Failure Detectors and Distributed Complexity Theory

Petr Kuznetsov

TU Berlin/Deutsche Telekom Laboratories

Jan 27th, 2009

BIRS workshop on Lower Bounds in Distributed Computing

Thinking “distributed” is hard

- Not natural
- Multitude of abstractions and models
- (Almost) no categorization/complexity classes

A theory of *distributed* computational complexity?

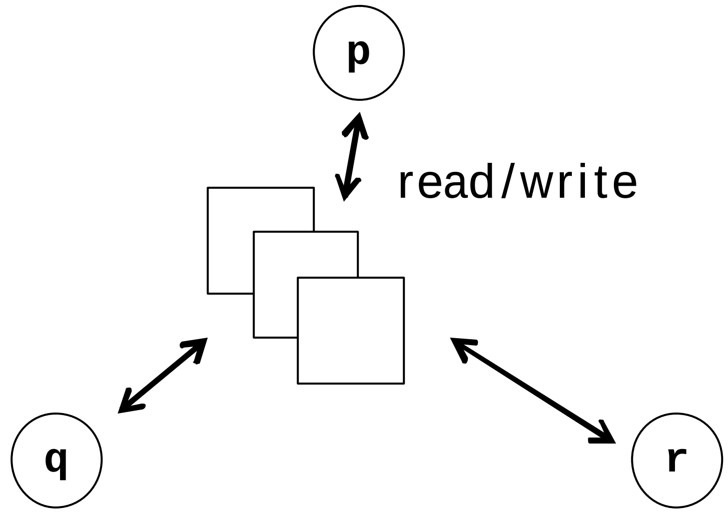
Blind Men and the Elephant

This talk

- Propose a hierarchy for *distributed tasks* based on *synchrony assumptions*
- Characterize the levels 1 and N: the classes of *universal* and the *easiest non-trivial* tasks
- Speculations on the complete characterization

Asynchronous RW memory system

- A set of *processes* communicate by reading and writing into shared memory
- Processes can fail by crashing
- No timing assumptions, i.e., no bounds on relative process speeds



The synchrony gap

Many fundamental problems are not solvable in asynchronous systems

- ✓Consensus [FLP85, LA87]
- ✓ Set agreement [HS93, SZ93, BG93]

But they *are* solvable in *synchronous* systems, where bounds on processing are known *a priori*

Modeling synchrony

- Explicit bounds on communication and relative processing speed [DDS86]
  - ✓Too coarse-grained

- Failure detectors [CHT96]
  - ✓An oracle providing hints on failure pattern: on where and when failures occurred
  - ✓Formally: FD D is a map from *failure pattern* to a set of failure detector histories

Failure detectors [CHT96]

- Distributed oracles providing hints on where and when failures occurred

- Formally: FD is a map from *failure patterns* to sets of *failure detector histories*

Failure detectors: examples

- *Perfect* P [CT96]

