Chapter 7: Introduction

Distributed Applications - Motivation

Why do we want to make our applications distributed?

- Applications are inherently distributed.
- A distributed system is more reliable.
- A distributed system performs better.
- A distributed system scales better.

Only paradigm to support use cases like Google, Facebook and Amazon!

Practical Concerns of Distributed Systems

"Easier to go from theory to practice than vice versa" [Fabian Kuhn]
but

- significant gap between fundamental results and practical applications
- high complexity: nondeterminism, failures
- large design space
- many tradeoffs: practical concerns vs. theoretical worst cases

Everybody wants to write applications as if they were *centralized*,
yet reap the benefits of distribution (very similar story to parallelism)
but

- ineffecient design (think RPC)
- leaky abstractions

Practical Concerns of Distributed Systems

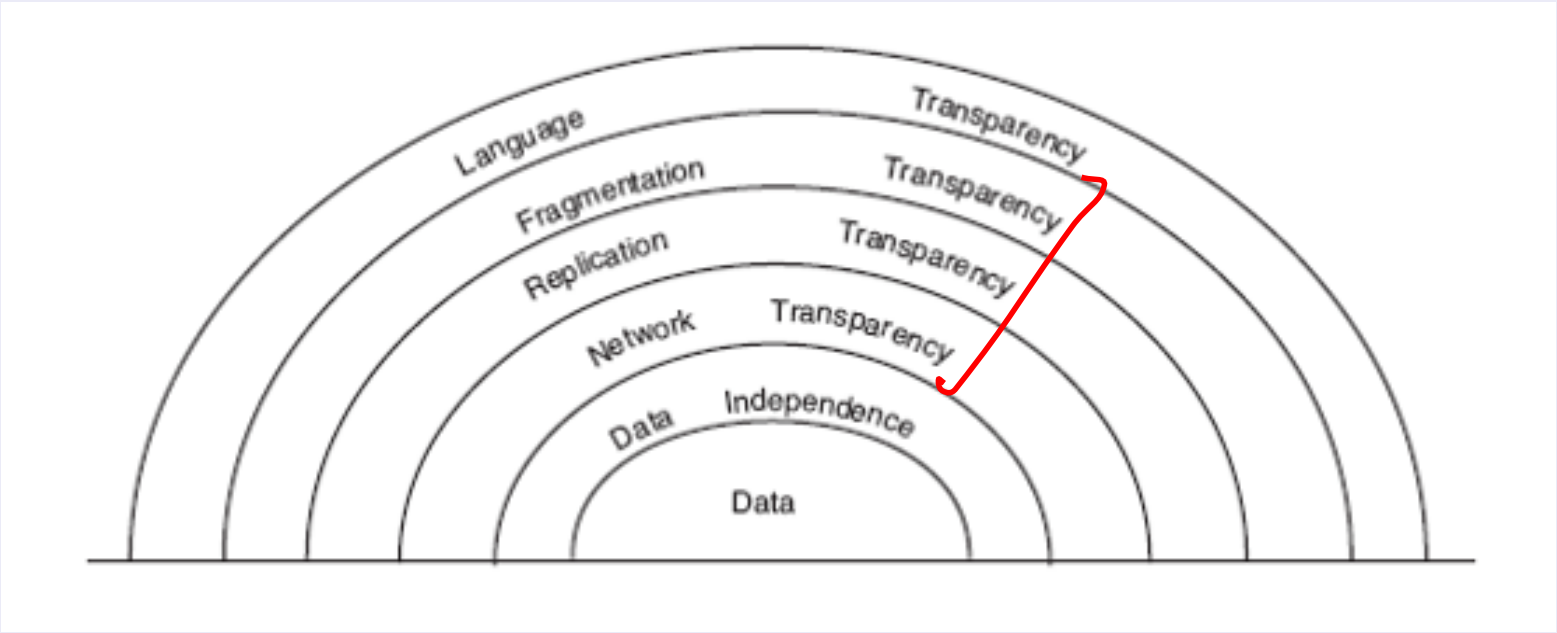
"Easier to go from theory to practice than vice versa" [Fabian Kuhn]
but

