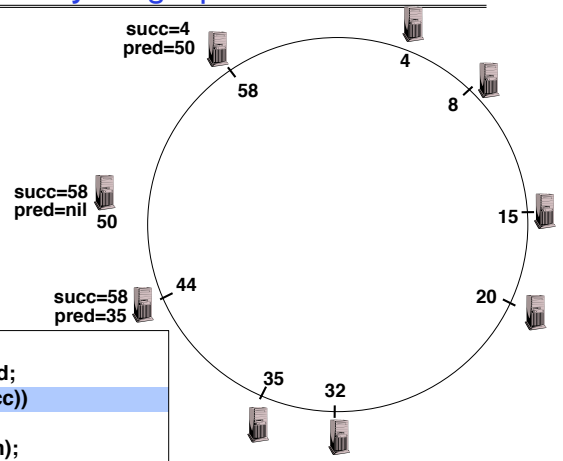
- n=44 runs stabilize()
- n's successor (58) returns x = 50




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

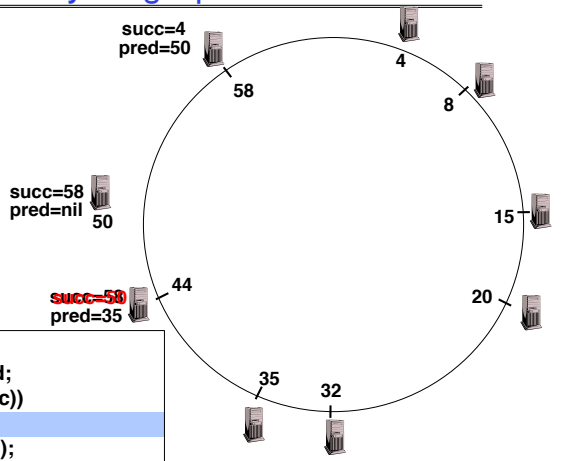
- n=44 runs stabilize()
  - x = 50
  - succ = 58




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

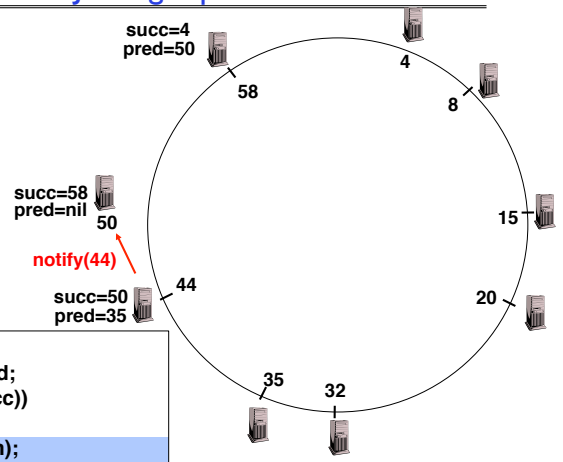
- n=44 runs stabilize()
  - x = 50
  - succ = 58
- n=44 sets succ=50




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

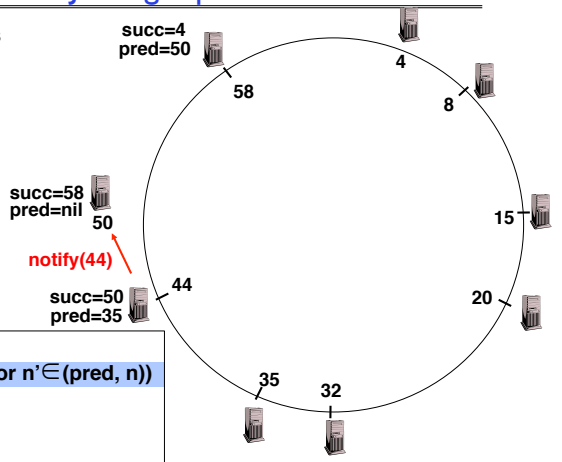
- n=44 runs stabilize()
- n=44 sends notify(44) to its successor




```
n.stabilize()
  x = succ.pred;
  if (x∈(n, succ))
    succ = x;
  succ.notify(n);
```


