Architectures for Big Data

Transactional (ACID) properties in distributed systems

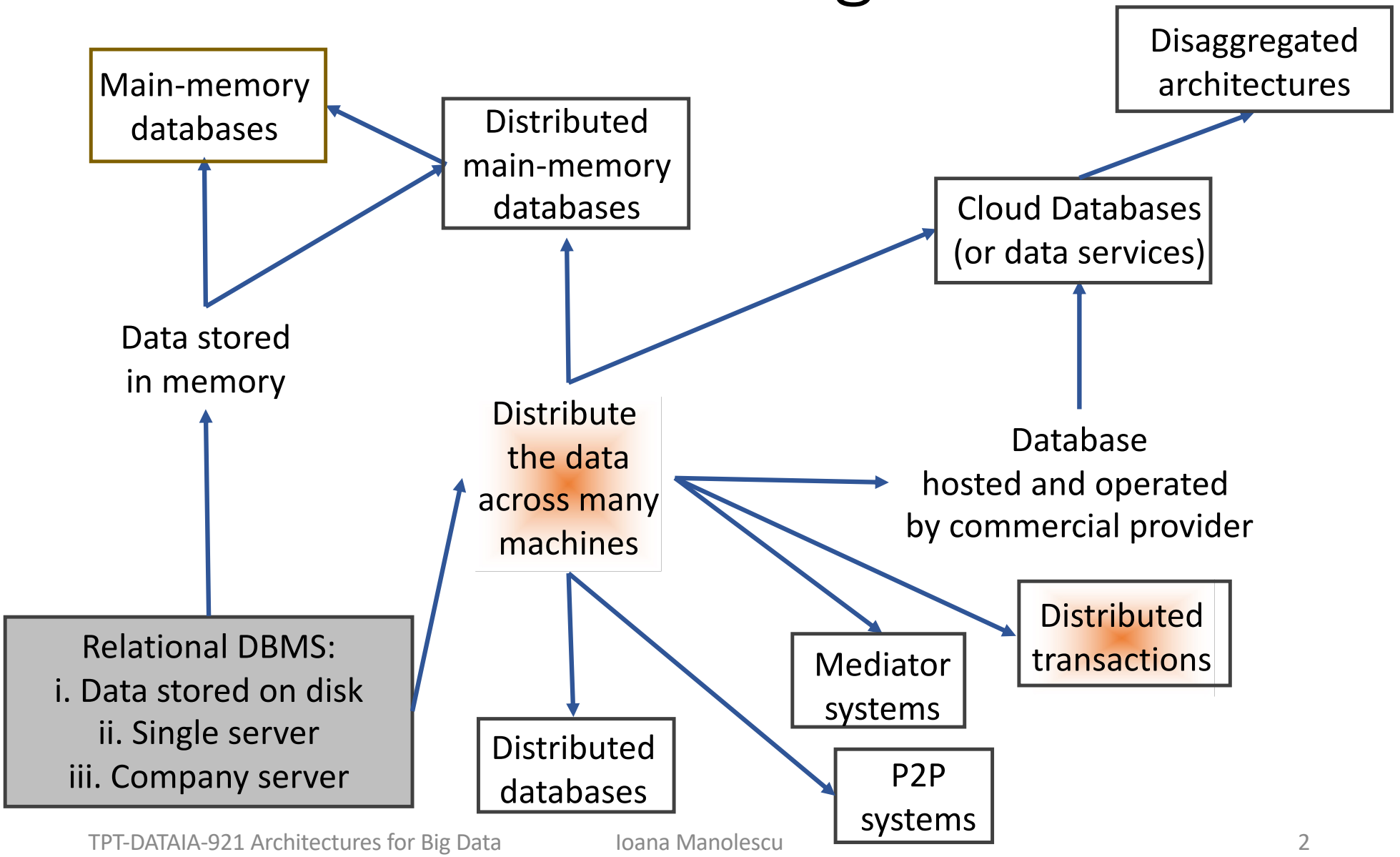
Ioana Manolescu

Inria Saclay & Ecole Polytechnique

ioana.manolescu@inria.fr

http://pages.saclay.inria.fr/ioana.manolescu

From databases to Big Data



Recall: Fundamental database properties: ACID

- **Atomicity**: either all operations involved in a transactions are done, or none of them is
  - E.g. bank payment
- **Consistency**: application-dependent constraint

