Technische Universität Darmstadt

Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 3: Distributed Algorithms

Section 1: Foundation: Motivation, Properties, Characteristics

Lecturer: Prof. Dr. Max Mühlhäuser

Distributed Algorithms Chapter

1. Introduction: Lack of common time & state, race conditions & observations
2. Synchronization: alignment of physical & *logical* clocks, 'global states'
3. Coordination: Failure Detectors, Mutual Exclusion, Leader Election
4. Cooperation: Multicast (On Different Topologies) & Consensus + Byzantine Generals
5. Local Algorithms: how to get along with local information only

Basic Problem

„Bermuda Triangle“ of Distributed Systems:

Basic Problem #1: **No Global State** accessible (with acceptable effort)

- No synchronized global variables, no global shared memory
- Message / agent travelling A → B: outdated state of A arrives at B

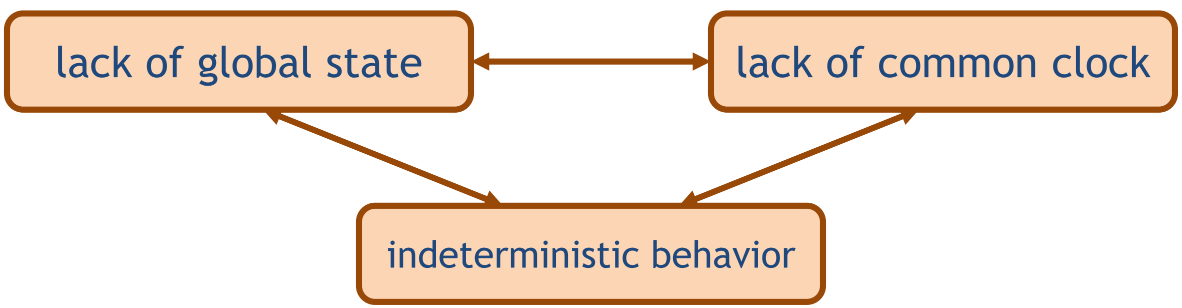
Basic Problem #2: **No Global Time** - clocks Not 100% synchronized

events E_A, E_B at A, B with recorded times $t(E_A) < t(E_B)$:

- May have happened at $t(E_A) > t(E_B)$!!
- When is it safe to „believe“ $t(E_A) < t(E_B)$?
- How to find out which is true? if undecidable: does distinction matter?

Basic Problem #3: **No Deterministic Behavior –** same execution ↔ different „results“ (at least internally)

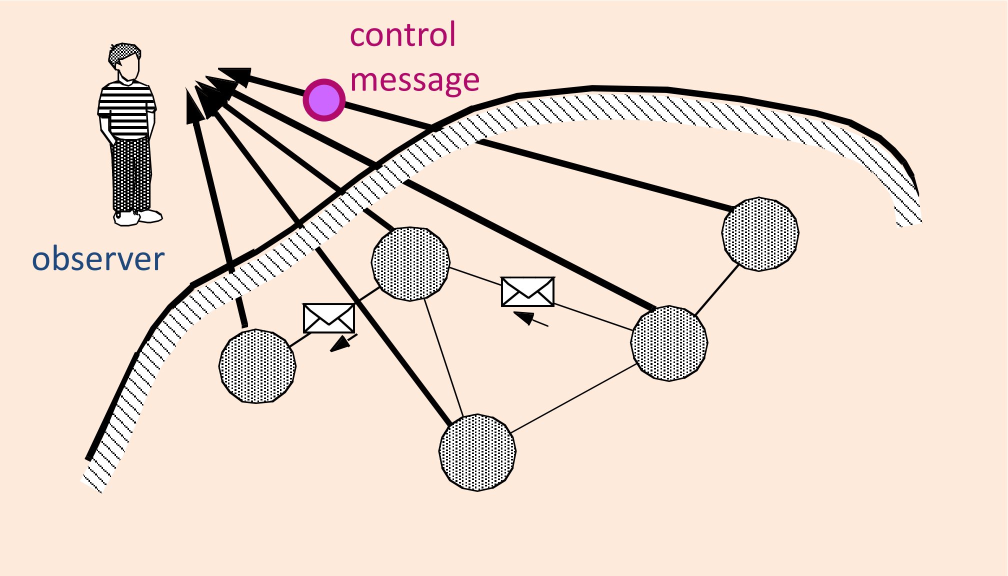
- „race“ conditions (messages from different senders, different threads arrive in different sequence)
- Erroneous underlying sys.: „correct program“ w/ unpredictable result!!



