UNIVERSITY OF LEICESTER

CO4219/CO7219 Internet and Cloud Computing Part II: Cloud Computing

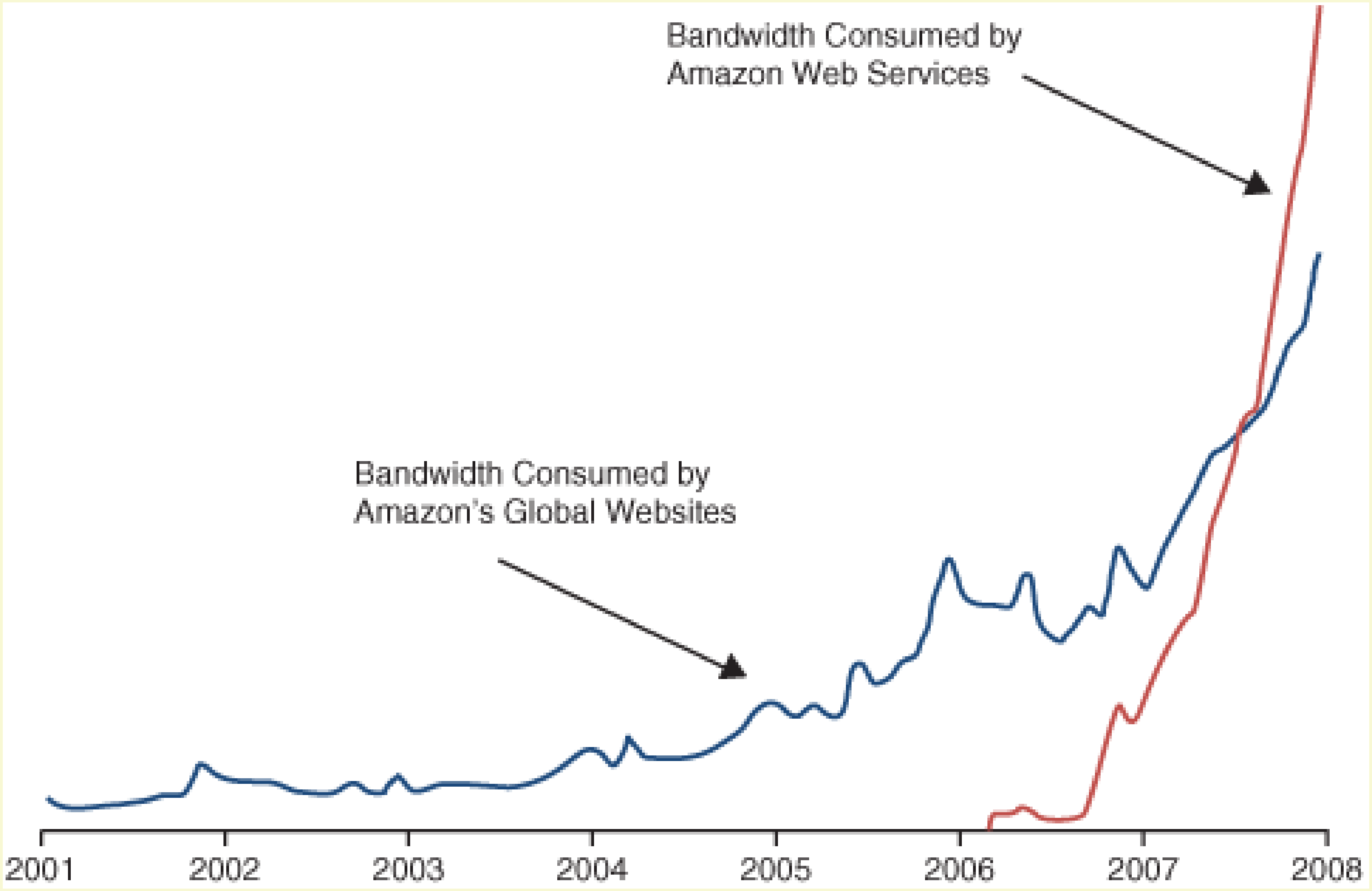
- Introduction to cloud computing
- Scalable distributed computation with MapReduce
- Hadoop implementation of MapReduce algorithms
- Privacy and security
- Fault-tolerance

Textbooks

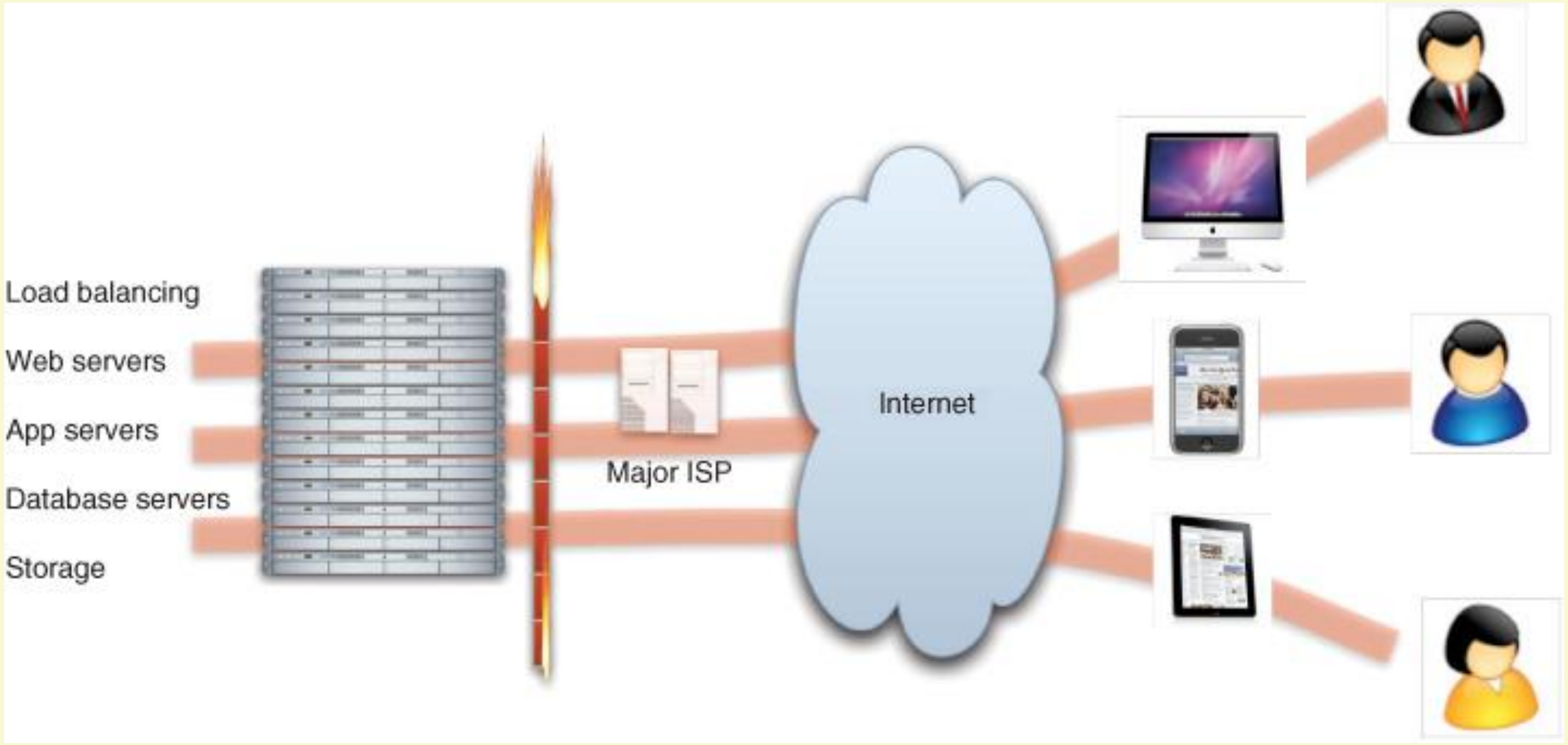
- J. Rosenberg, A. Mateos: **The Cloud at Your Service**: The when, how, and why of enterprise cloud computing. Manning Publications, 2010. (available via the Library in Safari Books Online)
- J. Lin, C. Dyer: **Data-Intensive Text Processing with MapReduce**. Morgan & Claypool Publishers, 2010. http://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf
- T. White: **Hadoop – The Definitive Guide**. 3rd Ed, O'Reilly, 2012.

What is cloud computing? (CAYS Ch. 1)

- Hottest buzzword in the IT world in the last few years
- Computing services offered by a third party, available for use when needed, that can be scaled dynamically, in response to changing needs
- First public cloud offering from Amazon attracted 500,000 customers in the first 18 months.