- significant gap between fundamental results and practical applications
- high complexity: nondeterminism, failures
- large design space
- many tradeoffs: practical concerns vs. theoretical worst cases

Everybody wants to write applications as if they were *centralized*,
yet reap the benefits of distribution (very similar story to parallelism)
but

- ineffecient design (think RPC)
- leaky abstractions

Example: Transparency in Distributed Databases



Typical CS approach: hide complexity by adding abstractions

Which of these types of transparency are likely to hold? None fully

Fallacies of distributed computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

established by Peter Deutsch, Bill Joy, Tom Lyon and James Gosling.

Considerations for distributed applications

On a conceptual level, we face a tradeoff:

- transparency and abstraction
- understandable models and algorithms

vs

- fundamental limitations
- efficiency

In practice, there are a number of useful rules

- Don't write distributed applications from scratch
- Don't invent your own algorithms (unless you know exactly what you are doing)
- Don't trust frameworks and tools blindly
  - Cannot beat fundamental science (FLP, CAP)
  - Make very specific tradeoffs and promises
  - Contain bugs in design and implementation

Considerations for distributed applications

On a conceptual level, we face a tradeoff:

- transparency and abstraction
- understandable models and algorithms

vs

- fundamental limitations
- efficiency

In practice, there are a number of useful rules

- Don't write distributed applications from scratch
- Don't invent your own algorithms (unless you know exactly what you are doing)
- Don't trust frameworks and tools blindly
  - Cannot beat fundamental science (FLP, CAP)
  - Make very specific tradeoffs and promises
  - Contain bugs in design and implementation

Fundamental problem: Mutable State

Programming languages:

```
i := i+1
```

SQL:

```
UPDATE Account
SET Balance = Balance*1.03
```

Changes have to

- be performed in a controlled manner
- become visible for all users in a consistent manner

Would an append-only behavior (with lifetime intervals) solve the problems?

Fundamental problem: Mutable State

Programming languages:

```
i := i+1
```

SQL:

```
UPDATE Account
SET Balance = Balance*1.03
```

Changes have to

- be performed in a controlled manner
- become visible for all users in a consistent manner

Would an append-only behavior (with lifetime intervals) solve the problems?

Fundamental problem: Mutable State

Programming languages:

```
i := i+1
```

SQL:

```
UPDATE Account
SET Balance = Balance*1.03
```

Changes have to

- be performed in a controlled manner
- become visible for all users in a consistent manner

Would an append-only behavior (with lifetime intervals) solve the problems?

Challenges of Mutable State

- Parallel access
  - same user, different locations: threads, multiprocessors, clusters, datacenters, mobile clients
  - different users: banks, flight bookings, supply chains, ...
- Complex modifications
  - associated updates: withdrawal from one accounts, depositing on another.
  - applications-level consistency: account may not go into overdraft, unavailable items may not be sold
- Error handling
  - crash/unavailablity, in particular during updates

Abstraction: Transactions

- Just one possible, but very useful and common abstraction
- Tackles the issues of keeping data consistent
- Provides well-defined concurrency and failure models
- Implementation details and system issues are hidden from the developer
- Generic approach, suitable for a wide range of workloads
- Major success factor of relational databases
- Catching up in popularity other areas,e.g.,
  - transactional file systems,
  - hardware-transactional RAM

Transactions: Concepts

- Application (or user) explicitly groups a sequence of operations that belongs together
- A small number of primitives
  - `BEGIN TRANSACTION / BOT`: explicit start
  - `COMMIT TRANSACTION`: finish and keep results
  - `ABORT TRANSACTION`: finish and discard results
- More complex models allow nested transaction, safepoints, ...
- Generic access to data items: `READ(X)` and `WRITE(X)`
- For the transaction's requests and effects on the underlying data certain properties are guaranteed: ACID properties.

