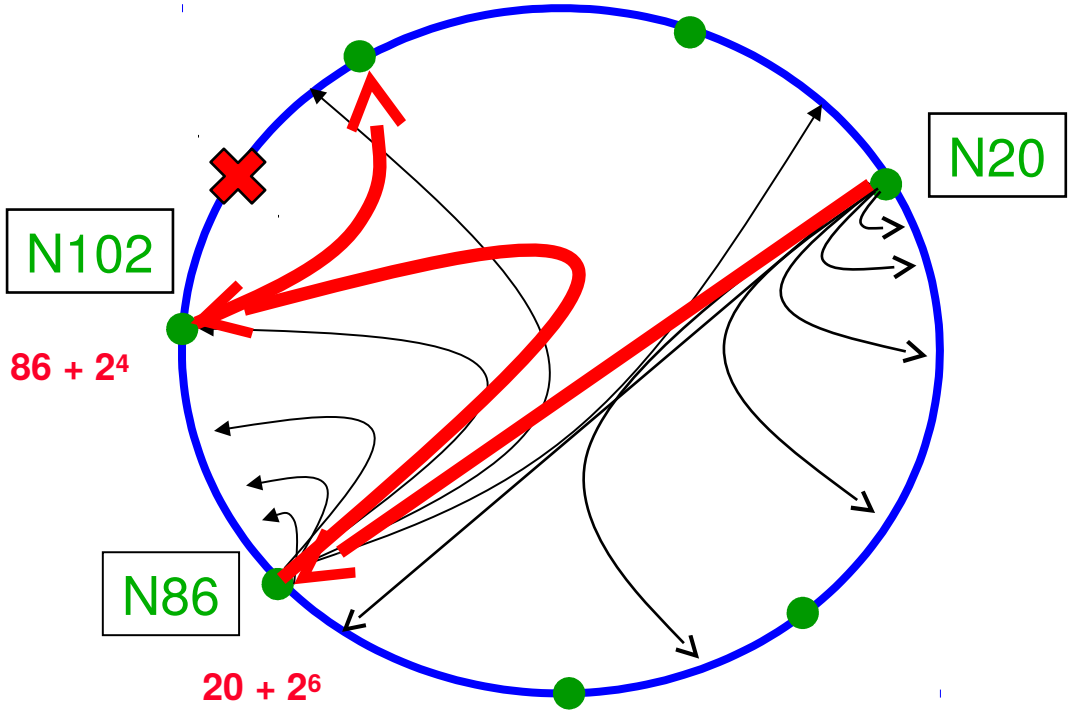
CSE 486/586 Distributed Systems

Consensus

Slides by Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap: Finger Table

- Finding a <key, value> using fingers



Let's Consider This…

Amazon EC2 Service Level Agreement

Effective Date: October 23, 2008

This Amazon EC2 Service Level Agreement ("SLA") is a policy governing the use of the Amazon Elastic Compute Cloud ("Amazon EC2") under the terms of the Amazon Web Services Customer Agreement (the "AWS Agreement") between Amazon Web Services, LLC ("AWS", "us" or "we") and users of AWS' services ("you"). This SLA applies separately to each account using Amazon EC2. Unless otherwise provided herein, this SLA is subject to the terms of the AWS Agreement and capitalized terms will have the meaning specified in the AWS Agreement. We reserve the right to change the terms of this SLA in accordance with the AWS Agreement.

Service Commitment

AWS wi[redacted]e Amazon EC2 available with an Annual Uptime Percentage (defined below) [redacted] at least 99.95% during the Service Year. [redacted] the event Amazon EC2 does not meet the Annual Uptime Percent[redacted]e a Service Credit as described below.

Definitions

- "Service Year" is the preceding 365 days from the date of an SLA claim.
- "Annual Uptime Percentage" is calculated by subtracting from 100% the percentage of 5 minute periods during the Service Year in which Amazon EC2 was in the state of "Region Unavailable." If you have been using Amazon EC2 for less than 365 days, your Service Year is still the preceding 365 days but any days prior to your use of the service will be deemed to have had 100% Region Availability. Any downtime occurring prior to a successful Service Credit claim cannot be used for future claims. Annual Uptime Percentage measurements exclude downtime resulting directly or indirectly from any Amazon EC2 SLA Exclusion (defined below).
- "Region Unavailable" and "Region Unavailability" means that more than one Availability Zone in which you are running an instance, within the same Region, is "Unavailable" to you.
- "Unavailable" means that all of your running instances have no external connectivity during a five minute period and you are unable to launch replacement instances.
- The "Eligible Credit Period" is a single month, and refers to the monthly billing cycle in which the most recent Region Unavailable event included in the SLA claim occurred.
- A "Service Credit" is a dollar credit, calculated as set forth below, that we may credit back to an eligible Amazon EC2 account.

