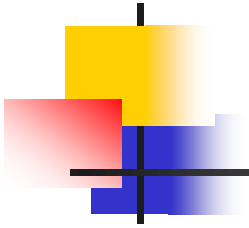
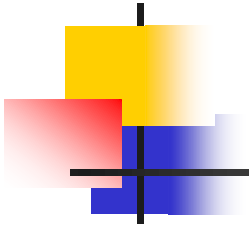
Introduction to Distributed Computing

Slide source acknowledgement: Arobinda Gupta and Pallab Dasgupta, IIT Kharagpur



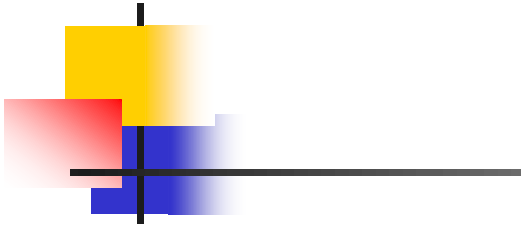
Talk Outline

- Introduction to Distributed Computing
  - What are they?
  - Why are they harder to design?
  - Importance of models
  - Complexity measures
  - Some classical problems
  - The notion of time and ordering of events
  - Some interesting examples
- Distributed System Realizations



Introduction

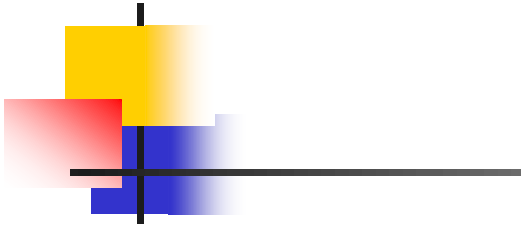
- Distributed Systems
  - Multiple independent computers that appear as one
  - Lamport's Definition
    - " You know you have one when the crash of a computer you have never heard of stops you from getting any work done."
  - A number of interconnected autonomous computers that provide services to meet the information processing needs of modern enterprises



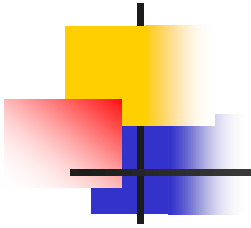
A more specific definition

A network of autonomous computers that communicate to perform some task

- Modes of communication
  - Message passing
  - Distributed shared memory
    - A common shared address space built over physical memory on different machines
  - Partially shared memory
    - Each node can read and write its own memory, and read its neighbors' memories



- A practical distributed system may have both
  - Computers that communicate by messages
  - Processes/threads on a computer that communicate by messages or shared memory
- We will focus on pure message passing systems in this talk

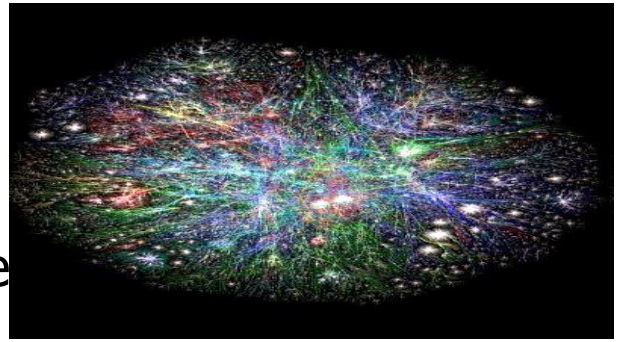


Characterizing Distributed Systems

- Multiple Autonomous Computers
  - each consisting of CPU's, local memory, stable storage, I/O paths connecting to the environment
  - Geographically Distributed
- Interconnections
  - some I/O paths interconnect computers that talk to each other
- Shared State
  - No shared memory
  - systems cooperate to maintain shared state
  - maintaining global invariants requires correct and coordinated operation of multiple computers.

Examples of Distributed Systems

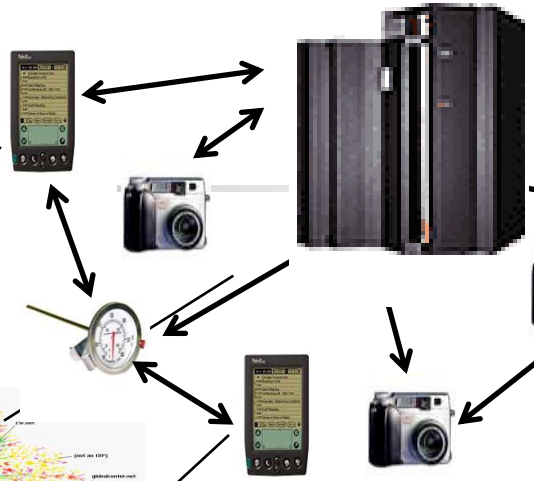
- Transactional applications - Banking systems
- Manufacturing and process control
- Inventory systems
- General purpose (university, office automation)
- Communication – email, IM, VoIP, social networks
- Distributed information systems
  - WWW
  - Cloud Computing Infrastructures
  - Federated and Distributed Database



