Introduction

Radu Nicolescu
Department of Computer Science
University of Auckland

29 July 2020

Organisation

- For additional details, see the DCO and the Canvas Syllabus
  - https://courseoutline.auckland.ac.nz/dco/course/COMPSCI/711/1205
  - https://canvas.auckland.ac.nz/courses/45914

- Introduction to several fundamental distributed algorithms
- Three assignments, totalling 30
- A bigger assignment, called project plus report, totalling 30

- Exam: theory of the distributed algorithms, 40
- Assignments: practical implementations, emulations of specific distributed algorithms on a single computer

Distributed algorithms

- Computing: computer systems, networking, computer architectures, computer programming, algorithms
- Parallel computing: multi-processors/cores, parallel systems, threading, concurrency, data sharing and races, parallel programming (data and task parallelism), parallel algorithms
- Distributed computing: distributed systems, message passing, communication channels, distributed architectures, distributed programming, distributed algorithms
  - concurrency of components
  - paramount messaging time
  - lack of global clock in the async case
  - independent failure of components

Overlap between parallel and distributed computing

- One litmus test:
  - Parallel computing: tight coupling between tasks, that cooperate on shared memory
  - Distributed computing: loose coupling between nodes, that cooperate by messaging
- Another difference – specific for algorithms:
  - In classical algorithms, the problem is encoded and given to the (one single) processing element
  - In parallel algorithms, the problem is encoded and partitioned, and then its chunks are given to the processing elements
  - In distributed algorithms, the problem is given by the network itself and solved by coordination protocols
- More:

https://en.wikipedia.org/wiki/Distributed_computing#Parallel_and_distributed_computing

Typical scenario in distributed computing

- computing nodes have local memories and unique IDs
- nodes are arranged in a network: graph, digraph, ring, ...
- neighbouring nodes can communicate by message passing
- independent failures are possible: nodes, communication channels
- the network topology (size, diameter, neighbourhoods) and other characteristics (e.g. latencies) are often unknown to individual nodes
- the network may or may not dynamically change
- nodes solve parts of a bigger problem or have individual problems but still need some sort of coordination – which is achieved by distributed algorithms

Recall from graph theory

- Graphs, edges, digraphs, arcs, degrees, connected, complete graphs, size, distance, diameter, radius, paths, weights/costs, shortest paths, spanning trees, ...

- geodesic distance between a pair of nodes = minimum length of a connecting path, aka length of a shortest path (hop count in unweighted graphs)
  - min

- diameter = maximum distance, between any pair of nodes
  - max min

- radius = minimum maximum distance, for any node to any other node (minimum attained at centers)
  - min max min

Typical graphs notations

Graph: (V, E)

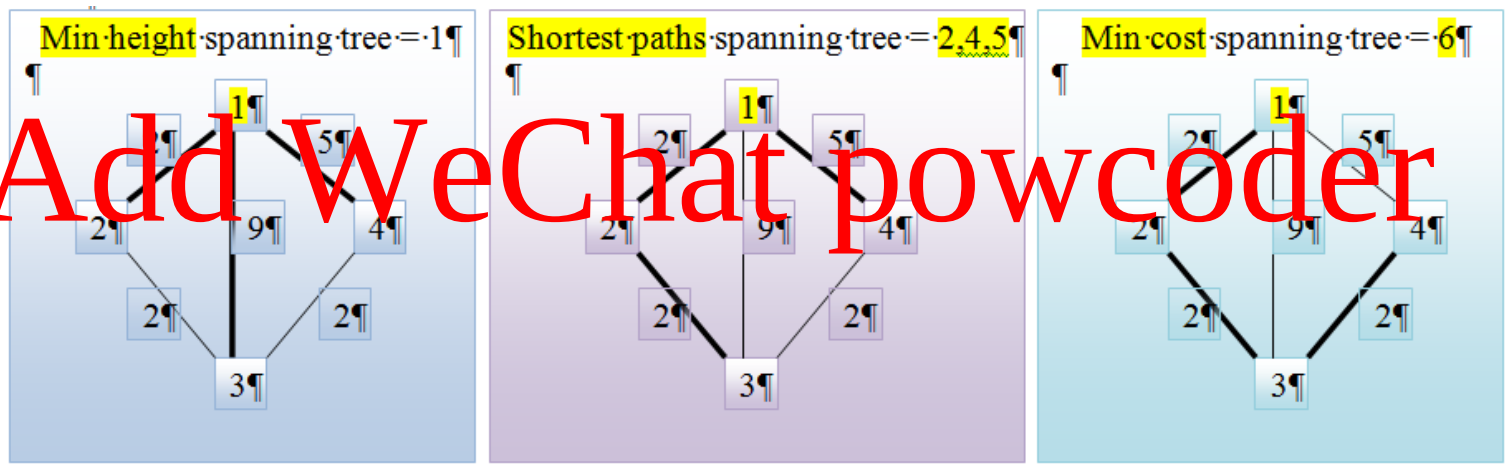
- V, set of nodes (vertices), $N = |V|$
- E, set of edges, $M = |E|$
- D, diameter = longest geodesic distance (longest shortest path) between any two nodes

- node eccentricity = longest geodesic distance (longest shortest path) from this node to other nodes

- D, diameter = maximum eccentricity, over all node pairs
- R, radius = minimum eccentricity, from any node
- centre (node) = node with minimum eccentricity (not unique)

Geodesics examples

- Three distinct spanning trees (rooted at 1) on the same undirected graph
- Diameter, radius, centre(s), of base graph?
- Diameter, radius, centre(s), of each spanning tree?

