CS 425 / ECE 428
Distributed Systems
Fall 2020

Indranil Gupta (Indy)

Lecture 6: Failure Detection and Membership, Grids

Jokes for this Topic

- (You will get these jokes as you start to understand the topic)
- **Why are neighbors on a ring deep in love? … Because one's heart beats for the other.**
- **The parents of the little gossip-style failure detection algorithm said that raising it was horror story… Because if it did not clean up after itself, one always saw ghosts.**
- **How did the little process know it had failed the exam? Because the teacher gave it a time out.**
- **What is the difference between failure detectors and CS425 Homeworks? Failure detectors must be complete but not accurate, while your solution to a CS425 homework must be accurate but not complete (your solutions have to be correct, but you need only attempt 8 out of 10 questions).**
- **What did the angry Taxi Driver say to the process that was consistently suspecting it? "You talkin' to me? You talkin' to me?" (ok, this joke makes sense only if you've seen the Robert De Niro movie "Taxi Driver")**

Exercises

1. Why is completeness more important to guarantee than accuracy (for failure detectors of crash failures)?
2. If heartbeating has 100% completeness but < 100% accuracy (for asynchronous networks), does pinging have 100% accuracy and 100% completeness?
3. A failure detection algorithm has each process send its heartbeats TO *k* other processes selected at *random (k* << *N,* number of processes in system). Heartbeats are not relayed, but instead recipients do the same action as ring heartbeat protocol recipients. Network is asynchronous.
   1. How many (max) simultaneous failures does this tolerate before it violates (i.e., without it violating) completeness?
   2. Is this 100% accurate?
4. In the previous question, if instead (each) process *i* asked those *k* random processes to send heartbeats to it (instead of the other way around), what would happen to the completeness and accuracy?

Exercises (2)

4. Due to limited resources, the membership list for gossip is only partial at each member. The membership list at each process is selected *uniformly at random* across the entire group and is of size *k* (somehow, the messages take care of it – don't worry about the protocol part). Processes don't fail or join. Each message is gossiped to *m* randomly selected neighbors (from the membership list), where $m < k$, and $m = O(\log(N))$, with the latter needed to ensure spread of gossip. The friend argues that due to random selection, the overall "behavior" of this protocol (in terms of dissemination time of gossips, etc.) is the same as in the case where all processes might have had full membership lists (known everyone in the group), and each gossip was sent to *m* neighbors. Is the friend right? If yes, then give a proof. If no, show why.
5. Explain why T_cleanup is needed in gossip-style heartbeating?
6. How does suspicion work in SWIM?
7. How does time-bounded pinging work in SWIM?
8. Derive the optimality bound equation.

A Challenge

- You’ve been put in charge of a datacenter, and your manager has told you, “Oh no! We don’t have any failures in our datacenter!”

- Do you believe him/her?

- What would be your first responsibility?
- Build a failure detector
- What are some things that could go wrong if you didn’t do this?

Failures are the Norm

… not the exception, in datacenters.

Say, the rate of failure of one machine (OS/disk/motherboard/network, etc.) is once every 10 years (120 months) on average.

When you have 120 servers in the DC, the mean time to failure (MTTF) of the next machine is 1 month.

When you have 12,000 servers in the DC, the MTTF is about once every 7.2 hours!

Soft crashes and failures are even more frequent!

To build a failure detector

- You have a few options

1. Hire 1000 people, each to monitor one machine in the datacenter and report to you when it fails.
2. Write a failure detector program (distributed) that automatically detects failures and reports to your workstation.

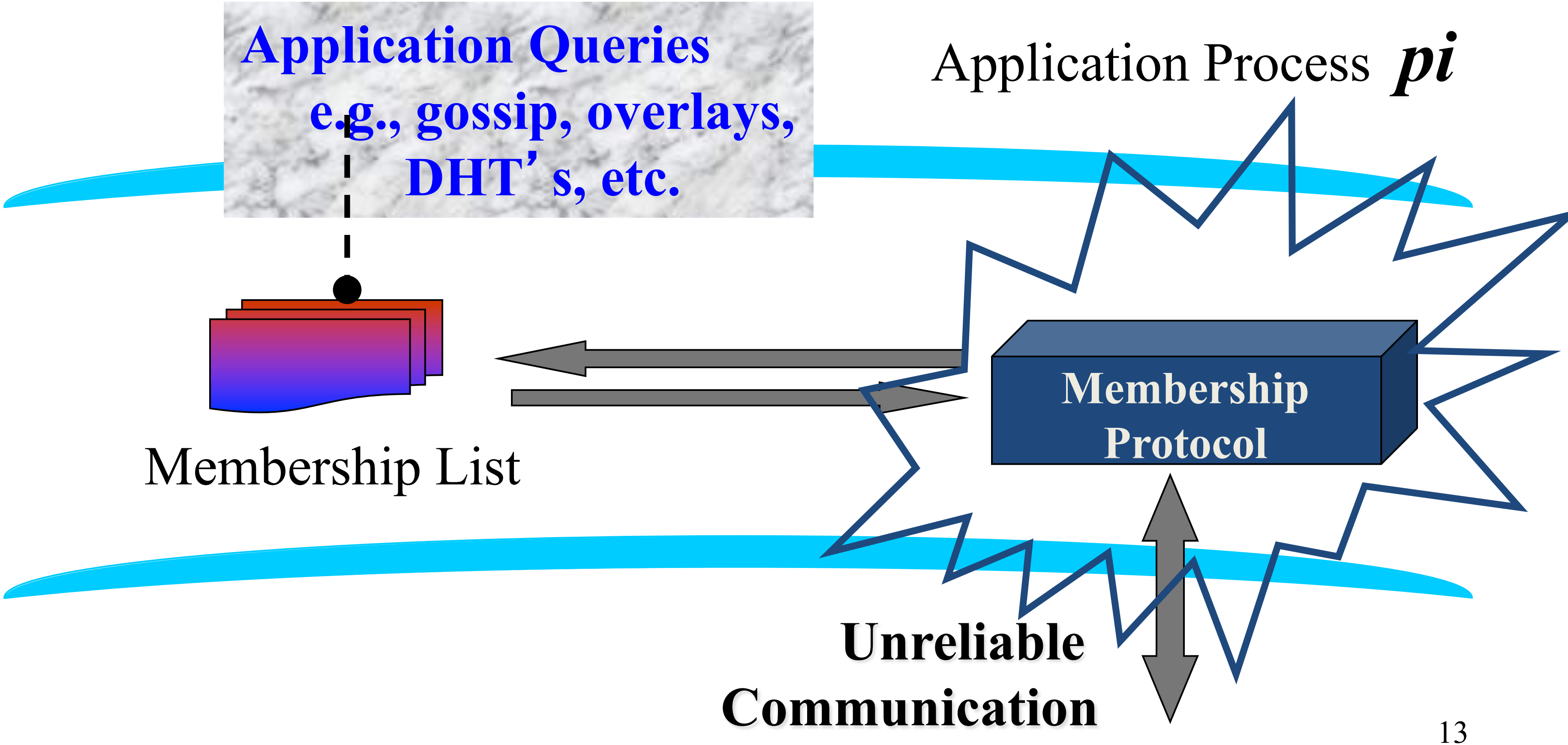
Which is more preferable, and why?

Target Settings

- Process ‘group’-based systems
  - Clouds/Datacenters
  - Replicated servers
  - Distributed databases

- Fail-stop (crash) process failures

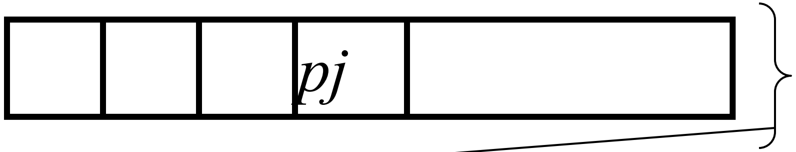
Group Membership Service



Two sub-protocols

Application Process ***pi***

Group Membership List

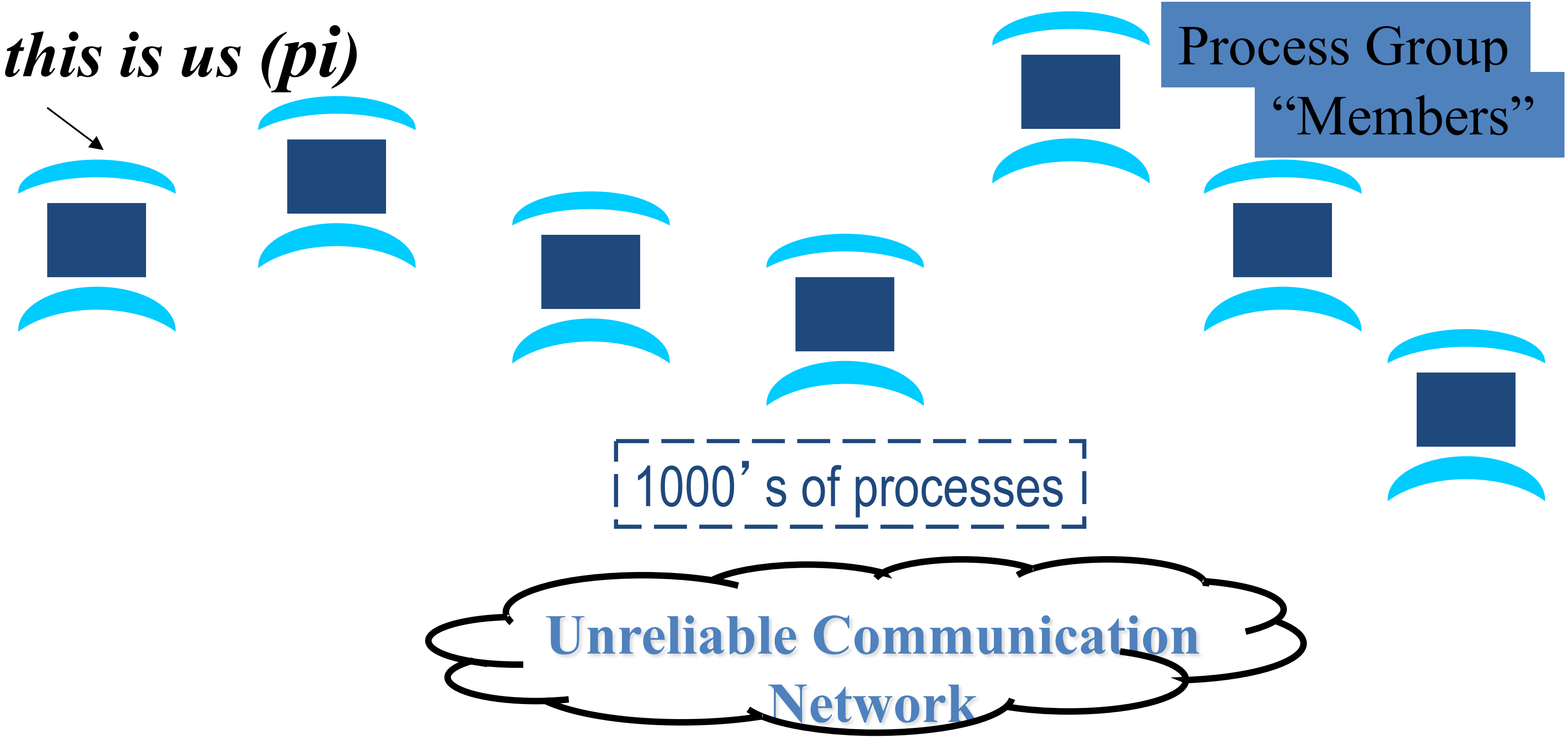


- ***Complete list all the time* (Strongly consistent)**
  - Virtual synchrony
- ***Almost-Complete* list (Weakly consistent)**
  - Gossip-style, SWIM, …
- **Or *Partial-random* list (other systems)**
  - SCAMP, T-MAN, Cyclon,…

