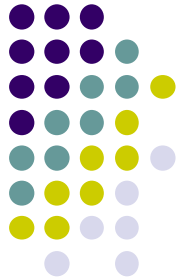
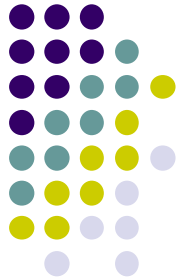


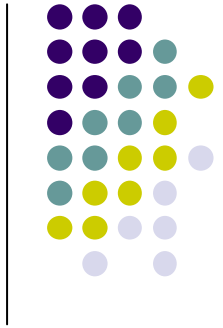
CS60002: Distributed Systems

Spring 2016

Course Structure

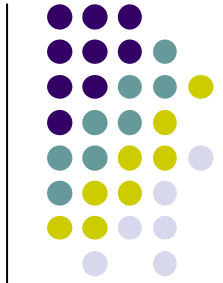


- Basic concepts
  - Models, complexity measures
- Fundamental problems/algorithms
  - Clocks and event ordering, global state collection, leader election, mutual exclusion, distributed graph algorithms, deadlock detection
- Basics of Fault-Tolerance in Distributed Systems
  - Fault models, types of tolerance, agreement/consensus, atomic commit
- Distributed databases
- Replication
- Distributed file systems
- Authentication in distributed systems – Kerberos
- Case studies of distributed systems in action around you



Basic Concepts

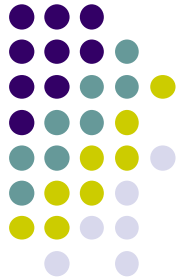
Distributed System



A broad definition

A set of autonomous processes that communicate among themselves to perform some task

- Modes of communication
  - Message passing
  - Shared memory
- Includes single machine with multiple communicating processes also

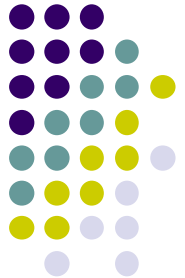


A more common definition

A network of autonomous computers that communicate by message passing to perform some task

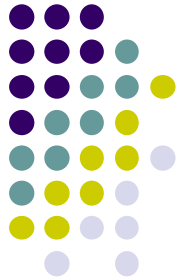
- A practical distributed system may have both
  - Computers that communicate by messages
  - Processes/threads on a computer that communicate by messages or shared memory

Advantages



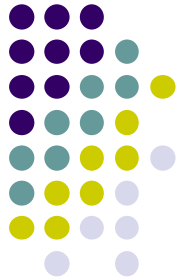
- Resource Sharing
- Higher throughput
- Handle inherent distribution in problem structure
- Fault Tolerance
- Scalability

Representing Distributed Systems



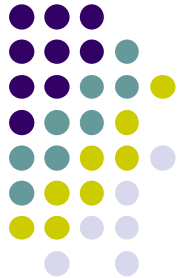
- Graph representation
  - Nodes = processes
  - Edges = communication links
  - Links can be bidirectional (undirected graph) or unidirectional (directed graph)
  - Links can have weights to represent different things (Ex. delay, length, bandwidth,…)
  - Links in the graph may or may not correspond with physical links

Why are They Harder to Design?

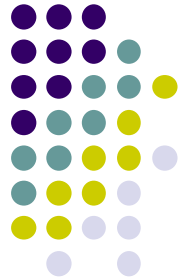


- Lack of global shared memory
  - Hard to find the global system state at any point
- Lack of global clock
  - Events cannot be started at the same time
  - Events cannot be ordered in time easily
- Hard to verify and prove
  - Arbitrary interleaving of actions makes the system hard to verify
  - Same problem is there for multi-process programs on a single machine
  - Harder here due to communication delays

Example: Lack of Global Memory



- Problem of Distributed Search
  - A set of elements distributed across multiple machines
  - A query comes at any one machine A for an element X
  - Need to search for X in the whole system
- Sequential algorithm is very simple
  - Search and update done on a single array in a single machine
  - No. of elements also known in a single variable