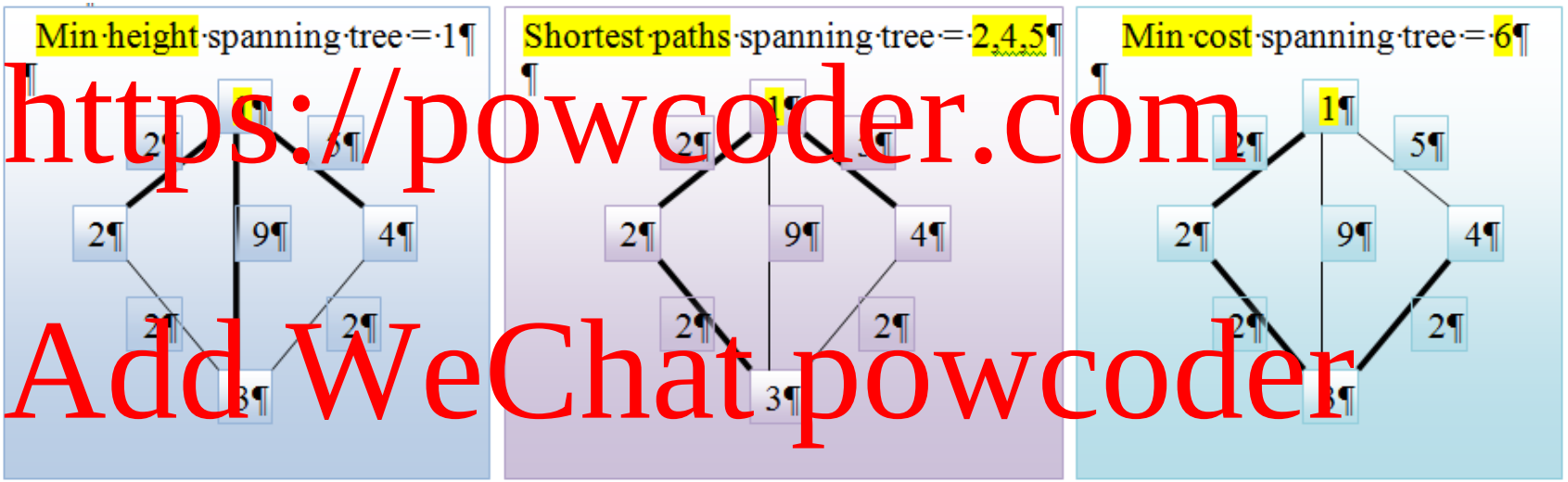


BFS and DFS spanning trees

- Reflexive edge: an edge that loops back to the same node
- Default assumption: unless specifically said so, graphs are irreflexive (no reflexive edges)
- Given a spanning tree, we define
  - Tree edge: an edge that belongs to the spanning tree
  - Frond edge: an edge that does not belong to the spanning tree
- BFS spanning tree characterisations
  - Each node at graph hop distance d from root appears at depth d in the tree
  - Frond edges only between nodes at depths differing by at most one (thus linking nodes on different branches)
- DFS spanning tree characterisation
  - Frond edges only between between nodes on the same branch (a frond links an ancestor with a descendant)

BFS and DFS spanning trees – examples

- Three distinct spanning trees (rooted at 1) on the same undirected graph
- Which one is BFS or DFS (start from root node 1)?



- Left is BFS; Right is DFS (when we first go to node 2)
- Middle is neither, but could be BFS, if we start from 2...
- ... or DFS, if we start from 3 or 4

Rounds and steps

- Nodes work in rounds (macro-steps), which have three sub-steps:
  1. Receive sub-step: receive incoming messages
  2. Process sub-step: change local node state
  3. Send sub-step: send outgoing messages
- Note: some algorithms expect null or empty messages, as an explicit confirmation that nothing was sent

Timing models

1. Synchronous models
   - nodes work totally synchronised – in lock-step
   - easiest to develop, but often unrealistic and less efficient
2. Asynchronous models
   - messages can take an arbitrary unbounded time to arrive
   - often unrealistic and sometimes impossible
3. Partially synchronous models
   - some time bounds or guarantees
   - more realistic, but most difficult

- Quiz: guess which models have been first studied?
  Heard of Nasreddin Hodja's lamp? ☺

Synchronous model – equivalent versions

- Synchronous model – version 1
  - all nodes: process takes 1 time unit
  - all messages: transit time (send $\longrightarrow$ receive) takes 0 time units