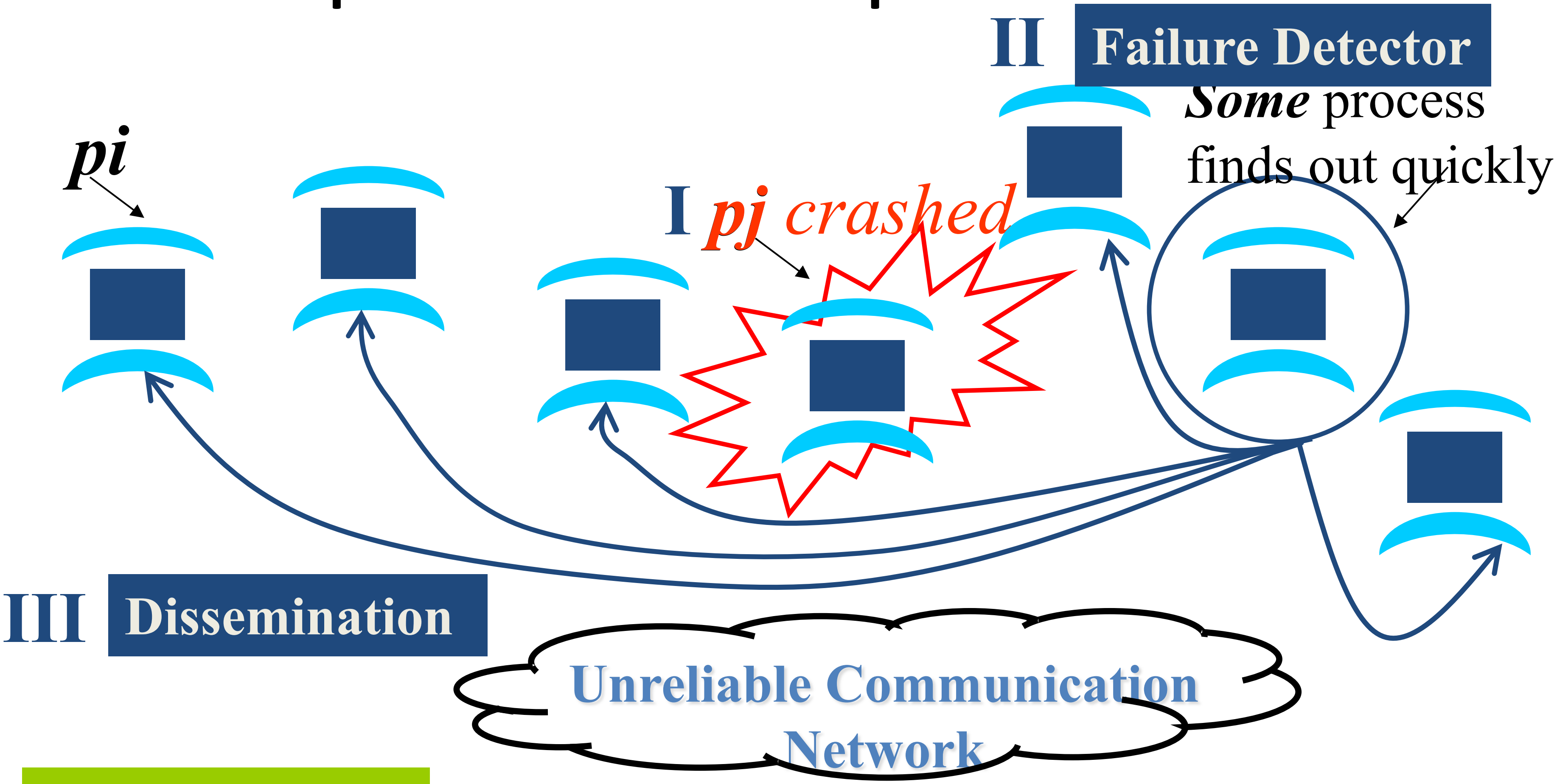
Focus of this series of lecture

Unreliable Communication

Large Group: Scalability A Goal



Group Membership Protocol



Fail-stop Failures only

Next

- How do you design a group membership protocol?

I. *pj* crashes

- Nothing we can do about it!
- A frequent occurrence
- Common case rather than exception
- Frequency goes up linearly with size of datacenter

II. Distributed Failure Detectors: Desirable Properties

- Completeness = each failure is detected
- Accuracy = there is no mistaken detection
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Distributed Failure Detectors: Properties

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failur
- Scale
  - Equal Load on each member
  - Network Message Load

Impossible together in lossy networks [Chandra and Toueg]

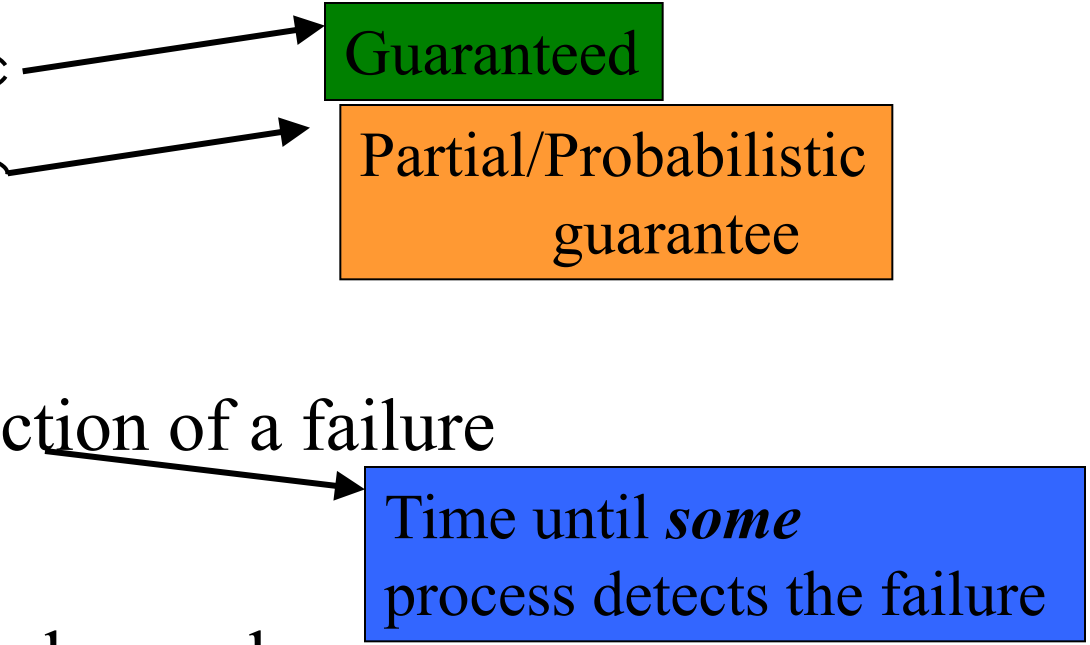
If possible, then can solve consensus! (but consensus is known to be unsolvable in asynchronous systems)

What Real Failure Detectors Prefer

- Completeness → Guaranteed
- Accuracy → Partial/Probabilistic guarantee
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