Bandwidth Consumed by Amazon Web Services
Bandwidth Consumed by Amazon's Global Websites
2001
2002
2003
2004
2005
2006
2007
2008

Origin of the "cloud" metaphor

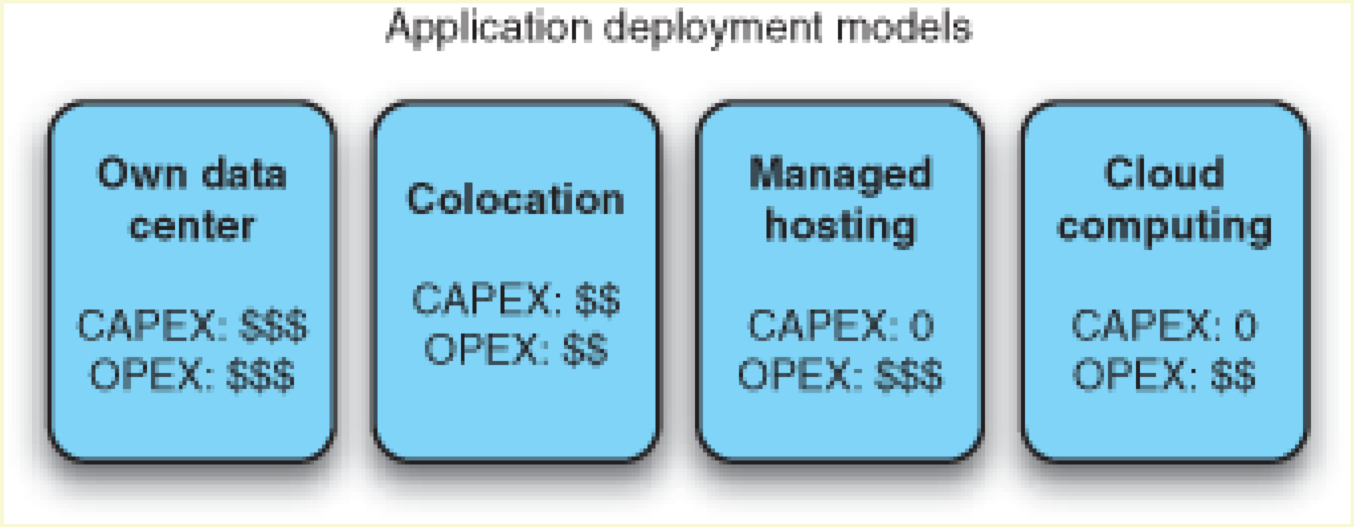


Five main principles of cloud computing

- **Pooled computing resources**, available to any subscribing user
- **Virtualized computing resources** to maximize hardware utilization
- **Elastic scaling** up or down according to need (dynamic scale without capital expenditure)
- **Automation** – build, deploy, configure, provision and move, without manual intervention
- **Metered billing** – per-usage business model, pay only for what you use

Pooled computing resources

- Hosting choices for IT organizations:



- CAPEX=capital expenditure, OPEX = operational exp.
- Cloud computing exploits economies of scale

UNIVERSITY OF LEICESTER

Virtualization of compute resources

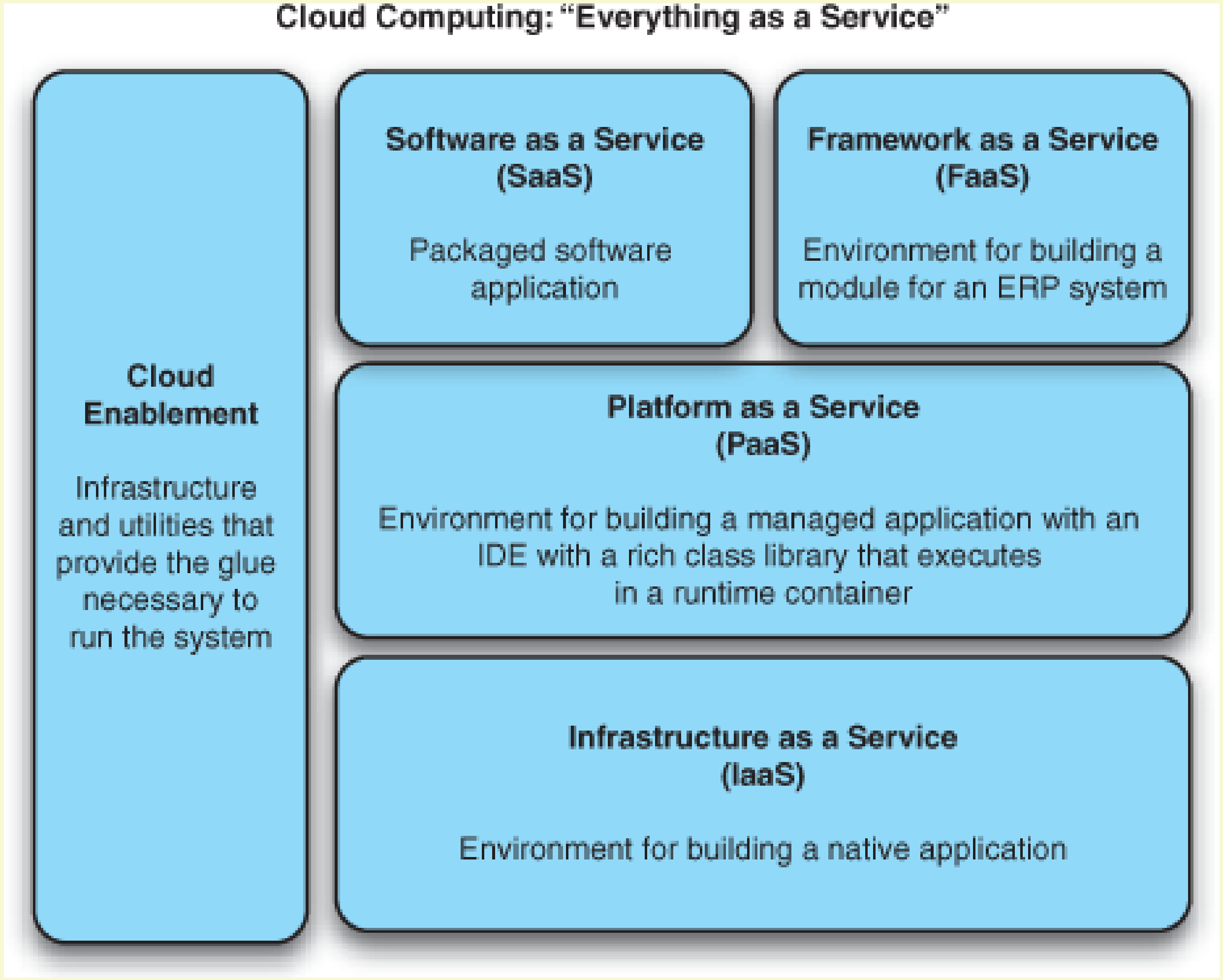
- Each physical server is partitioned into many virtual servers
- Each virtual server can run an operating system and applications
- Shift to multicore servers strengthens the impact of virtualization
- Virtualization increases utilization of physical servers dramatically

UNIVERSITY OF LEICESTER

Benefits from moving to the cloud

- Economic benefits, incl. change from CAPEX to OPEX (for example, a medium-sized company reduced the monthly cost for server hosting from $2,200 to $250 by moving to the cloud)
- Agility benefits from not having to procure and provision servers
- Efficiency benefits that may lead to competitive advantages
- Security (can be) stronger and better in the cloud

X-as-a-Service taxonomy


Cloud Computing: "Everything as a Service"
Cloud Enablement
Infrastructure and utilities that provide the glue necessary to run the system
Software as a Service (SaaS)
Packaged software application
Framework as a Service (FaaS)
Environment for building a module for an ERP system
Platform as a Service (PaaS)
Environment for building a managed application with an IDE with a rich class library that executes in a runtime container
Infrastructure as a Service (IaaS)
Environment for building a native application

Infrastructure as a Service (IaaS)

- Also Hardware as a Service (HaaS)
- IaaS provider supplies virtual machine images of different OS flavours
- User can bring online and use instances of virtual machines as needed
- Charged for use of VMs, storage and bandwidth
- Example: Amazon Elastic Compute Cloud (EC2), Microsoft Azure

Platform as a Service (PaaS)

- Platform abstracts away interaction with the virtual operating system.
- Simpler to use than IaaS, but less flexible and requires coding in specific languages supported by the PaaS provider
- Examples: Google AppEngine, Microsoft Azure

Software as a Service (SaaS)

- Software and data hosted in the cloud, available on demand
- Examples: Gmail, Google Docs, Salesforce CRM
- Framework as a Service (FaaS) allows customers to augment and enhance the capabilities of the base SaaS system, creating custom, specialised applications

Technological underpinnings (CAYS Ch.2.1)

- Data center with servers on a network
- Virtualized servers
- Access API (provision and decommission servers etc)
- Storage (machine images, applications, persistent data)
- Database (needed for many applications)
- Elasticity (ability to expand and contract applications)