- Synchronous model – version 2
  - all nodes: process takes 0 time units
  - all messages: transit time (send $\longrightarrow$ receive) takes 1 time units

- The second (and equivalent) version ensures that synchronised models are a particular case of asynchronous models

Asynchronous model

- Asynchronous model
  - all nodes: process takes 0 time units
    - each message (individually): transit time (send $\longrightarrow$ receive) takes any real number of time units
      - or, normalised: any real number in the $[0, 1]$ interval
    - thus synchronous models (version 2) are a special case
    - often, a FIFO guarantee, which however may affect the above timing assumption (see next slide)
- Time complexity (worst-case) = supremum of all possible normalised async runs
  - NOTE: async time complexity $\geq$ sync time complexity as the sync run is just one of the all possible runs

Asynchronous model

- Asynchronous model with FIFO channels
  - the FIFO guarantee: faster messages cannot overtake earlier slower messages sent over the same channel
  - congestion (pileup) may occur and this should be accounted for
  - the timing assumption of the previous slide only applies to the top (oldest) message in the FIFO queue
  - after the top message is delivered, the next top message is delivered after an additional arbitrary delay (in $\mathbb{R}$ or $[0, 1]$)... [Lynch]
  - thus, a sequence of n messages may take n time units until the last is delivered
  - essentially, a FIFO "channel" may not be a simple channel, but need some middleware (to manage the message queue)
  - suggestion to develop robust models, who do not rely on any implicit FIFO assumption [Tel]

Nondeterminism

- In general, both models are non-deterministic
  - Sync and async: often a choice needs to be made between different requests (e.g. to consider an order among neighbours)
  - Async: messages delivery times can vary (even very widely)

- However, all executions must arrive to a valid decision (not necessarily the same, but valid)
  - If always same decisions, then the system is called confluent

Starting and stopping options

- Starting options
  - Single-source or centralised or diffusing: starts from a designated initiator node
  - Many-source or decentralised: starts from many nodes, often from all nodes at the same time
- Termination options
  - Easy to notice from outside, but how do the nodes themselves know this?
  - Sometimes one node (often the initiator) takes the final decision and can notify the rest (usually omitted phase)
  - In general, this can be very challenging and requires sophisticated control algorithms for termination detection

Echo algorithm

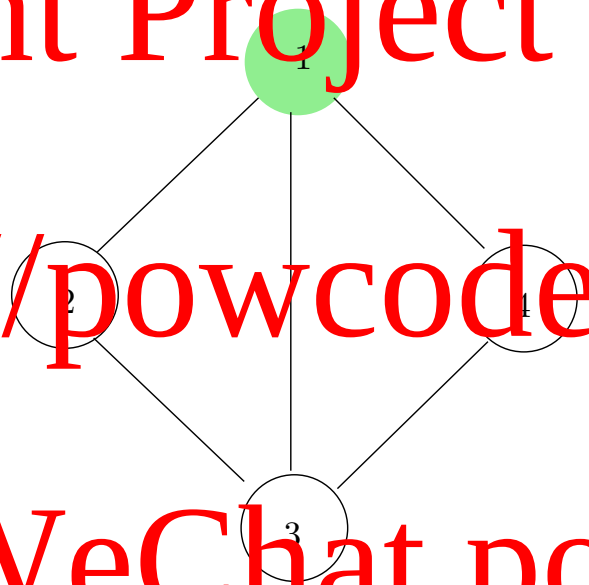
- Echo is a fundamental diffusing (single source) algorithm, which is the basis for several others
- combines two phases (imagine a stone thrown into a pond creating waves which echo back)
  - a broadcast, "top-down" phase, which builds child-to-parent links in a spanning tree
  - a convergecast, "bottom-up" or echo phase, which confirms the termination
  - parent-to-child links can be built either at convergecast time or by additional confirmation messages immediately after the broadcast (not shown in the next slides)
- after receiving echo tokens from all its neighbours, the source node decides the termination (and can optionally start a third phase, to inform the rest of this termination)

Echo algorithm

- Echo algorithm is an instance of a large family known as wave algorithms (not further discussed here)

- In the sync mode, the broadcast phase of Echo determines a BFS spanning tree – Echo is also known as SyncBFS
- In the async mode, the broadcast phase of Echo determines a spanning tree, but not necessarily a BFS spanning tree