  Outputs a list of suspected processes

  ✓ No process is suspected before it fails

  ✓ Eventually, all faulty processes are always suspected

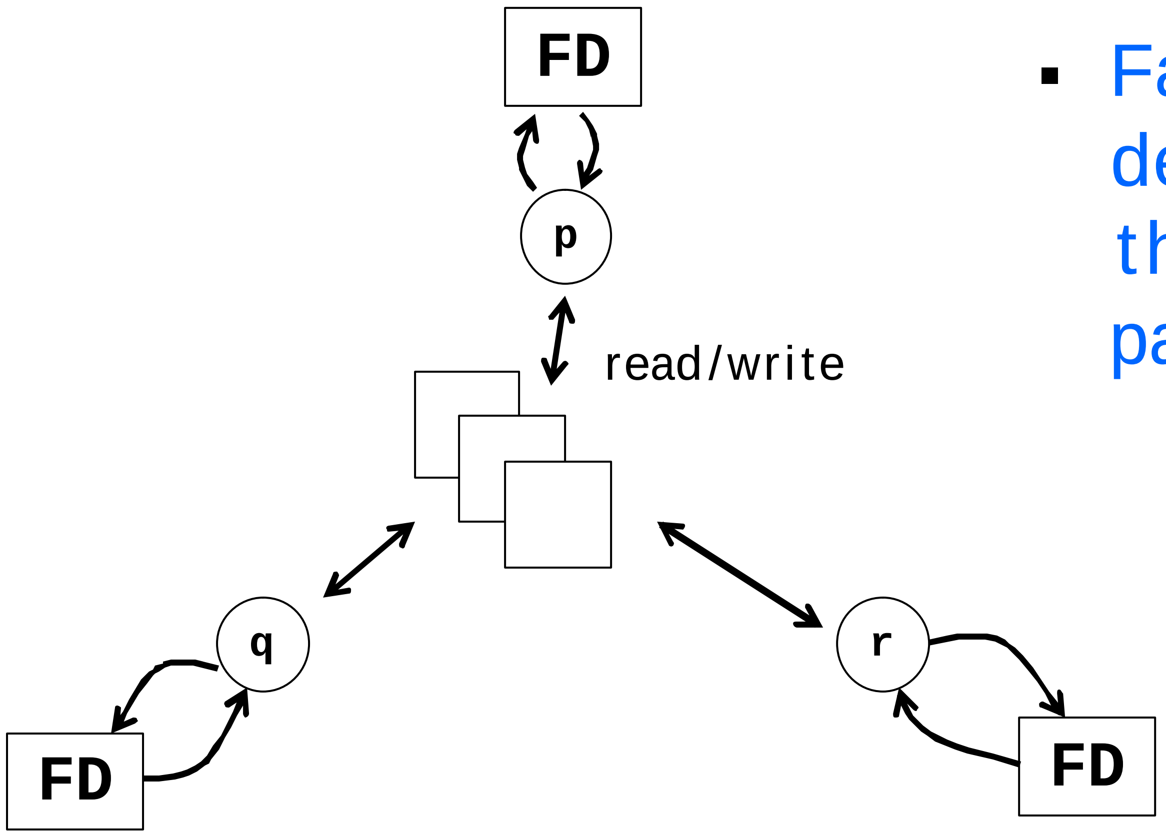
- *Eventual leader* Ω [CHT96]

  Outputs a process identifier

  ✓ Eventually, the *same correct* process is always output

Model

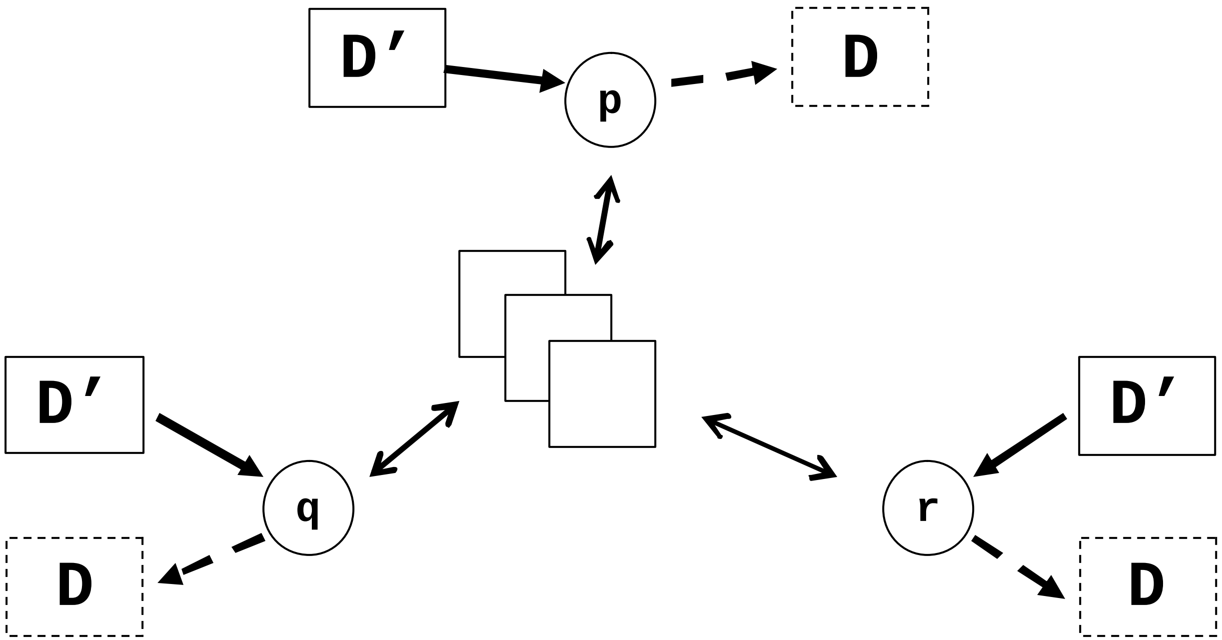
Read-write shared-memory system with failure detectors [CHT96]