Data Centers

- Servers often mounted in 19-inch rack cabinets (42 slots), in single rows with corridors between them
- Need unwavering power – backup batteries and diesel generators to handle brownouts and outages
- Cooling via air-conditioning or water cooling
- Ample network bandwidth
- Physical and logical security
- Disaster recovery contingencies

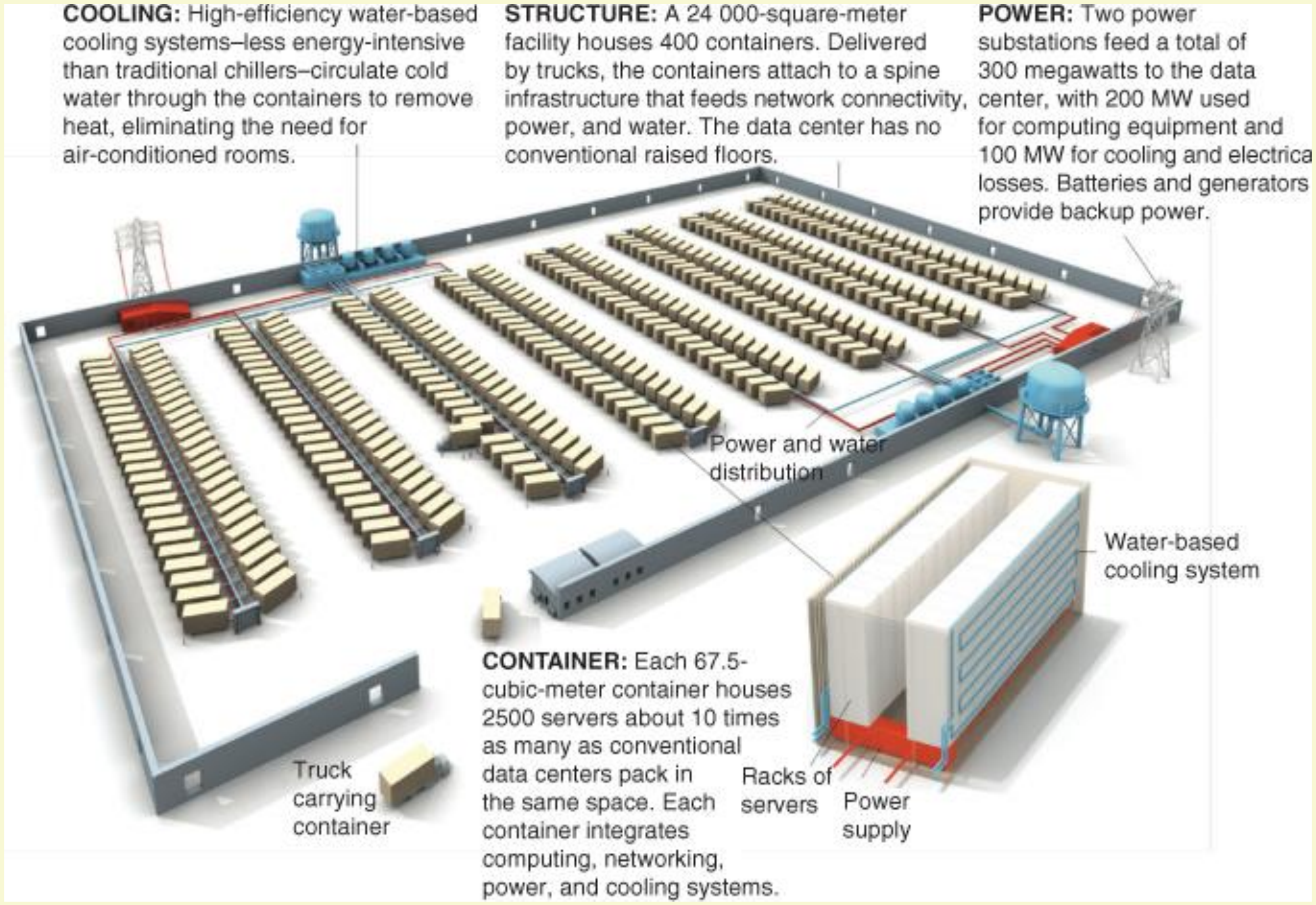
Data Center Scale

- Traditional data centers cost $100-200 million
- Cloud data centers cost $500 million or more
- Geographic proximity to heavy usage areas and access to cheap power.
- Electricity costs of $30 million per year (data centers consume 0.5% of the world's electricity)
- Volume discounts: Amazon spent $90 million for 50,000 servers from Rackable/SGI in 2008, which would cost $215 million without discount

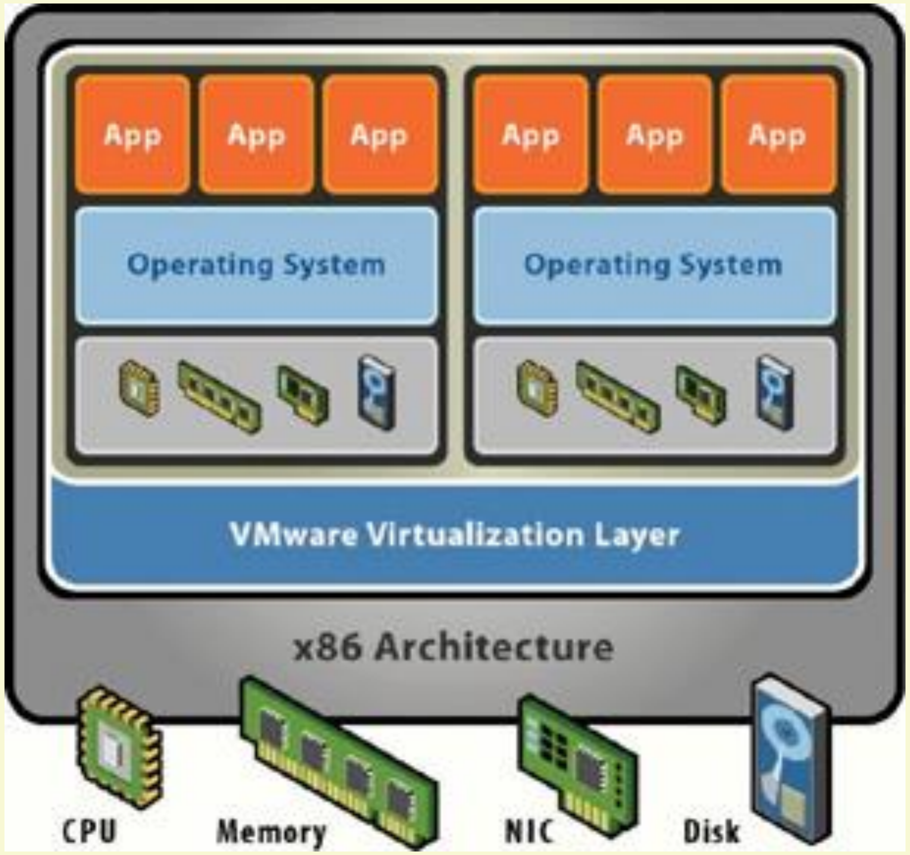
Google Data Centers

- Cheap servers built from components
- High-efficiency power supplies, power management features (voltage/frequency scaling): Power usage efficiency 1.125 (compared to a typical 2.5)
- Dalles, Oregon location near Dalles Dam for cheap power and cooling
- Google have data centers in about 25 locations with 450,000 servers (as of 2007)

Expandable, modular cloud data center


COOLING: High-efficiency water-based cooling systems–less energy-intensive than traditional chillers–circulate cold water through the containers to remove heat, eliminating the need for air-conditioned rooms.
STRUCTURE: A 24 000-square-meter facility houses 400 containers. Delivered by trucks, the containers attach to a spine infrastructure that feeds network connectivity, power, and water. The data center has no conventional raised floors.
POWER: Two power substations feed a total of 300 megawatts to the data center, with 200 MW used for computing equipment and 100 MW for cooling and electrica losses. Batteries and generators provide backup power.
Power and water distribution
Water-based cooling system
CONTAINER: Each 67.5-cubic-meter container houses 2500 servers about 10 times as many as conventional data centers pack in the same space. Each container integrates computing, networking, power, and cooling systems.
Racks of servers
Power supply
Truck carrying container

Virtualization



- Guest operating systems access virtual hardware resources
- Virtualization leads to higher utilization and increases flexibility and agility

Databases in the cloud

- Conventional databases: Relational database management systems (RDBMS), also called SQL databases
- RDMBS do not easily scale to cloud-scale data sets and workloads (e.g. because of referential integrity)
- Since 1998, rapidly growing movement towards non-SQL type databases, e.g. key-value databases (Amazon SimpleDB, Google BigTable)
- NoSQL systems employ a distributed architecture

Computing in the Cloud (DITPM, Ch. 1)

- “Big data”: vast repositories of public and private data
- Gathering, analyzing, monitoring, filtering, searching, organizing web content must tackle big-data problems
- Business intelligence and analysis of user behaviour data: data warehousing, data mining, analytics.
- High-energy physics, astronomy, bioinformatics, …