What Real Failure Detectors Prefer

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load



What Real Failure Detectors Prefer

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee

Time until ***some*** process detects the failure

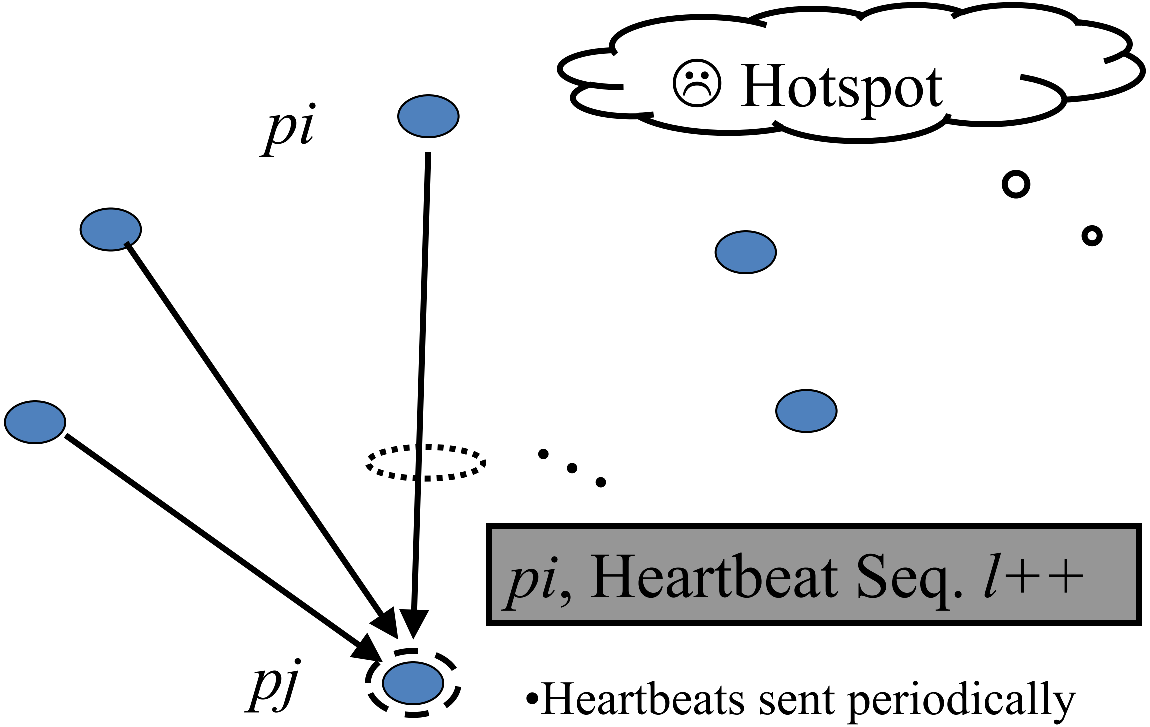
No bottlenecks/single failure point

Failure Detector Properties

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

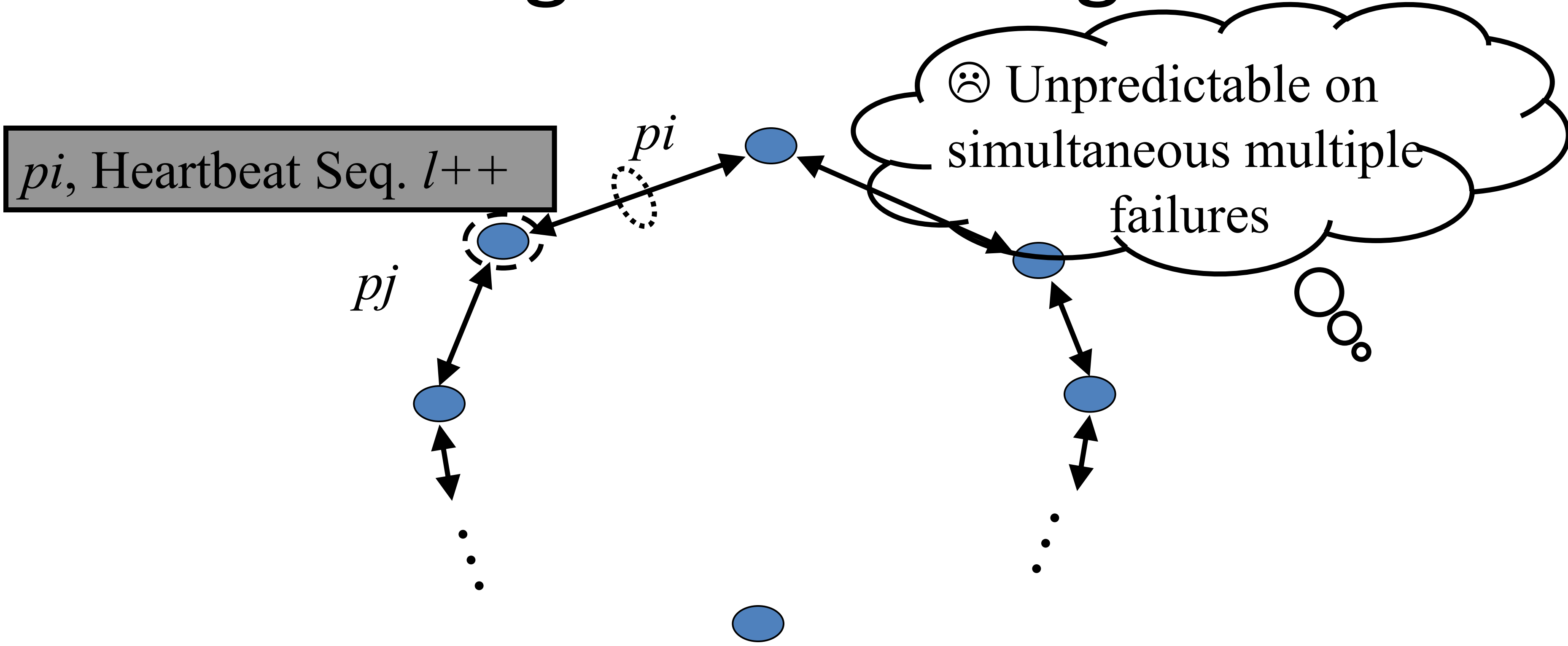
In spite of arbitrary simultaneous process failures

Centralized Heartbeating

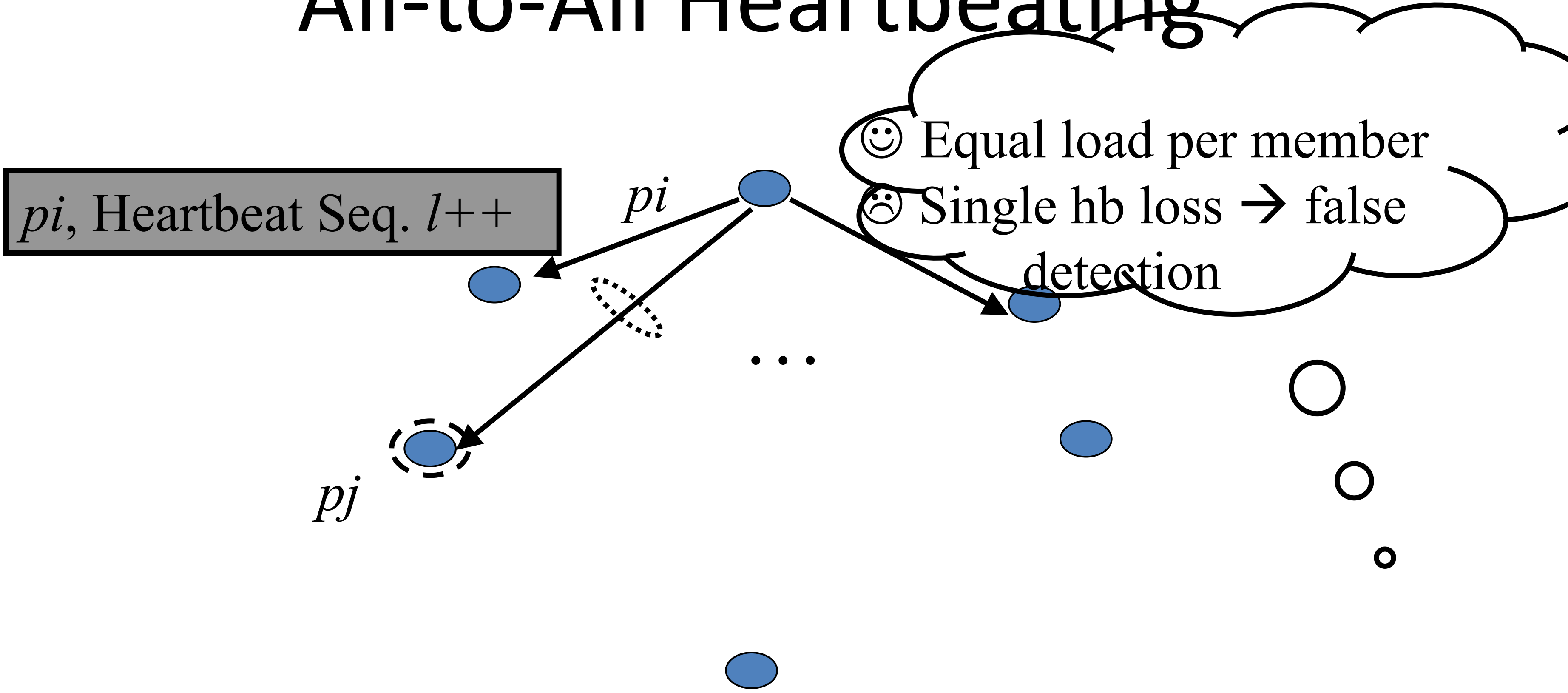


- Heartbeats sent periodically
- If heartbeat not received from *pi* within timeout, mark *pi* as failed

Ring Heartbeating



All-to-All Heartbeating

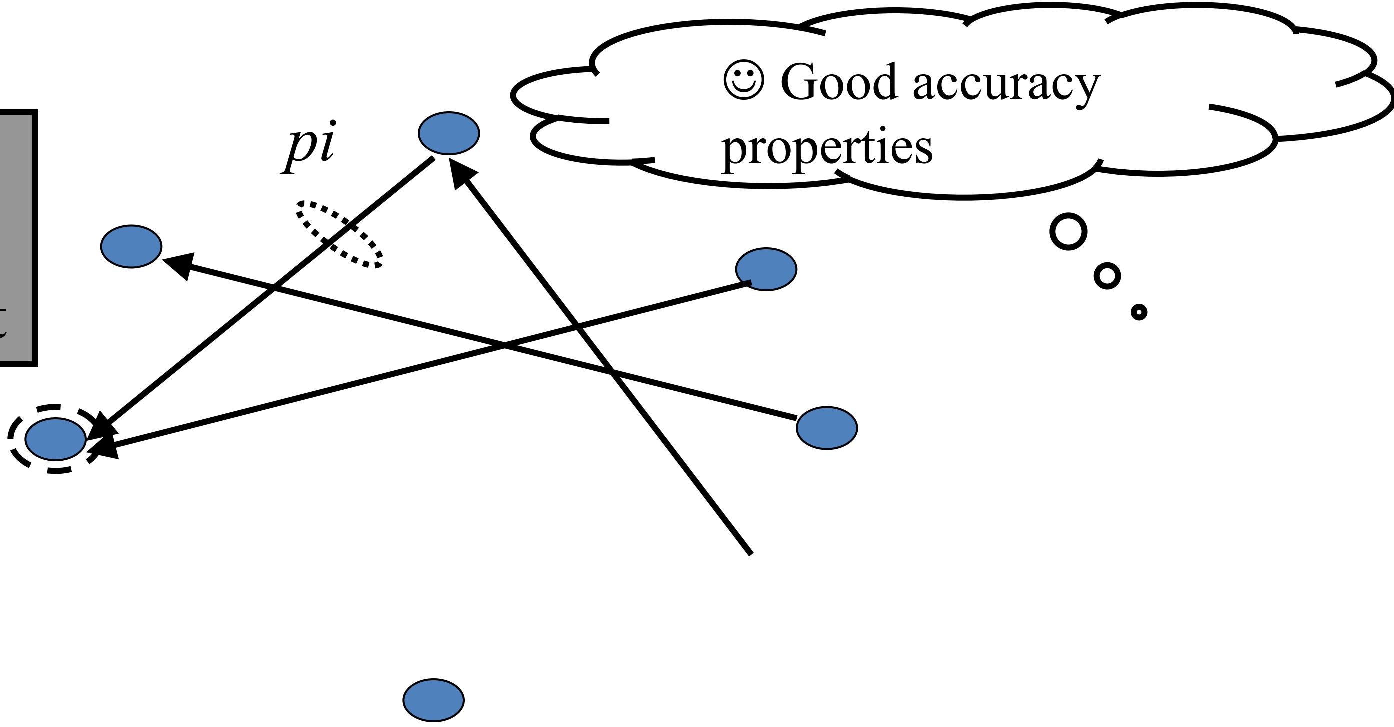


Next

- How do we increase the robustness of all-to-all heartbeating?

Gossip-style Heartbeating

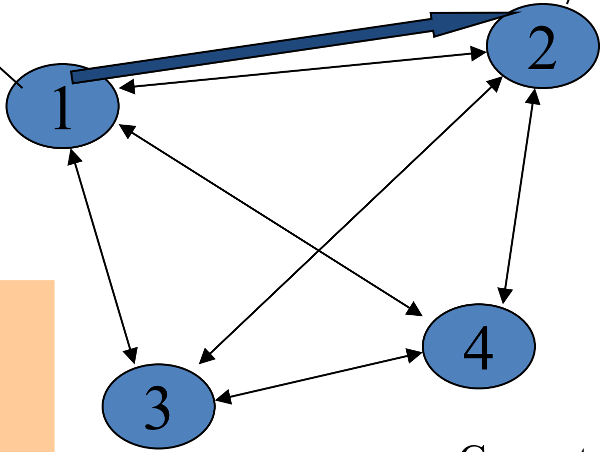
Array of
Heartbeat Seq. *l*
for member subset



Gossip-Style Failure Detection

1	10120	66
2	10103	62
3	10098	63
4	10111	65

Address — Heartbeat Counter — Time (local)



1	10118	64
2	10110	64
3	10090	58
4	10111	65

1	10120	70
2	10110	64
3	10098	70
4	10111	65

Current time : 70 at node 2

(asynchronous clocks)

Protocol:

- Nodes periodically gossip their membership list: pick random nodes, send it list
- On receipt, it is *merged* with local membership list
- When an entry times out, member is marked as failed

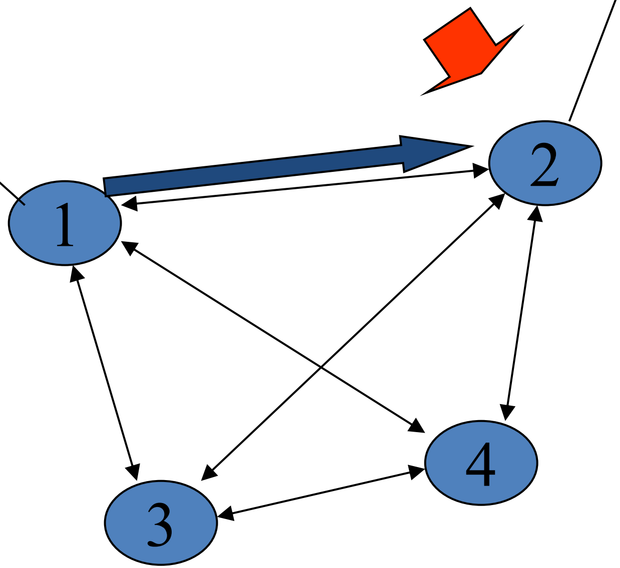
Gossip-Style Failure Detection

- If the heartbeat has not increased for more than T_{fail} seconds,
  the member is considered failed
- And after a further $T_{cleanup}$ seconds, it will delete the member from the list
- Why an additional timeout? Why not delete right away?