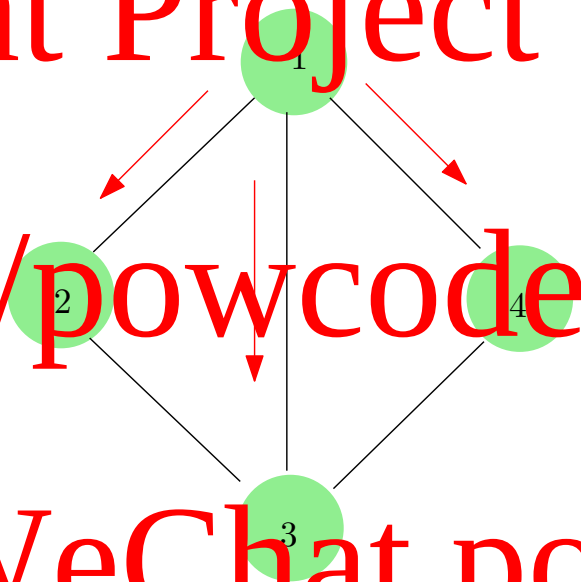
Echo Algorithm - Sync



Time Units = 0

Messages = 0

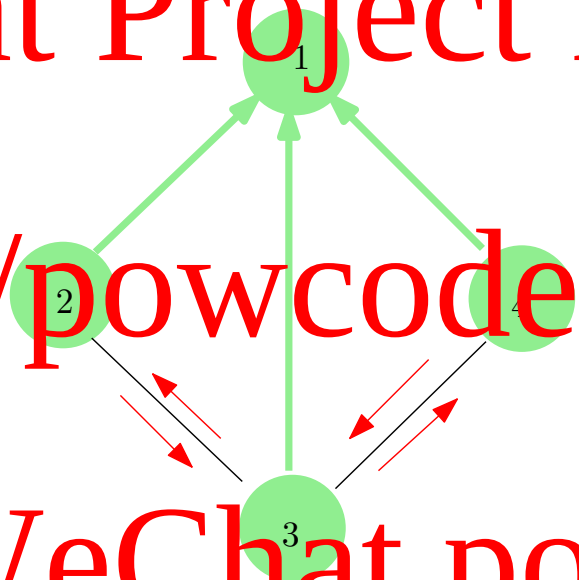
Echo Algorithm - Sync



Time Units = 1

Messages = 3

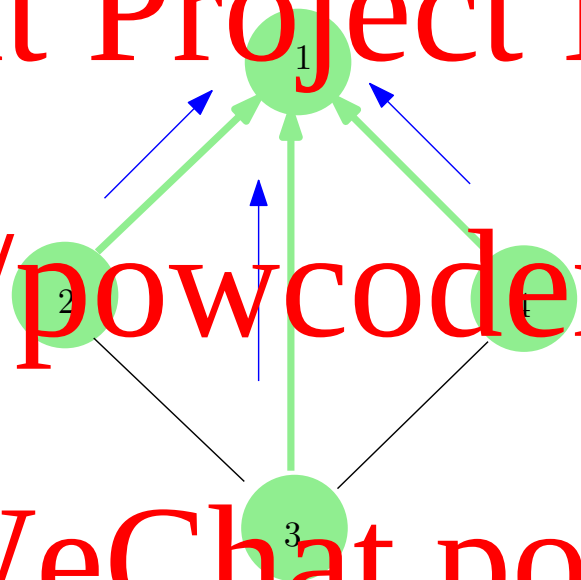
Echo Algorithm - Sync



Time Units = 2

Messages = 7

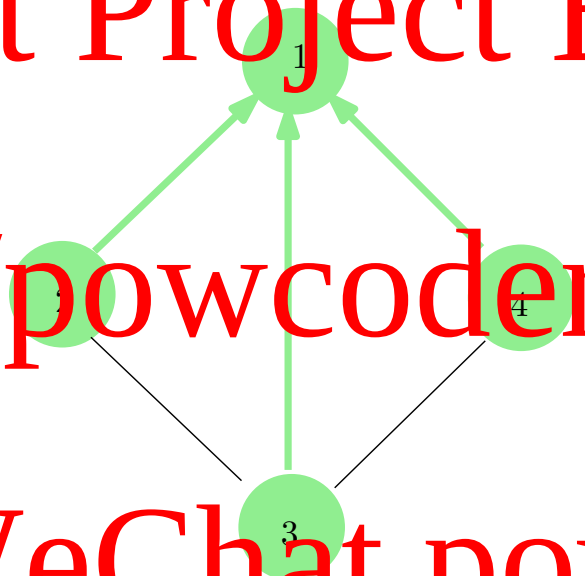
Echo Algorithm - Sync



Time Units = 3

Messages = 10

Echo Algorithm - Sync



Time Units $= 3 \leq 2D + 1$

Messages $= 10 = 2|E|$

Echo programs (Tel)

- Each node has a list, Neigh, indicating its neighbours
- There are two different programs: (1) for the initiator, and (2) for the non-initiators
- Here, the programs are high-level pseudocode, not in the basic Receive/Process/Send state-machine format (but can be reduced to this)
- With the exception of the initiator's first step, nodes become active only after receiving at least one message; i.e. nodes are idle (passive) between sending and receiving new messages
- Exercise: try to translate the following pseudocodes into a state machine format

Echo program for initiator (Tel)


```
1  let parent = null
2  let rec = 0
3
4  for q in Neigh do
5      send tok to q
6
7  while rec < |Neigh| do
8      receive tok
9      rec += 1
10
11 decide
```