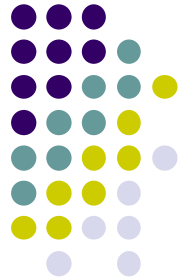


- A distributed algorithm has more hurdles to solve
  - How to send the query to all other machines?
  - Do all machines even know all other machines?
  - How to get back the result of the search in each m/c?
  - Handling updates (both add/delete of elements at a machine and add/remove of machines) – adds more complexity

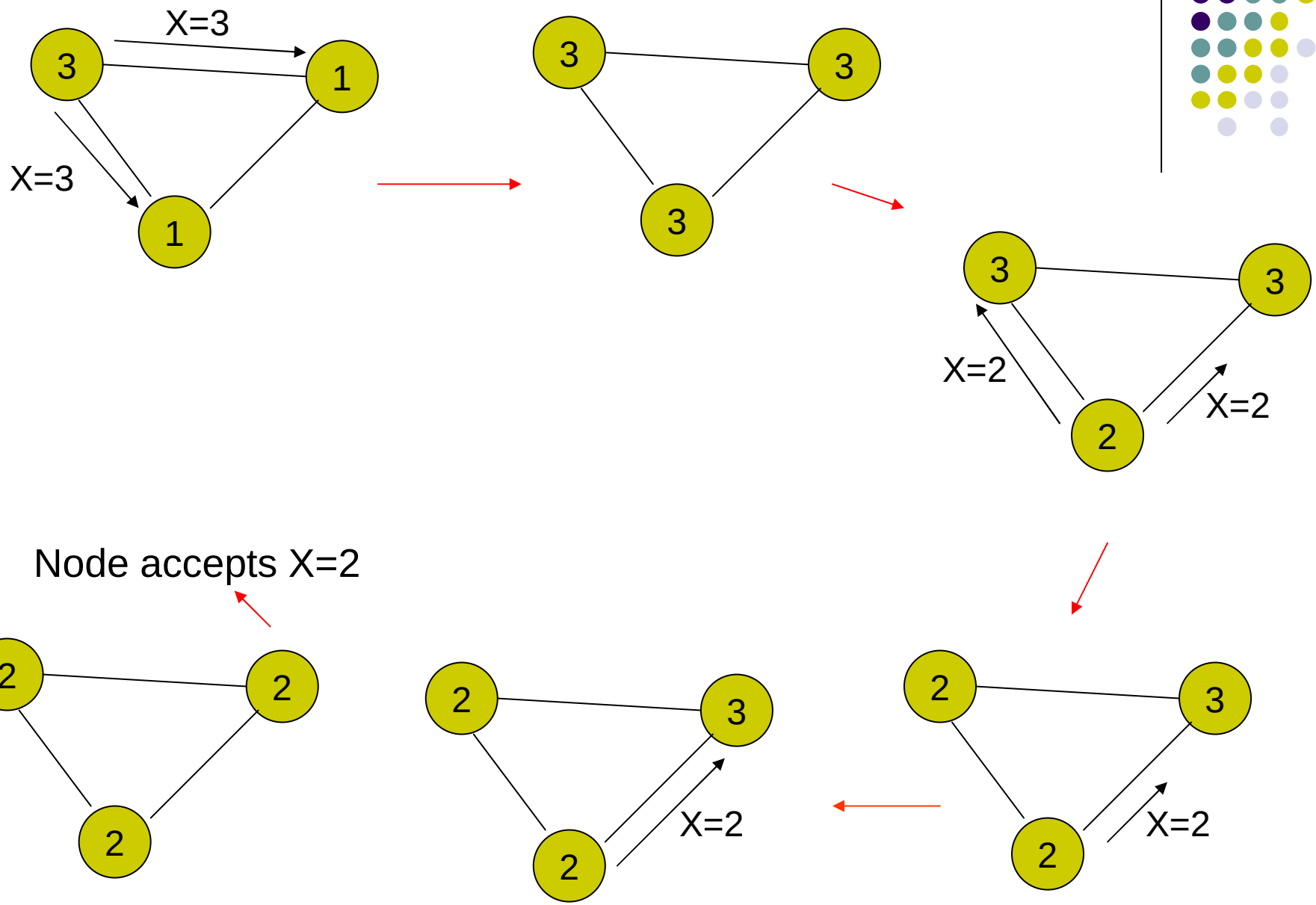
Main problem

No one place (global memory) that a machine can look up to see the current system state (what machines, what elements, how many elements)