Gossip-Style Failure Detection

- What if an entry pointing to a failed node is deleted right after T_{fail} (=24) seconds?

1	10120	66
2	10103	62
3	10098	55
4	10111	65



1	10120	66
2	10110	64
3 / 4	10098 / 10111	75 / 65
4	10111	65

Current time : 75 at node 2

Analysis/Discussion

- Well-known result: a gossip takes O(log(N)) time to propagate.
- So: Given sufficient bandwidth, a single heartbeat takes O(log(N)) time to propagate.
- So: N heartbeats take:
  - O(log(N)) time to propagate, if bandwidth allowed per node is allowed to be O(N)
  - O(N.log(N)) time to propagate, if bandwidth allowed per node is only O(1)
  - What about O(k) bandwidth?
- What happens if gossip period T_{gossip} is decreased?
- What happens to $P_{mistake}$ (false positive rate) as T_{fail} ,$T_{cleanup}$ is increased?
- Tradeoff: False positive rate vs. detection time vs. bandwidth

Next

- So, is this the best we can do? What is the best we can do?

Failure Detector Properties …

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Are application-defined Requirements

- Completeness → Guarantee always
- Accuracy → Probability *PM(T)*
- Speed → *T* time units
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Are application-defined Requirements

- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

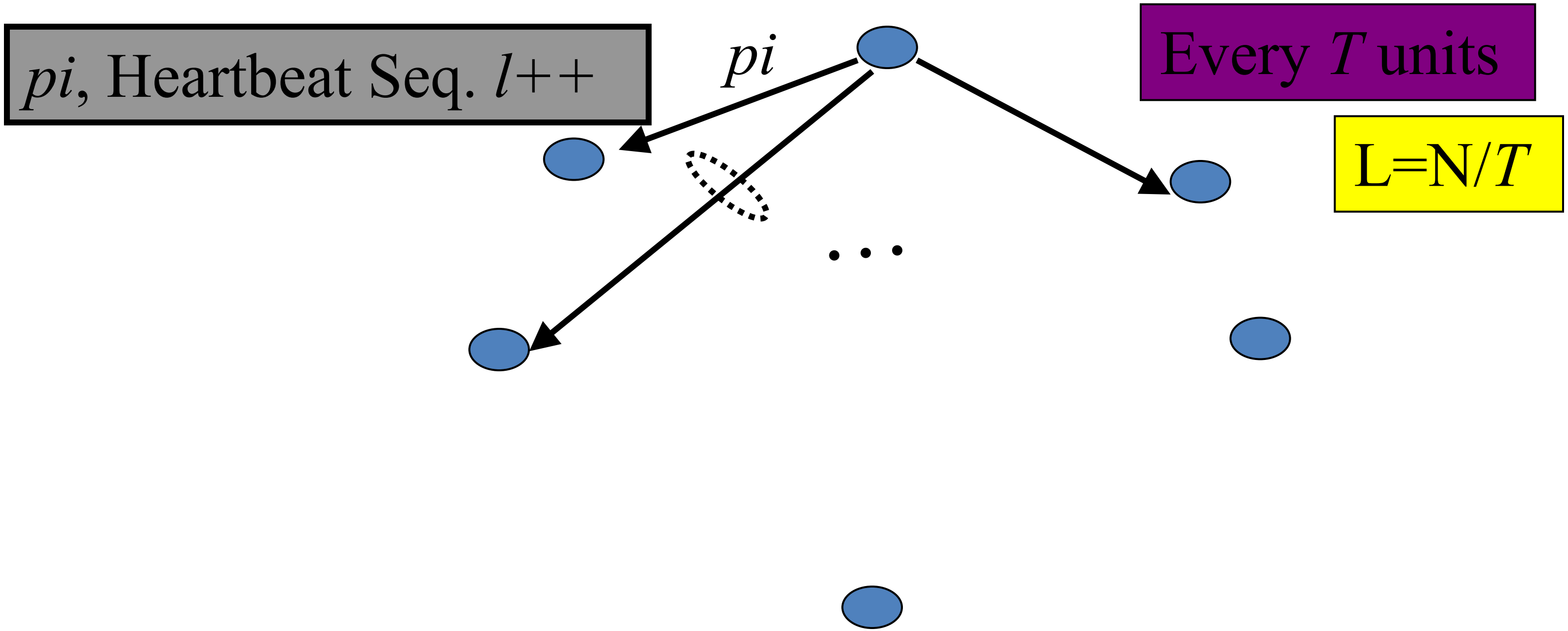
Guarantee always

Probability *PM(T)*

T time units

N*L: Compare this across protocols

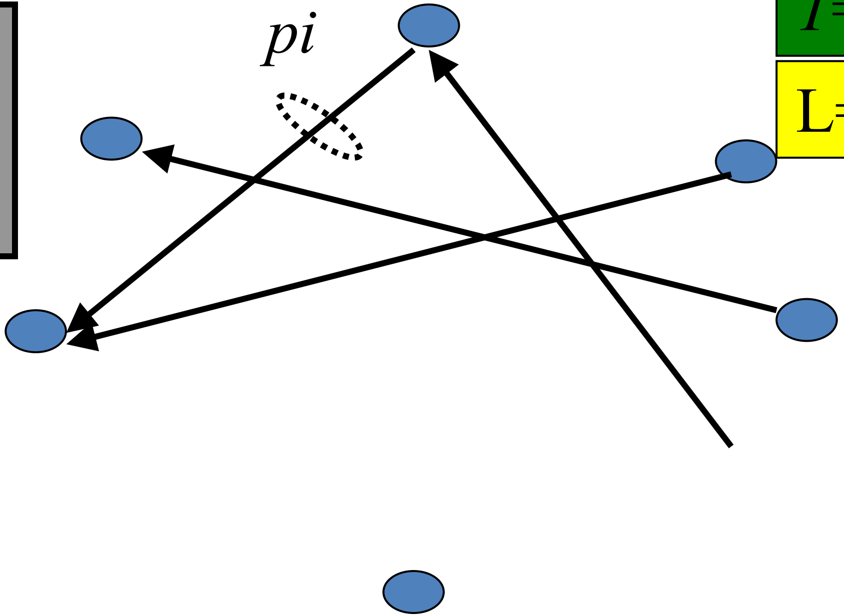
All-to-All Heartbeating



Gossip-style Heartbeating

Array of Heartbeat Seq. *l* for member subset

Every tg units =gossip period, send O(N) gossip message



T=logN * tg

L=N/tg=N*logN/*T*

What's the Best/Optimal we can do?

- *Worst case* load L* per member in the group (messages per second)
  - as a function of *T*, *PM(T)*, N
  - Independent Message Loss probability p_{ml}

- $$L^* = \frac{\log(PM(T))}{\log(p_{ml})} \cdot \frac{1}{T}$$

Heartbeating

- Optimal L is independent of N (!)
- All-to-all and gossip-based: sub-optimal
    - L=O(N/T)
    - try to achieve simultaneous detection at ***all*** processes
    - fail to distinguish *Failure Detection* and *Dissemination* components

➲ Can we reach this bound?

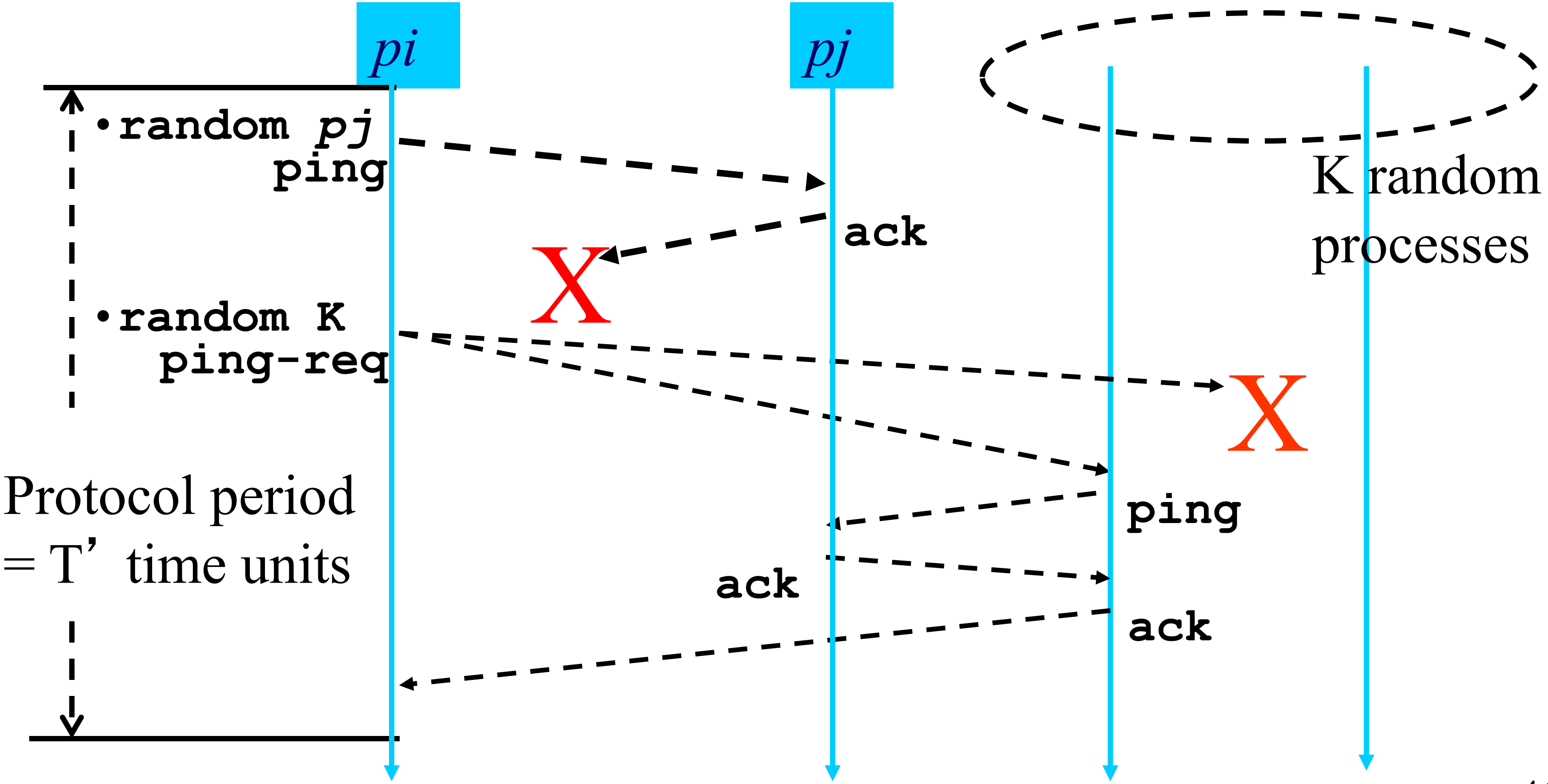
➲ Key:

- Separate the two components
- Use a non heartbeat-based Failure Detection Component

Next

- Is there a better failure detector?