Big ideas in MapReduce cloud computing

- Scale "out", not "up": use large numbers of low-end servers, not expensive high-end servers
- Assume failures are common: If mean-time between failures (MTBF) is 1,000 days, a 10,000-server cluster has 10 server failures per day on average
- Move processing to the data – exploit data locality
- Process data sequentially, avoid random access
- Hide system-level details from application developer
- Seamless scalability

MapReduce basics (DITPM CH. 2)

- Divide and conquer: partition a large problem into many subproblems, process the subproblems in parallel, then combine the solutions
- MapReduce provides a simple abstraction that hides many system-level details from the programmer
- Storage is typically handled by a distributed file system that sits underneath MapReduce
- MapReduce is a programming model, an execution framework, and its implementation (e.g Hadoop)

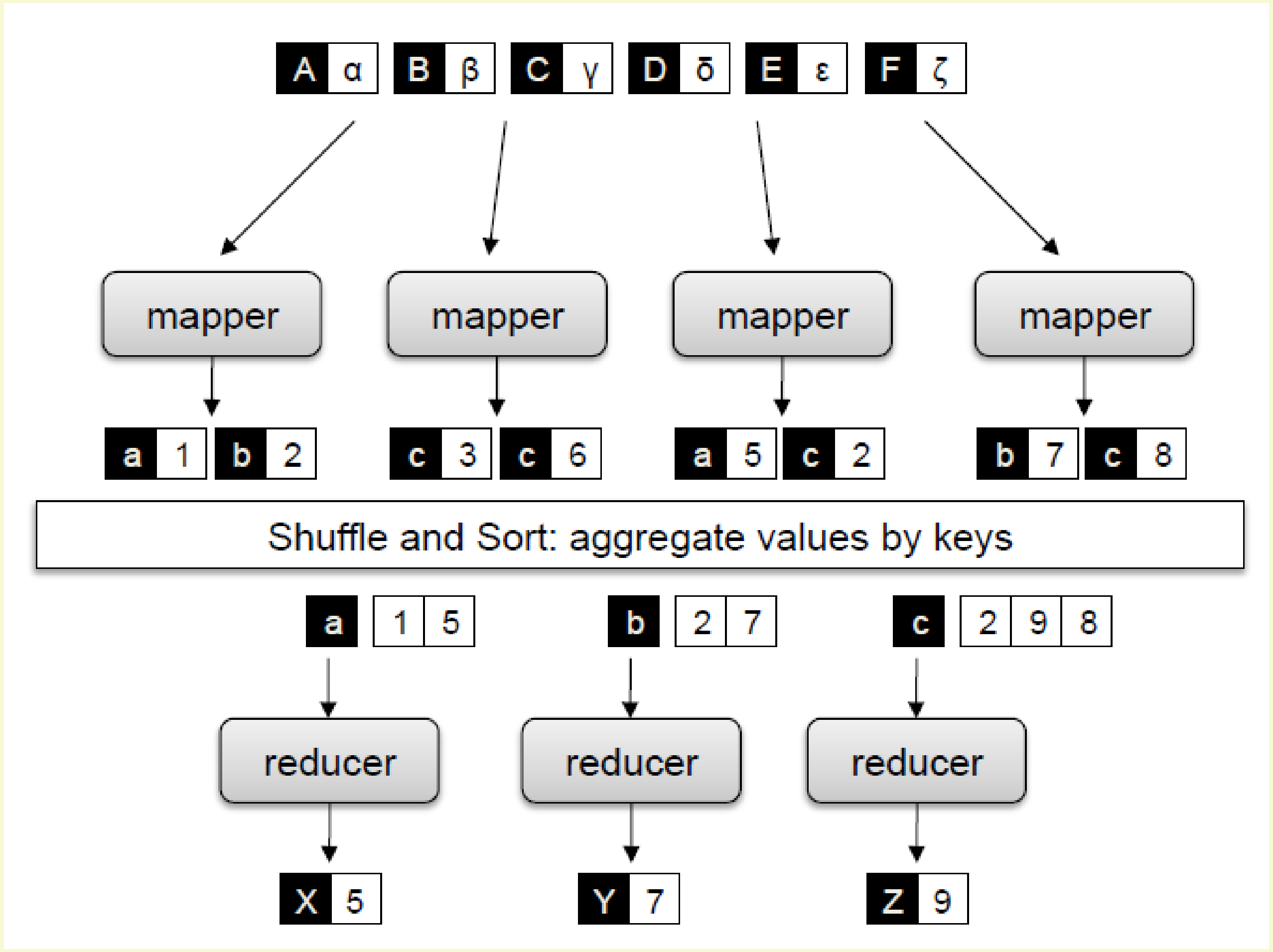
Mappers and reducers

- Input processing in two stages
  1. User-specified computation (in parallel) over all input records in the data set (Map)
  2. Aggregation of results by another user-specified computation (Reduce)
- More complex algorithms can be decomposed into a sequence of MapReduce jobs
- Map and Reduce operate on key-value pairs

Signatures of mapper and reducer

- Convention: [T] denotes a list of elements of type T
- map: (k1,v1) → [(k2,v2)]
  Map key-value pair (k1,v1) to a list of key-value pairs
- reduce: (k2, [v2]) → [(k3,v3)]
  Reduce one key with a list of values to a list of key-value pairs
- All mapper outputs with the same key are passed to the same reducer
- Each reducer’s output is written to the file system

Simplified view of mappers and reducers



DITPM p. 22

Simple WordCount in MapReduce

```
• class Mapper
      method MAP(docid a, doc d)
            for all term t in doc d do
                  EMIT(term t, count 1)

• class Reducer
      method REDUCE(term t, counts [c1,c2,…])
            sum ← 0
            for all count c in counts [c1, c2, …] do
                  sum ← sum + c
            EMIT (term t, count sum)
```

Comments on implementation details

- Mapper and Reducer are classes that implement MAP and REDUCE methods
- Hadoop initializes a Mapper object for each map task, and calls the MAP method on each key-value pair (and similarly for Reducer)
- Programmer provides hint on number of Mappers, and can specify precisely the number of Reducers
- Mappers and Reducers are allowed to have side effects

More on implementation details

- Input key-value pairs, intermediate key-value pairs (mapper output and reducer input), and output key-value pairs can all be different
- MapReduce jobs with no reducers are possible – output is then one file per mapper
- In the most common case, input to a MapReduce job comes from data stored on the distributed file system, but databases such as BigTable can also be used for input and output

UNIVERSITY OF LEICESTER

Execution Framework

- MapReduce program consists of code for mappers and reducers (and combiners and partitioners), plus configuration parameters (e.g. Input/output)
- Developer submits MapReduce job to the submission node (in Hadoop: job tracker)
- Execution framework (“runtime”) takes care of everything else (distributed code execution on clusters ranging from one node to thousands)

Scheduling

- MapReduce jobs are broken down into smaller units called tasks (e.g. map tasks and reduce tasks)
- Scheduler maintains task queue and tracks progress of running tasks
- Speed of a MapReduce job is sensitive to unusually slow tasks (“stragglers”).
- Speculative execution: Execute identical copy of a task on a different machine, use the first result.

Data/code co-location

- Input and output data is stored in a distributed file system
- To achieve data locality, the scheduler starts a task on the node that holds a particular block of data needed by the task.
- If this is not possible, necessary data is streamed over the network (preferably to a node in the same rack)

Synchronization

- In MapReduce, synchronization is accomplished by a "barrier" between the map and reduce phases.
- Barrier groups intermediate key-value pairs by key and routes them to the right reducer: "shuffle and sort"
- For m mappers and r reducers, up to m * r distinct copy operations may be needed.
- Reduce phase cannot start until map phase is complete.