  E.g. every client has a single birthdate
- **Isolation**: concurrent operations on the database are executed as if each ran alone on the system
  - E.g. if a debit and a credit operation run concurrently, the final result is still correct
- **Durability**: data will not be lost nor corrupted even in the presence of system failure during operation execution

Jim Gray, ACM Turing Award 1998 for « fundamental contributions to databases and transaction management »

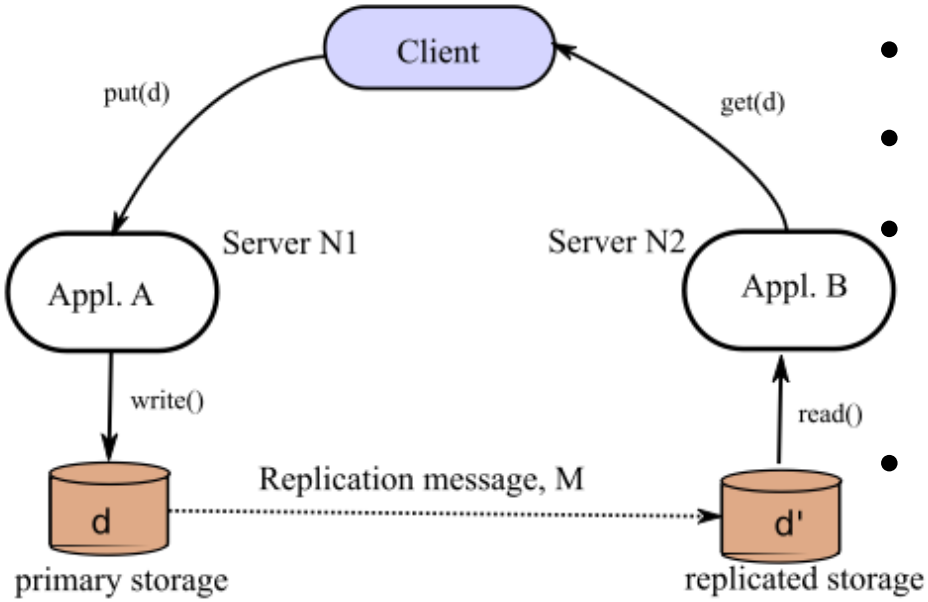
Limits of ACIDity in large distributed systems: the **CAP theorem**

- Eric Brewer, « Symposium on Principles of Distributed Computing », 2000 (conjecture)
- Proved in 2002

- No distributed system can simultaneously provide

1. **Consistency** (all nodes see the same data at the same time)
2. **Availability** (node failures do not prevent survivors from continuing to operate)
3. **Partition tolerance** (the system continues to operate despite arbitrary message loss)

CAP theorem by example



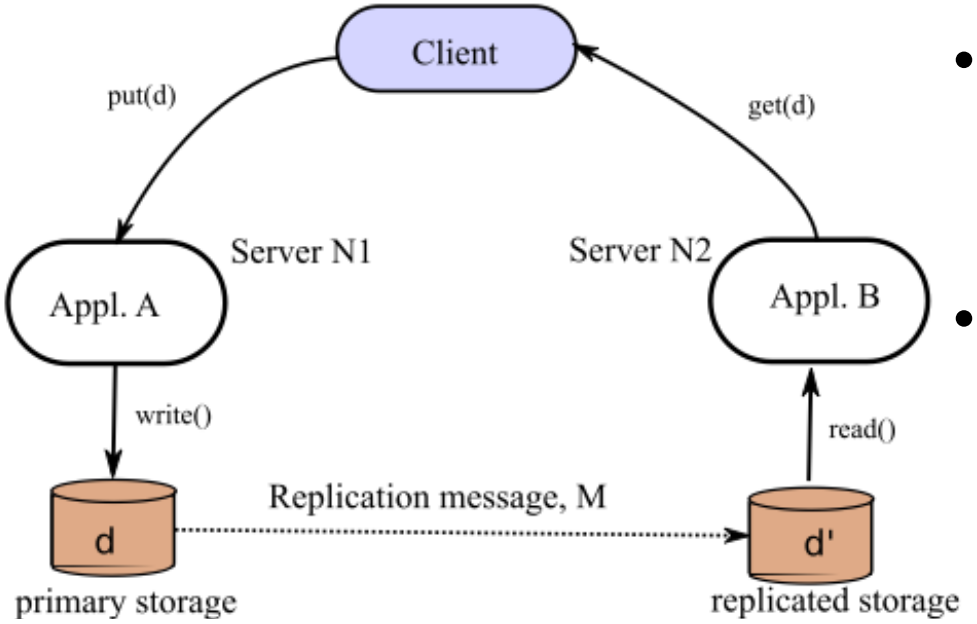
- Primary and replica store
- Applications A and B on servers
- Client writes a new d value through A, which propagates d to the replica (replacing the old d')
- Subsequently, client reads from B