One Reason: Impossibility of Consensus

- Q: Should Ethan give an A to everyone in CSE 486/586?
  - Input: everyone says either yes/no.
  - Output: an agreement of yes or no.
- Bad news
  - Asynchronous systems cannot guarantee that they will reach consensus with even one faulty process.
- Many consensus problems
  - Reliable, totally-ordered multicast (what we saw already)
  - Mutual exclusion, leader election, *etc.* (what we will see)
  - Cannot reach consensus.

The Consensus Problem

- N processes
- Each process *p* has
  - input variable x_p : initially either 0 or 1
  - output variable y_p : initially *b* (*b*=undecided) – can be changed only once
- Consensus problem: Design a protocol so that either
  - all non-faulty processes set their output variables to 0
  - Or all non-faulty processes set their output variables to 1
  - There is at least one initial state that leads to each possible outcome

Assumptions (System Model)

- The only process failures are *crash-stop*.
- Synchronous systems have bounds on
  - Message delays
  - Max time for each process step
  - *e.g.*, multiprocessor (with common clock across processors)
- Asynchronous systems have no such bounds
  - *E.g.*, the Internet

Example: State Machine Replication

- Run multiple copies of a state machine
- For what?
  - Reliability
- All copies agree on the order of execution.
- Many mission-critical systems operate like this.
  - Air traffic control systems, Warship control systems, *etc.*

First: Synchronous Systems

- Every process starts with an initial input value (0 or 1).
- Every process keeps the history of values received so far.
- The protocol proceeds in rounds.
- At each round, everyone multicasts their history.
- After all the rounds are done, pick the minimum.

First: Synchronous Systems

- Assume that at most *f* processes crash
  - Proceed in *f* + 1 rounds (with timeout)
  - Use basic multicast (B-multicast)
- $Values^r_i$: the set of proposed values known to process $p = P_i$ at the beginning of round r.
- Initially, $Values^0_i = \{\}$; $Values^1_i = \{ v_i = x_p \}$

```
for round r = 1 to f + 1 do:
    B-multicast(Values^r_i)
    Values^(r+1)_i ← Values^r_i
    for each v_j received do:
        Values^(r+1)_i = Values^(r+1)_i ∪ v_j
y_p = d_i = minimum(Values^(f+1)_i)
```

Why Does It Work?

Assume that two non-faulty processes differ in their final set of values. By contradiction:

- Suppose p_i and p_j are these processes.
- Assume that p_i possesses a value v that p_j does not.
- Intuition: p_j must have consistently missed v in all rounds.
  - In the final round, some third process, p_k, sent v to pi, and crashed before sending v to p_j.
  - Any process sending v in the penultimate round must have crashed; otherwise, both p_k and p_j should have received v.
  - Iterating, we infer at least one crash in each preceding round.
  - But we have assumed at most f crashes can occur and there are $f + 1$ rounds → contradiction.

Second: Asynchronous Systems

- Messages have arbitrary delay, processes arbitrarily slow
- Impossible to achieve consensus
  - Even a single failure is enough to prevent the system from reaching consensus!
  - A slow process is indistinguishable from a crashed process
- Impossibility applies to any protocol that claims to solve consensus
- Proved in a now-famous result by Fischer, Lynch and Patterson, 1983 (FLP) [1]
  - Stopped many distributed system designers dead in their tracks
  - A lot of claims of “reliability” vanished overnight!

Are We Doomed?

- Asynchronous systems (*i.e.*, systems with arbitrary delay) cannot guarantee that they will reach consensus with even one faulty process.
- Key word: "guarantee"
  - Does not mean that processes can never reach consensus if one is faulty
  - Allows room for reaching agreement with some probability greater than zero
  - In practice many systems reach consensus.
- How do we get around this?
  - Two key things in the result: one faulty process & arbitrary delay