Basic Problems, extended

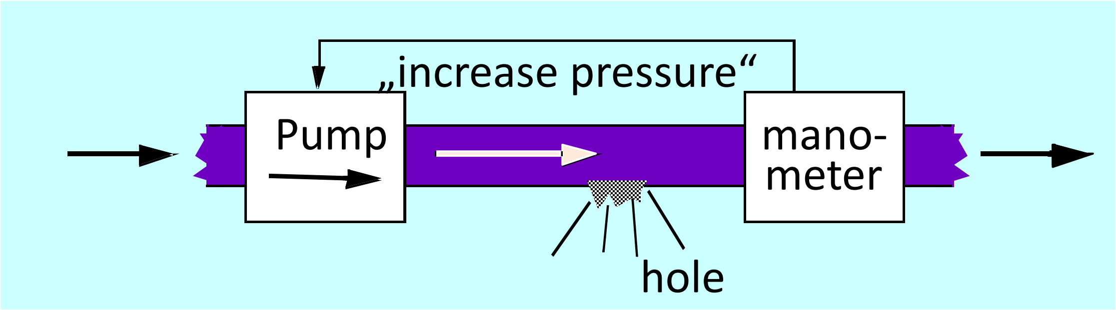
Lack of global state & time:

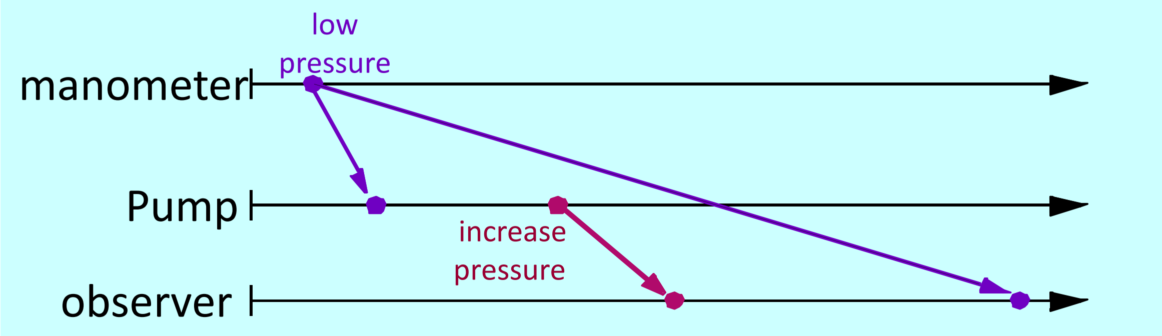
- Observer (process) informed via messages (varying delay)
  → messages report about different snapshot times of the past!



Example: Process Control (Causality)

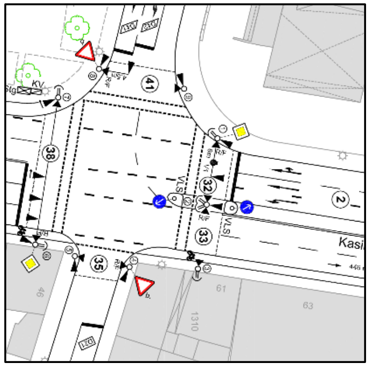
- One of the resulting problems: **causality** difficult to identify
- Imagine pump as shown below:
  - observer may infer:
    [„increase preasure“ command → „low pressure“] ⇒ high pressure caused hole!