SWIM Failure Detector Protocol



Detection Time

- Prob. of being pinged in T'= $1-(1-\frac{1}{N})^{N-1} = 1-e^{-1}$
- $\mathrm{E}[T] = \mathrm{T'}.\frac{e}{e-1}$
- Completeness: *Any* alive member detects failure
  - Eventually
  - By using a trick: within worst case *O(N)* protocol periods

Accuracy, Load

- *PM(T)* is exponential in -*K*. Also depends on *pml* (and *pf*)
  - See paper
- $\frac{L}{L^*} < 28$ $\frac{E[L]}{L^*} < 8$ for up to 15 % loss rates

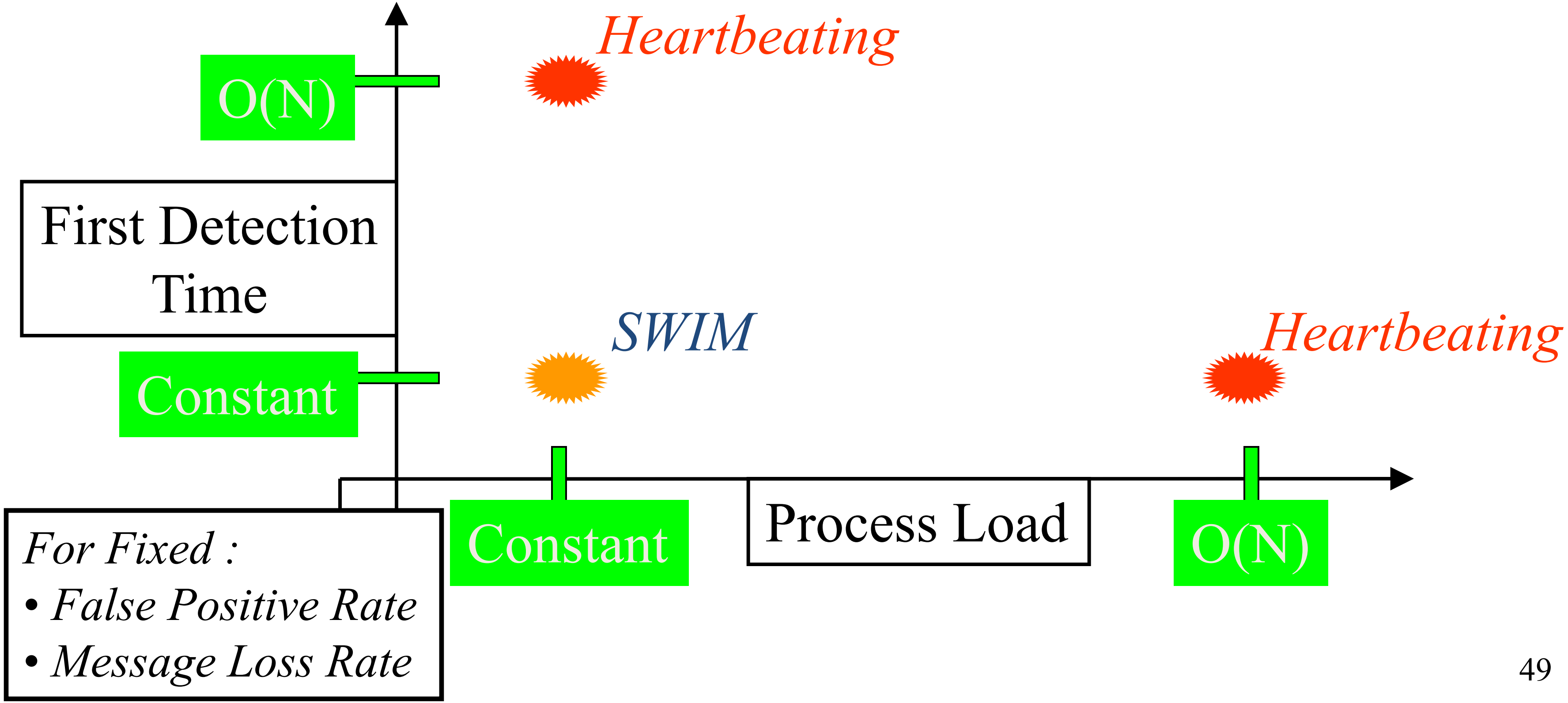
SWIM Failure Detector

Parameter	SWIM
First Detection Time	• Expected $\left[\frac{e}{e-1}\right]$ periods • Constant (independent of group size)
Process Load	• Constant per period • < 8 L* for 15% loss
False Positive Rate	• Tunable (via K) • Falls exponentially as load is scaled
Completeness	• Deterministic time-bounded • Within O(log(N)) periods w.h.p.

Time-bounded Completeness

- Key: select each membership element once as a ping target in a traversal
  - Round-robin pinging
  - Random permutation of list after each traversal
- Each failure is detected in worst case 2N-1 (local) protocol periods
- Preserves FD properties

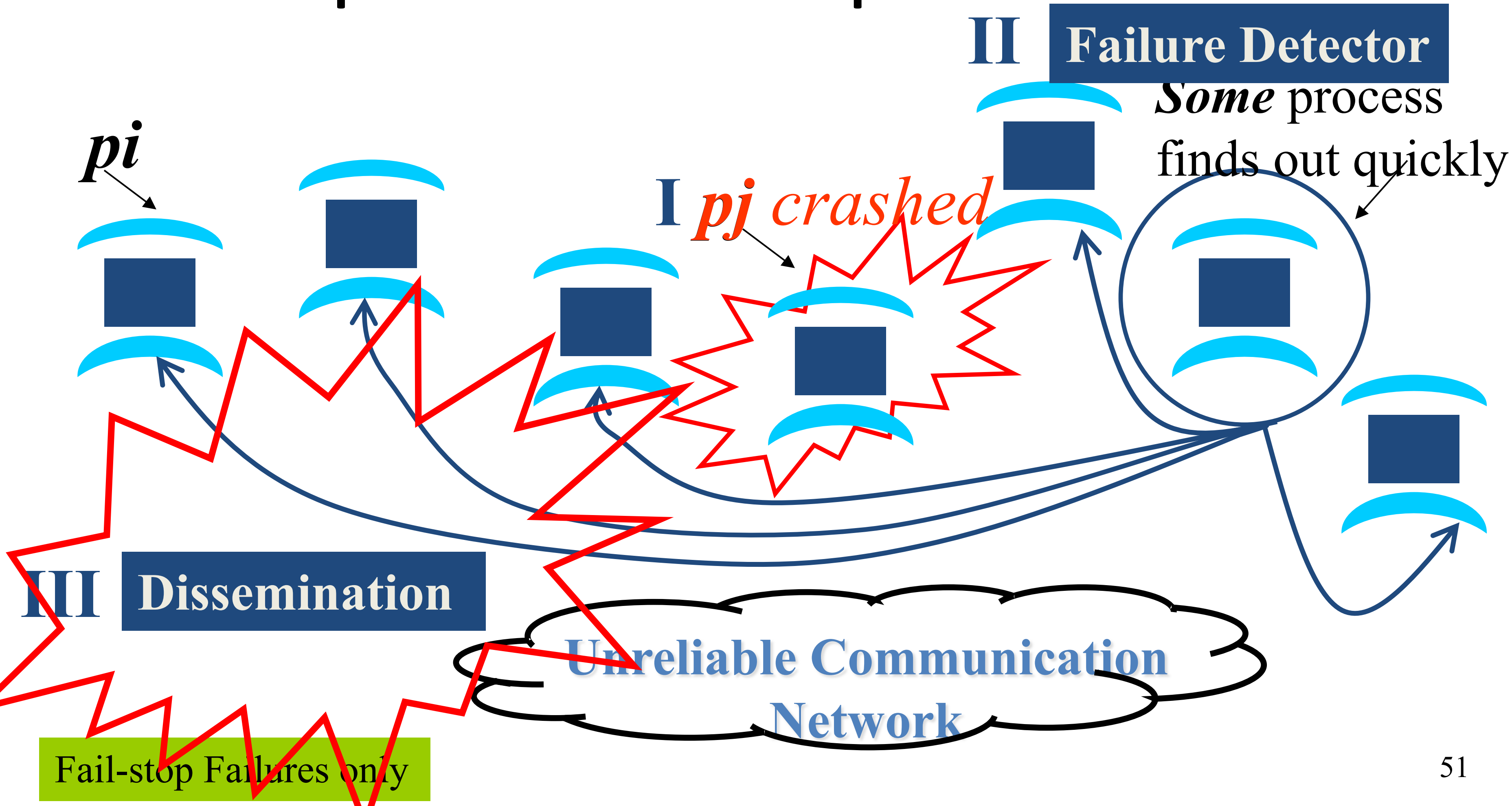
SWIM versus Heartbeating



Next

- How do failure detectors fit into the big picture of a group membership protocol?
- What are the missing blocks?