Techniques to Overcome Impossibility

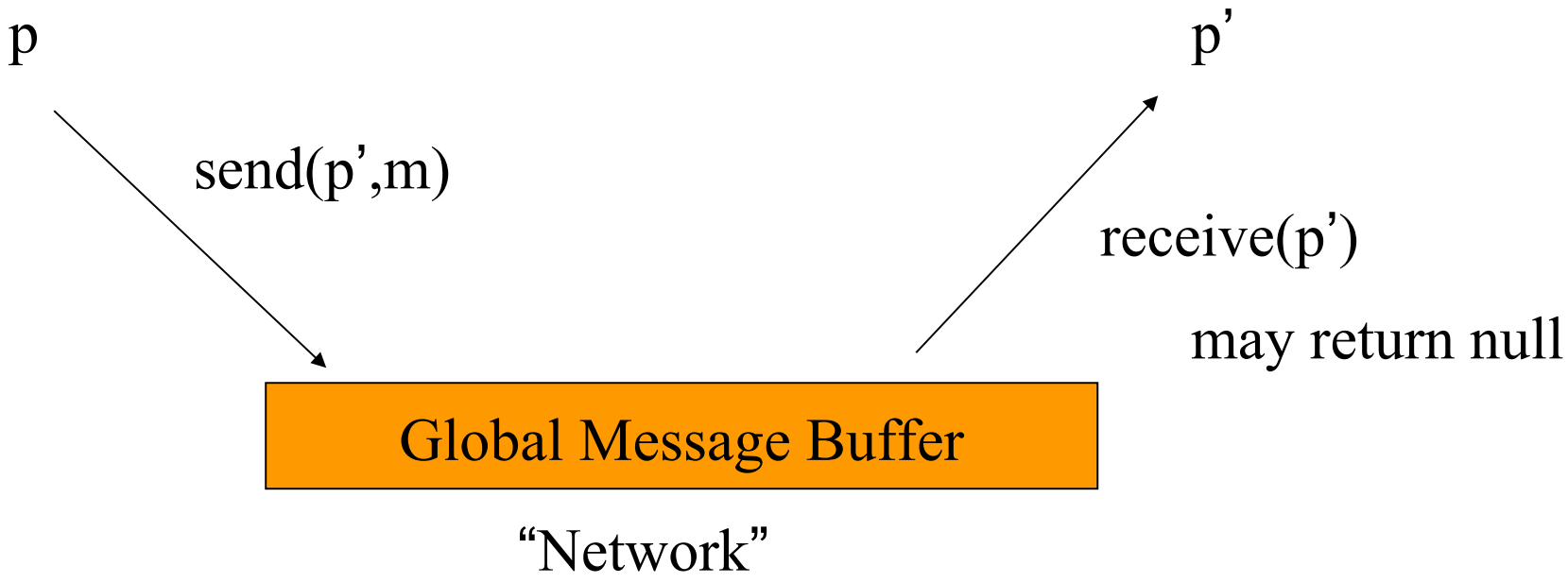
- Technique 1: masking faults (*crash-stop*)
  - For example, use persistent storage and keep local checkpoints
  - Upon failure, restart the process and restore from the last checkpoint.
  - This masks fault, but may introduce arbitrary delays.
- Technique 2: failure detectors
  - For example, if a process is slow, mark it as a failed process.
  - Then actually kill it somehow, or discard all the messages from that point on (*fail-silent*)
  - This effectively turns an asynchronous system into a "synchronous system"
  - Failure detectors might not be 100% accurate and requires a long timeout value to be reasonably accurate.

Recall

- Each process *p* has a state
  - program counter, registers, stack, local variables
  - input register x_p : initially either 0 or 1
  - output register y_p : initially *b* (undecided)
- Consensus Problem: Design a protocol so that either
  - all non-faulty processes set their output variables to 0
  - all non-faulty processes set their output variables to 1
  - (No trivial solutions allowed)