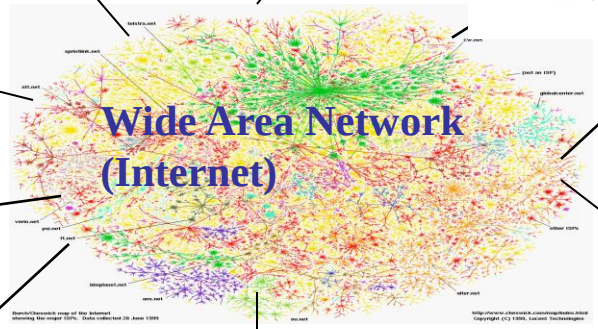
Next Generation Information Infrastructure

Distance Learning

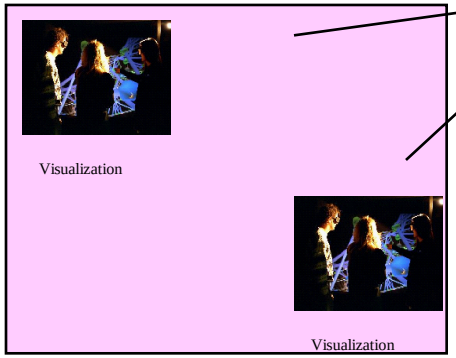
Electronic Commerce



DeviceNets & SensorNets

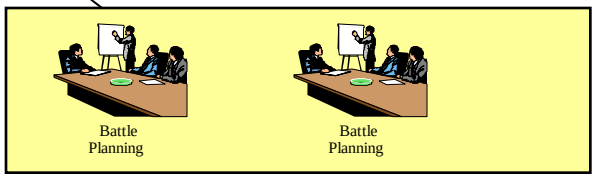


Wide Area Network (Internet)



Collaborative Multimedia (Telemedicine)

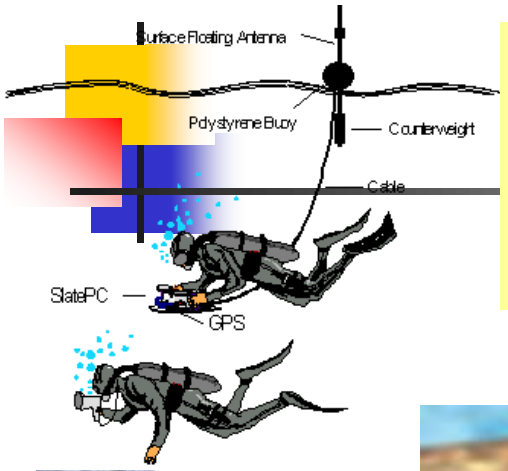
Server farms

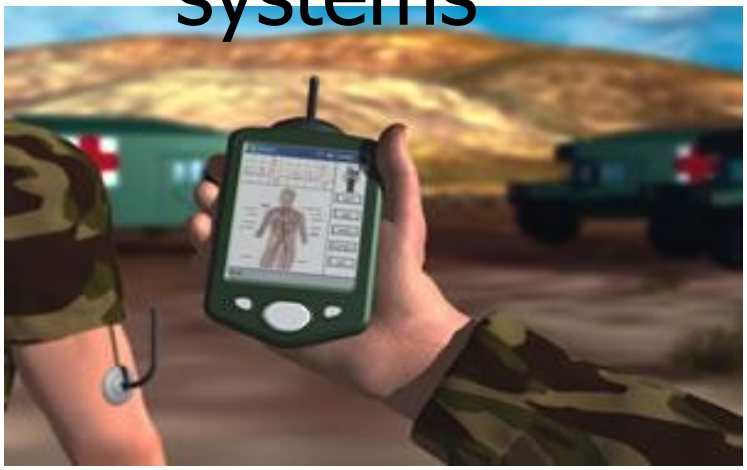


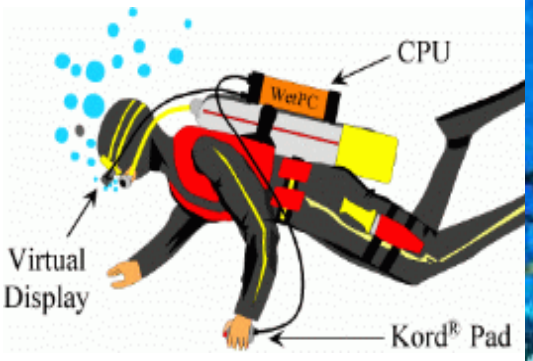
Collaborative Task Clients

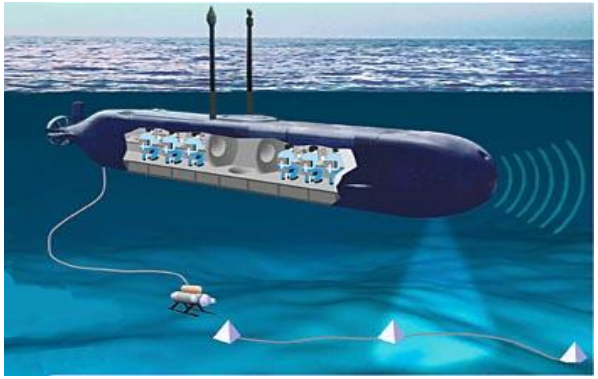
Requirements - Availability, Reliability, Quality-of-Service, Cost-effectiveness, Security