Transaction Examples

(Example 1a) Debit/credit

Consider a debit/credit-program of a bank which transfers a certain amount of money between two accounts.
Executing the program will give us the following transaction T:

```
BEGIN
% Withdraw
READ current value VA of account A from disk into T's local main memory;
decrement VA by amount X;
WRITE new value VA' = VA - X of account A from T's local main memory onto disk;
% Deposit
READ current value VB of account B from disk into T's local main memory;
increment VB by amount X;
WRITE new value VB' = VB + X of account B from T's local main memory onto disk;
COMMIT;
```

Discussion of Exampe 1a

- Assume when executing `T` the system runs into a failure, e.g. after writing `A` and before reading `B`. A customer of the bank has lost `X` money!
- Assume debit/credit-transaction `T1` is running concurrently to a transaction `T2`, which computes the balance of the accounts `A` and `B`. Then the `READ` and `WRITE` accesses of both transactions may be interleaved. Assume that `T2` is executed after `T1` writing `A` and before `T1` writing `B`, then the balance computed will be incorrect.

Discussion of Exampe 1a

- Assume when executing `T` the system runs into a failure, e.g. after writing `A` and before reading `B`. A customer of the bank has lost `X` money!
- Assume debit/credit-transaction `T1` is running concurrently to a transaction `T2`, which computes the balance of the accounts `A` and `B`. Then the `READ` and `WRITE` accesses of both transactions may be interleaved. Assume that `T2` is executed after `T1` writing `A` and before `T1` writing `B`, then the balance computed will be incorrect.

(Example.1b) Distributed debit/credit

Assume that

- Two different branches of the bank are involved
- Each branch maintains it own servers and stores its data there

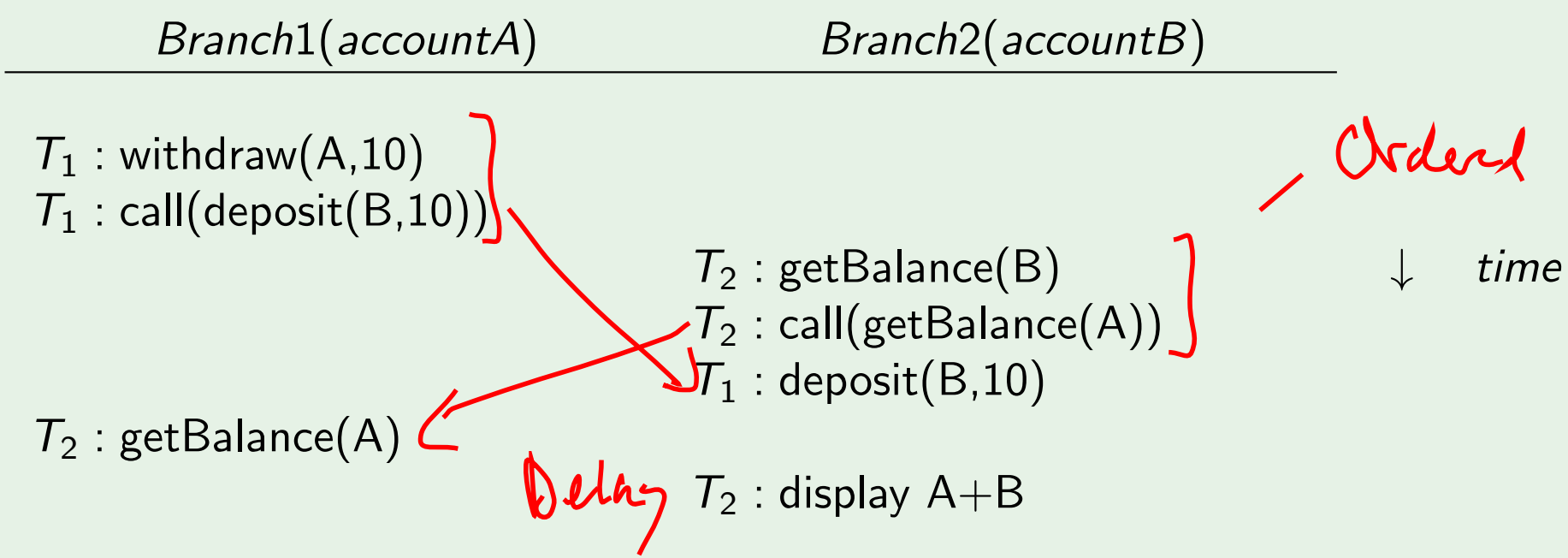
Assume further,

- At Branch1 a debit/credit-transaction is started
- At Branch2 a balancing transaction is started,
- Both transactions involve the same accounts.