Joining Operation

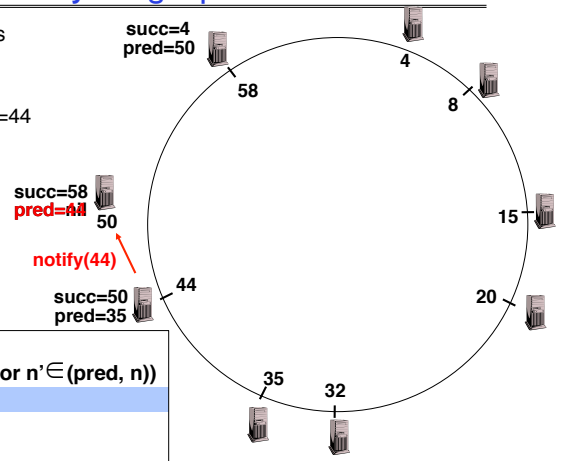
- n=50 processes notify(44)
  - pred = nil




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```


Joining Operation

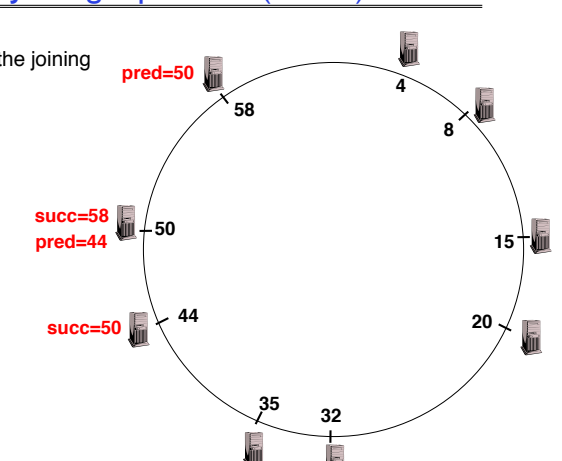
- n=50 processes notify(44)
  - pred = nil
- n=50 sets pred=44




```
n.notify(n')
  if (pred = nil or n'∈(pred, n))
    pred = n'
```


Joining Operation (cont'd)

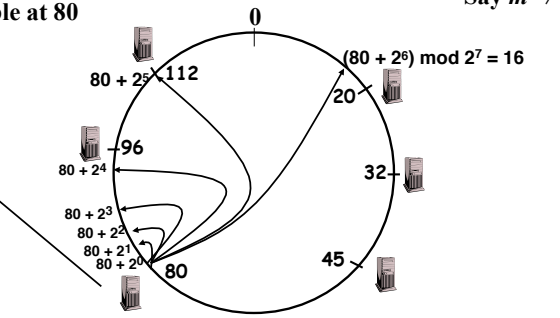
- This completes the joining operation!



Achieving Efficiency: *finger tables*

Finger Table at 80

i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	112
6	20



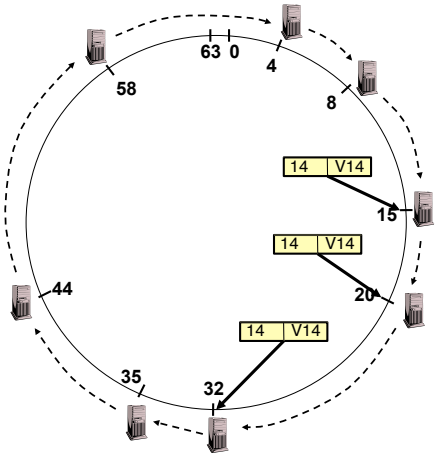
*i*th entry at node with id *n* is first peer with id >= $n + 2^i \pmod{2^m}$

Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
- In the pred() reply message, node A can send its k-1 successors to its predecessor B
- Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list
- If k = log(M), lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system

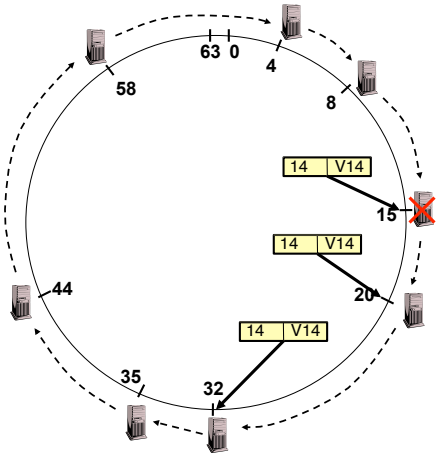
Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example: replicate (K14, V14) on nodes 20 and 32



Storage Fault Tolerance

- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed
- Will need to add a new replica on node 35