What if a failure occurs in the system?

Communication missed between primary and replica

1. If we want **Partition tolerance** (let the system function) → the Client reads old data (**no Consistency**)
2. If we want **Consistency**, e.g. make the write+replica msg an atomic transaction (to avoid missed communications) → **no Availability** (we may wait for the msg forever if failure)

CAP theorem: what can we do?

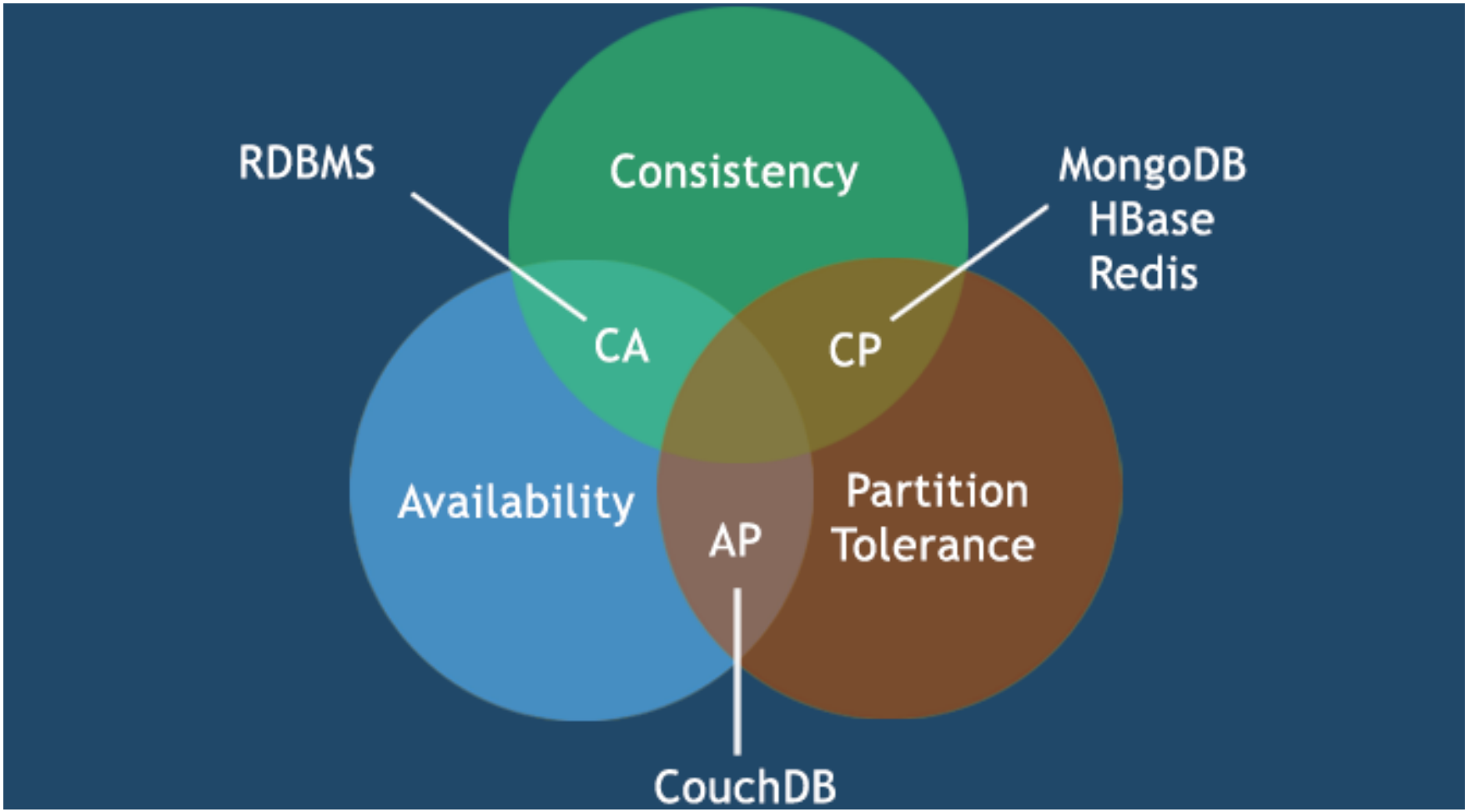


- **Partition tolerance**: we must have it (cannot block if one machine fails)
- Then one must *trade some consistency for availability*

Eventual consistency model:

- The replication message is asynchronous (non-blocking)
- N1 keeps sending the message until acknowledge by N2 (*eventually* the replica and primary store are consistent)
- In the mean time, the client works on inconsistent data (« I had already removed this from the basket once! »)

NoSQL systems vs. CAP theorem



Modern systems (e.g. NoSQL) arose exactly because partition tolerance is a must in large-scale distributed systems

More on CAP theorem

- ACID properties focus on consistency: business databases (sales, administration...)
- **BASE**: Basically Available, Soft state, Eventually consistent
  - Modern NoSQL systems are typically BASE
- "Partition" in fact corresponds to a **timeout** (when do we decide that we waited enough)
  - Different nodes in the system may have different opinion on whether there is a partition
  - Each node can go in "partition mode"
- Different systems provide different ACID/BASE guarantees. Important to understand them!

Choices in the ACID-BASE spectrum

- **Yahoo! PNUTS**: give up strong consistency to avoid high latency. The master copy is always "nearby" the user
- **Facebook**: the master copy is always remote, however updates go directly to the master copy and *this is also where users' reads go for 20 seconds*. After that, the user traffic reverts to the closer copy.