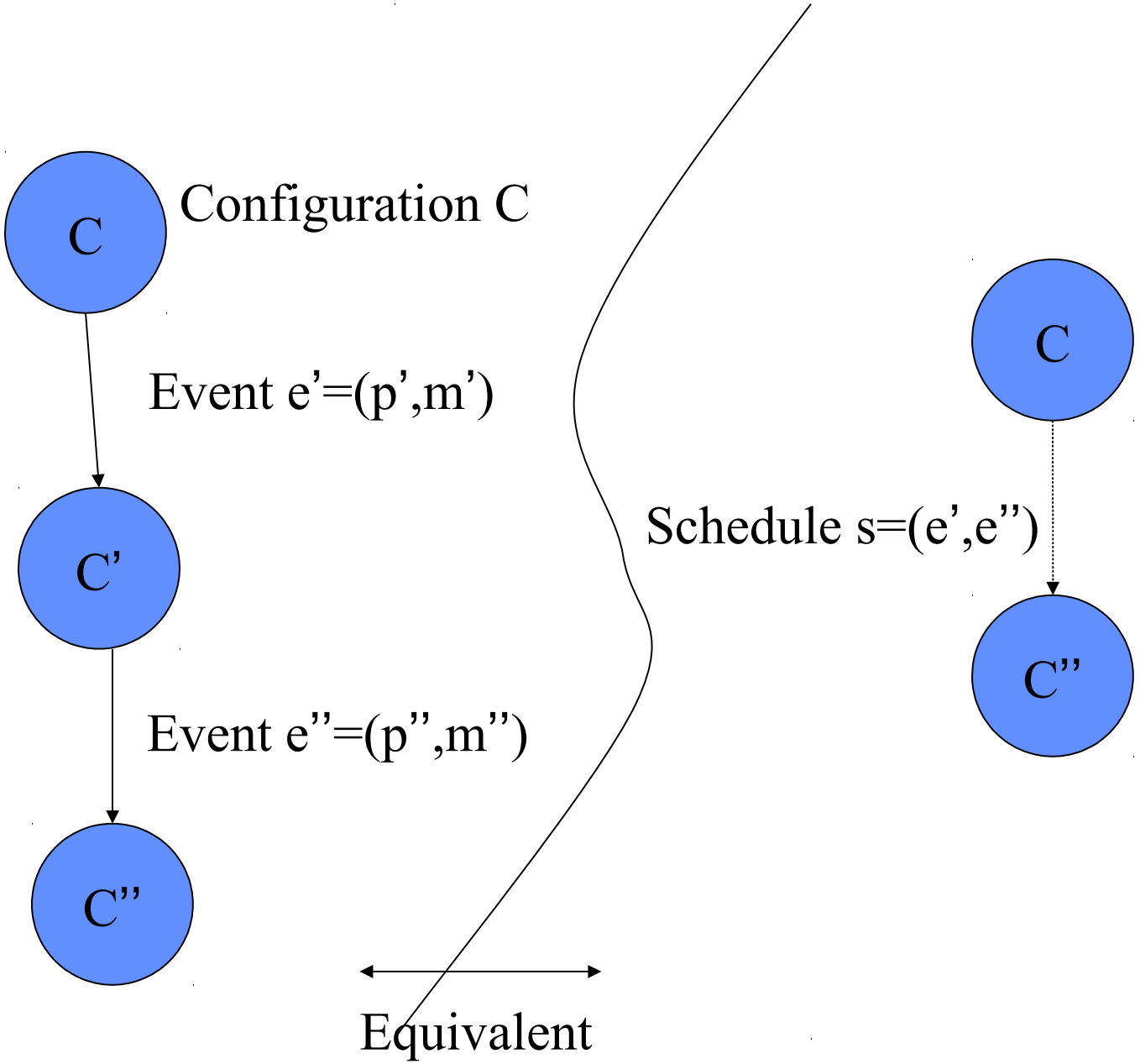
Proof of Impossibility: Reminder

- State machine
  - Forget real time, everything is in steps & state transitions.
  - Equally applicable to a single process or distributed processes
- A state S1 is reachable from another state S0 if there is a sequence of events from S0 to S1.
- There is an initial state with an initial set of input values.


p
p'
send(p',m)
receive(p')
may return null
Global Message Buffer
"Network"

Different Definition of "State"

- State of a process
- Configuration: Global state
  - Collection of states, one per process
  - State of the global buffer
- Each Event is an atomic collection of three sub-steps:
  - receipt of a message by a process (say *p*), and
  - processing of the message, and
  - the sending of all necessary messages by *p*
- Note: this event is different from Lamport events
- Schedule: sequence of events


C
Configuration C
Event e'=(p',m')
C'
Event e''=(p'',m'')
C''
Schedule s=(e',e'')
C
C''
Equivalent