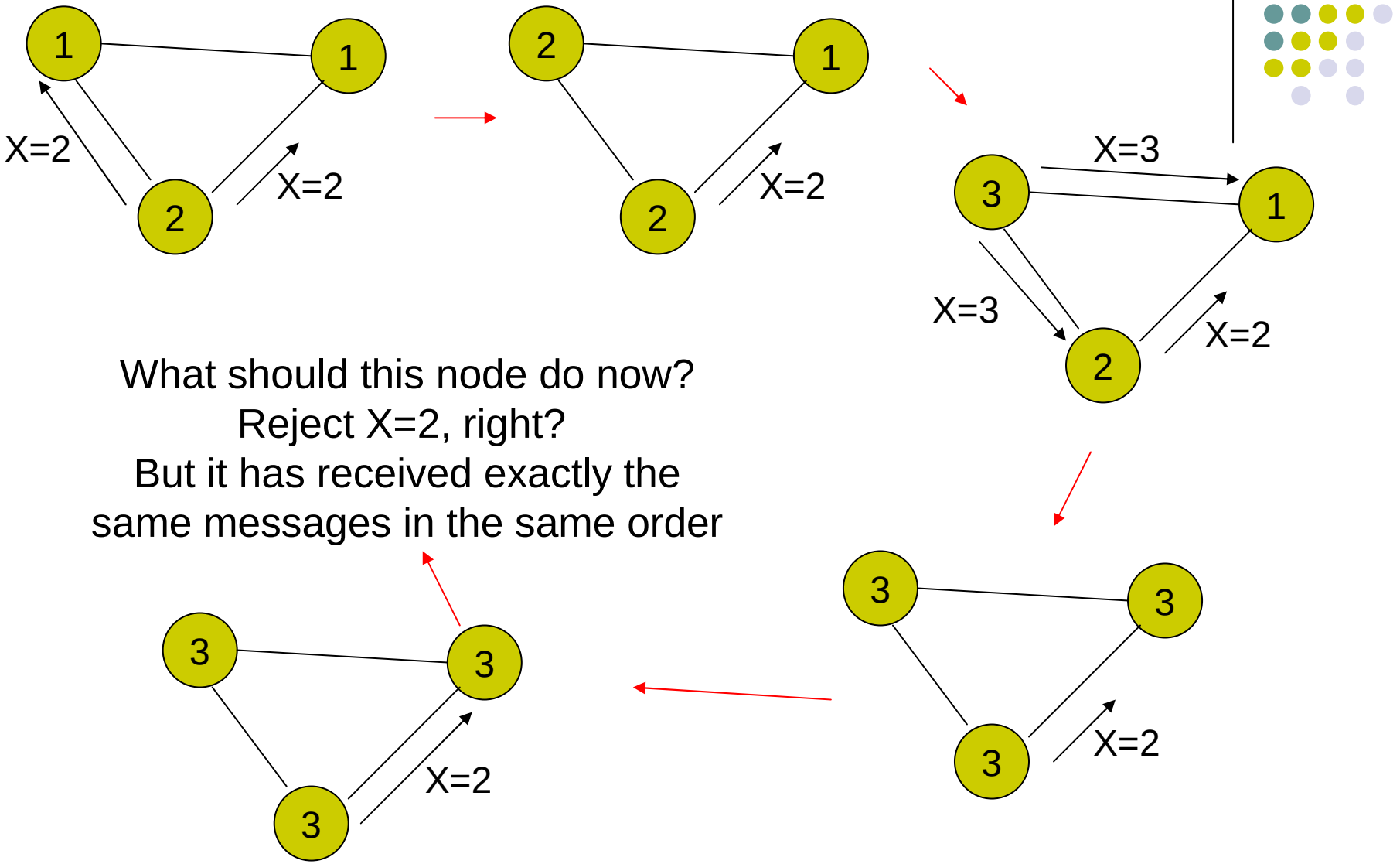
Example: Lack of Global Clock



- Problem of Distributed Replication
  - 3 machines A, B, C have copies of a data X, say initialized to 1
  - Query/Updates can happen in any m/c
  - Need to make the copies consistent within short time in case of update at any one machine
  - Naïve algorithm
    - On an update, a machine sends the updated value to the other replicas
    - A replica, on receiving an update, applies it


