Arrow and Ivy

Ordering Shared Objects in Distributed Networks

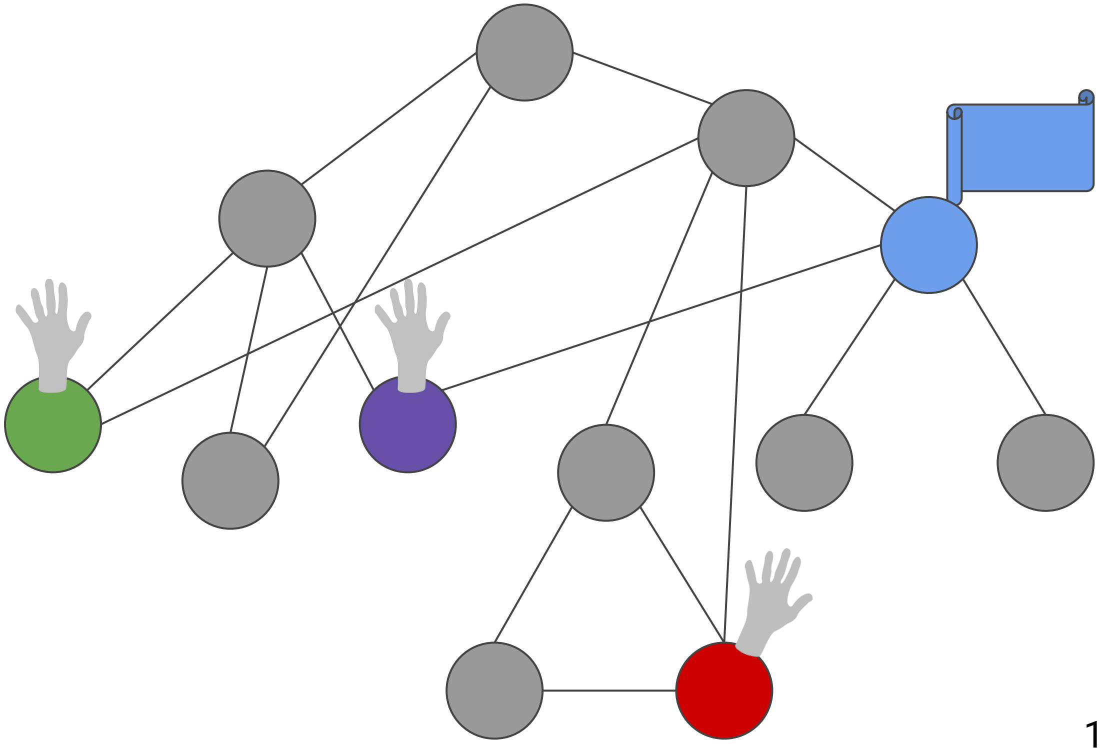
Arash Pourdamghani

January 2021
University of Vienna

Input

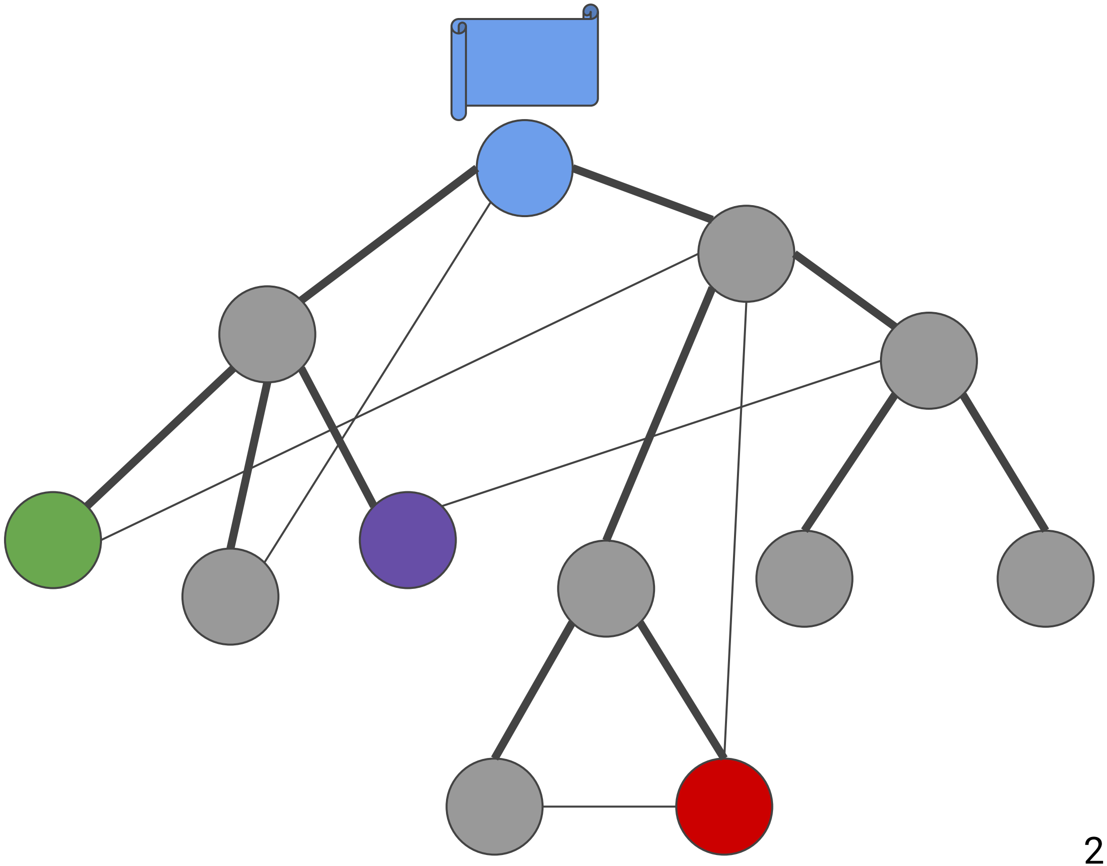
- Underlying Network
- Shared Object
  - Variable
  - Data Structure