Mobile & ubiquitous distributed systems

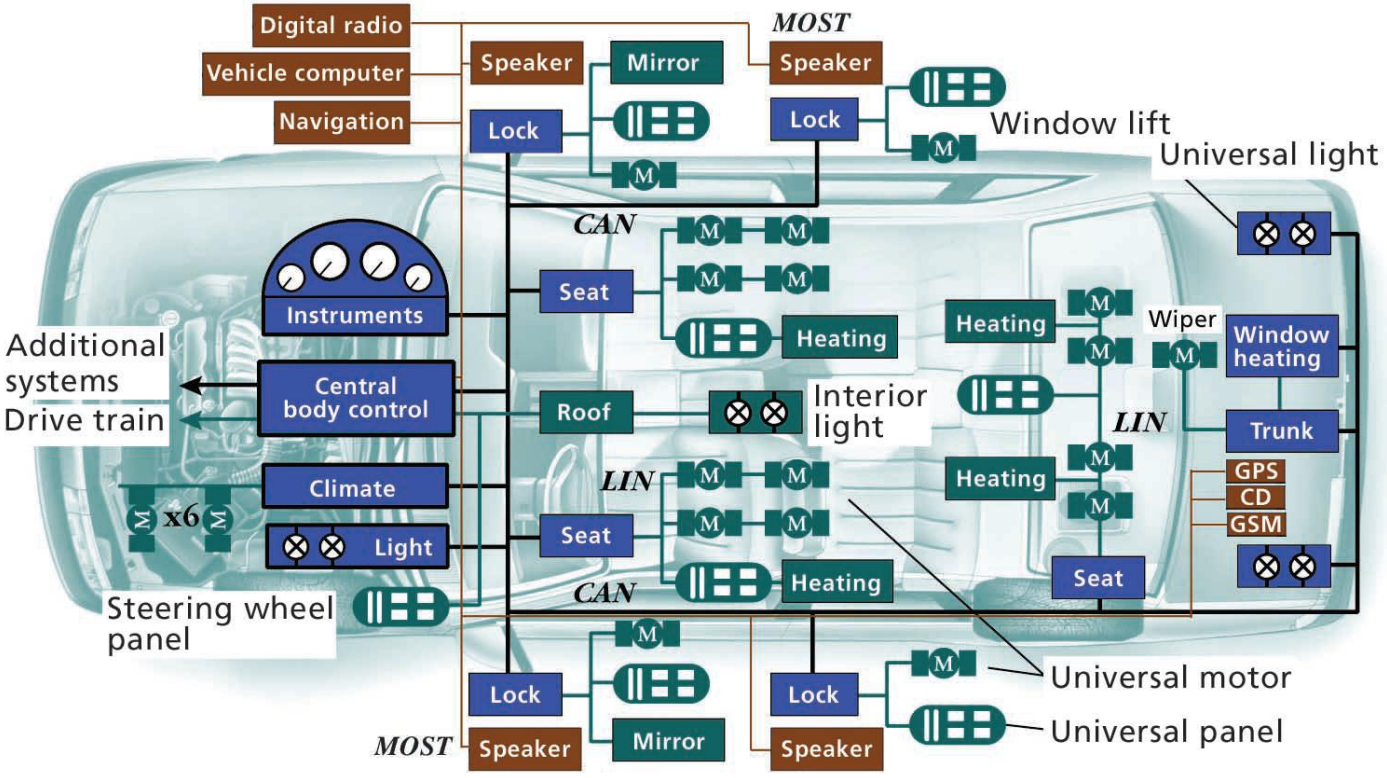








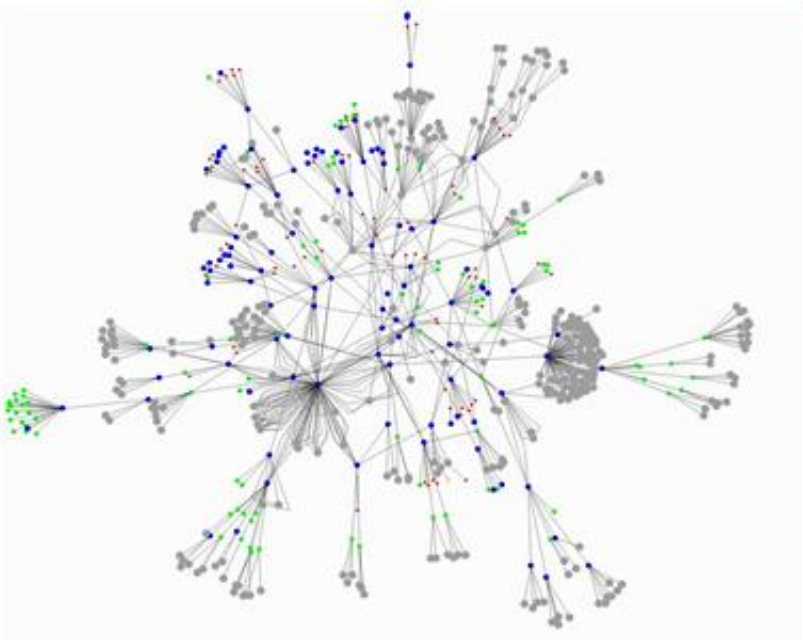
Example: *Automotive Control*



CAN Controller area network
GPS Global Positioning System
GSM Global System for Mobile Communications
LIN Local interconnect network
MOST Media-oriented systems transport