X=3
3
1
X=3
1
3
3
3
3
3
X=2
2
X=2
Node accepts X=2
2
2
2
2
3
2
X=2
2
3
2
X=2

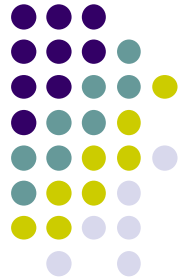
But then, consider the following scenario



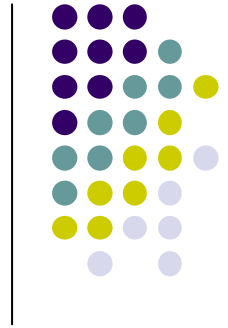
What should this node do now?
Reject X=2, right?
But it has received exactly the same messages in the same order

Could be easily solved if all nodes had a synchronized global clock

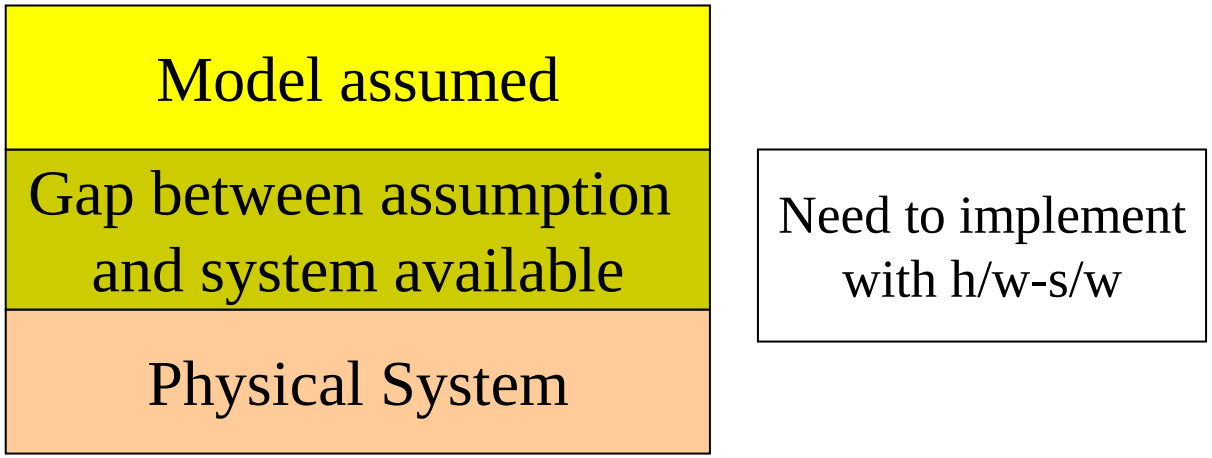
Models for Distributed Algorithms



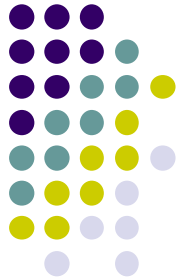
- Informally, guarantees that one can assume the underlying system will (or will not!) give
  - Topology : completely connected, ring, tree, arbitrary,…
  - Communication : shared memory/message passing (Reliable? Delay? FIFO? Broadcast/multicast?…)
  - Synchronous/asynchronous
  - Failure possible or not
    - What all can fail?
    - Failure models (crash, omission, Byzantine, timing,…)
  - Unique Ids
  - Other Knowledge : no. of nodes, diameter



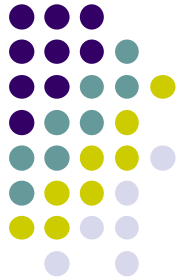
- Less assumptions => weaker model
- A distributed algorithm needs to specify the model on which it is supposed to work
- The model may not match the underlying physical system always