- Requests:
  - Read
  - Write
  - Read-Modify-Write



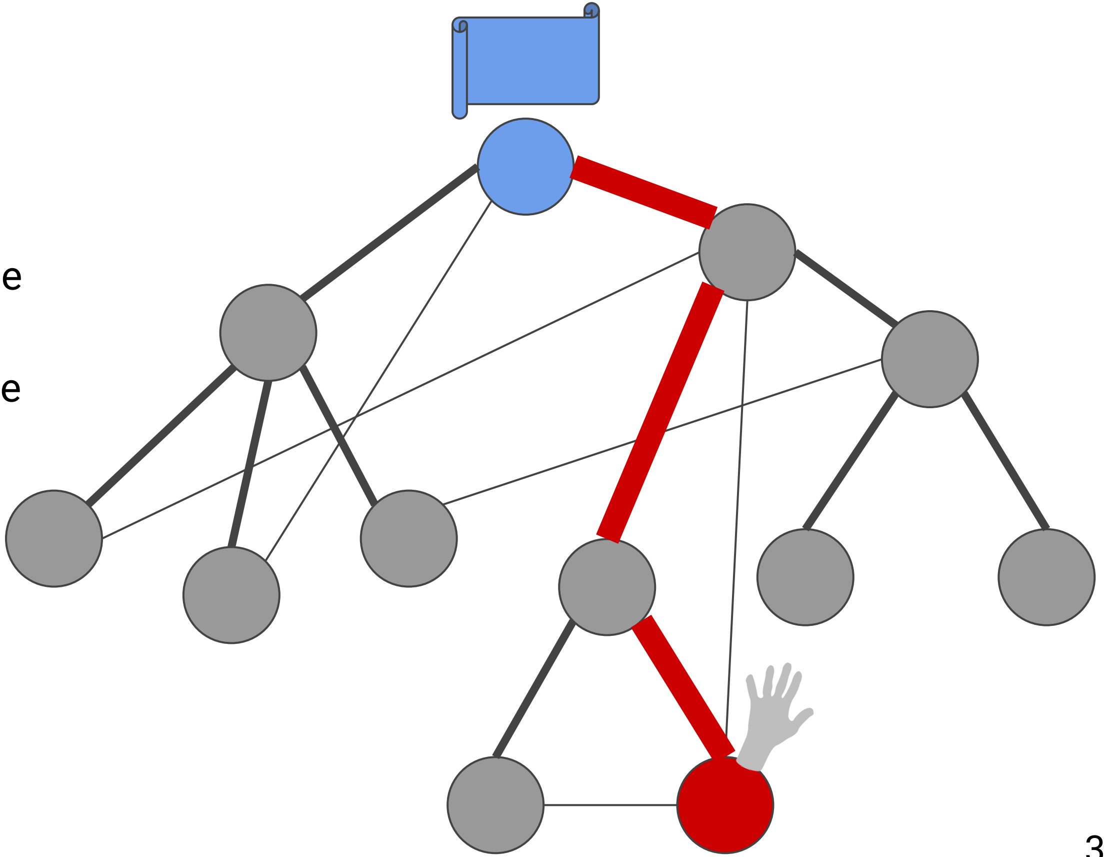
Solutions

- Choosing a spanning Tree:
  - Optimal Choice?
  - Root
- Without Modification:
  - Central Location
  - Home Based
- With Modification:
  - Arrow
  - Ivy
  - Arvy