Transactions shall have access to accounts on remote server via remote procedure calls (RPCs) (which are synchronous) → blocking
The following procedures may exist

- *withdraw*(*account*, *amount*),
- *deposit*(*account*, *amount*)
- *getBalance*(*account*).

A possible interleaving when both transactions are running in parallel.



An incorrect balance will be displayed!

A possible interleaving when both transactions are running in parallel.

*Branch*1(*accountA*)	*Branch*2(*accountB*)	
T_1 : withdraw(A,10)		
T_1 : call(deposit(B,10))		
	T_2 : getBalance(B)	↓ *time*
	T_2 : call(getBalance(A))	
	T_1 : deposit(B,10)	
T_2 : getBalance(A)		
	T_2 : display A+B	

An incorrect balance will be displayed!

(Example 1c) Distributed debit/credit

Assume that

- Two different branches of the bank are involved
- Each branch maintains it own servers and stores its data there

Assume further,

- At Branch1 a debit/credit-transaction is started
- At Branch2 a balancing transaction is started,
- Both transactions involve the same accounts.

Communication is explicitly implemented by exchanging messages between the involved servers.

Finally assume, that each transaction implements exclusive access to both accounts during execution.

(Example 1c) Distributed debit/credit

Assume that

- Two different branches of the bank are involved
- Each branch maintains it own servers and stores its data there

Assume further,

- At Branch1 a debit/credit-transaction is started
- At Branch2 a balancing transaction is started,
- Both transactions involve the same accounts.

Communication is explicitly implemented by exchanging messages between the involved servers. asynchronous / non-blocking

Finally assume, that each transaction implements exclusive access to both accounts during execution.

(Example 1c) Distributed debit/credit

Assume that

- Two different branches of the bank are involved
- Each branch maintains it own servers and stores its data there