Complexity Measures

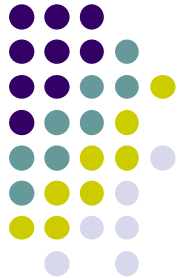


- Message complexity : total no. of messages sent
- Communication complexity/Bit Complexity : total no. of bits sent
- Time complexity : For synchronous systems, no. of rounds. For asynchronous systems, different definitions are there
  - Asynchronous round, no. of steps for central scheduler based systems etc.
- Space complexity : total no. of bits needed for storage at all the nodes



- Complexity depends on model
  - Ex: Message complexity may differ by O(n) if broadcast is assumed instead of unicast
- Should be careful in comparing/translating complexities across models

Example: Distributed Search Again

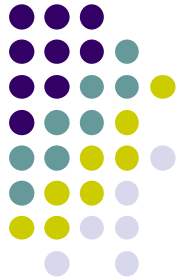


- Assume that all elements are distinct
- Network represented by graph G with n nodes and m edges

Model 1

Asynchronous, completely connected topology, reliable communication

- Algorithm:
  - Send query to all neighbors
  - Wait for reply from all, or till one node says Found
  - A node, on receiving a query for X, does local search for X and replies Found/Not found.
- Worst case message complexity = 2(n – 1) per query



Model 2

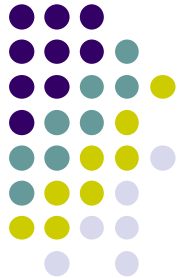
Asynchronous, completely connected topology, unreliable communication

- Algorithm:
  - Send query to all neighbors
  - Wait for reply from all, or till one node says Found
  - A node, on receiving a query for X, does local search for X and replies Found/Not found.
  - If no reply within some time, send query again
- Problems!
  - How long to wait for? No bound on message delay!
  - Message can be lost again and again, so this still does not solve the problem.
  - In fact, impossible to solve (may not terminate)!!

Model 3

Synchronous, completely connected topology, reliable communication

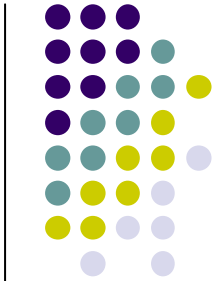
- Maximum one-way message delay = α
- Maximum search time at each m/c = β
- Algorithm:
  - Send query to all neighbors
  - Wait for reply from all for T = $2\alpha + \beta$, or till one node says Found
  - A node, on receiving a query for X, does local search for X and replies Found if found, does not reply if not found
  - If no reply received within T, return “Not found”
  - Message complexity = n -1 if not found, n if found
  - Message complexity reduced, possibly at the cost of more time



Model 4

Asynchronous, reliable communication, but not completely connected

- How to send the query to all?
- Algorithm (first attempt):
  - Querying node A sends query for X to all its neighbors
  - Any other node, on receiving query for X, first searches for X. If found, send back “Found” to A. If not, send back “Not found” to A, and also forward the query to all its neighbors other than the one it received from (flooding)
  - Eventually all nodes get it and reply
  - Message complexity – O(nm) (why?)



- But are we done?
  - Suppose X is not there. A gets many “Not found” messages. How does it know if all nodes have replied? (*Termination Detection*)
- Lets change (strengthen) the model
  - Suppose A knows n, the total number of nodes
    - A can now count the number of messages received. Termination if at least one “Found” message, or n “Not found” messages
    - Message complexity – O(nm)
  - Suppose A knows upper bound on network diameter and synchronous system
    - Can be done with O(m) messages only
- Can you do it without changing the model?

So Which Model to Choose?

- Ideally, as close to the physical system available as possible
  - The algorithm can directly run on the system
- Should be implementable on the physical system by additional h/w-s/w
  - Ex., reliable communication (say TCP) over an unreliable physical system
- Sometimes, start with a strong model, then weaken it
  - Easier to design algorithms on a stronger model (more guarantees from the system)
  - Helps in understanding the behavior of the system
  - Can use this knowledge to then design algorithms on the weaker model