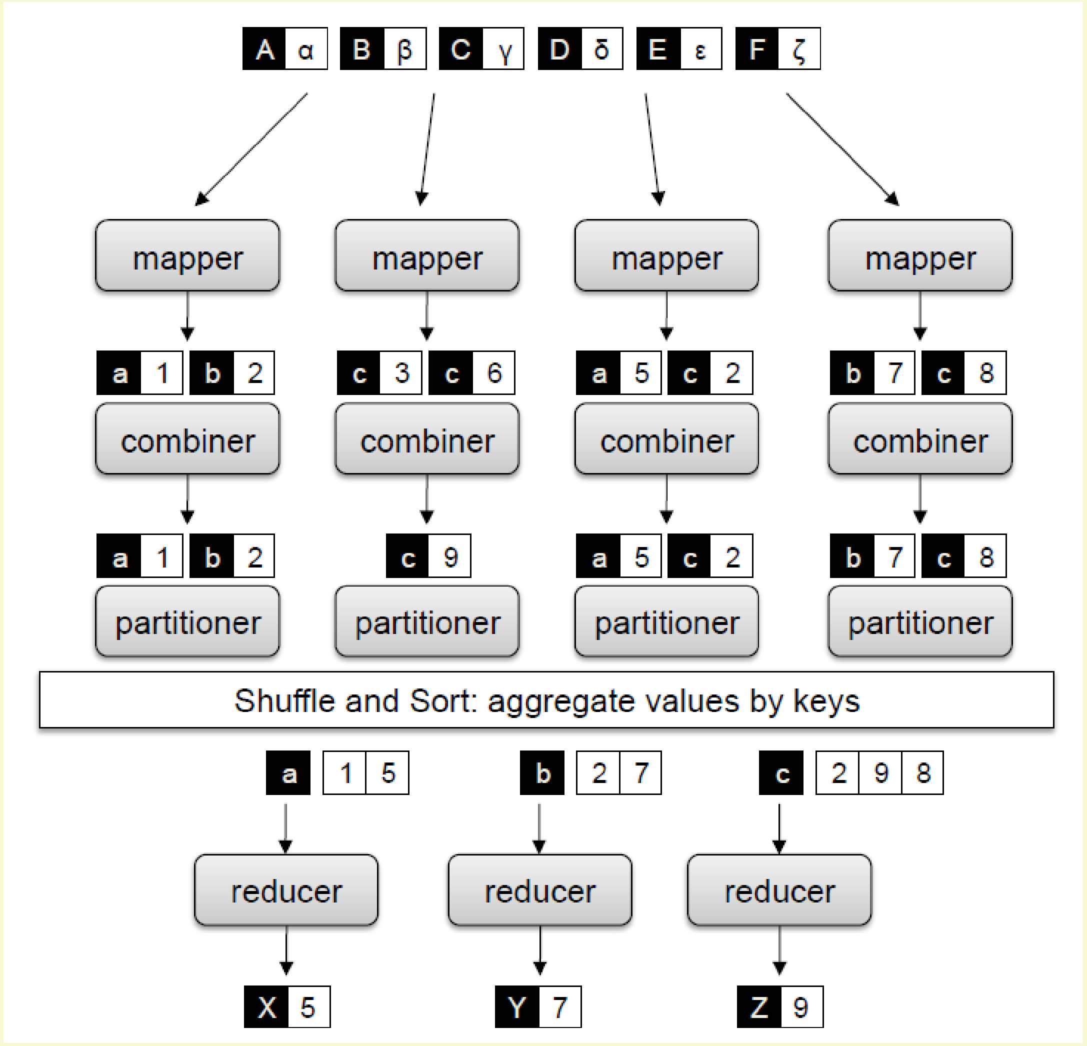
Error and fault handling

- MapReduce runtime must be resilient to hardware and software errors:
  - hardware faults
  - disk failures
  - RAM errors
  - Planned data center outages (maintenance, upgrade)
  - Unplanned data center outages (e.g. power failure)

Partitioners and Combiners

- In addition to mappers and reducers, we have:
  - Partitioners: divide intermediate key space and assign intermediate key-value pairs to reducers
  - Combiners: “mini-reducers” that allow local aggregation on the output of mappers, prior to the “shuffle and sort”
  - Hadoop runtime uses combiners at its discretion: combiner may be invoked zero, one, or multiple times

Mapper, combiner, partitioner, reducer



Note: In Hadoop, partitioners are actually executed before combiners.

DITPM p. 28

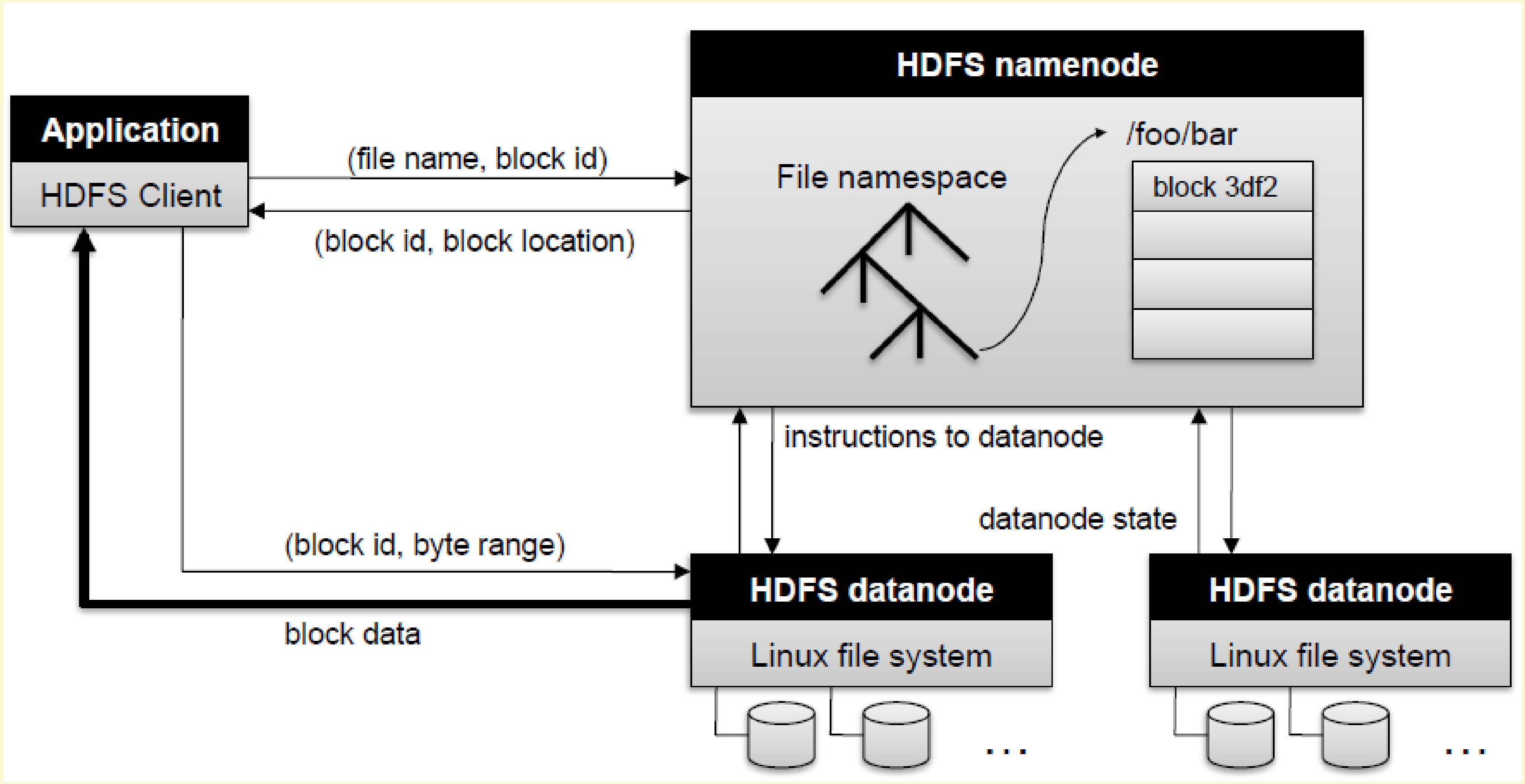
Storage in HPC

- Conventional High-Performance Computing (HPC):
  - Storage is viewed as a distinct and separate component from computation.
  - Use e.g. network-attached storage (NAS) or storage area networks (SAN)
  - Compute node fetches input from storage, processes the data, writes back results
  - Link between compute nodes and storage nodes becomes bottleneck as data size increases

Distributed File System (DFS)

- Abandon separation of computation and storage
- HDFS (Hadoop Distributed File System) is an open-source implementation of the Google File System (GFS) that supports Hadoop
- Divide user data into blocks of 64 MB.
- Master-slave architecture: master maintains file namespace (metadata, directory structure, file to block mapping, block location, access permissions), slaves manage actual data blocks

HDFS Architecture



HDFS Reliability

- By default, HDFS stores three separate copies of each data block on servers in different physical racks
- This makes HDFS resilient to datanode crashes and network failures that bring a rack offline
- Replicated blocks also make it easier to co-locate data and processing in MapReduce
- Namenode directs creation of additional copies of a block as needed.

HDFS design choices

- Support modest number of **large** files:
  - Many small files would exceed the namenode’s memory.
  - Mappers in MapReduce process (part of) a single file: Mapping over many small files would be inefficient (recall the m * r copy operations!)
- Support batch oriented workloads (long streaming reads and large sequential writes). No data caching.