- Failure detectors depend only on the failure pattern

Comparing failure detectors

D is *weaker* than D′ if there exists an algorithm that *emulates* D using D′



O is weaker than P:

Output the smallest non-suspected process

The weakest failure detector

D is *the weakest failure detector to solve problem* M if and only if:

- ✓D is *sufficient:* can be used to solve M
- ✓D is *necessary:* weaker than *any* failure detector D' that can be used to solve M

An idea

Classify distributed abstractions based on the minimal synchrony assumptions to solve them (WFD)

An idea

Classify distributed tasks based on the minimal synchrony assumptions to solve them (WFD)

Distributed tasks (I,O,?)

- I – set of input vectors
- O – set of output vectors
- Task specification ? : I? 2^{O} –relation between I and O

k-set agreement

- Each process in $\{P_0,\ldots P_N\}$ *decides* on a value in $\{0,\ldots,N\}$
- Not more than k distinct values are decided
- If i is decided, then Pi *participated*

✓Equivalent to k parallel consensuses, at least one returns [GRRT06]

✓k=1: Consensus [FLP85]

✓k<N+1 not solvable in asynchronous systems [HS93,BG93,SZ93]

✓k=N+1 trivially solvable

A synchrony-based hierarchy of (N+1)-process tasks

Level	Task		
N	Consensus	—	*universal* tasks [Her91,CHT96,DFG05] - O
N-1	2-set agreement	—	?
⋮	⋮		
2	(N-1)-set agreement	—	?
1	N-set agreement	—	The easiest unsolvable task [Zie08] - anti-O
0	(N+1)-set agreement, (2N+1)-renaming	←	Trivial tasks [HS94,BG97] no synchrony assumptions

Outline

- Level N: the WFD for consensus
- Level 1: the easiest nontrivial task
- Levels 2 - N-1: the easiest k-resilient impossible task?

Level N: WFD for consensus [CHT96]

- Eventual leader FD (Ω) is sufficient for solving consensus [DDS87,Lam90,CT96]

- Showing the necessity of Ω:
  - ✓Let D be any FD sufficient to solve consensus
  - ✓Let A be the corresponding algorithm

  - ✓Present a *reduction algorithm* that extracts Ω, given D and A

Reduction algorithm

Two parallel threads:

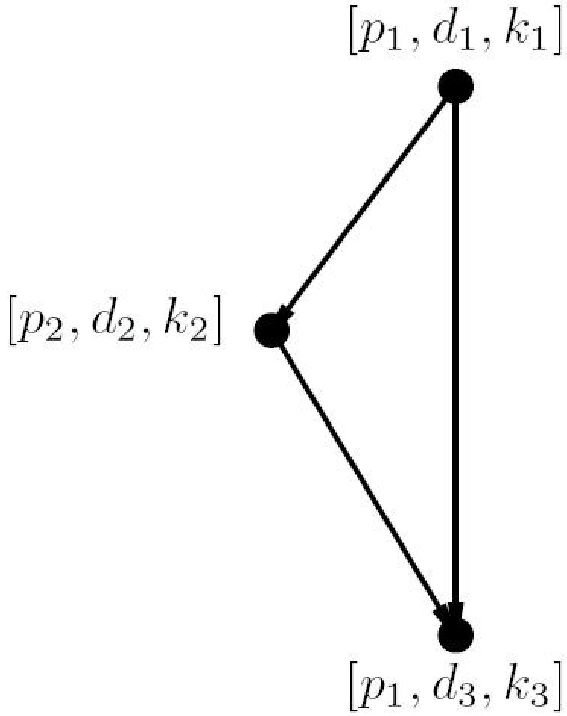
- Query D, exchange the returned values and temporal relations among them: build ever growing sample of the FD output (DAG)