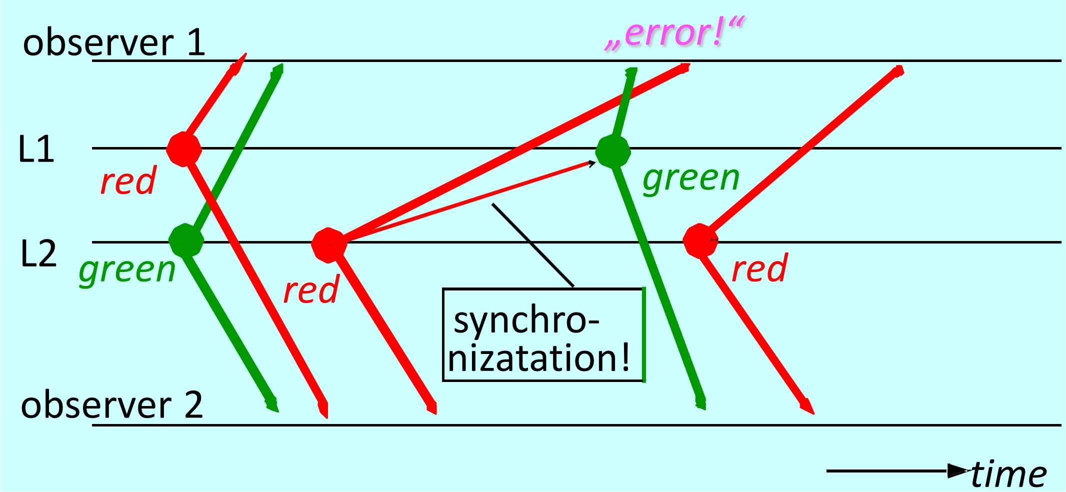


Example: Traffic Lights (Causality)

- Another example: distributed traffic light control (here: 2 lights)



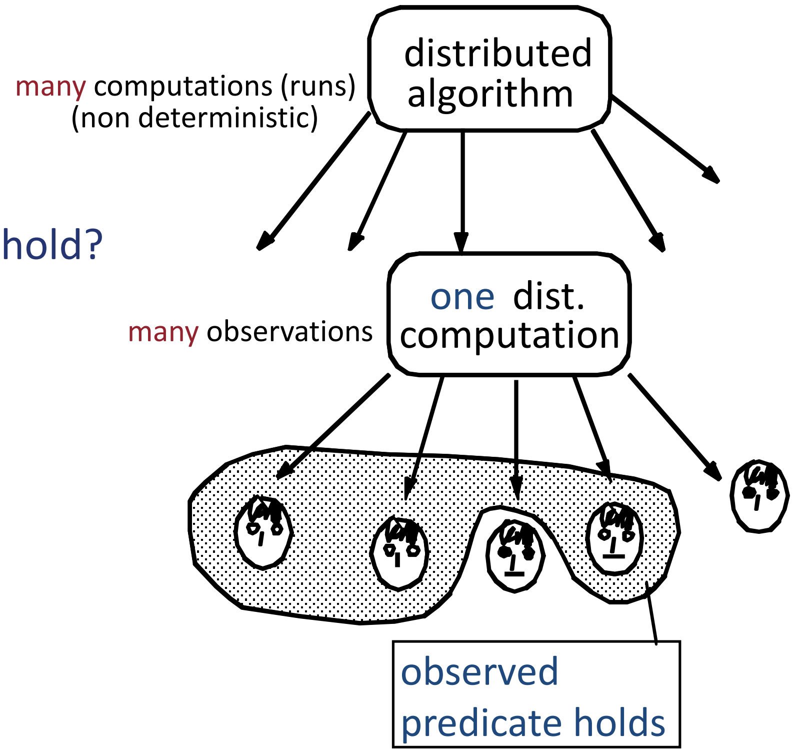
- Observers 1 & 2 see incorrect vs. correct causality „snapshots“



Basic Problems, extended

(1) no global time & (2) no global state & (3) indeterminism →
N observers may even observe identical execution differently

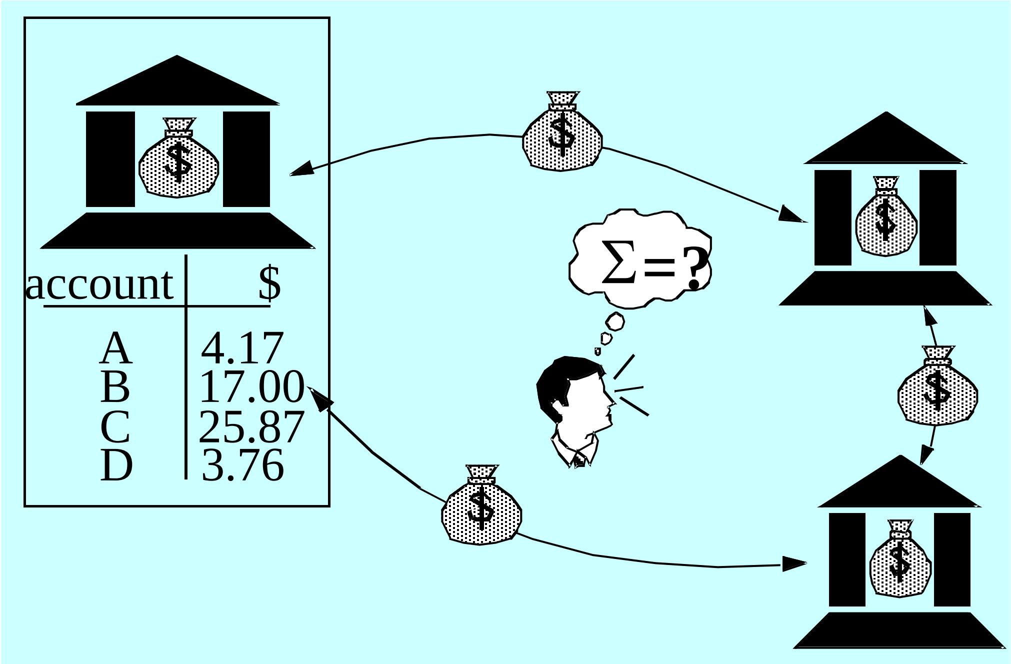
Hence: 1 Algorithm
→Many (differing!) executions
→Each with many (differing!) observations