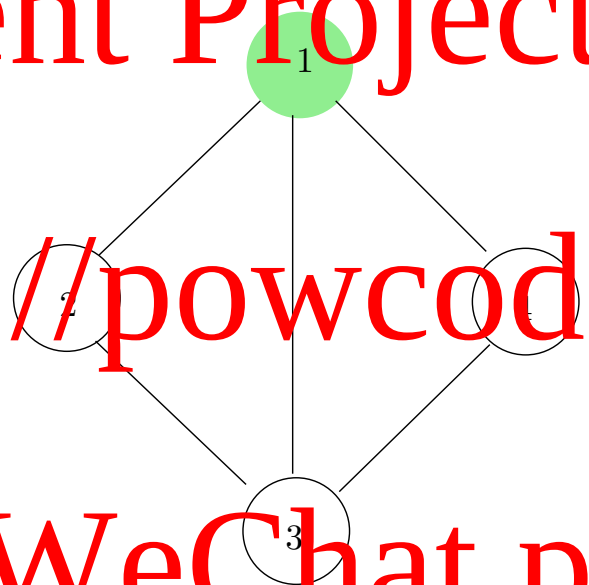
Echo program for non-initiators (Tel)

```
1  let parent = null
2  let rec = 0
3
4  receive tok from q                    // non-deterministic choice!
5  parent = q
6  rec += 1
7
8  for q in Neigh \ parent do
9      send tok to q
10
11 while rec < |Neigh| do                // count all received tokens
12     receive tok                       // forward and return
13     rec += 1
14
15 send tok to parent
```

Sync vs. Async

- Using the informal complexity measures appearing in the preceding slides (there are others), we conclude
- Sync: time complexity $= O(D)$, message complexity $= O(|E|)$
- The same Echo programs run also in the async mode and obtain valid results
- However, the runtime complexity measure changes (drastically)
- Async: time complexity$=O(|V|)$, message complexity$=O(|E|)$
- Why?
- For the time complexity, we take the supremum over all possible normalised executions (delivery time in $[0, 1]$)