- Use the DAG to simulate runs of A and extract the output of O

Building the DAGs

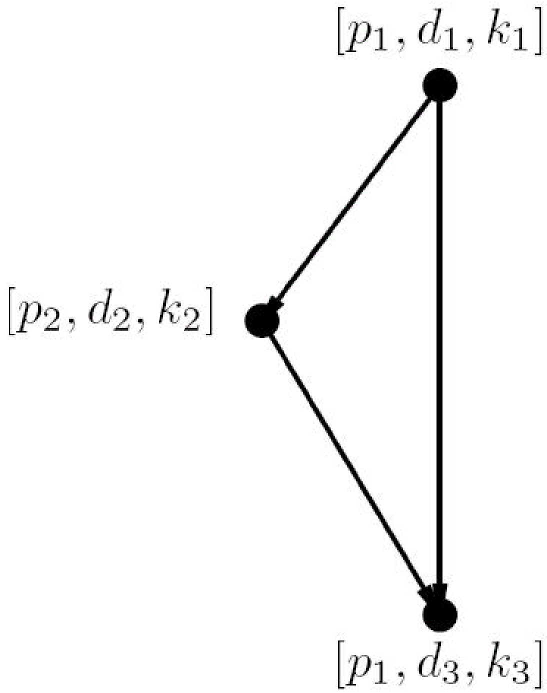
- Each process
  - ✓Queries D and updates the DAG
  - ✓Write DAG in the shared memory, read other DAGs and merge

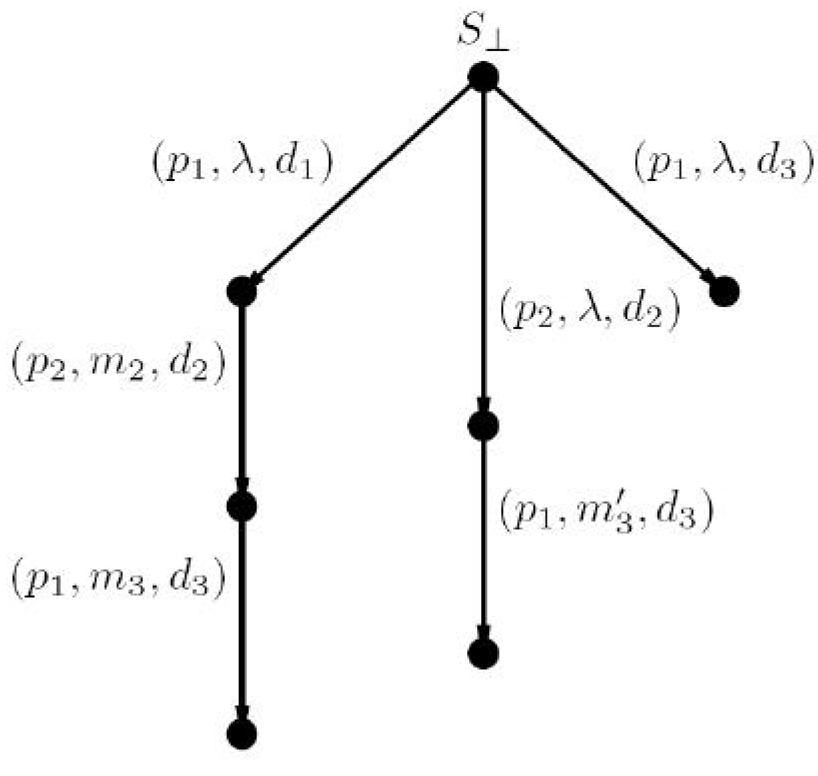
- Properties of DAGs
  - ✓Each vertex has an extension in which each correct process appears infinitely often
  - ✓Correct process eventually agree on the growing subDAGs