- In traffic lights example, e.g.: does „#green lights < 2“ hold?
  - Holds for reality (invisible in DistSys) & observer 2
  - Does not always hold for observer 1
- important? **yes**, e.g.:
  - Above example: observe „correctness“ criterion
  - Global „snapshots“: observe ‚global state‘
  - Termination
  - Garbage collection
  - Consistent checkpoints
  - Deadlock
- Imagine problem of distributed debugging!

Example: Banking (Global Snapshot = Money in Circulation)

- Task: count total amount of money in circulation, for entire economy
- Important: can‘t stop distributed process for counting purpose!
  - Important for all of TK1-3.x: can‘t (or don‘t want to) stop app during orthogonal ‚basic‘ algorithm
  - Hence, distributed basic algorithm has to work in an ‚uninterrupted‘ overall system
- Global state *or* global time would solve the issue
  - Both are not possible
  - We need a different approach
    - See next subchapter
- Further issues include:
  - Termination?
  - Correctness?
  - Coping with
    - msg loss
    - (fraud?)



Correctness in distributed systems

How to address **correctness** for distributed programs/algorithms?

- First, a reminder re. **concurrent** programs/algorithms (where message passing is not considered)

Systems with high degree of concurrency:

- need some atomicity model
- … which is often based on "external effects" (for us: send and receive)

Simplistic example: Two concurrent processes operating on same variable X:

- P1: inc (X); // increase x by 1
- P2: inc (X);

- What happens when P1 and P2 are executed concurrently?
- i.e., will the following hold: $X = X_{old} + 2$?
  - Not necessarily if processes are not atomic: imagine …
    - inc(x) implemented in non-atomic way as inc(x) = {read (x); x:=x+1; write (x)};
    - read and write operations of P1 and P2 **intertwined** → $X = X_{old} + 1$!

Correctness in distributed systems

- Indeterminism and message-based communication mean:
  - For correctness, we need to consider every possible alternative
  - There are a lot of execution orders
- How to prove correctness?
  - Sometimes not as difficult as it seems
- yet, “overall correctness” hard to tackle; rather: prove … “correctness with respect to …” → defined in terms of properties
- Important properties:
  - Safety
  - Liveness
  - Fairness

Properties: Safety

Safety Properties:

- Are kept throughout computation and are always true
- Intuition: safety talks about "something bad"
  - Try to prove: "Bad thing" cannot happen
  - (or, if it does happen, we will know within finite number of steps)
- Examples:
  - **Deadlock freedom:** There is always a process that can execute another instruction (however, not necessarily does it execute)
  - **Mutual exclusion (Mutex):** It is not allowed for two different processes to enter the same critical section (i.e. code or memory region) concurrently
    - However: This holds even if of the critical sections are ever executed at all
  - **Mutex II:** a critical region may not be entered by ≥ 1 process at the same time

(Stability):

- Property always holds *once it has become true*
- If X holds at some time then X holds for the rest of the execution

Properties: Liveness

Liveness properties:

- Guarantee progress in computation
- Intuition: Liveness talks about “something good“
  - “Good thing” *must eventually* happen (in finite number of steps)
- Examples
  - **No starvation:** Any process that wishes to execute an instruction will eventually be able to execute
  - **Termination:** Program or process eventually terminates
  - Mutex: One of the processes will enter critical section
    - Note the difference to safety!
- Note: term “eventually” is very weak here
  - In terms of states it means that the “desired state” is reachable from every prior state of the system, but it does not say *when* it will actually be reached
  - Depending on model (state-based or else), “finite number of steps” not easy to tackle

Properties: Fairness

Fairness properties: … are stricter than liveness

- Weak fairness: If a process waits on a certain request, then eventually it will be granted
  - Again “eventually“: i.e., no time limit; difference to liveness not obvious
- Strong fairness: If a process performs request sufficiently frequently, then eventually it will be granted sufficiently often
  - “sufficiently often” also interpreted differently in different models
- Linear waiting: If a process performs a request then it will be granted *before any other process gets its request granted twice*
- FIFO (≈ linear waiting): If a process performs a request, then it will be granted before *any other process that asked later (*)* gets its request granted

Note: All above are easy to implement in a centralized system

(*) However, in a distributed system it is not clear what “before” or “later” mean

intuitive fairness with strict notion of before/later **cannot** be generally implemented

Model of a Distributed System

(remember pragmatic definition DistSys :== $\{AS_i\} \cup$ CSS)

For distributed algorithms and formal specifications, we define:

DistSys :== collection *P* of *N* processes p_i, $i = 1,2,\ldots N$

- Each process p_i has a state s_i consisting of its variables (which it transforms as it executes)
- Processes communicate only by sending messages over a network
- Only *three possible actions* of processes:
  - Send
  - Receive
  - Change own state
- Event:
  - The occurrence of a single action that a process carries out as it executes i.e., Send, Receive, change state

Remarks about complexity

(remember sequential algorithms: O-Notation etc. → time & space complexity)

Distributed Algorithms: ‚more than time & space complexity'

- time & space complexity extended by **#_of_computers C** (parallelism)
  - quite often: problem size **N grows proportional to C** for basic distributed algorithms
- **Communication complexity (discussed since about 1980):** concerned with amount of data transferred over the net for solving a problem (quasi the # of bits)
  - In reality, issues may be more subtle:
    - there may be a tradeoff between "message complexity" (# of msg's M = F(problem size N)) and "message *size* complexity" (msg size = F(problem size N)) [with N = # of nodes or # of processes)
    - relation to connectivity (# of (hard/soft) links - often: should-but-does-not grow as fct. of C^2)
- **Time complexity revised:** how does computation „time" grow with N
  - On one node? on the sum of all nodes?
    - … and how is degree of parallel computing considered?
    - is N related to C? ("integer factorization": N ≠ F(C); "distributed snapshot": N=C)
  - Literature: computational complexity class NC (Nick's class):
    - Set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors
- **Usually tradeoffs** between most of the above
  … *and* maybe between complexity and „accuracy" (near-optimal, heuristics, …)

Complexity, Concurrency, Speedup

Concurrency plays a role …:

- In „vonNeumann“ computers - scheduling introduces „quasi-concurrency“
- In parallel computers (SMP, Multicore, …)
- In DistSys: ‘level of concurrency’ (parallelism) essential for performance

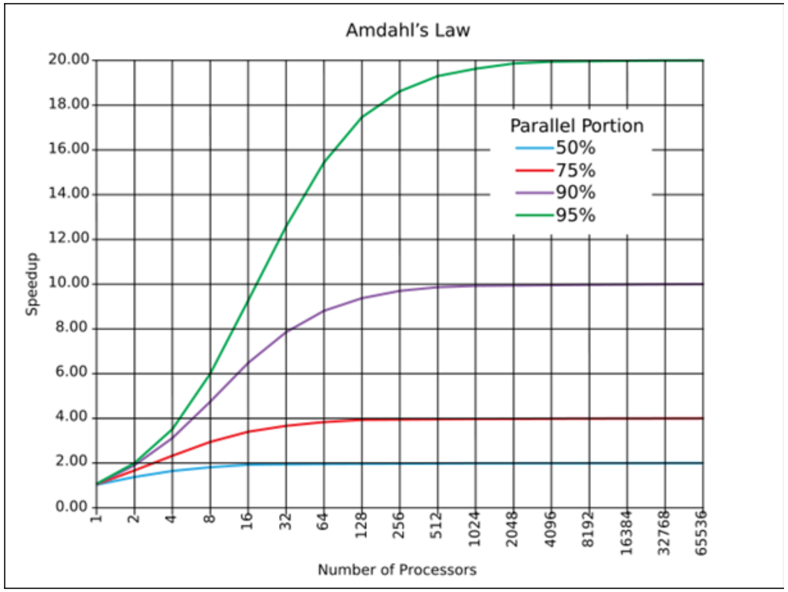
Therefore, for complexity & performance issues in Distributed Systems …