Source: Leen and Hefferman, IEEE Computer, Jan 2002

Peer to Peer Systems

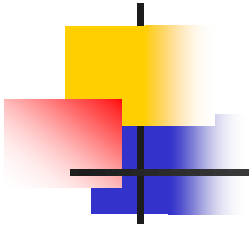


P2P File Sharing
- Napster, Gnutella, Kazaa, eDonkey, BitTorrent
- Chord, CAN, Pastry/Tapestry, Kademlia

P2P Communications
- MSN, Skype, Social Networking Apps

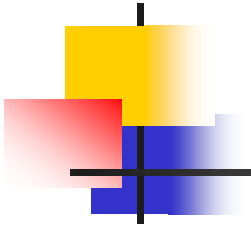
P2P Distributed Computing
- Seti@home

Use the vast resources of machines at the edge of the Internet to build a network that allows resource sharing without any central authority.



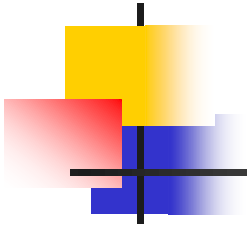
Why Distributed Computing?

- Inherent distribution
  - Bridge customers, suppliers, and companies at different sites.
- Speedup - improved performance
- Fault tolerance
- Resource Sharing
  - Exploitation of special hardware
- Scalability
- Flexibility



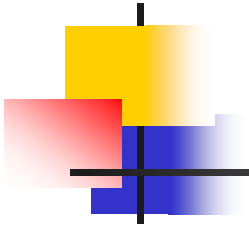
Why are Distributed Systems Hard?

- Scale
  - numeric, geographic, administrative
- Loss of control over parts of the system
- Unreliability of message passing
  - unreliable communication, insecure communication, costly communication
- Failure
  - Parts of the system are down or inaccessible
  - Independent failure is desirable



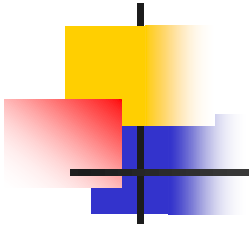
Design goals of a distributed system

- Sharing
  - HW, SW, services, applications
- Concurrency
  - compete vs. cooperate
- Scalability
  - avoids centralization
- Fault tolerance/availability
- Transparency
  - location, migration, replication, failure, concurrency



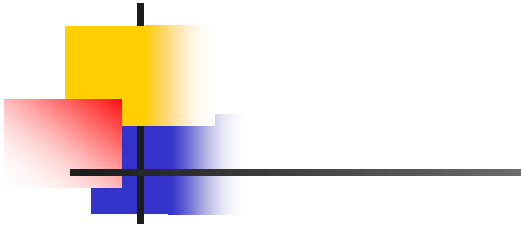
Distributed Algorithms

- Algorithms that run on distributed systems to perform some desired task
- Examples
  - Algorithms for mutual exclusion, for creating a spanning tree of a network, for building routing tables in the Internet, for scheduling jobs on different machines, for disseminating information to multiple nodes
  - Many many more…

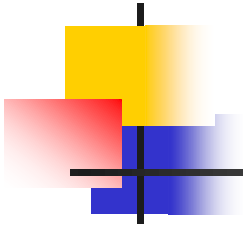


Why are They Harder to Design?

- Lack of global shared memory
  - No one place where the global system state can be accessed at any point
- Lack of global clock
  - Events cannot be started at the same time
  - Events cannot be ordered in time easily
  - Note that if we had a global shared memory, we could build a global clock easily

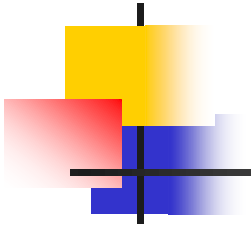


- Hard to verify and prove
  - Arbitrary interleaving of actions of different processes makes the system hard to verify
  - Same problem is there for multi-process programs on a single machine
  - Harder here due to communication delays that introduce additional non-determinism

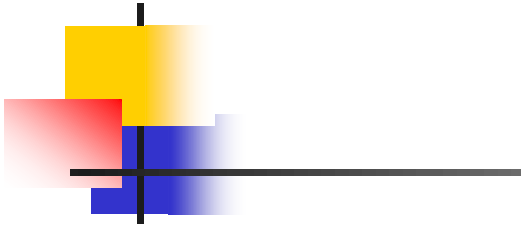


Example: Lack of Global Memory

- Problem of Distributed Search
  - A set of elements distributed across multiple machines (no duplicates)
  - Query for element X at any one machine A
  - A needs to search for X in the whole system
- Sequential algorithm is very simple
  - Search done on a single array in a single machine
  - No. of elements also known in a single variable