Administrivia

- Midterm 3 coming up on Wen 11/29 6:30-8PM
  - All topics up to and including Lecture 24
    - » Focus will be on Lectures 17 – 25 and associated readings, and Projects 3
    - » But expect 20-30% questions from materials from Lectures 1-16
  - Closed book
  - 2 sides hand-written notes both sides

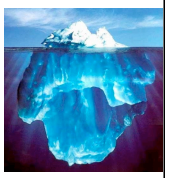
BREAK

Outline

- Quorum consensus
- Chord
- Mesos

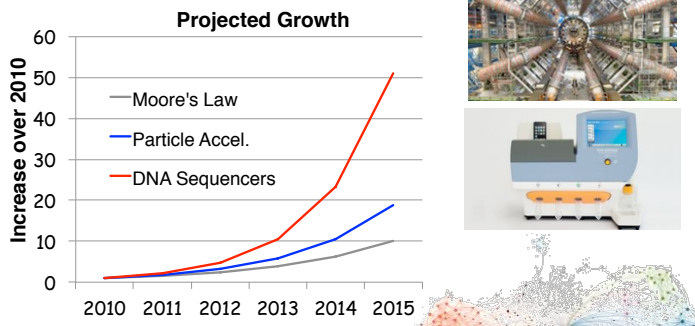
Data Deluge

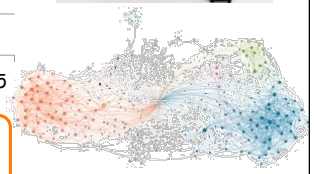
- Billions of users connected through the net
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year
  - FB building Exabyte ($2^{60} \cong 10^{18}$) data centers



- It's all happening online – could record every:
  - Click, ad impression, billing event, server request, transaction, network msg, fault, fast forward, pause, skip, …
- User Generated Content (Web & Mobile)
  - Facebook, Instagram, Yelp, TripAdvisor, Twitter, YouTube, …

Data Grows Faster than Moore's Law

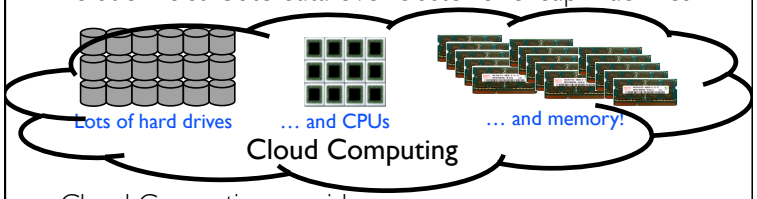




And Moore's law is ending!!

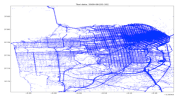
The Big Data Solution: Cloud Computing

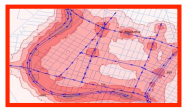
- One machine can not process or even store all the data!
- Solution: distribute data over cluster of cheap machines



- Cloud Computing provides:
  - Illusion of infinite resources
  - Short-term, on-demand resource allocation
  - Can be much less expensive than owning computers
  - Access to latest technologies (SSDs, GPUs, …)

What Can You do with Big Data?





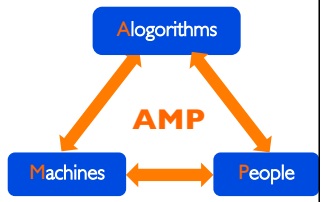
Crowdsourcing + Physical modeling + Sensing + Data Assimilation

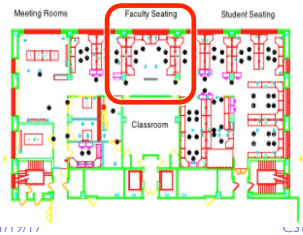
=

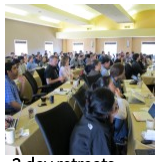
http://traffic.berkeley.edu

The Berkeley AMPLab

- January 2011 – 2016
  - 8 faculty
  - > 50 students
  - 3 software engineer team
- Organized for collaboration

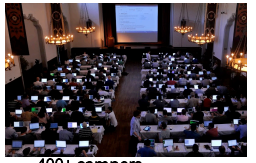






3 day retreats
(twice a year)

AMPCamp (since 2012)



400+ campers
(100s companies)

The Berkeley AMPLab

- Governmental and industrial funding:

Goal: Next generation of open source data analytics stack for industry & academia: Berkeley Data Analytics Stack (BDAS)