Simulation

- Each path in the DAG implies a simulated run – build a *simulation tree*





Valences

- A simulated finite run is v-valent if it has an extension with v decided
  - ✓Univalent – one decision is reachable
  - ✓Bivalent – two decisions are reachable

- Case 1: the initial state stays univalent
  - ✓If i is always decided, then always output Pi as the leader
- Case 2: the initial state is bivalent

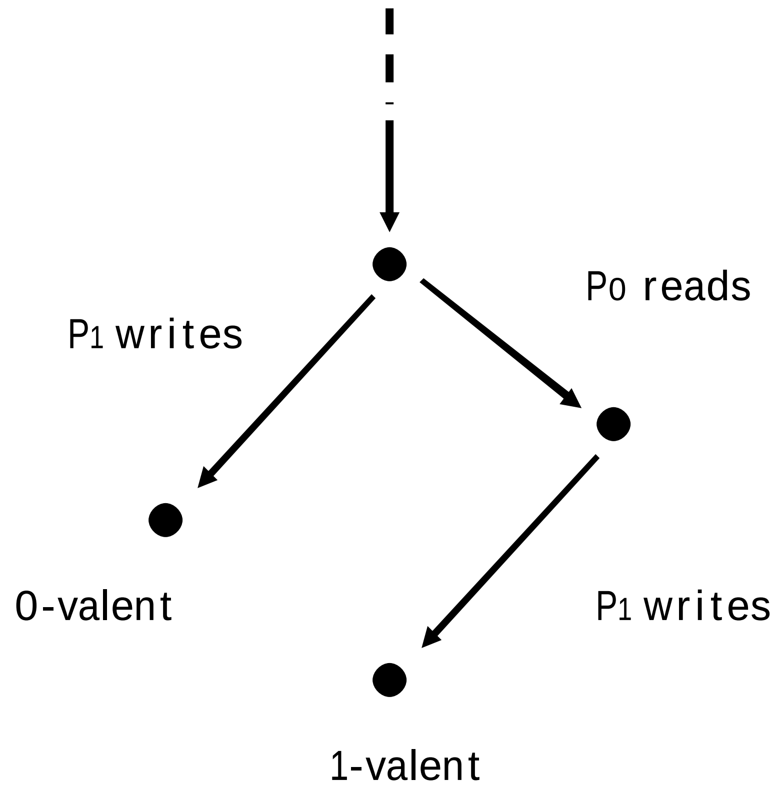
Bivalent initial state

Then the simulation tree has a critical run that hides the decision in a local state of a process

A la FLP:

- simulate a fair run starting from the root
- there is a bivalent run R and a process Pi such that any descendant R extended with a step of Pi is univalent

A hook



- P_0 must be correct!
- Output P_0 as the leader

Eventually

- Univalent initial state: the correct process proposing the decision is forever output
- Bivalent initial state: the correct process hiding the decision in *the first* hook is forever output

O is emulated!

Outline

- Level N: the WFD for consensus
- Level 1: the easiest nontrivial task
- Levels 2 - N-1: the easiest k-resilient impossible task?

Level 1: The weakest FD "ever"

- The failure detector that is:
  - ✓Nontrivial: sufficient to solve *some* unsolvable task: (N+1)-process N-set agreement
  - ✓Necessary: weaker than *any* nontrivial failure detector

(Populates the class of the easiest unsolvable tasks)

The candidate: anti-Omega [Zie08]

- Outputs a single process, eventually:
  - ✓Some correct process is never output
- Equivalent to N-vector of O's: at least one correct (elects a correct leader)
- Solves N-set agreement
  - ✓Run N parallel consensus instances, each using one position in N-vector O, decide on the value returned by the first decided instance

Two observations:

Let A solve a non-trivial task using a failure detector D and let G be any DAG based on D

- There exists an *asynchronous* algorithm A' that simulates runs of A using G instead of D
  - ✓Each finite run of A' simulates a run of A
- For all DAGs, A' has at least one non-deciding run