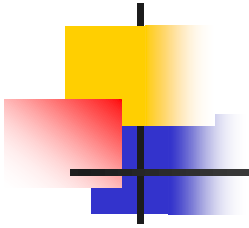


- A distributed algorithm has more hurdles to solve
  - How to send the query to all other m/cs?
  - Do all machines even know all other m/cs?
  - How to get back the result of the search in each m/c?
  - Handling updates (both add/delete of elements at a machine and add/remove of machines) – adds more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



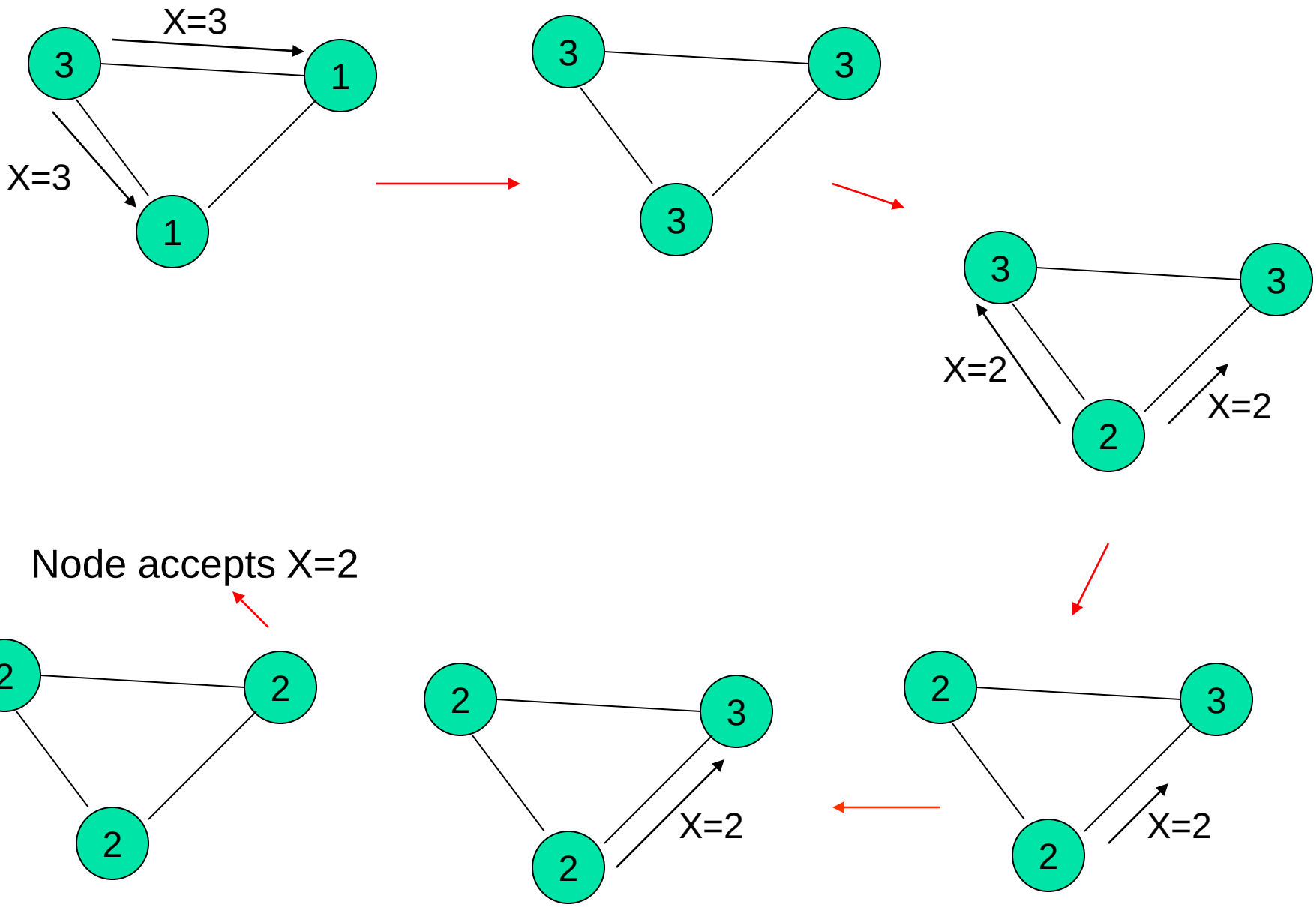
Main problem

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

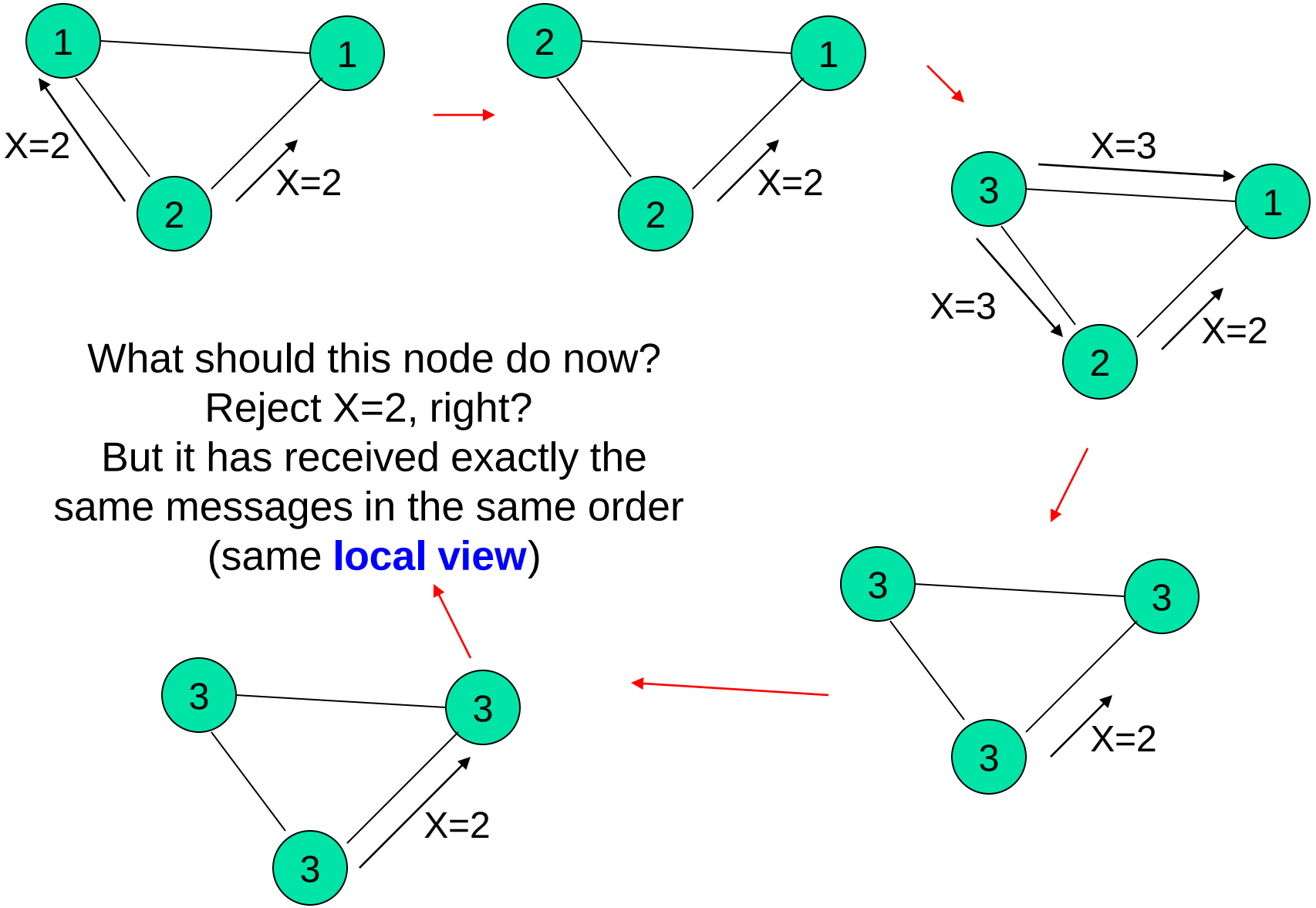


Example: Lack of Global Clock

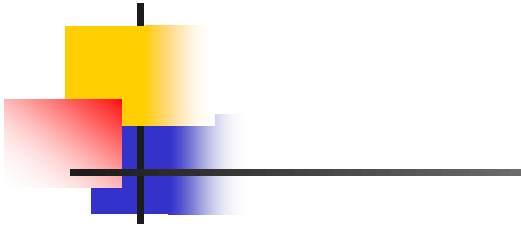
- Problem of Distributed Replication
  - 3 machines A, B, C have copies of a data X, say initialized to 1
  - Query/Updates can happen in any m/c
  - Need to make the copies consistent in case of update at any one machine
  - Naïve algorithm