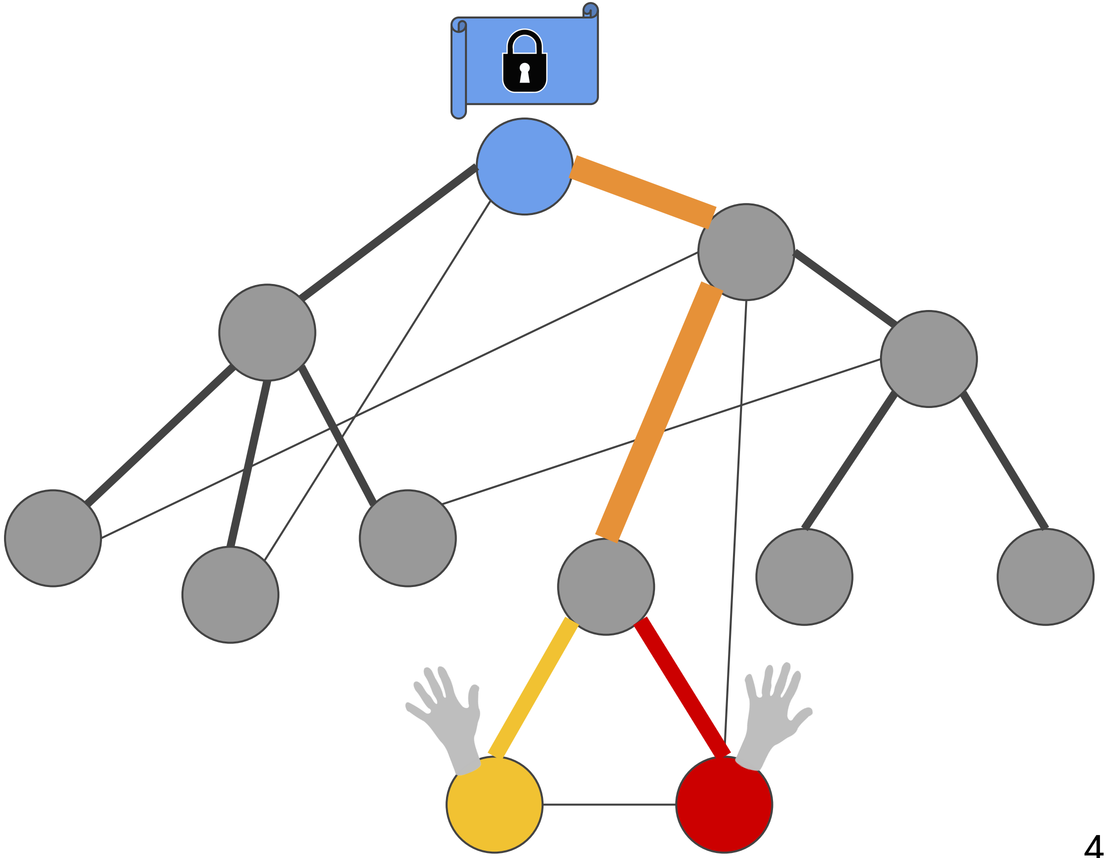
Central Location

- Process:
  - Send requests to root
  - Root Process Requests
  - Send back result down tree
- Bad example:
  - All requests from one node
- Solutions:
  - Route back directly
  - Moving the root