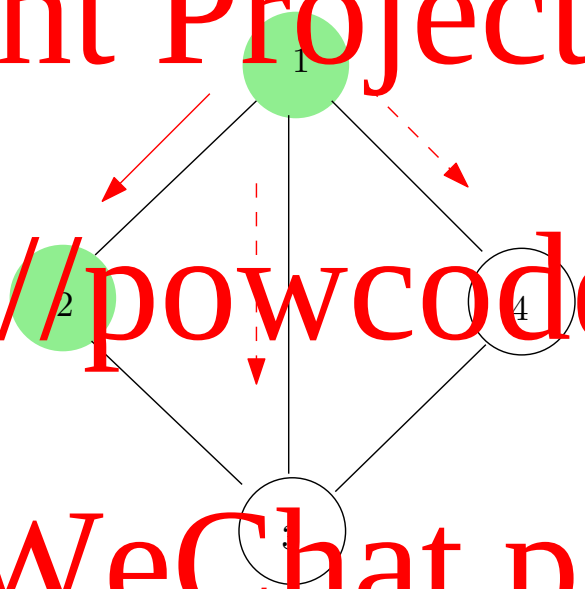
Echo Algorithm - Async



Time Units = 0

Messages = 0

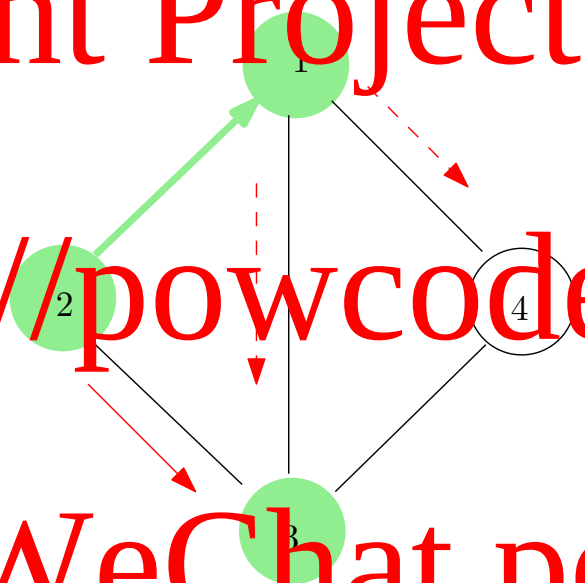
Echo Algorithm - Async



Time Units $= \varepsilon$

Messages $= 3$

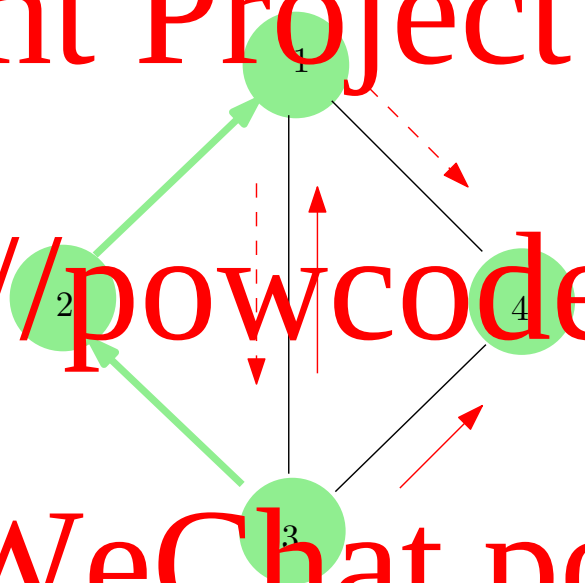
Echo Algorithm - Async



Time Units $= 2\varepsilon$

Messages $= 4$

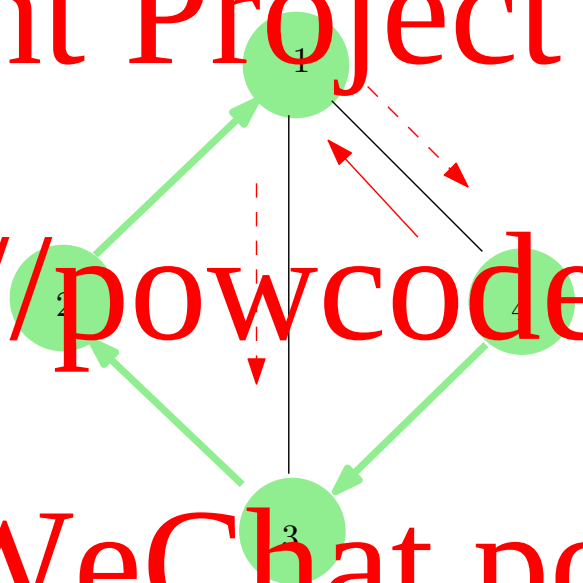
Echo Algorithm - Async



Time Units $= 3\varepsilon$

Messages $= 6$

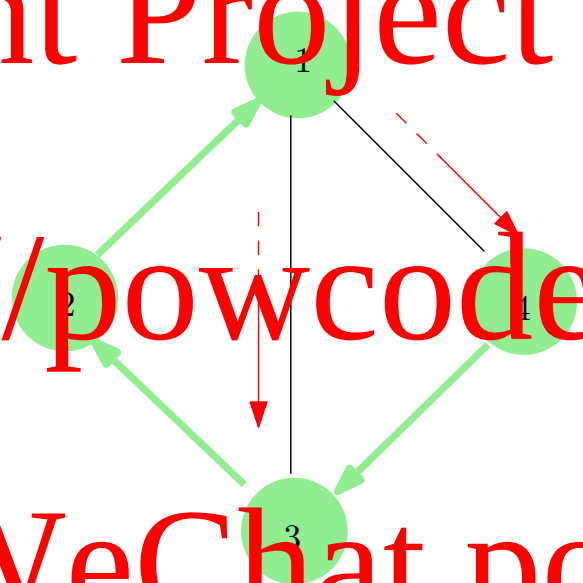
Echo Algorithm - Async



Time Units $= 4\varepsilon$

Messages $= 7$