    - On an update, a machine sends the updated value to the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


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

But then, consider the following scenario



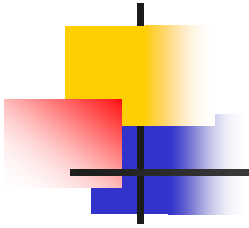
What should this node do now?
Reject X=2, right?
But it has received exactly the same messages in the same order
(same **local view**)



- Could be easily solved if all nodes had a synchronized global clock
  - Just timestamp each event with the clock value and order events according to timestamps
- But impossible to perfectly synchronize clocks in multiple machines
  - Message delays cannot be estimated exactly

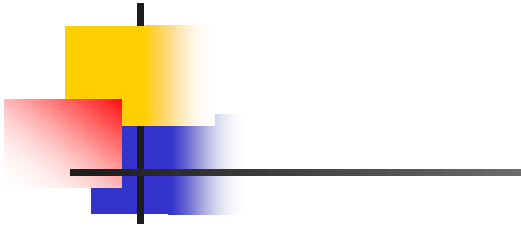
Classifying Distributed Systems

- Based on degree of synchrony
  - Synchronous
  - Asynchronous
  - Partially synchronous
- Based on communication medium
  - Message Passing
  - Shared Memory
- Fault model
  - Crash failures
  - Byzantine failures