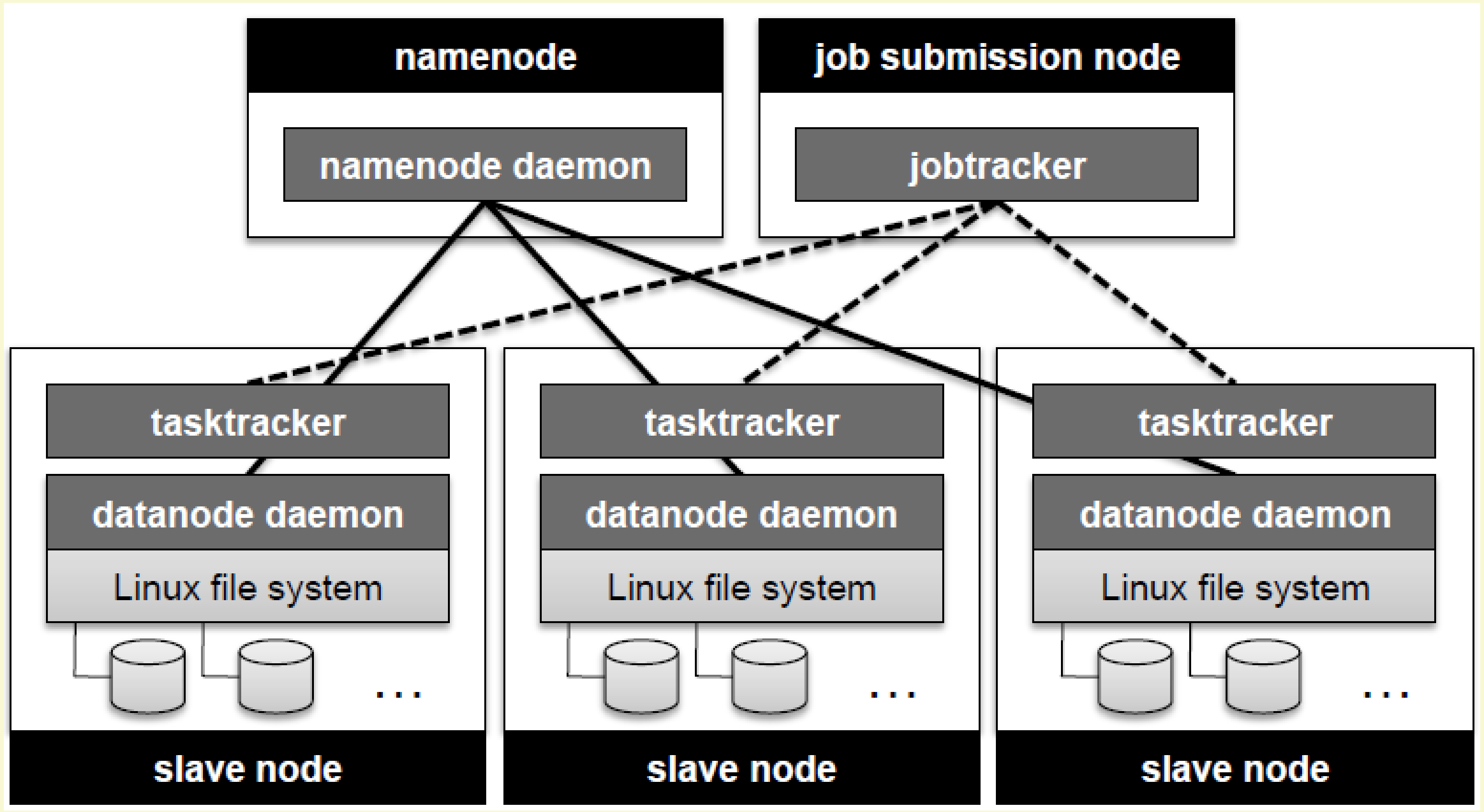
HDFS Design Choices (cont.)

- Applications must be aware of the characteristics of the distributed file system (only a subset of possible file operations is supported)
- Deployed in an environment of cooperative users. Assume data center environment where only authorized users have access. No strict enforcement of file permissions.
- Self-monitoring and self-healing mechanisms to robustly cope with common failure modes.

Single-master design of HDFS

- Single point of failure: If the namenode goes down, the entire file system becomes unavailable.
- Why the single-master design?
  - Simple implementation, file system consistency
  - No data is ever moved through the namenode, so it does not become a bottleneck.
  - MTBF can be on the order of months, and a warm standby node can be used to quickly switch over in case of a failure

Hadoop Cluster Architecture



Nodes in a Hadoop cluster

- namenode: runs namenode daemon
- job submission node: runs job tracker (receives user submitted jobs, coordinates execution of mappers and reducers)
- slave node:
  - task tracker (execute tasks of a MapReduce job)
  - datanode daemon (serves HDFS data)

MapReduce Algorithm Design (DITPM,Ch.3)

- Programming techniques to control execution and flow of data:
  - Use complex data structures as keys and values
  - Execute user-specified initialization and termination code in mapper and reducer
  - Preserve state in mappers and reducers
  - Control sort order of intermediate keys (to determine order of processing by reducer)
  - Control the partitioning of the key space

Local Aggregation

- Reduce the number and size of key-value pairs that need to be shuffled from mappers to reducers
- Can be done using separate combiners or with "in-mapper combining"
- Recall:
  - Combiners are "mini-reducers" that process the output of mappers locally.
  - Hadoop does not guarantee that combiners are actually executed.

Original mapper for Wordcount example

-
  ```
  class Mapper
      method Map(docid a, doc d)
          for all term t in doc d do
              Emit(term t, count 1)
  ```
- Outputs (word,1) for every word in the document.
- Can we change this to output only one key-value pair for each word in the document?

Aggregation on per-document basis

- **class** Mapper

```
method MAP(docid a, doc d)
    H ← new ASSOCIATIVEARRAY  // default val 0
    for all term t in doc d do
        H{t} ← H{t} + 1
    for all term t in H do
        EMIT(term t, count H{t})
```

- Outputs only one key-value pair per term.

Aggregation across multiple documents

-
  ```
  class Mapper
      method Initialize
          H ← new AssociativeArray  // default val 0
      method Map(docid a, doc d)
          for all term t in doc d do
              H{t} ← H{t} + 1
      method Close
          for all term t in H do
              Emit(term t, count H{t})
  ```
- This is the “in-mapper combining” design pattern

Advantages of “in-mapper combining”

- Advantages:
  - Provides control over when local aggregation occurs and how it takes place
  - Typically more efficient than using separate combiners (key-value pairs need to be generated)
- Disadvantages:
  - Potential for order-dependent bugs
  - Need sufficient main memory for intermediate results (can use “flush when full” to address this)

Mean Value Computation

- Task:
  - Given: Input of key-value pairs, keys are strings and values are numbers
  - For each key k, compute the mean of all values v such that (k,v) is an input key-value pair
- Need to take care when writing the Combiner as MEAN(1,2,3,4,5) ≠ MEAN(MEAN (1,2), MEAN (3,4,5))
- Idea: Use (sum,count) pairs as intermediate values

Mean Value Computation: Mapper

-
  ```
  class Mapper
      method MAP(string t, integer r)
          EMIT(string t, pair(r, 1))
  ```

- Mapper outputs one key-value pair for each input key-value pair

Mean Value Computation: Combiner

```
• class Combiner
      method Combine(string t, pairs [(s1,c1),(s2,c2)...])
            sum ← 0
            cnt ← 0
            for all pair(s,c) in pairs[(s1,c1),...] do
                  sum ← sum + s
                  cnt ← cnt + c
            Emit(string t, pair (sum,cnt))
```

Mean Value Computation: Reducer

```
class Reducer
    method REDUCE(string t, pairs [(s1,c1),(s2,c2)...])
        sum ← 0
        cnt ← 0
        for all pair(s,c) in pairs[(s1,c1),...] do
            sum ← sum + s
            cnt ← cnt + c
        $r_{avg}$ ← sum/cnt
        EMIT(string t, double $r_{avg}$)
```

Mapper with in-mapper combining

- **class** Mapper

```
    method Initialize
        S ← new AssociativeArray  // default val 0
        C ← new AssociativeArray  // default val 0
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    method Close
        for all string t in S do
            Emit(string t, pair(S{t},C{t}))
```

Building word co-occurrence matrices

- Application problem: For each pair of unique words in a text corpus, calculate the number of times that the pair occurs together in a sentence/paragraph/ document.
- Many applications in text mining, information retrieval, natural language processing, data mining, etc.
- Idea: Use complex key or value types

“pairs” pattern: pairs as keys

- **class** Mapper

```
method MAP(docid a, doc d)
    for all term w in doc d do
        for all term u in NEIGHBORS(w) do
            EMIT(pair(w,u), count 1)
```

- **class** Reducer

```
method REDUCE(pair p, counts [c1,c2,…])
    s ← 0
    for all count c in counts [c1, c2, …] do
        s ← s + c
    EMIT (pair p, count s)
```

‘stripes’ pattern (stripes as values): Mapper

- **class** Mapper

```
    method MAP(docid a, doc d)
        for all term w in doc d do
            H ← new ASSOCIATIVEARRAY
            for all term u in NEIGHBORS(w) do
                H{u} ← H{u} + 1
            EMIT(term w, stripe H)
```