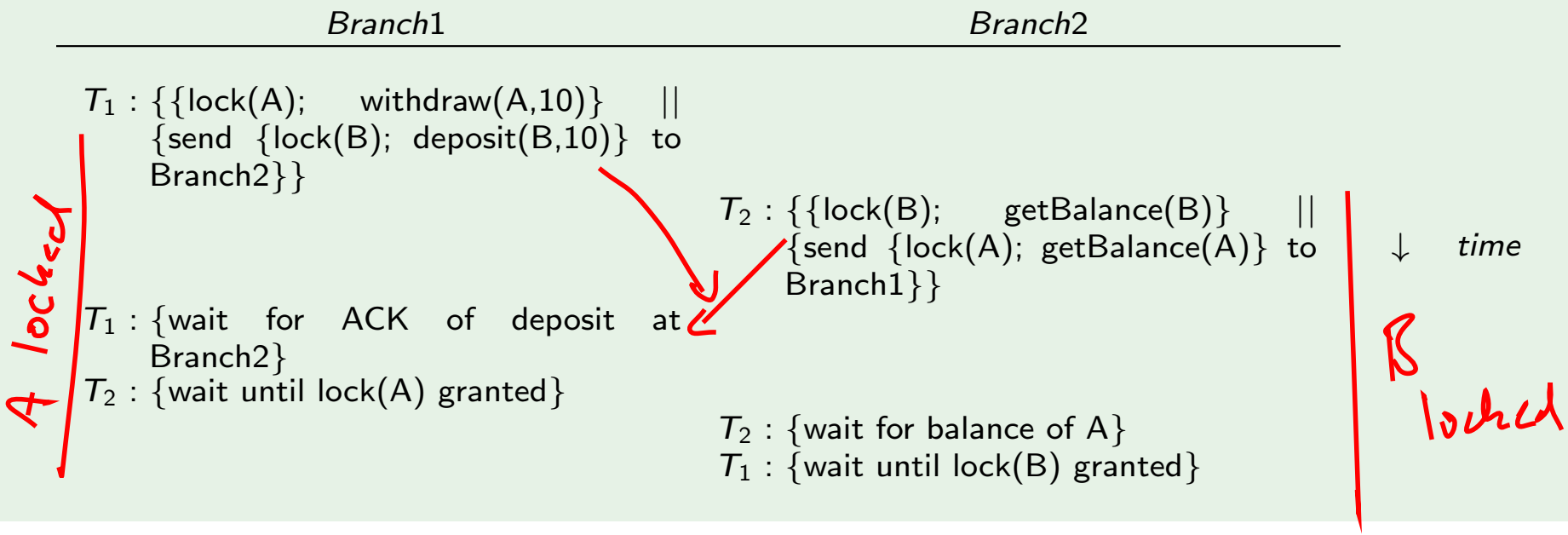
Assume further,

- At Branch1 a debit/credit-transaction is started
- At Branch2 a balancing transaction is started,
- Both transactions involve the same accounts.

Communication is explicitly implemented by exchanging messages between the involved servers.

Finally assume, that each transaction implements exclusive access to both accounts during execution.

A possible interleaving



A deadlock has occured which is difficult to detect!

A possible interleaving

*Branch*1	*Branch*2	
T_1 : {{lock(A); withdraw(A,10)} || {send {lock(B); deposit(B,10)} to Branch2}}		
	T_2 : {{lock(B); getBalance(B)} || {send {lock(A); getBalance(A)} to Branch1}}	↓ *time*
T_1 : {wait for ACK of deposit at Branch2}		
T_2 : {wait until lock(A) granted}		
	T_2 : {wait for balance of A}	
	T_1 : {wait until lock(B) granted}	

A deadlock has occured which is difficult to detect!

(Example 2) Electronic commerce

Consider the following purchasing activity, which covers several different servers and service located at different sites:

- A client connects to a bookstore's server and starts browsing and querying the catalog.
- The client gradually fills a shopping card with items intended to purchase.
- When the client is about to check out she makes final decisions what to purchase.
- The client provides all necessary information for placing a legally binding order, e.g. shipping address and credit card.
- The merchants's server forwards the payment information to the customer's bank or to a clearinghouse. When the payment is accepted, the inventory is updated, shipping is initiated and the client is notified about successful completion of her order.

Discussion of Example 2

- The final step of the purchasing is the most critical one. Several servers maintained by different institutions are involved.
- Most importantly it has to be guaranteed that
  - *either* all the tasks of the final step are processed correctly,
  - *or* the whole purchasing activity is undone.

(Example 3) Mobile computing

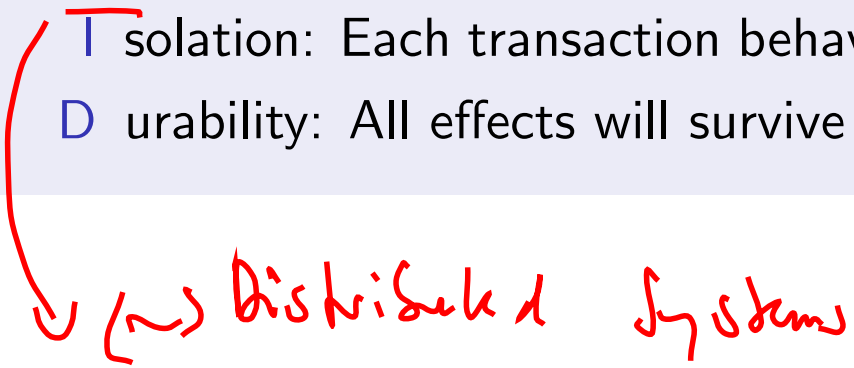
Assume that the described purchasing activity is performed via a smartphone. Then the situation gets even more complicated.