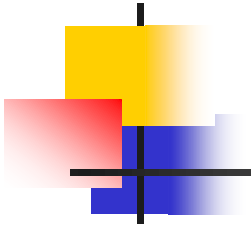


Models for Distributed Algorithms

- Informally, guarantees that one can assume the underlying system will give
  - Topology
    - Arbitrary, completely connected, ring, tree, …
  - Communication
    - Shared memory
    - Message passing (Reliable? Delay? FIFO? Broadcast/multicast?…)
  - Failure possible or not
    - What all can fail?
    - Failure models (crash, omission, Byzantine…)

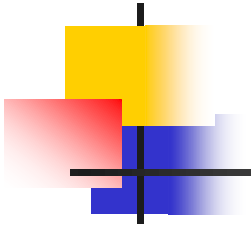


- Synchrony
  - Partially Synchronous systems make some timing assumptions about the system
    - Ex. max message transmission delay, max processing delay at nodes
  - Synchronous systems allow computation to proceed in rounds

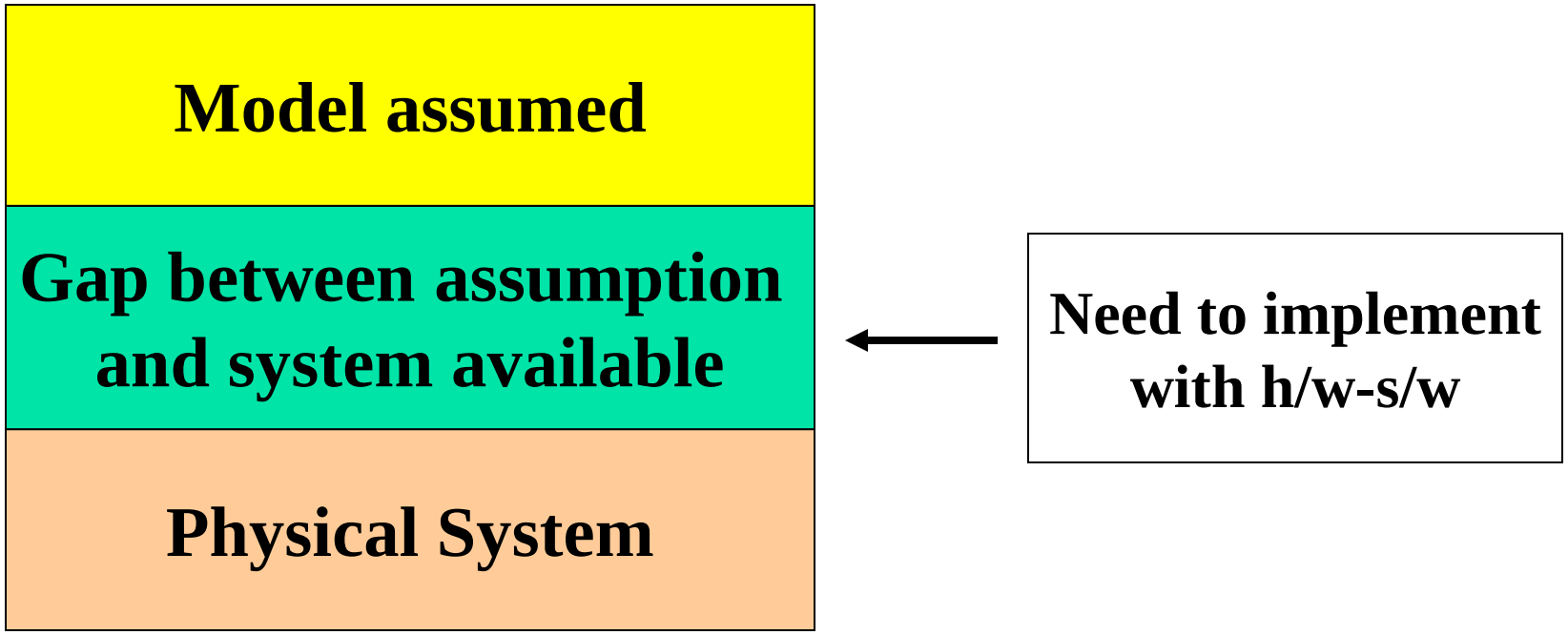


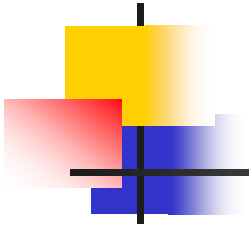
- Knowledge of number of nodes in the system
  - Exact or upper bound
- Knowledge of diameter of the network
- Others…

Less assumptions => weaker model



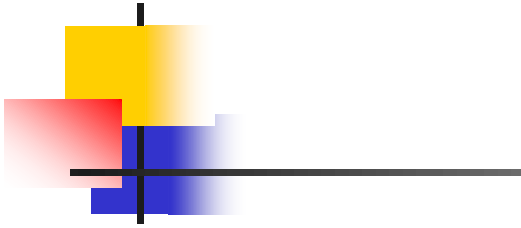
- A distributed algorithm needs to specify the model on which it is supposed to work
  - It will assume that the guarantees given by the model hold
- The model may not match the underlying physical system always
  - Need to put in additional hardware/software to implement the algorithm