- Mapper emits pairs of a term and an associative array with co-occurring terms

‘stripes’ pattern (stripes as values): Reducer

```
class Reducer
    method REDUCE(term w, stripes [H1,H2,…])
        Hf ← new ASSOCIATIVEARRAY
        for all stripe H in stripes [H1, H2, …] do
            SUM(Hf,H)
        EMIT (term w, stripe Hf)
```

- Here, SUM(Hf,H) adds H{t} to Hf{t} for all t.
- Each final key-value pair corresponds to a row of the word co-occurrence matrix

Comparison between ‘pairs’ and ‘stripes’

- Pairs algorithm generates more key-value pairs
- Stripes approach is more compact, but the value type is more complex, causing more serialization and deserialization overhead
- Stripes approach assumes that associative array fits into the main memory (scalability bottleneck)
- Both approaches can benefit from the use of combiners (or in-mapper combining)
  - More opportunities for local aggregation with stripes approach, due to smaller key space

Speed Comparison: pairs vs stripes

- Corpus of 2.27 million documents, 5.7 GB in total
- Hadoop cluster with 19 slave nodes, each with 2 single-core processors
- Stripe approach much faster: 11 minutes compared to 62 minutes for pairs approach
- In pairs approach, 2.6 billion intermediate key-value pairs (reduced to 1.1 billion by combiners)
- In stripes aproach, 653 million intermediate key-value pairs (reduced to 28.8 million by combiners)

Computing relative frequencies

- In the co-occurence matrix, m_{ij} is the absolute frequency of word w_j co-occurring with word w_i
- What if we want to know the relative frequency of word w_j co-occurring with word w_i?

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')} = \frac{N(w_i, w_j)}{N(w_i, *)}$$

Here, $N(w_i,w_j)$ is the absolute frequency of word w_j co-occurring with word w_i

Relative frequencies with stripes approach

- Using the “stripes” approach, computing relative frequencies is straightforward:
  - Reducer gets w_i as key and absolute frequencies $N(w_i,w_j)$ for all w_j as values
  - Add up all $N(w_i,w_j)$ to get total frequency $N(w_i,*)$
  - Then calculate relative frequency for each w_j by dividing $N(w_i,w_j)$ by total frequency $N(w_i,*)$
- Requires that associative array fits into memory

Relative frequencies with pairs approach

- Reducer receives (w_i,w_j) as key and count as value
- To compute f(w_j|w_i) we need N(w_i,*).
- To compute N(w_i,*) on the reducer, we need to:
  – Ensure all pairs (w_i,w_j) for any w_j are routed to the same reducer (using a partitioner)
  – Ensure the intermediate keys (w_i,w_j) are sorted first by w_i, then by w_j.
  – Build associative array in memory on reducer.

Avoiding the memory bottleneck

- Is it possible to compute relative frequencies using the “pairs” approach without building the associative array on the reducer?
- Yes, by coordinating several mechanisms in MapReduce:
  - Mapper outputs extra key (word,*) with value 1 for every pair of co-occuring words
  - Use sort order to ensure that (word,*) values arrive before (word,word2) values at reducer

Example processing in reducer

key	values	Reducer action
(dog,*)	[12,14,27]	Calculate N(dog,*)=12+14+27=53
(dog,ant)	[1,2,3]	f(ant|dog)= (1+2+3)/53 = 6/53
…		
(dog,zebra)	[1,2,1,1]	f(zebra|dog)=(1+2+1+1)/53 = 5/53
(door,*)	[24,7]	Calculate N(door,*)=24+7=31
…		

- Partitioners ensures that all pairs (dog,...) are sent to the same reducer
- Reducer maintains state between calls

Summary: “Order inversion” design pattern

- By proper coordination, we can access the result of a computation (e.g. an aggregate statistic) before processing the data.
- Convert problem of sequencing computations into a sorting problem
- By controlling how keys are sorted, we can present data to the reducer in the required order
- This greatly reduces memory requirements of the reducer.

Order inversion for relative frequencies

- Summary of techniques we used:
  - Emit special key-value pair for each co-occurring word
  - Control sort order of intermediate key
  - Define custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer
  - Preserve state across multiple keys in the reducer

Secondary sorting

- MapReduce sorts intermediate results by key
- What if we want to sort by value?
- Example:
  - Input is dump of sensor records (ti,mi,ri) where ti is time stamp, mi is the sensor, and ri is the sensor reading.
  - We want to output for each sensor mi its sequence of sensor readings

Secondary sorting – first attempt

- The mappers could output key-value pairs with key mi and value (ti,ri).
- The sensor readings of the same sensor all arrive at the same reducer, but in arbitrary order.
- To sort the readings by time, the reducer needs to keep them all in memory until the last key-value pair is processed => memory bottleneck.

Secondary sorting – “value-to-key conversion”

- Mapper makes timestamp part of the key and emits pairs with key (mi,ti) and value ri.
- Partitioner ensures that all pairs (mi,...) are sent to same reducer
- Sort order ensures that pairs (mi,ti) are sorted first by mi and then by ti
- Reducer receives all sensor readings for a sensor in sorted order (but must preserve state between calls)

Relational joins

- In relational databases we often want to join two tables on a column value, for example:
  - Table S: (student, module)
  - Table R: (module, convenor)
  - Join (written in SQL):
    Select * from S,R where S.module = R.module
- How can we do this in MapReduce (without a database)?

Scenario with two tables

- Table S contains tuples (ki,si,Si), where ki is the column we wish to join on, si is the unique id in S, and Si is remaining data.
- Table T contains tuples (ki,ti,Ti)
- Example scenario:
  - S stores user profiles (ki is username)
  - T stores logs of online activity
  - Joining S and T allows an analyst to break down online activity by demographics

Reduce-side join

- Map over both data sets
- Emit join column as intermediate key, and the tuple itself as intermediate value
- Since key-value pairs with the same key are routed to the same reducer, the reducer can join the tuples as required
- This works fine for a one-to-one join where each tuple from S is joined with at most one from T

Example for reduce-side one-to-one join

Key	Values	Reducer action
k1	[(s2,S9),(t8,T7)]	Emit (k1,(s2,S9,t8,T7))
k2	[(t3,T12)]	-
k3	[(t4,T19),(s3,S4)]	Emit(k3,(s3,S4,t4,T19))

- Values from S and T can arrive in either order
- Keys with only one value do not produce a join result

Reduce-side one-to-many join

- If one tuple from S is joined with many from T, they will arrive in arbitrary order at the reducer, but we would like the tuple from S to come first.
- We can again use value-to-key conversion to ensure that this happens:
  - Mappers emit ((ki,si),Si) and ((kj,tj),Tj)
  - Sort order is such that (ki,si) comes before (ki,tj)
  - Partitioner must ensure that all pairs (ki,...) arrive at the same reducer

Reduce-side many-to-many join

- With approach from one-to-many join, tuples with key k1 arrive at the reducer in this order:
  - ((k1,s1), S1)
  - ((k1,s2),S2)
  - ….
  - ((k1,s10),S10)
  - ((k1,t1),T1)
  - …
  - ((k1,t17),T17)

- Reducer keeps all (si,Si) for key k1 in memory,
- And then outputs joined tuple pairs for each (ti,Ti) that is processed

Map-side join

- Assume that both S and T are sorted by the same key (the join key) and partitioned into corresponding blocks (e.g. as result of a previous MapReduce step)
- Each mapper reads the i-th block of S (local) and the i-th block of T (remote read), performs the join, and emits the joined tuples
- Reducers only pass the data on

Memory-backed join

- In some cases, one of the two relations, say T, fits completely into main memory.
- Each mapper reads T into main memory (e.g. into an associative array) in the initialization method.
- When the mapper processes a tuple from S, it performs the join with all tuples from T and emits the joined tuples
- For large T, distributed storage in the main memory of several servers can be considered.