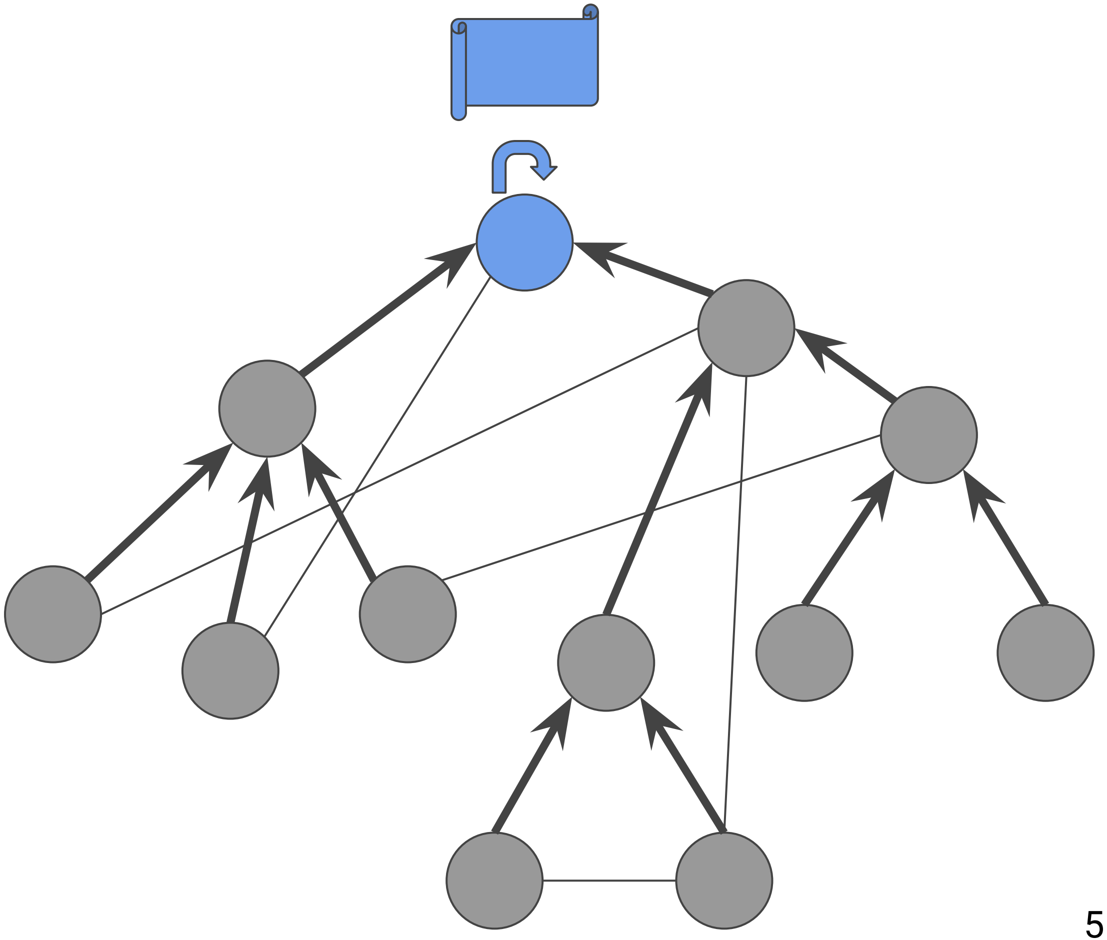
Home-Based

- Process:
  - Known home base
  - Request a lock
  - Process Locally
- Benefits:
  - Mobile networks
- Bad example:
  - Triangle routing
- Solution:
  - Moving the root



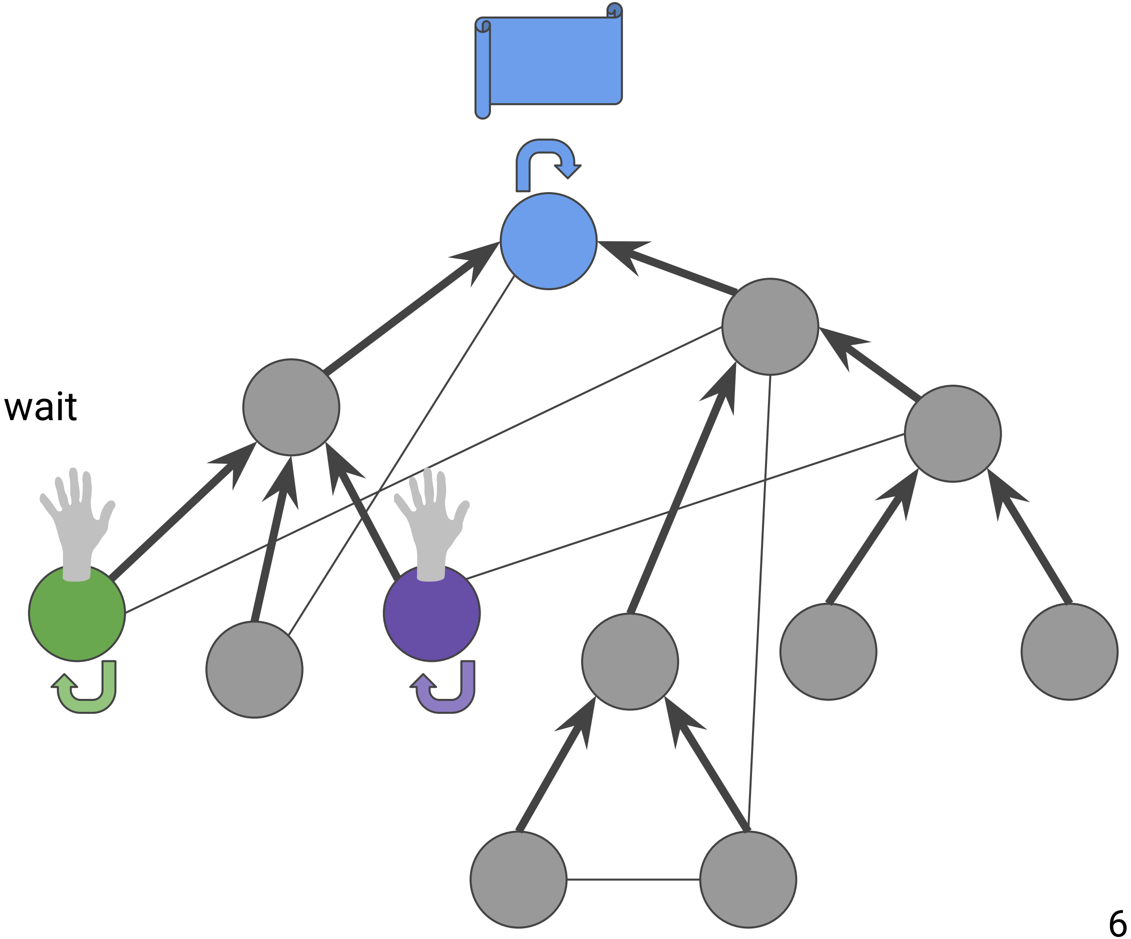
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph



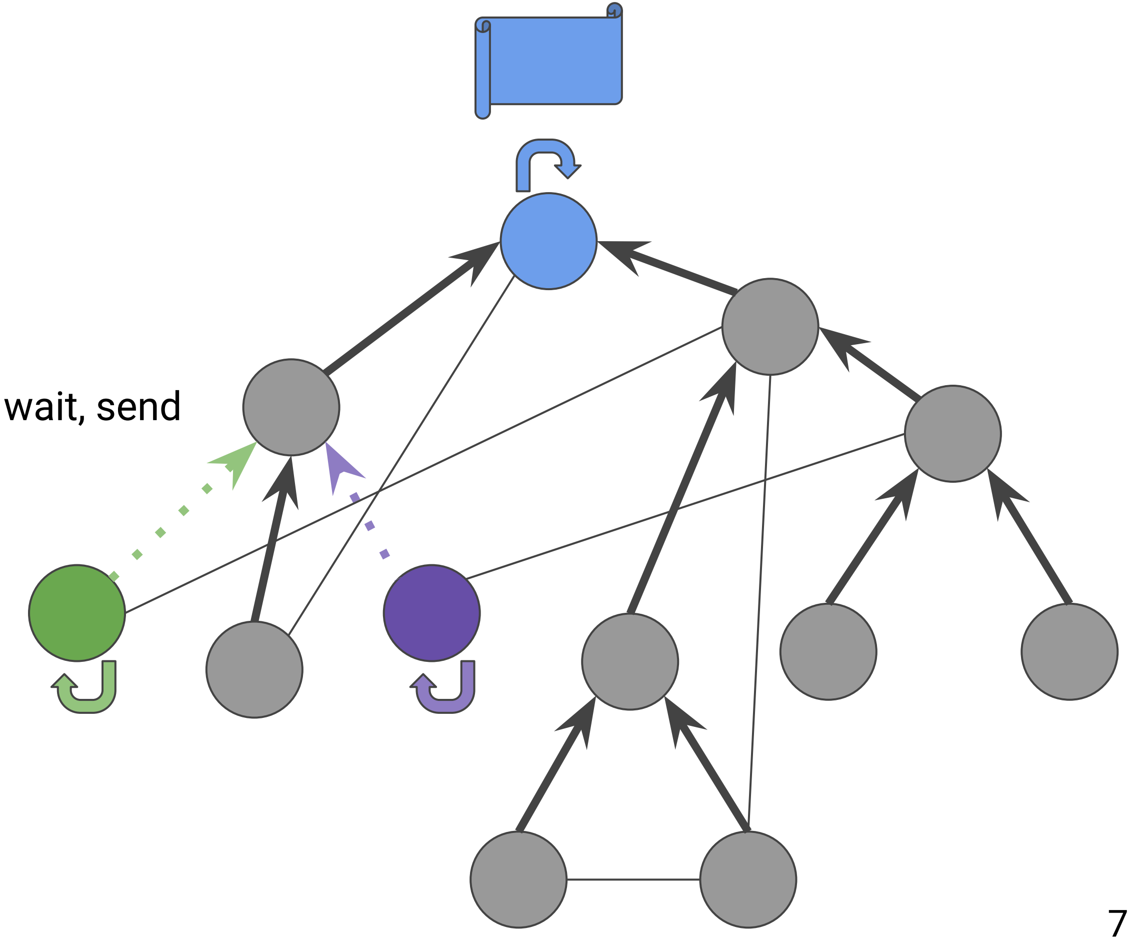
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait



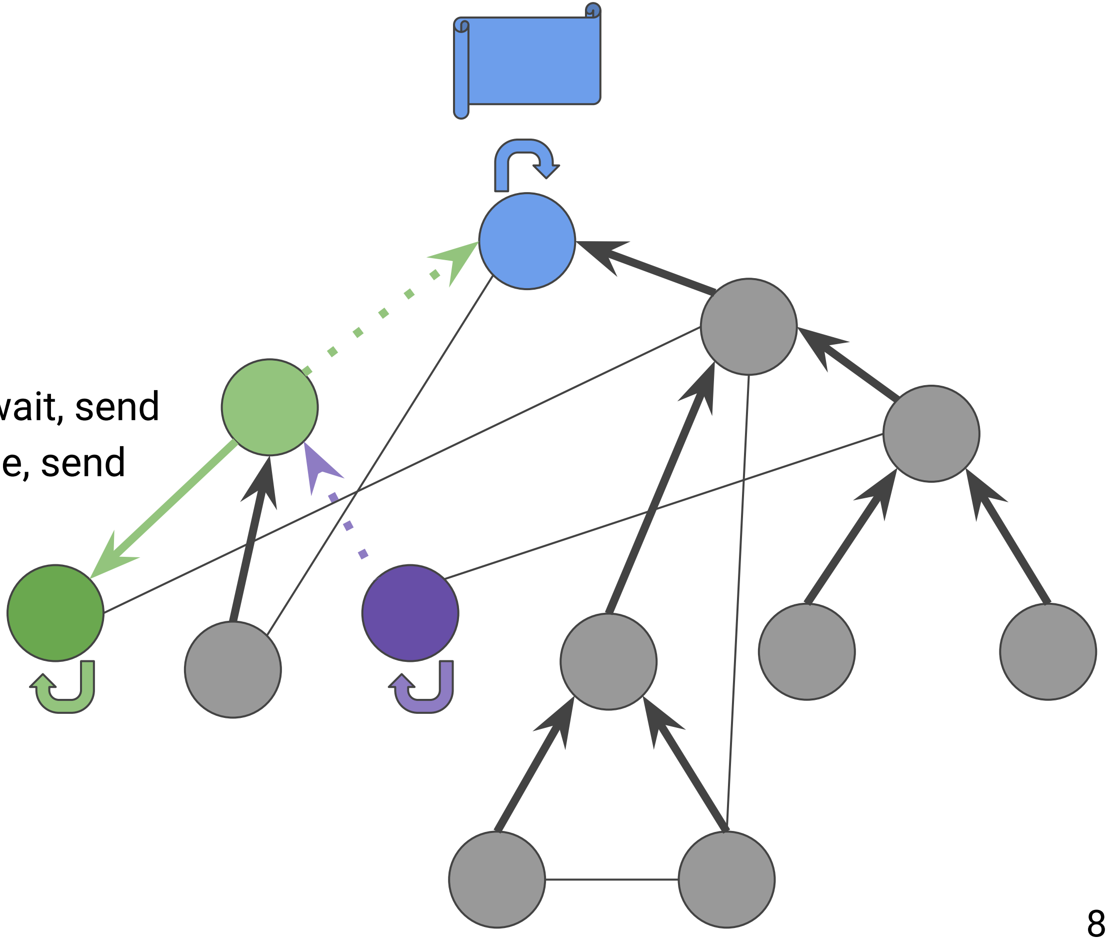
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait, send



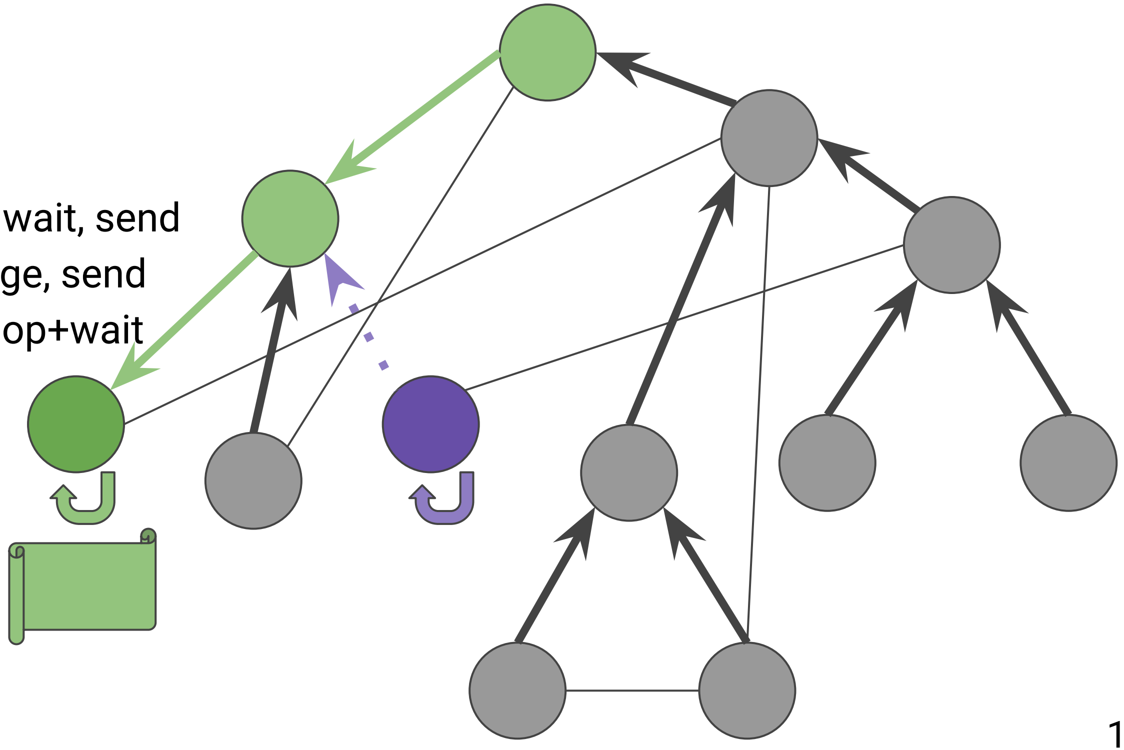
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait, send
  - Arrival: reverse arrival edge, send