Model assumed
Gap between assumption and system available
Physical System
Need to implement with h/w-s/w



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
  - Overhead of implementation to be carefully considered



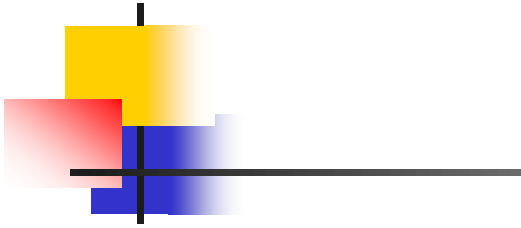
- But sometimes, start with a strong model (even if somewhat impractical to implement)
  - Easier to design algorithms on a stronger model (more guarantees from the system)
  - Helps in understanding the behavior of the system
  - Can use this knowledge to then
    - Design algorithms on a weaker model, or
    - Prove impossibility results on what can be solved on a weaker model

Example: Distributed Search Again

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

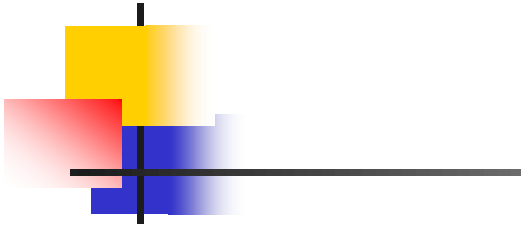
Model 1: *Asynchronous, completely connected topology, reliable communication*

- 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 messages per query = 2(n – 1)



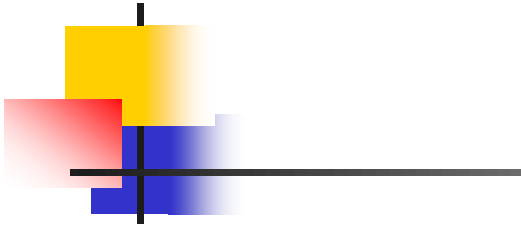
Model 2 : *Asynchronous, completely connected topology, unreliable communication*

- 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*

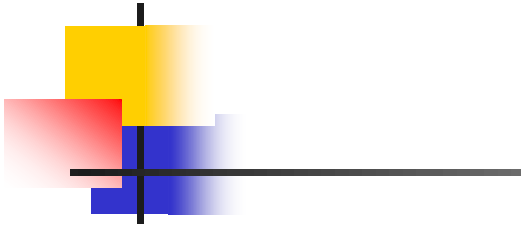
- Maximum one-way message delay = α
- Maximum search time at each m/c = β
- Algorithm:
  - Send query to all neighbors
  - Wait for reply from all for T = $2\alpha + \beta$, or till one node says Found
  - A node, on receiving a query for X, does local search for X and replies Found if found, does not 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



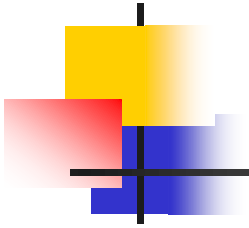
Model 4: *Asynchronous, reliable communication, but not completely connected*

- 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 ?

- 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 ?

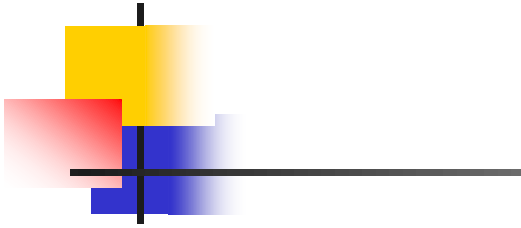


- Suppose A knows upper bound on network diameter and synchronous system
  - Can be done with O(m) messages only
- Can you do it without changing the model?
  - Try building and using a spanning tree!
  - What would be the message complexity?

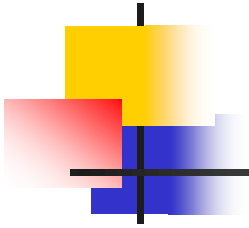


Complexity Measures

- Space complexity
  - Total no. of bits needed for storage at all the nodes
- Message complexity
  - Total no. of messages sent
  - Can be deceptive sometimes if message size is non-constant
- Communication complexity/Bit Complexity
  - Total no. of bits sent

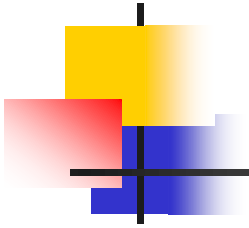


- Time complexity
  - For synchronous systems, no. of rounds
    - For asynchronous systems, what is time anyway?
      - Remember that there is no global clock
      - Different notions of time complexity measures exist
      - Should be careful when comparing the time complexities of two algorithms
        - Check if the definitions of time are the same



Some Classical Problems

- Ordering events in the absence of a global clock
- Capturing the global state
- Termination detection
- Mutual exclusion
- Leader election
- Clock synchronization
- Constructing spanning trees (and other graph structures)
- Agreement protocols



Distributed Algorithms in Action

- Domain Name System (DNS)
- Internet routing protocols
- Search engines
- Cloud computing
- High performance computing systems
- Distributed file systems (NFS, HDFS)
- Single sign-on login (Kerberos)
- Many many more….