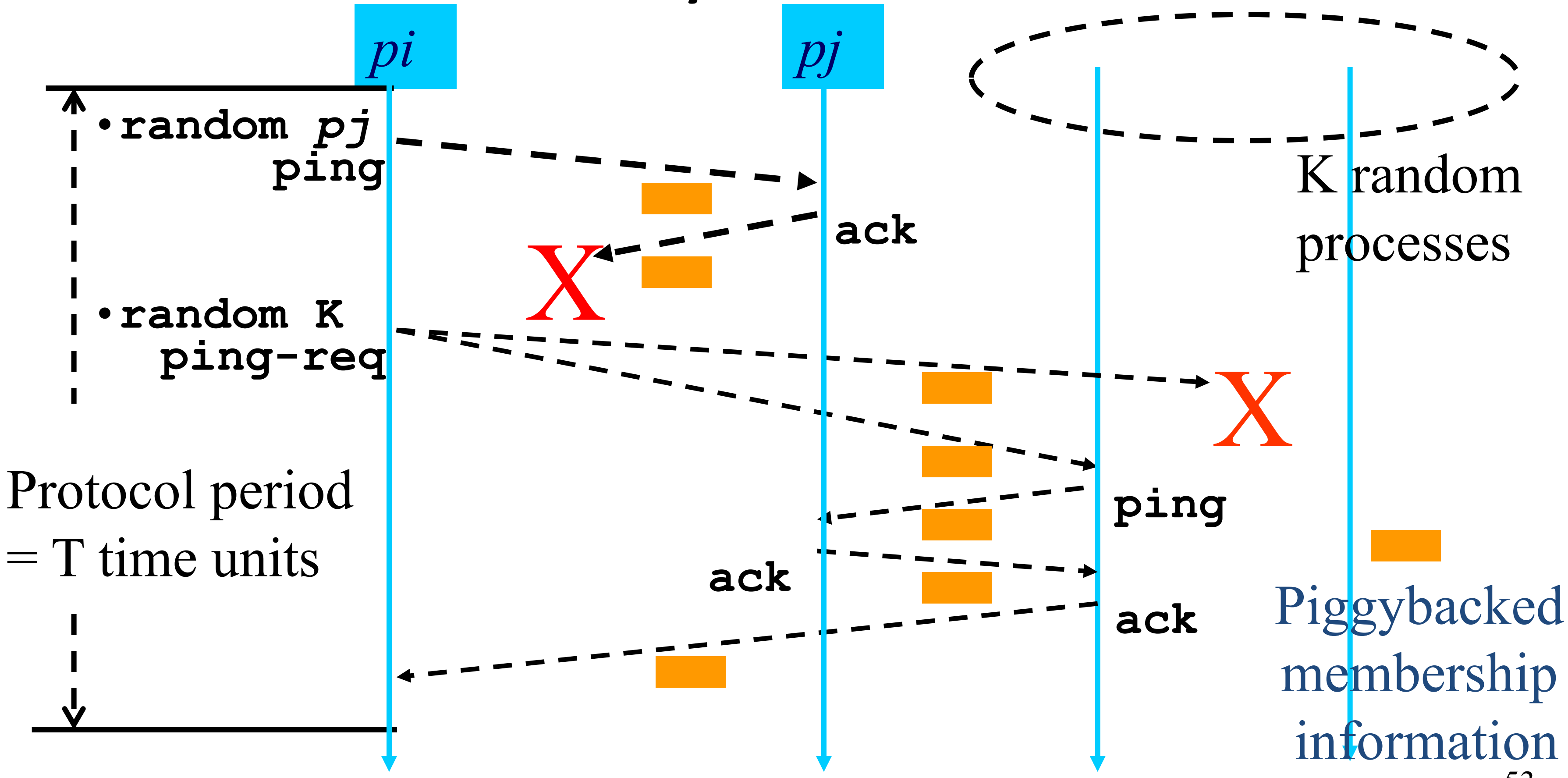
Group Membership Protocol



Dissemination Options

- Multicast (Hardware / IP)
  - unreliable
  - multiple simultaneous multicasts
- Point-to-point (TCP / UDP)
  - expensive
- Zero extra messages: Piggyback on Failure Detector messages
  - Infection-style Dissemination

Infection-style Dissemination



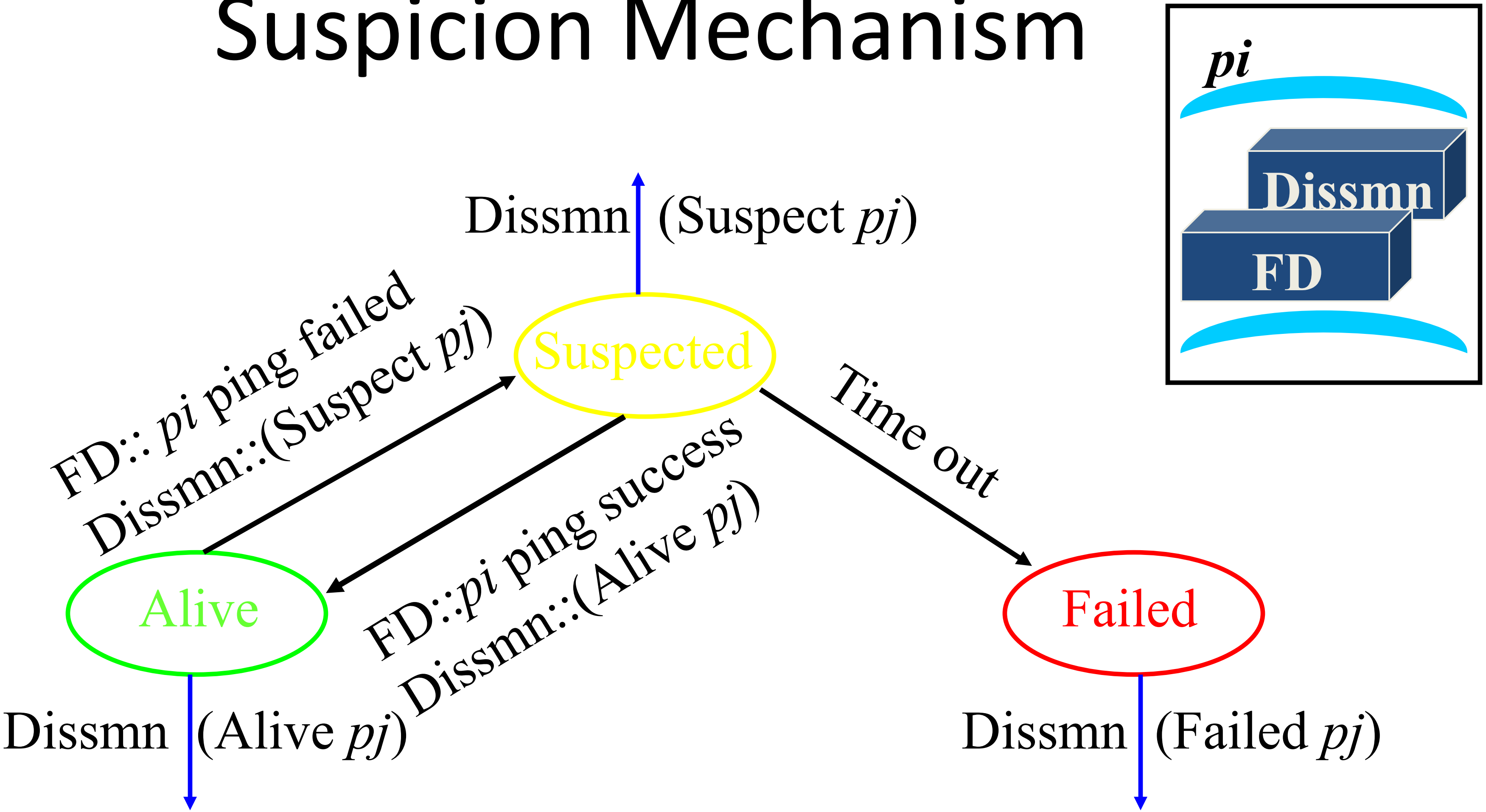
Infection-style Dissemination

- Epidemic/Gossip style dissemination
  - After $\lambda.\log(N)$ protocol periods, $N^{-(2\lambda-2)}$ processes would not have heard about an update
- Maintain a buffer of recently joined/evicted processes
  - Piggyback from this buffer
  - Prefer recent updates
- Buffer elements are garbage collected after a while
  - After $\lambda.\log(N)$ protocol periods, i.e., once they've propagated through the system; this defines weak consistency

Suspicion Mechanism

- False detections, due to
  - Perturbed processes
  - Packet losses, e.g., from congestion
- Indirect pinging may not solve the problem
- Key: *suspect* a process before *declaring* it as failed in the group

Suspicion Mechanism



Suspicion Mechanism

- Distinguish multiple suspicions of a process
  - Per-process *incarnation number*
  - *Inc* # for *pi* can be incremented only by *pi*
    - e.g., when it receives a (Suspect, *pi*) message
  - Somewhat similar to DSDV (routing protocol in ad-hoc nets)
- Higher inc# notifications over-ride lower inc#'s
- Within an inc#: (Suspect inc #) > (Alive, inc #)
- (Failed, inc #) overrides everything else

SWIM In Industry

- First used in Oasis/CoralCDN
- Implemented open-source by Hashicorp Inc.
  - Called "Serf"
  - Later "Consul"
- Today: Uber implemented it, uses it for failure detection in their infrastructure
  - See "ringpop" system

Wrap Up

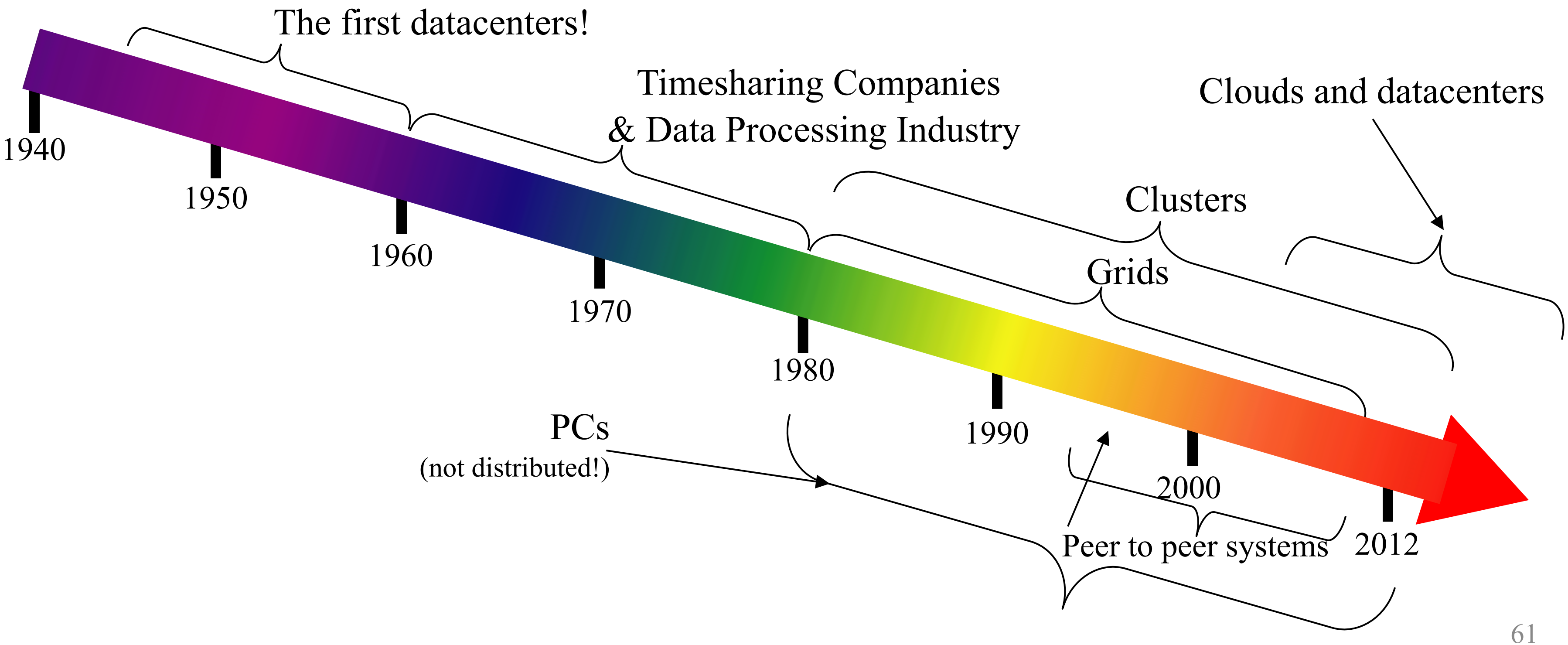
- Failures the norm, not the exception in datacenters
- Every distributed system uses a failure detector
- Many distributed systems use a membership service

- Ring failure detection underlies
  – IBM SP2 and many other similar clusters/machines

- Gossip-style failure detection underlies
  – Amazon EC2/S3 (rumored!)