- … we may ‘borrow’ from a *very* old dispute about concurrent programming
- .. where speedup treats possible parallelization on multiple processors/nodes
  - **Speedup** :== ratio between
    - a sequential algorithm ***Seq*** and
    - a parallelized version of it, ***Par***
- (next slide skipped, repeated from Cloud Computing chapter)

Complexity, Concurrency, Speedup

Amdahl's Law:

- … is 'frustrating'
- … divides program into parallelizable portion $p \in (0,1)$ and portion a requiring sequential computation
- $a + p = 1$ (i.e.: $p = 1-a$)
- On P processors, the parallelizable fraction can be computed in one P^{th} of the time; overall time: $a + p/P$
  - Since a+p=1, this yields the following speedup (1 vs. P processors):
  - Considering communication overhead o(P):
  - Basically bound by 1/a
  - E.g., a=5% → speedup bound to 20fold (only!!)
- Bottomline result: The **sequential fraction** of a program (in terms of run time) determines the **upper bound for speedup**!



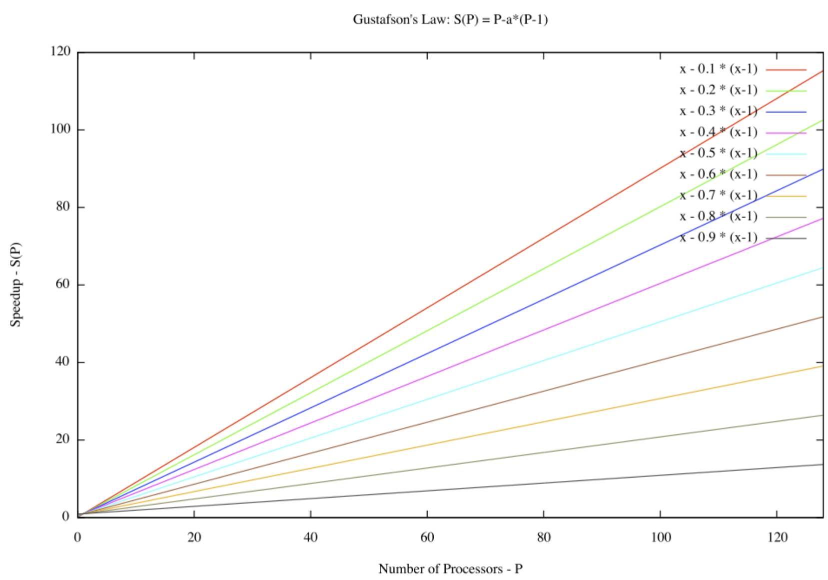
$$speedup\ s = \frac{Seq}{Par} = \frac{a+p}{a+\frac{p}{P}} = \frac{1}{a+\frac{1-a}{P}}$$

$$s = \frac{1}{a + o(P) + \frac{1-a}{P}} \leq \frac{1}{a}$$

Concurrency

Gustafson's Law: basically intended as counter argument to Amdahl's Law

- Much more promising results,
- Relies on observation: *problem* growth and 'growth' of *no. of processors* are correlated



Basic Problem

„Bermuda Triangle“ of Distributed Systems:

Basic Problem #1: **No Global State** accessable (with acceptable effort)

- no synchronized global variables, no global shared memory
- message / agent travelling A → B: out-dated state of A arrives at B

Basic Problem #2: **No Global Time** - clocks Not 100% synchronized

events E_A, E_B at A, B with recorded times $t(E_A) < t(E_B)$:

- may have happened at $t(E_A) > t(E_B)$!!
- when is it safe to „believe“ $t(E_A) < t(E_B)$?
- how to find out which is true? if undecidable: does distinction matter?

Basic Problem #3: **No Deterministic Behavior –** same execution ↔ different „results“ (at least internally)

- „race“ conditions (messages from different senders, different threads arrive in different sequence)
- erroneous underlying sys.: „correct program“ w/ unpredictable result!!