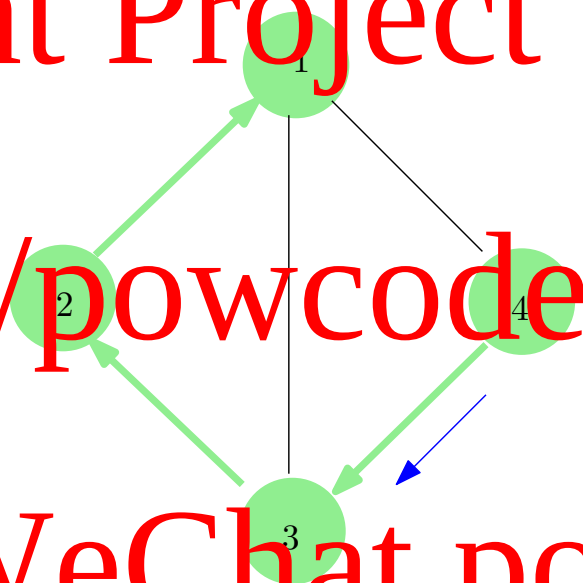
Echo Algorithm - Async



Time Units = 1

Messages = 7

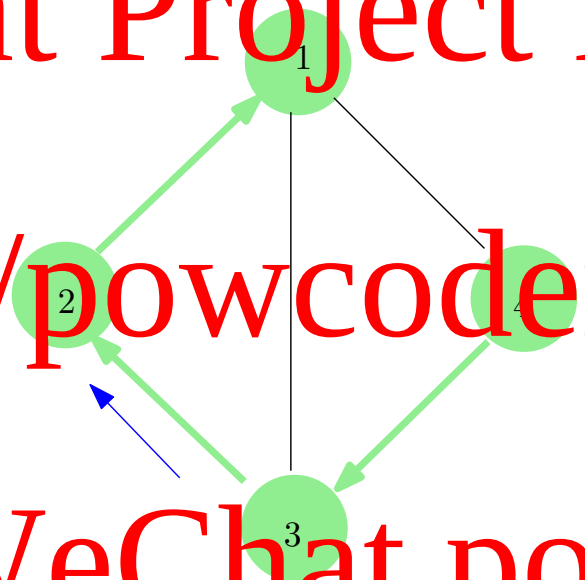
Echo Algorithm - Async



Time Units = 2

Messages = 8

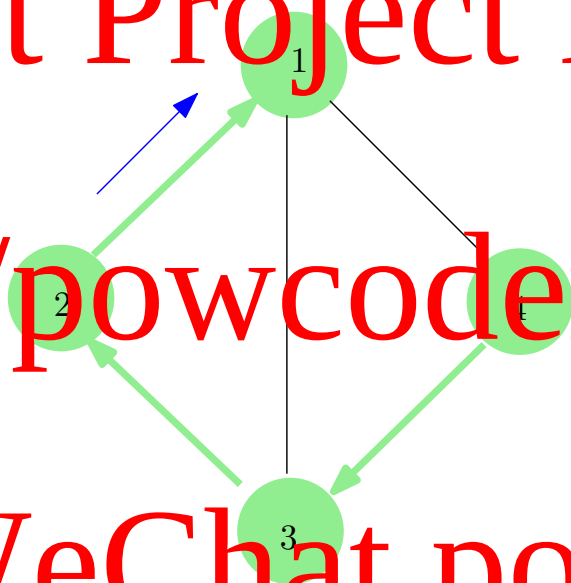
Echo Algorithm - Async



Time Units = 3

Messages = 9

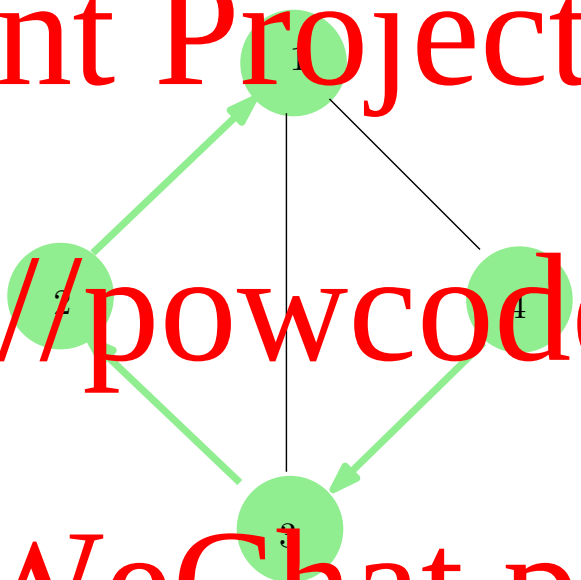
Echo Algorithm - Async



Time Units = 4

Messages = 10

Echo Algorithm - Async



Total Time Units = 4

Time Units on Broadcast = $1 \leq D$

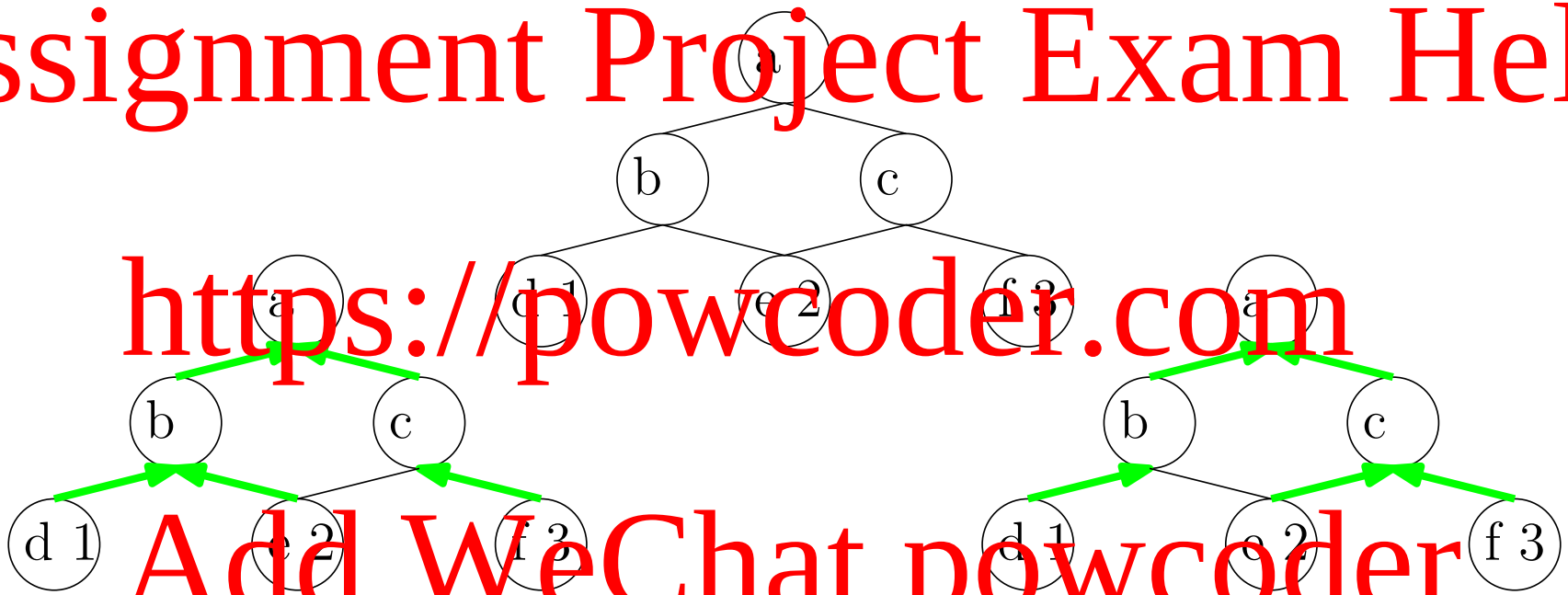
Time Units on Convergecast = $3 = |V| - 1$