A’: asynchronous simulation of A

Let G be *any* DAG build as in CHT (with failure pattern F)

To simulate the next step of Pi

- Wait for the first vertex [Pi,d] of the DAG G that succeeds all causally preceding (in G) steps of A
  - ✓Perform the step of A using d

A': asynchronous simulation of A

The simulated run R' *could have happened* when A is run with failure pattern F

- R' can be *unfair:* some correct (in F) process may appear only finitely often
- But *safe:* cannot produce incorrect decisions

For each G, not all runs of A’ are deciding

- Otherwise, A’ (using G) solves a non-trivial task – a contradiction
- Let DAG G be constructed for some failure pattern F
- There exists a run of A’ using G that never decides: some faulty process (but correct in F) takes only finitely many simulated steps

Locating the non-deciding run

Each process runs two threads:

- Construct an ever-growing DAG G
- Locally simulate multiple runs of A using A' and G:
  - ✓ Do DFS on the "first" non-deciding run of A
  - ✓ Output the last process to be simulated

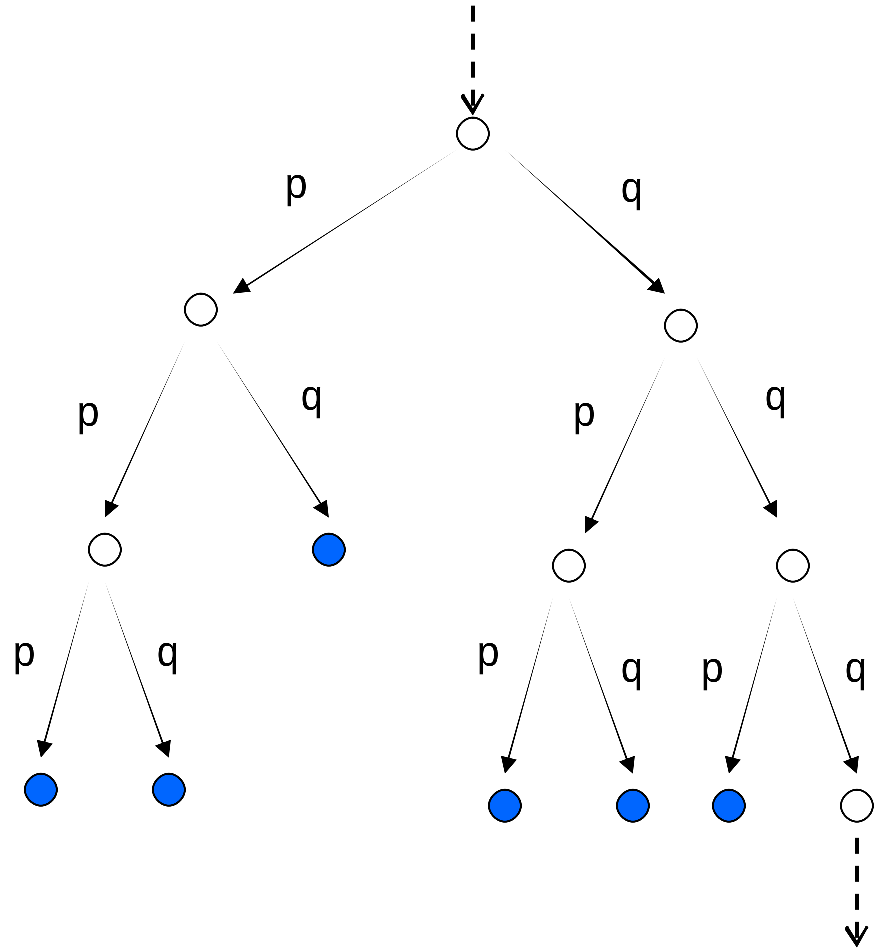
Two cases

- Some correct process gets stuck on waiting for a vertex of Pi to appear in G
  - ✓Every correct process eventually gets stuck too
  - ✓Pi is faulty: anti-O is extracted!

- No correct process ever gets stuck
  - ✓Correct processes go along the same never-deciding simulated run R'
  - ✓An issue: DFS does not prevent arbitrary "branching away" from R'

- The non-deciding run: q,q,q,...