Choices in the ACID-BASE spectrum

- **Amazon DynamoDB, Cassandra, Ryak**:
  - In normal functioning, there is a master node for each data item
  - All writes to a data item are sent to its master node, then synchronously replicated to W other nodes
  - All reads requests are synchronously sent to R nodes.
    - R+W <= N, thus there may be inconsistent reads.
    - Cassandra allows « weak reads » (W=1) and also « quorum reads » (better consistency, 4x slower)
  - In some situations, e.g., failure of a master node, or load balancing may, updates for 1 data item may end up on different master nodes
    - Potential inconsistencies

What do to in case of inconsistency?

1. Merge copies: find a commonly agreed upon version
   - Concurrent Versioning Systems (CVS, SVN, GIT) do this pretty well but not always
   - Some conflicts remain to be solved by the user
2. Limit the operation set to have fewer conflicts and/or easier to solve
   - E.g., Google Docs solves conflicts by allowing *only style change and add/delete text*
   - E.g., using only *commutative* operations: there is always a way to rearrange a set of operations in a preferred consistent global order
     1. Addition is commutative
     2. Addition with a bounds check is not

From CAP to PACELC

CAP states that a network partition causes the system to have to decided between less availability and less consistency

- No network partition → no problem (we can have ACID!)

However, in a Big Data system, replication (usually across a WAN) is required to guarantee against eventual component failure.

A more global way to think about performance trade-offs is

PACELC (« passelk »):

- If there is a network **P**artition, how does the system trade between **A**vailability and **C**onsistency?
- **E**lse, how does the system trade between **L**atency and **C**onsistency?