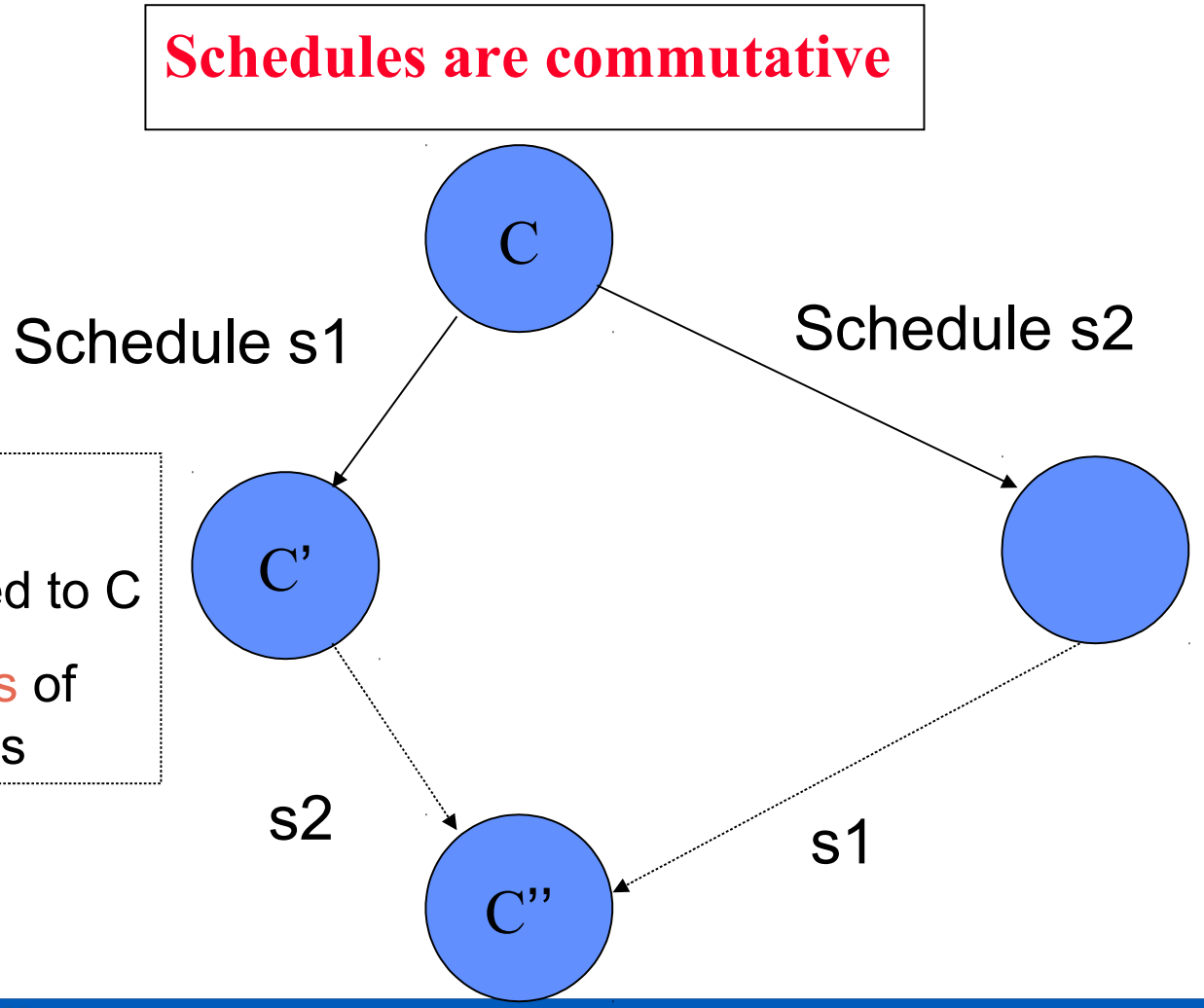
State Valencies

- Let Configuration C have a set of reachable decision values V
  - If |V| = 2, C is bivalent
  - If |V| = 1, C is said to be 0-valent or 1-valent, as appropriate
- Bivalent means that the outcome is unpredictable (but still doesn't mean that consensus is not guaranteed). There are three possibilities:
  - Unanimous 0
  - Unanimous 1
  - Mixture of 0 and 1 values

Guaranteeing Consensus

- If we want to say that a protocol guarantees consensus (with one faulty process and arbitrary delays), we must be able to say the following:
- Consider all possible input sets (*i.e.*, all initial configurations).
- For each input set, the protocol should produce either 0 or 1 even with one failure for all possible execution paths.
  - *i.e.*, no "mixture of 0 and 1 values"
- The impossibility result: We can't do that.
  - *i.e.*, there always exists an execution path that will produce a mixture of values.

Lemma 1



s1 and s2

- can each be applied to C
- Involve disjoint sets of receiving processes

The Theorem

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent configuration, there is always another reachable bivalent configuration
- Insight: It is not possible to distinguish a faulty node from a slow node.
- Theorem (Impossibility of Consensus): There is always an execution path in an asynchronous distributed system (for any algorithm) such that the group of processes never reaches consensus (*i.e.*, always remains bivalent).

Summary

- Consensus: reaching an agreement
- Possible in synchronous systems.
- Asynchronous systems cannot guarantee that they will reach consensus with even one faulty process.

References

[1] Fischer, Lynch, and Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM. Vol. 32 No. 2. April 1985. **Required Reading.** http://groups.csail.mit.edu/tds/papers/Lynch/pods83-flp.pdf

Acknowledgements

- These slides are by Steve Ko, used and (lightly) modified by Ethan Blanton with permission.
- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).