Messages = 10

Echo algorithm revisited

- Like other members of the wave algorithm family, Echo can be adapted to determine:
  - The size of the network (number of nodes)
  - The number of nodes which have a given property
  - The maximum, minimum or sum of all values contained in nodes (assuming that each node contains a numerical value)
  - In general, functions which are associative and commutative, such as + (why?)
  - A "leader" (e.g. the node with the highest ID)
- A simple "trick": values can be attached to the tokens!
  - Forward token with null/zero value
  - Return token to parent with subtree value

Echo/size program – associativity and commutativity



Non-determnistic but confluent evaluations – all return 6!

- Left: (1+2)+3, (2+1)+3, 3+(1+2), 3+(2+1)
- Right: 1+(2+3), 1+(3+2), (2+3)+1, (3+2)+1

Echo/size program for initiator


```
1  let parent = null
2  let rec = 0
3  let size = 1
4
5  for q in Neigh do
6      send (tok,0) to q
7
8  while rec < |Neigh| do
9      receive (tok,s) // order irrelevant; ⊕ commutative
10     rec += 1
11     size += s
12
13 decide size
```

Echo/size program for non-initiators

```
1  let parent = null
2  let rec = 0
3  let size = 1
4
5  receive (tok,s) from q   // choice irrelevant; + associative
6  parent = q
7  rec += 1
8
9  for q in Neigh \ parent do
10     send (tok,0) to q
11
12 while rec < |Neigh| do
13     receive (tok,s)          // fan-out tokens: s=0
14     rec += 1                 // fan-in tokens: s=subtree size
15     size += s
16
17 send (tok,size) to parent // only children really contribute
```

Further algorithms

- Besides some basic fundamental algorithms, we intend to cover (or have a good look at) some of the most challenging distributed algorithms

  - Distributed MST (Minimal Spanning Tree). Dijkstra prize 2004
  - Byzantine agreement: "the crown jewel of distributed algorithms" – Dijkstra prize 2005, 2001

  - The relation between Byzantine algorithms and blockchains
  - all these have practical significance and applications

Project and practical work

- Practical work is necessary to properly understand the concepts
- Unfortunately, we cannot afford to experiment with real distributed systems (sorry for this ☹)

- We need to play the "Game of Distribution" and simulate or, better, emulate distributed systems on a single lab machine

- For uniformity and fairness in marking, we use .NET specifically with C# (or F#)
- The final exam is focused on concepts, does not contain "technological" questions (related to .NET)

Prerequisites (cf. 335)

- Familiarity with C#, at least to the level presented in the C# 5.0 Specifications, Chapter Introduction (pp 1–32):

  `https://www.microsoft.com/en-nz/download/details.aspx?id=7029`

- Familiarity with basic .NET API or ability to learn this on the fly, as needed
- Familiarity with the specific API that we will use to emulate distributed systems

- Familiarity with searching and reading the MSDN library
- Familiarity with Linqpad `http://www.linqpad.net/`

How to emulate sync and async distributed systems?

- Single process/task emulation by well defined calls among "nodes"
- Multi-task emulation via channels (or pipelines or actors)
- Multi-process emulation by HTTP/REST calls (e.g. Nancy, Sinatra)

Readings

- Textbooks (in library, also limited previews on Google books)
  - Lynch, Nancy (1996). Distributed Algorithms. Morgan Kaufmann Publishers. ISBN 978-1-55860-348-6.
  - Tel, Gerard (2000). Introduction to Distributed Algorithms, Second Edition, Cambridge University Press, ISBN 978-0-52179-483-1.
  - Fokkink, Wan (2013). Distributed Algorithms - An Intuitive Approach, MIT Press (Second Edition 2018 also available).
- Research and overview articles (generally overlapping the textbooks) will be indicated for each topic

Readings 2

- My articles (in collaboration) on bio-inspired models for distributed algorithms may offer additional views
  - network discovery
  - firing squad synchronisation (graphs and digraphs)
  - BFS, BFS, edge and node disjoint paths
  - dynamic programming (DP), belief propagation (BP)
  - image processing (stereo, skeletonisation, region growing)
  - Byzantine agreement
  - formal verification
  - hard problems (SAT, TSP, QSAT)
- Pre-print versions published in the CDMTCS research reports

`https://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/secondcgi.pl?date`