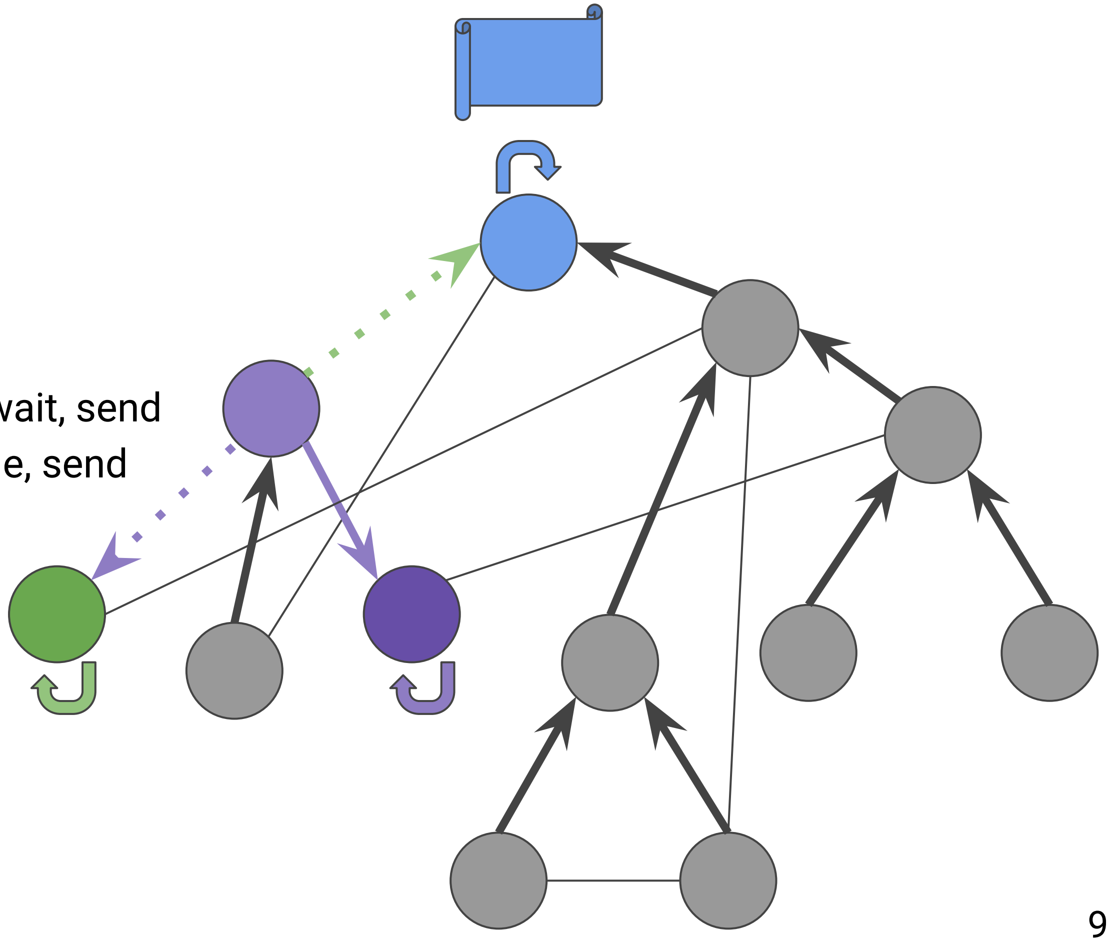
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait, send
  - Arrival: reverse arrival edge, send
  - Root: Send, delete self-loop+wait