- The smartphone might be temporarily disconnected from the mobile network. Thus it is not guaranteed that the state of the catalog as seen by the client reflects the state of the catalog at the server.
- If the client enters a dead spot during processing of the final step of the purchasing activity, confusion may arise, e.g. the purchasing is started again resulting in double orders.

Transaction Guarantees

ACID properties

- **A** tomicity: A transaction is executed completely or not at all.
- **C** onsistency: Consistency constraints defined on the data are preserved.
- **I** solation: Each transaction behaves as if it were operating alone on the data.
- **D** urability: All effects will survive all software and hardware failures.



Discussion of ACID guarantees

- classical, "all-inclusive" guarantee in (relational) database systems
- solves the problems demonstrated in Examples (and more)
- well-established theory and clear semantics
- mature and well-engineered implementations
  - *Recovery* for *A*, *D*
  - *Concurrency Control* for *I*

What do you think of this abstraction, in particular after part 1 of the lecture?

We are looking at a fork in the road:

- provide ACID, but limit scalability and availability
- favour scalability and availability, but sacrifice on isolation/consistency: NoSQL, BASE

The course will cover both!

Discussion of ACID guarantees

- classical, "all-inclusive" guarantee in (relational) database systems
- solves the problems demonstrated in Examples (and more)
- well-established theory and clear semantics
- mature and well-engineered implementations
  - *Recovery* for *A*, *D*
  - *Concurrency Control* for *I*

What do you think of this abstraction, in particular after part 1 of the lecture?

We are looking at a fork in the road:

- provide ACID, but limit scalability and availability
- favour scalability and availability, but sacrifice on isolation/consistency: NoSQL, BASE

The course will cover both!

Discussion of ACID guarantees

- classical, "all-inclusive" guarantee in (relational) database systems
- solves the problems demonstrated in Examples (and more)
- well-established theory and clear semantics
- mature and well-engineered implementations
  - *Recovery* for *A*, *D*
  - *Concurrency Control* for *I*

What do you think of this abstraction, in particular after part 1 of the lecture?

We are looking at a fork in the road:

- provide ACID, but limit scalability and availability → 1980 – 2000
- favour scalability and availability, but sacrifice on isolation/consistency: NoSQL, BASE → 2000 – 2010

The course will cover both!

Data-Centric Distributed Applications

Union of two technologies:

- Database Systems + Distributed Systems
- Database systems provide
  - data independence (physical & logical)
  - centralized and controlled data access
  - integration
- Distributed System provide
  - distribution
  - scaling
  - reliability

Data-Centric Distributed Applications

Union of two technologies:

- Database Systems + Distributed Systems
- Database systems provide
  - data independence (physical & logical)
  - centralized and controlled data access
  - integration
- Distributed System provide
  - distribution
  - scaling
  - reliability

Goals of Data-Centric Distributed Applications

1. Transparent management of distributed and replicated data
2. Reliability/availability through distributed transactions
3. Improved performance
4. Easier and more economical system expansion

Focus of this course: Distributed transactions!
If you are interested in the other aspects, visit my course in the winter term.

Challenges of Distributed/Replicated Data

- Storing copies of data on different nodes enables availability, performance and reliability
- Data needs be consistent
  - Synchronizing concurrent access
  - Detecting and recovering from failures
  - Deadlock management
- Fundamental conflicts between requirements (see CAP theorem)

Goals of Data-Centric Distributed Applications

1. Transparent management of distributed and replicated data
2. Reliability/availability through distributed transactions
3. Improved performance
4. Easier and more economical system expansion

Focus of this course: Distributed transactions!
If you are interested in the other aspects, visit my course in the winter term.

Challenges of Distributed/Replicated Data

- Storing copies of data on different nodes enables availability, performance and reliability
- Data needs be consistent
  - Synchronizing concurrent access
  - Detecting and recovering from failures → hard
  - Deadlock management
- Fundamental conflicts between requirements (see CAP theorem)