Summary of design patterns seen so far

- In-mapper combining: perform local aggregation in the mapper
- “pairs” and “stripes”: keep track of joint events or of all events that co-occur with the same event
- Order inversion: convert sequencing of computations into a sorting problem
- Value-to-key conversion: move part of the value into the key and exploit MapReduce framework for sorting

Useful MapReduce programming techniques

- Construct complex keys or values that bring together the data necessary for a computation
- Execute user-specified initialization and termination code in mapper or reducer
- Preserve state across multiple inputs in mapper and reducer
- Control sort order of intermediate keys
- Control partitioning of intermediate key space

Security and the private cloud (CAYS, Ch. 4)

- Cloud computing enables startups and small and medium-size enterprises to launch services without huge prior investment
- Large enterprises tend to be more reluctant to move to the cloud due to security concerns
- Alternatives to public clouds: private clouds, and virtual private clouds

Security concerns slowing cloud adoption

- Users spanning different corporations and trust levels interact with the same set of compute resources
- Public cloud offerings are exposed to more attacks
- Many nations have laws requiring SaaS providers to keep customer data and copyrighted material within national boundaries
- Some businesses may not like the ability of a country to get access to their data (e.g. via USA PATRIOT Act)

Major cloud data center security

- Physical security
  - House data centers in nondescript facilities (security by obscurity) with extensive setback and military-grade perimeter control berms
  - Physical access is strictly controlled (security staff, video surveillance, intrusion detection systems)
  - Authorized staff use two-factor authentication at least three times to access data-center floors
  - Log and audit all access by employees

Physical security

- Perimeter security (razor wire)
- Biometric authentication (palm reader)
- Access control (man trap)
- Most public cloud providers have SAS 70 Type II certification

Access control measures

- Typical access control for initial sign-up:
  - Billing validation: check billing address of credit card used for payment
  - Identity verification via phone (out of band): generate PIN through browser, enter via phone
  - Create sign-in credentials (strong passwords, ideally multi-factor such as RSA SecurID)
- Use secret authentication key for each API call (digital signature)

Cloud data security

- In many ways, the cloud is more secure than most data centers within an organisation:
  - Centralizing data in the cloud means less leakage than distributing it all over the organization
  - For centralized data it is easier to monitor access and usage
- But: If there is a breach in the cloud, centralized data can mean a more comprehensive and damaging data theft

Cloud versus local data centers

- If an incident occurs, the cloud provides a faster and more comprehensive means of response (state-of-the-art intrusion detection, fast acquisition of forensic data, short downtime)
- Cloud providers are providing more and better built-in verification (e.g. MD5 hash on all stored S3 objects in Amazon's S3)
- Local data centers will become more expensive and less reliable compared to the biggest cloud providers
- Cloud becomes cheaper, more secure, more reliable

Network security in the cloud

- All public clouds provide a firewall (which allows restricting traffic by protocol, service port, or source IP address (block))
- Many prominent public cloud providers have strong capabilities of defending against DDoS (distributed denial of service) attacks
- User applications run on virtual servers, and the host OS can prevent address spoofing and packet sniffing

Data storage security

- Cloud provider's disk virtualization layer can guarantee that data is never exposed to another user
- Still, it is good practice to store data in an encrypted file system on top of the virtualized disk device
- Access control lists can limit access to storage containers
- Data and control messages can be transferred securely via SSL-encrypted access to storage API

Private cloud

- A computing architecture that provides hosted services to a specific group of people behind a firewall.
- A private cloud uses virtualization, automation, and distributed computing to provide on-demand elastic computing capacity to internal users.

Principles of cloud computing revisited

- Principles that remain the same as for public cloud:
  - Virtualization: high utilization of assets
  - Elasticity: dynamic scale without CAPEX
  - Automation: build, deploy, configure, provision, move without manual intervention
- Principles that do not necessarily hold:
  - Pooled resources (available to any subscribing users)
  - Metered billing (pay for what you use)

Primary considerations for private cloud

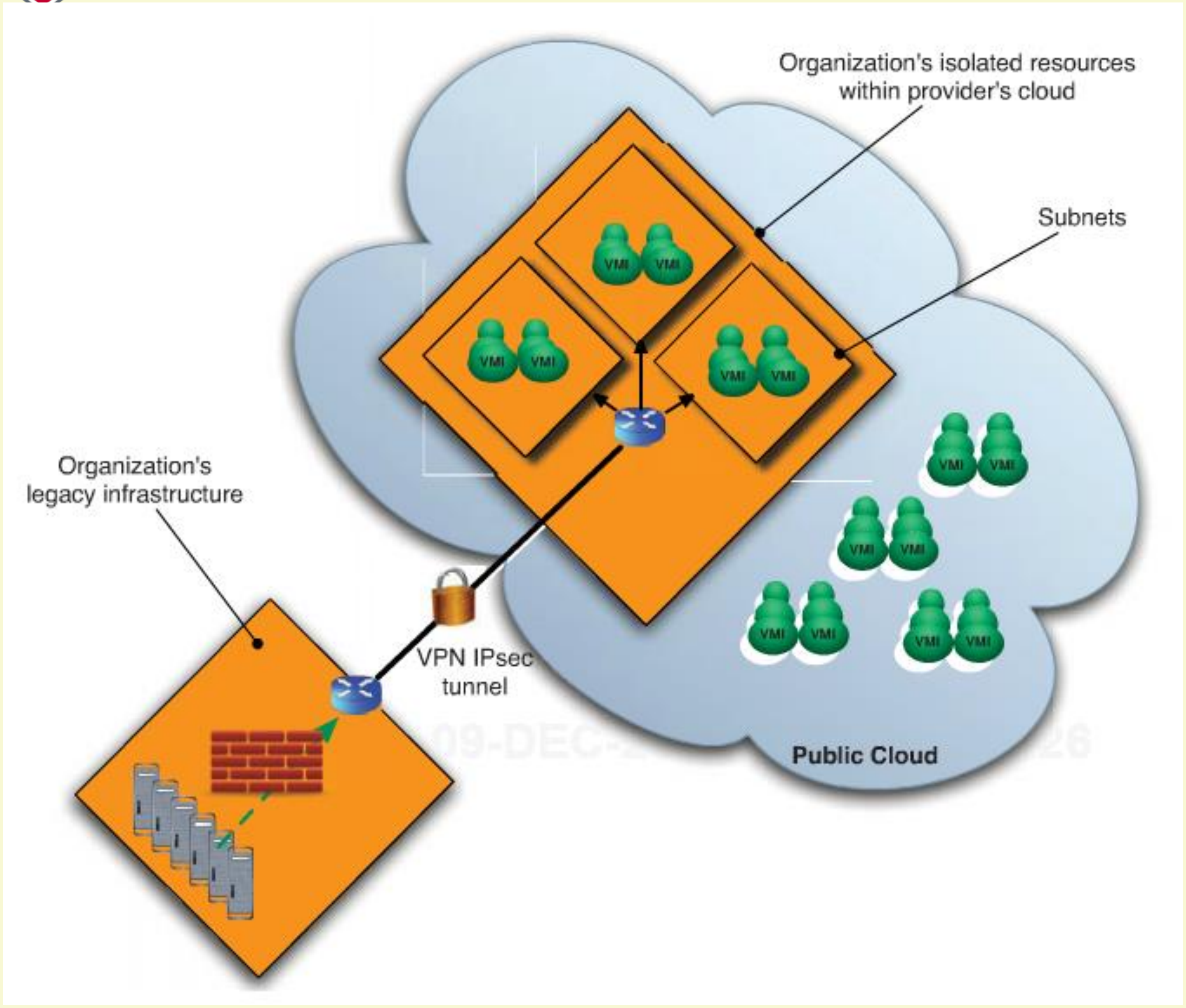
- **Security:** applications that require direct control and custody over data for security or privacy reasons
- **Availability:** applications that require access to a defined set of computing resources that cannot be guaranteed in a shared resource pool environment
- **User community:** Organization with a large number of users who need access to utility computing resources
- **Economies of scale:** Existing data center and hardware resources that can be used, and ability to purchase equipment at favourable pricing levels

Concerns about deploying a private cloud

- Private clouds are small: Few corporate data centers see anything close to the type of volume seen by cloud-computing providers
- Legacy applications don’t “cloudify” easily
- On-premises doesn’t necessarily mean more secure
- Do what you do best – private clouds will always be many steps behind the public clouds (whose providers constantly optimize how they operate)

Virtual private cloud (VPC)

- A VPC is a secure and seamless bridge between an organization’s existing IT infrastructure and a provider’s public cloud
- Idea:
  - Organisation that has its own IT infrastructure adds additional web-facing servers to an application when the traffic exceeds the on-premise capacity
  - Back-end, database servers, authentication servers etc. remain in organisation’s own data center


Organization's isolated resources within provider's cloud
Subnets
VMI
Organization's legacy infrastructure
VPN IPsec tunnel
Public Cloud

CAYS p. 93

Virtual private cloud security

- An organisation’s existing infrastructure is connected to a set of isolated cloud-computing resources via a virtual private network connection
- The organisation’s existing management capabilities and security services extend to the virtual private cloud and protect information there in the same way as in their own data center (including firewalls, intrusion-detection systems, etc)

Usage scenarios of virtual private clouds

- **Expanding corporate applications into the cloud** to reduce total cost of ownership (email systems, financial systems, CRM applications, etc.)
- **Elastically scaling websites in the cloud**: Temporarily add web servers when traffic load exceeds on-premises capacity (“cloudbursting”)
- **Disaster recovery**: Periodically back up mission-critical data to the VPC, and in the event of a disaster, quickly launch replacement compute capacity in the cloud to ensure business continuity

Cloud Security - Summary

- Security remains the biggest fear factor for larger organisations considering a move to the cloud, but there are many features and mechanisms that can (at least partially) address these concerns in the public cloud
- For organisations with enough scale, buying power and expertise, private clouds offer advantages of increased control, predictability and security
- Virtual private clouds represent a hybrid model that can be considered as an alternative