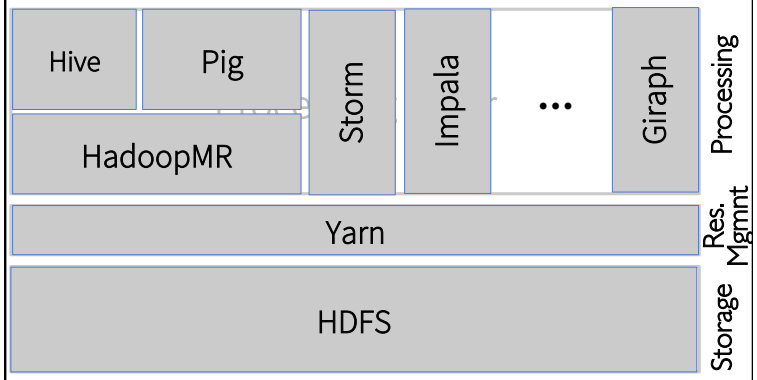
Generic Big Data Stack

Processing Layer

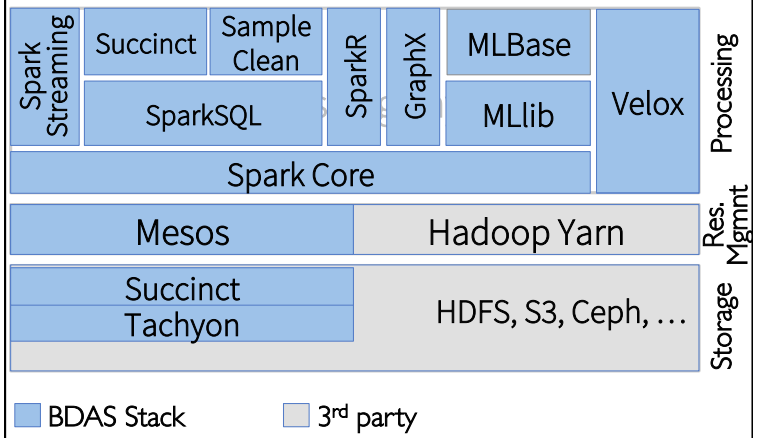
Resource Management Layer

Storage Layer

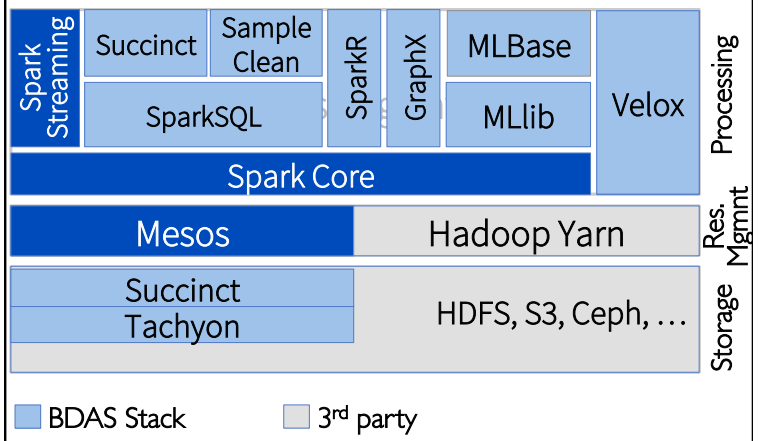
Hadoop Stack



BDAS Stack



Today's Lecture



Summary

- Mesos
- Spark

A Short History

- Started at UC Berkeley in Spring 2009
  - A class project of cs294 (Cloud Computing: Infrastructure, Services, and Applications)
- Open Source: 2010
- Apache Project: 2011
- Today: one of the most popular cluster resource management systems (OS for datacenters)

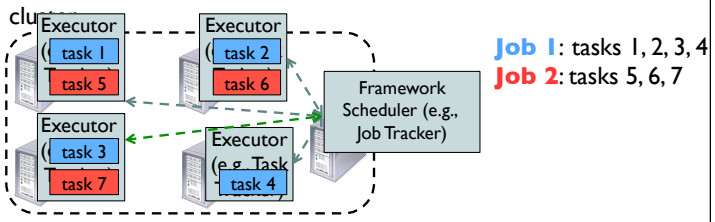
Motivation

- Rapid innovation in cloud computing (aka 2008)

- No single framework optimal for all applications
- Each framework runs on its dedicated cluster or cluster partition