Grid Computing

"A Cloudy History of Time"

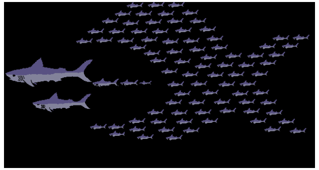


"A Cloudy History of Time"

First large datacenters: ENIAC, ORDVAC, ILLIAC
Many used vacuum tubes and mechanical relays

1940 1950 1960 1970 1980 1990 2000 2012

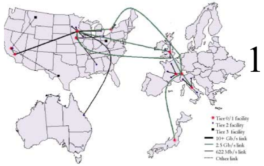
Berkeley NOW Project
Supercomputers
Server Farms (e.g., Oceano)



P2P Systems (90s-00s)
- Many Millions of users
- Many GB per day

Data Processing Industry
- 1968: $70 M. 1978: $3.15 Billion

Timesharing Industry (1975):
- Market Share: Honeywell 34%, IBM 15%,
- Xerox 10%, CDC 10%, DEC 10%, UNIVAC 10%
- Honeywell 6000 & 635, IBM 370/168, Xerox 940 & Sigma 9, DEC PDP-10, UNIVAC 1108



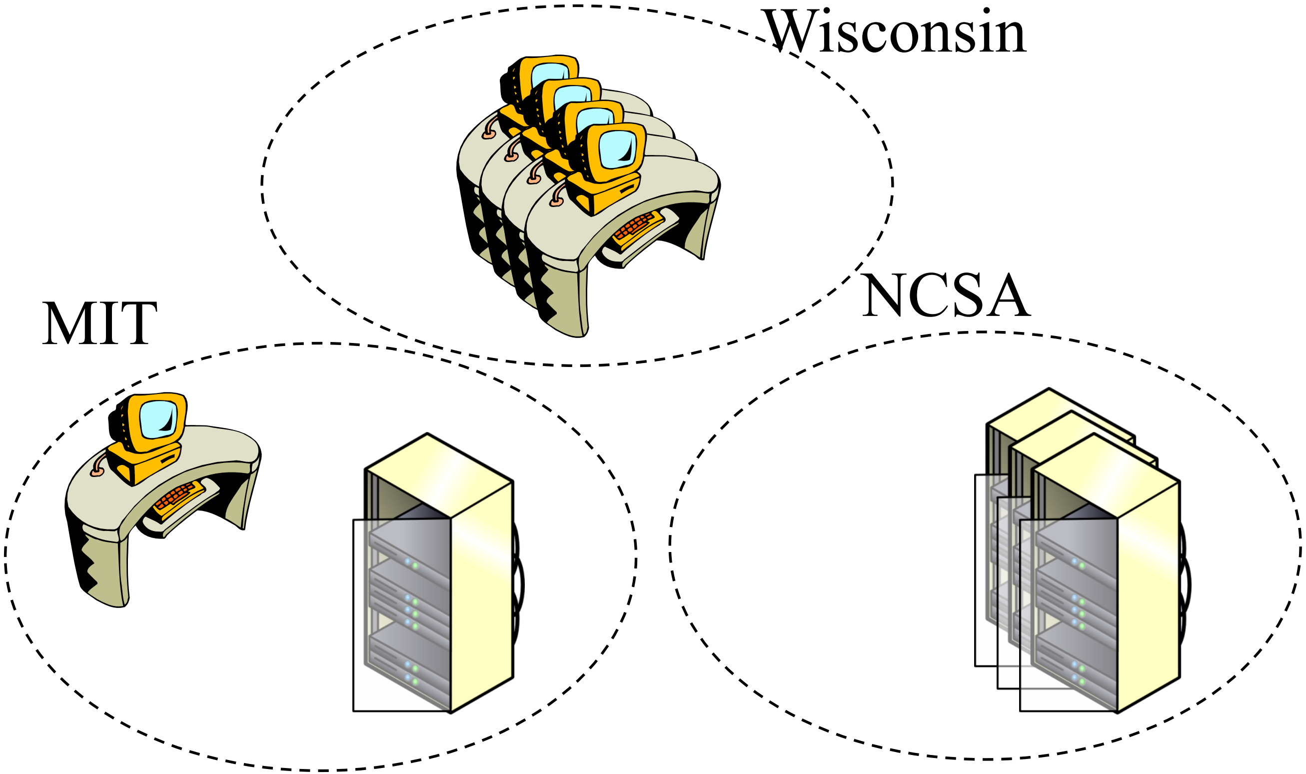
Grids (1980s-2000s):
- GriPhyN (1970s-80s)
- Open Science Grid and Lambda Rail (2000s)
- Globus & other standards (1990s-2000s)

Clouds

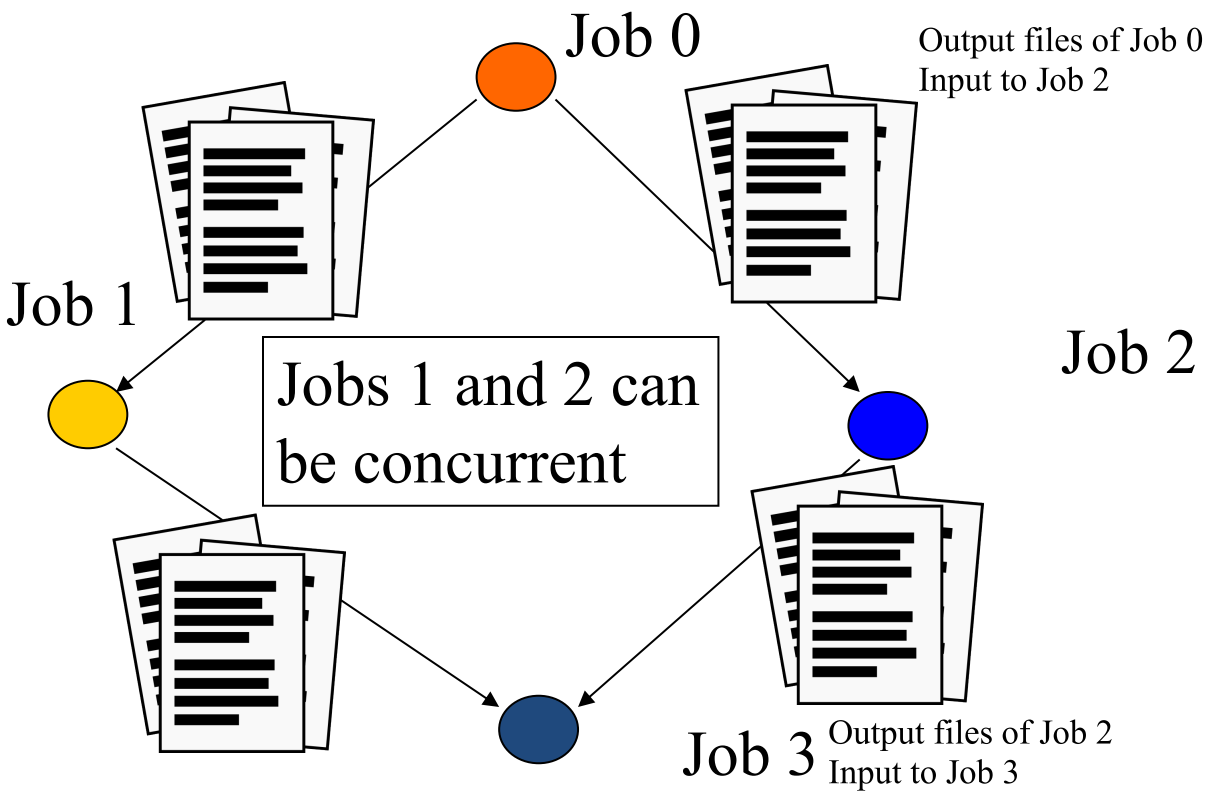
Example: Rapid Atmospheric Modeling System, ColoState U

- Hurricane Georges, 17 days in Sept 1998
  - "RAMS modeled the mesoscale convective complex that dropped so much rain, in good agreement with recorded data"
  - Used 5 km spacing instead of the usual 10 km
  - Ran on 256+ processors
- Computation-intenstive computing (or HPC = high performance computing)
- *Can one run such a program without access to a supercomputer?*