- But steps of p are always simulated!



Solution: fairness increase

- For each prefix: simulate all extensions containing steps of subsets S1,S2,... of increasing sizes
  - ✓First all solo extensions, then all 2-process extensions, etc.
  - ✓Each next iteration simulates all runs simulated before

- Eventually, the first never deciding run R can only branch to deciding extensions with steps in inf(R)

Eventually

All correct processes either:

- Forever wait for some faulty process Pi
  - ✓Output Pi

- Forever simulates steps of processes in S, some correct process not in S
  - ✓Output processes in S

Outline

- Level N: the WFD for consensus
- Level 1: the easiest nontrivial task
- Levels 2 to N-1: some speculations

Generalization to k-set agreement?

CHT:

- (Partially) re-constructs FLP
- Running k CHTs in parallel (for each consensus instance)?
  - ✓Running k FLPs to prove the impossibility of k-set agreement?
- Does not populate level N+1-k

Generalization to k-set agreement?

- Plain impossibility of k-set agreement for k<N is not enough
- Cannot wait forever until one process takes a step

WFD for k-set agreement: upper bound

- Anti-Omega-k: outputs a set of N+1-k, eventually some correct process is never output
- Equivalent to k-vector-Omega

Relating wait-freedom and k-resiliency: BG-simulation

- For each (N+1)-process protocol P

  k+1 *simulators* produce a k-resilient run of P

- (N+1)-process k-resilient k-set agreement is impossible [BG93]

A k-resilient non-deciding run

- Let A be any algorithm that that solves k-set agreement using D
- Let G be any DAG

- There exists at least one never-deciding k-resilient run of A' (using G)

- Suppose not. Then k+1 processes solve k-set agreement (using BG simulation of A')

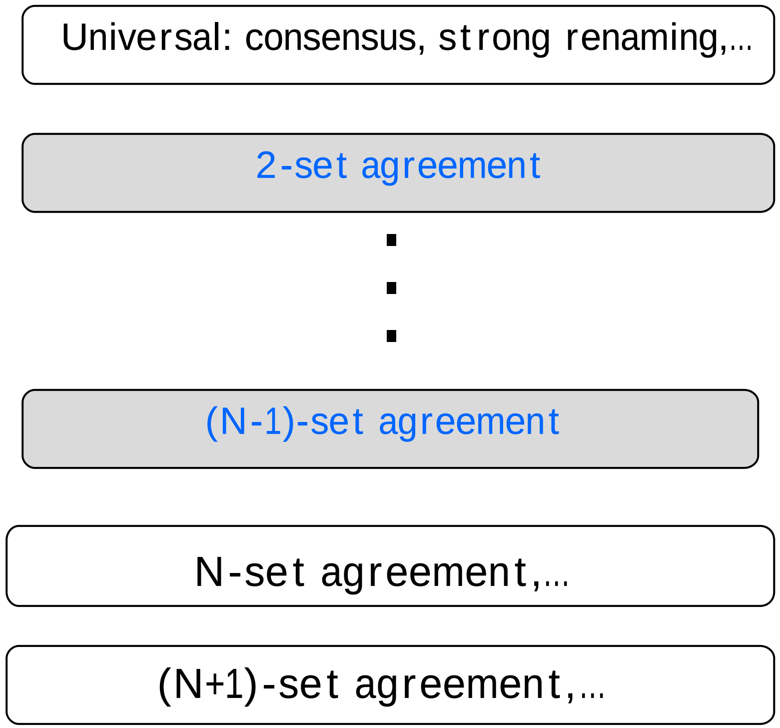
A possible reduction

- Eventually agree on the same ever-growing "fairest" non-deciding simulated run R
- Output the set S of N+1-k processes taking the most number of steps in R
  - ✓Some correct process not in S!

- Achieving convergence
- Dealing with faulty processes (=N+1-k)

Conclusions

- Conjecture: a hierarchy of tasks based on WFDs
- Bottom and top levels are characterized
- Filling the gap: new insights needed



Thank you!