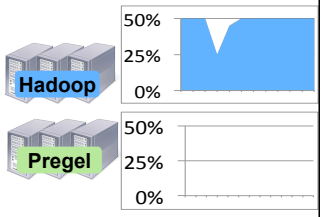
Computation Model: Frameworks

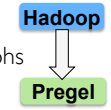
- A *framework* (e.g., Hadoop, MPI) manages one or more *jobs* in a computer cluster
- A *job* consists of one or more *tasks*
- A *task* (e.g., map, reduce) is implemented by one or more processes running on a single machine



One Framework Per Cluster Challenges

- Inefficient resource usage
  - E.g., Hadoop cannot use available resources from Pregel's cluster
  - No opportunity for stat. multiplexing
- Hard to share data
  - Copy or access remotely, expensive
- Hard to cooperate
  - E.g., Not easy for Pregel to use graphs generated by Hadoop

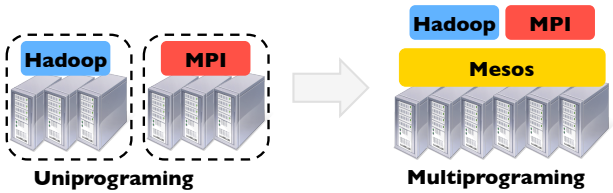




Need to run multiple frameworks on same cluster

Solution: Apache Mesos

- Common resource sharing layer
  - abstracts ("virtualizes") resources to frameworks
  - enable diverse frameworks to share cluster



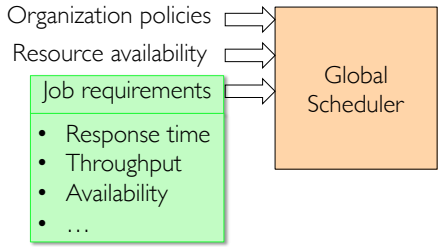
Fine Grained Resource Sharing

- Task granularity both in **time** & **space**
  - Multiplex node/time between tasks belonging to different jobs/frameworks
- Tasks typically short; median ~= 10 sec, minutes
- Why fine grained?
  - Improve data locality
  - Easier to handle node failures

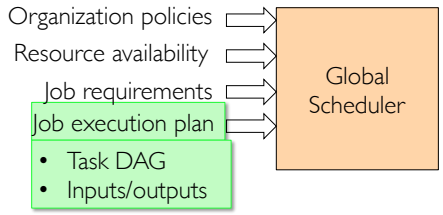
Mesos Goals

- **High utilization** of resources
- **Support diverse frameworks** (existing & future)
- **Scalability** to 10,000's of nodes
- **Reliability** in face of node failures
- Focus of this talk: ***resource management & scheduling***

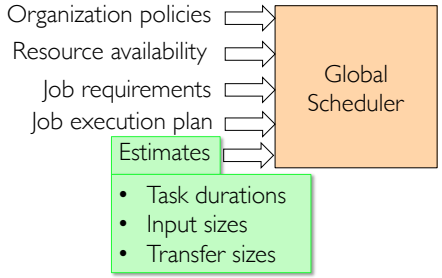
Approach: Global Scheduler



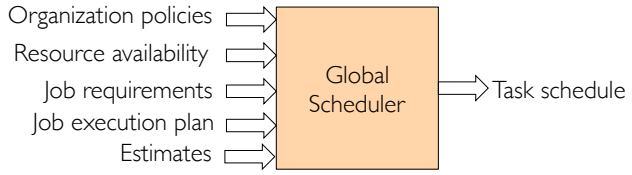
Approach: Global Scheduler



Approach: Global Scheduler

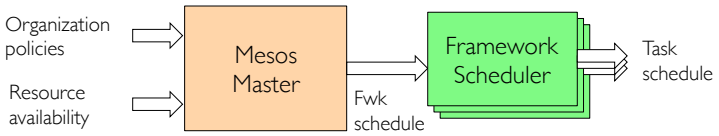


Approach: Global Scheduler



- Advantages: can achieve optimal schedule
- Disadvantages:
  - Complexity → hard to scale and ensure resilience
  - Hard to anticipate future frameworks' requirements
  - Need to refactor existing frameworks