Distributed Computing Resources



An Application Coded by a Physicist



An Application Coded by a Physicist

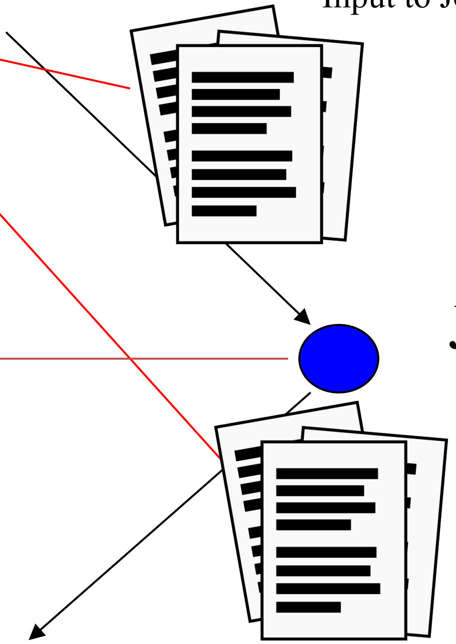
Several GBs

May take several hours/days

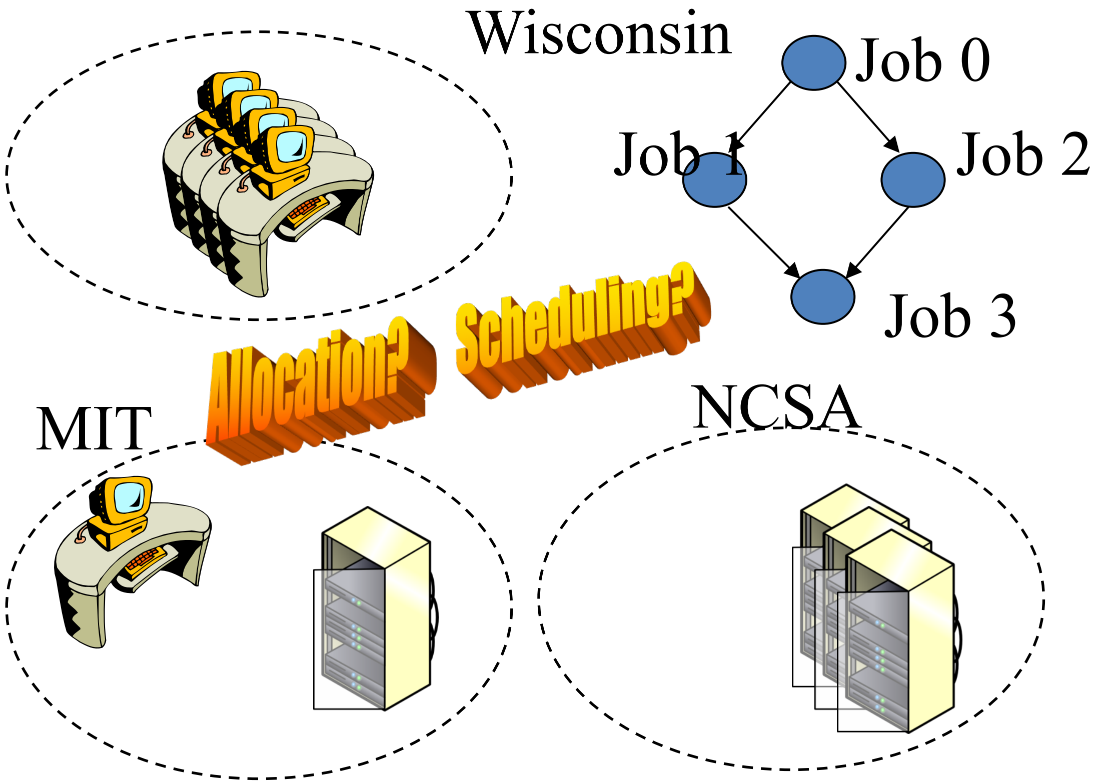
4 stages of a job

- Init
- Stage in
- Execute
- Stage out
- Publish

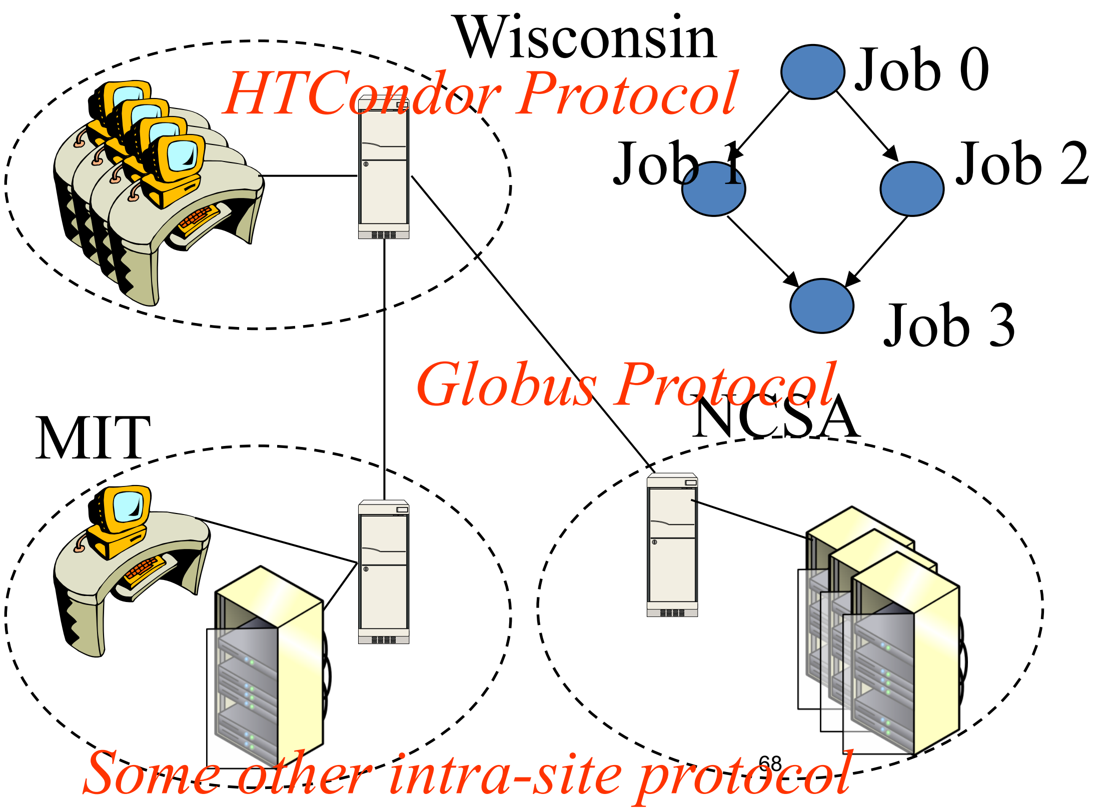
Computation Intensive,
so Massively Parallel



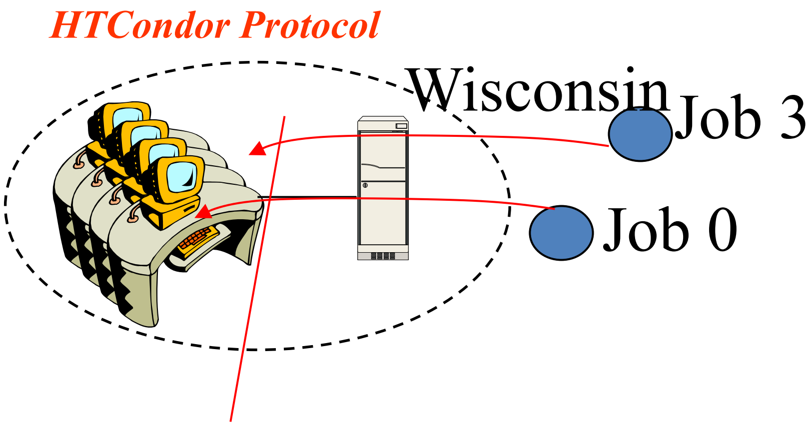
Scheduling Problem



2-level Scheduling Infrastructure



Intra-site Protocol



Internal Allocation & Scheduling
Monitoring
Distribution and Publishing of Files

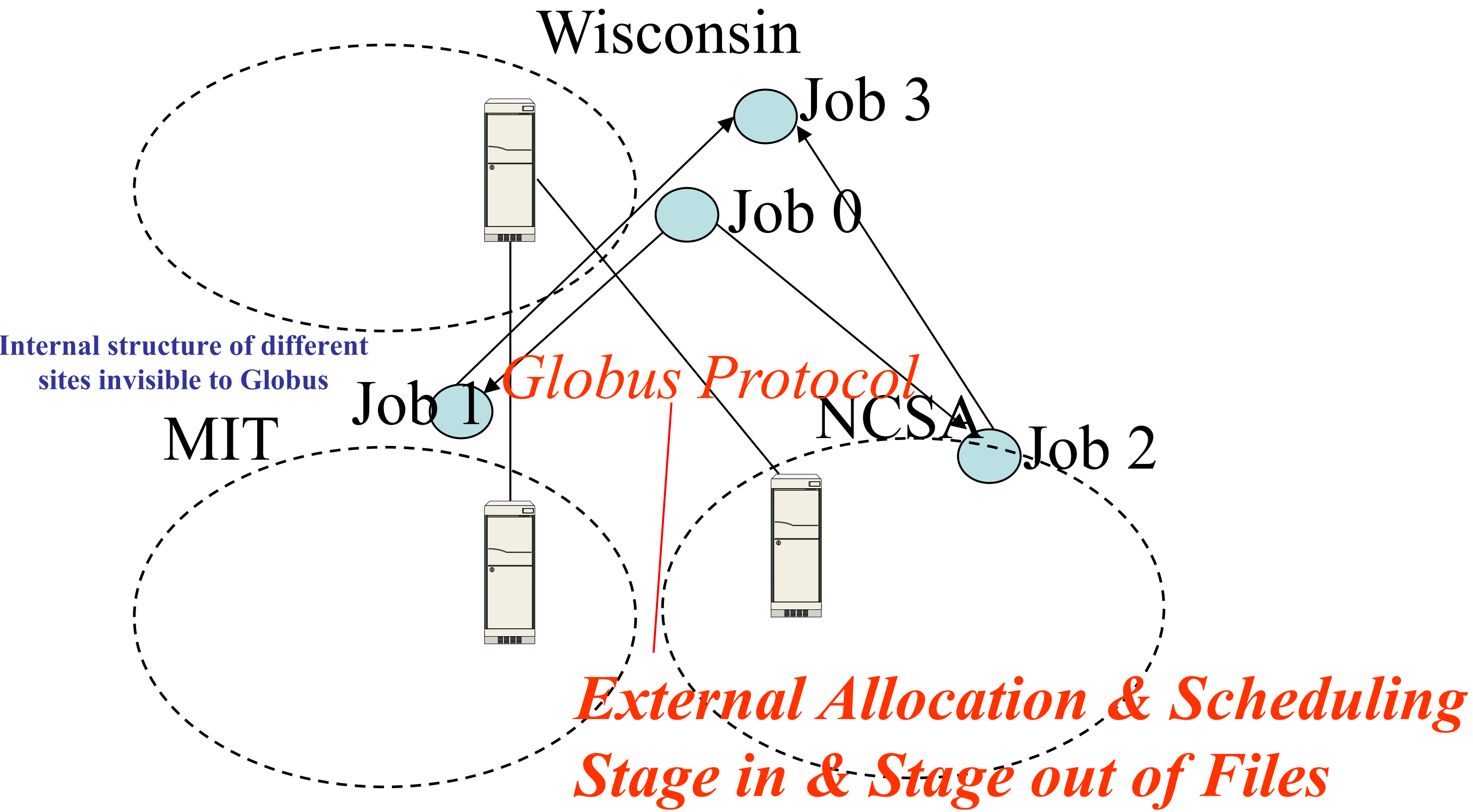
Condor (now HTCondor)

- High-throughput computing system from U. Wisconsin Madison
- Belongs to a class of "Cycle-scavenging" systems
  - SETI@Home and Folding@Home are other systems in this category

Such systems

- Run on a lot of workstations
- When workstation is free, ask site's central server (or Globus) for tasks
- If user hits a keystroke or mouse click, stop task
  - Either kill task or ask server to reschedule task
- Can also run on dedicated machines

Inter-site Protocol



Globus

- Globus Alliance involves universities, national US research labs, and some companies
- Standardized several things, especially software tools
- Separately, but related: Open Grid Forum
- Globus Alliance has developed the Globus Toolkit

http://toolkit.globus.org/toolkit/

Globus Toolkit

- Open-source
- Consists of several components
  - GridFTP: Wide-area transfer of bulk data
  - GRAM5 (Grid Resource Allocation Manager): submit, locate, cancel, and manage jobs
    - Not a scheduler
    - Globus communicates with the schedulers in intra-site protocols like HTCondor or Portable Batch System (PBS)
  - RLS (Replica Location Service): Naming service that translates from a file/dir name to a target location (or another file/dir name)
  - Libraries like XIO to provide a standard API for all Grid IO functionalities
  - Grid Security Infrastructure (GSI)

Security Issues

- Important in Grids because they are *federated,* i.e., no single entity controls the entire infrastructure

- Single sign-on: collective job set should require once-only user authentication
- Mapping to local security mechanisms: some sites use Kerberos, others using Unix
- Delegation: credentials to access resources inherited by subcomputations, e.g., job 0 to job 1
- Community authorization: e.g., third-party authentication

- These are also important in clouds, but less so because clouds are typically run under a central control
- In clouds the focus is on failures, scale, on-demand nature

Summary

- Grid computing focuses on computation-intensive computing (HPC)
- Though often federated, architecture and key concepts have a lot in common with that of clouds
- Are Grids/HPC converging towards clouds?
  - E.g., Compare OpenStack and Globus