Our Approach: Distributed Scheduler



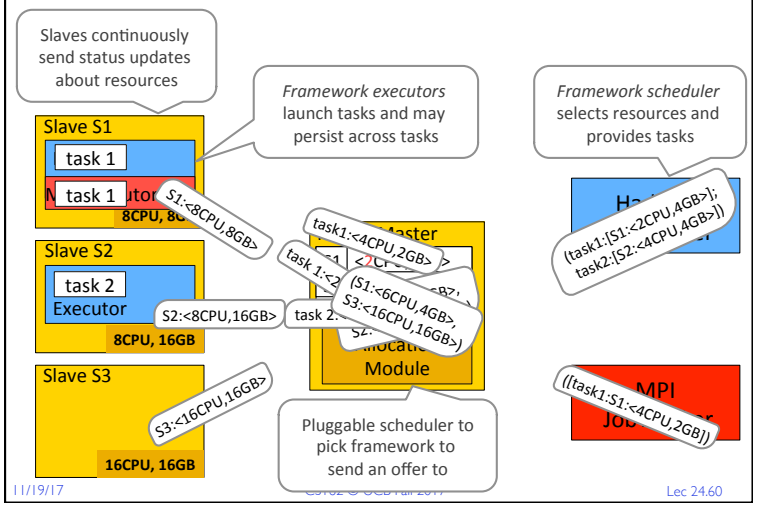
- Advantages:
  - Simple → easier to scale and make resilient
  - Easy to port existing frameworks, support new ones
- Disadvantages:
  - Distributed scheduling decision → not optimal

Resource Offers

- Unit of allocation: ***resource offer***
  - Vector of available resources on a node
  - E.g., node1: <1CPU, 1GB>, node2: <4CPU, 16GB>
- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

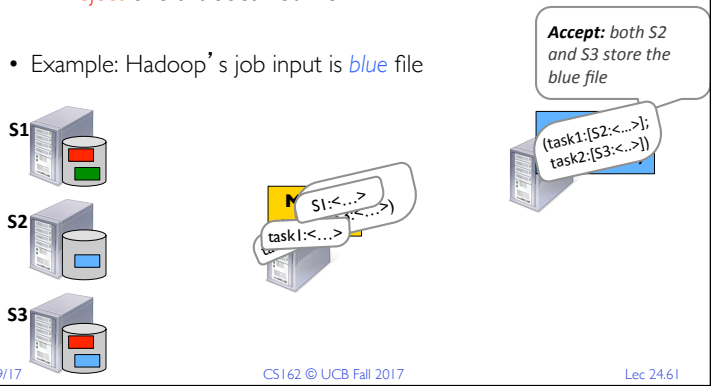
Push task scheduling to frameworks

Mesos Architecture: Example



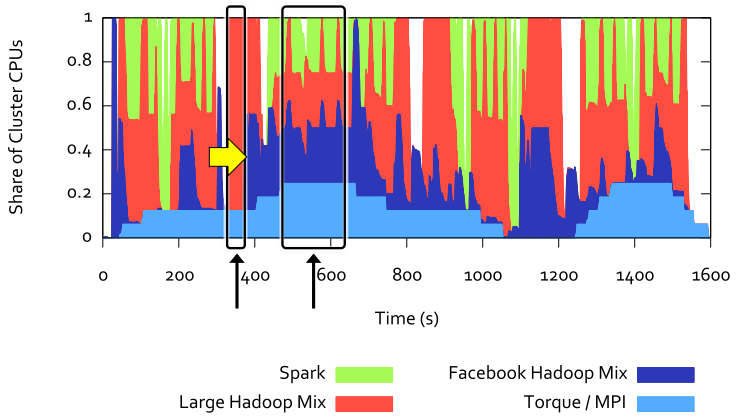
Why does it Work?

- A framework can just wait for an offer that matches its constraints or preferences!
  - Reject offers it does not like
- Example: Hadoop's job input is *blue* file



Dynamic Resource Sharing

- 100 node cluster



Apache Mesos Today

- Hundreds of of contributors
- Hundreds of deployments in productions
  - E.g., Twitter, GE, Apple
  - Managing 10K node datacenterts!
- Mesosphere, startup to commercialize Apache Spark

MESOSPHERE

Summary (1/2)

- Quarum consensus:
  - N replicase
  - Write to W replicas, read from R, where W + R > N
  - Tolerate one failure
- Chord:
  - Highly scalable distributed lookup protocol
  - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
  - Guarantees that a tuple is found in O(log(*M*)) steps
  - Highly resilient: works with high probability even if half of nodes fail

Summary (2/2)

- Mesos: large scale resource management systems:
  - Two-level schedulers
  - First level: performance isolation across multiple frameworks, models
  - Second level: frameworks decide which tasks to schedule, when, and where