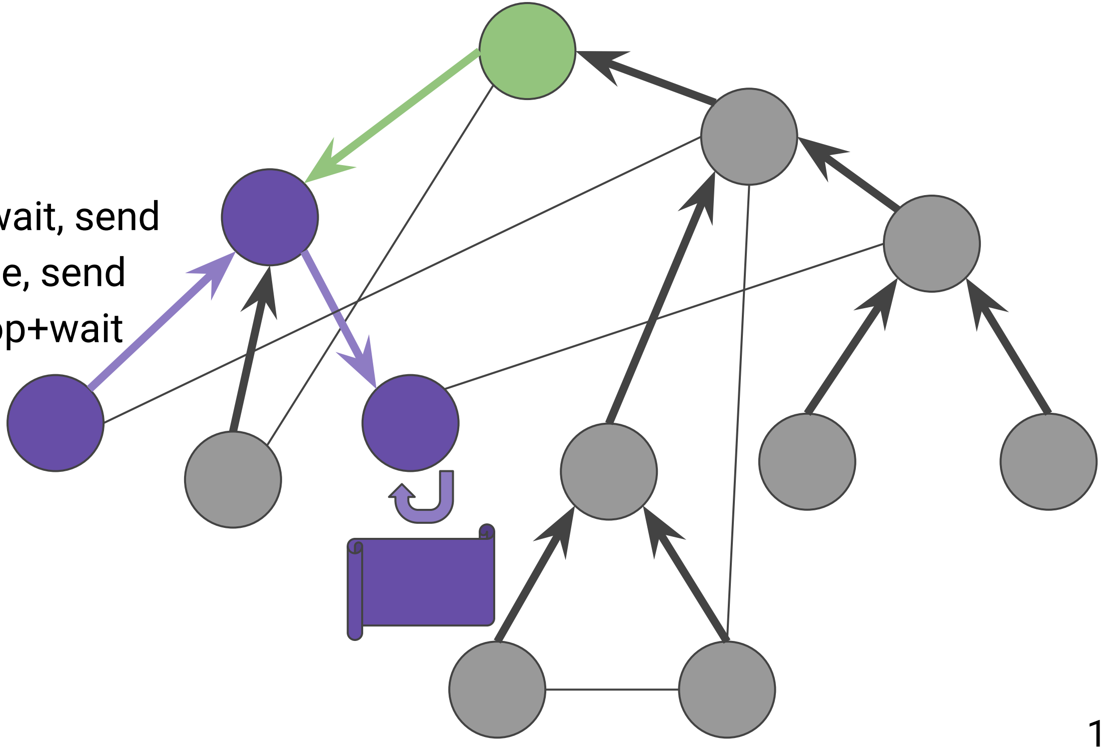
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait, send
  - Arrival: reverse arrival edge, send



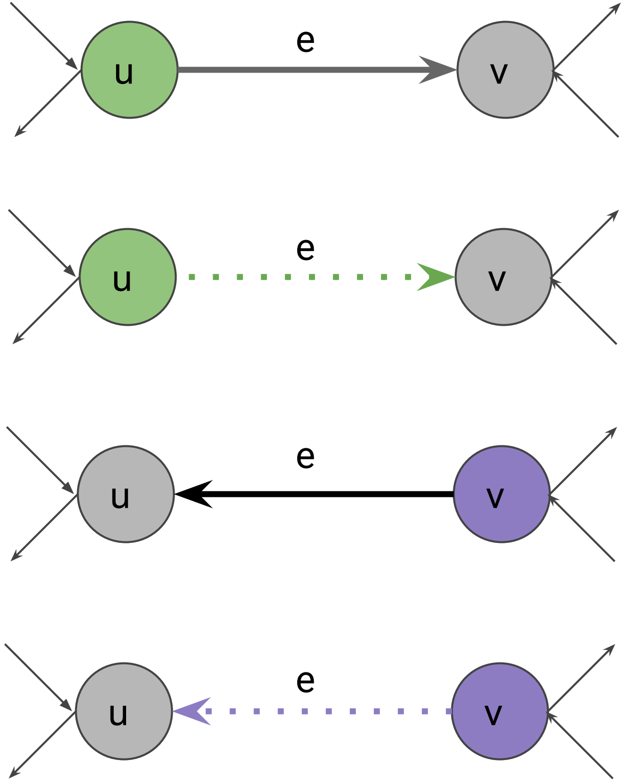
Arrow: Protocol

- Directing tree towards root:
  - Self loop for root
  - Keep underlying graph
- Finding process:
  - Initiator: Make Self-loop, wait, send
  - Arrival: reverse arrival edge, send
  - Root: Send, delete self-loop+wait



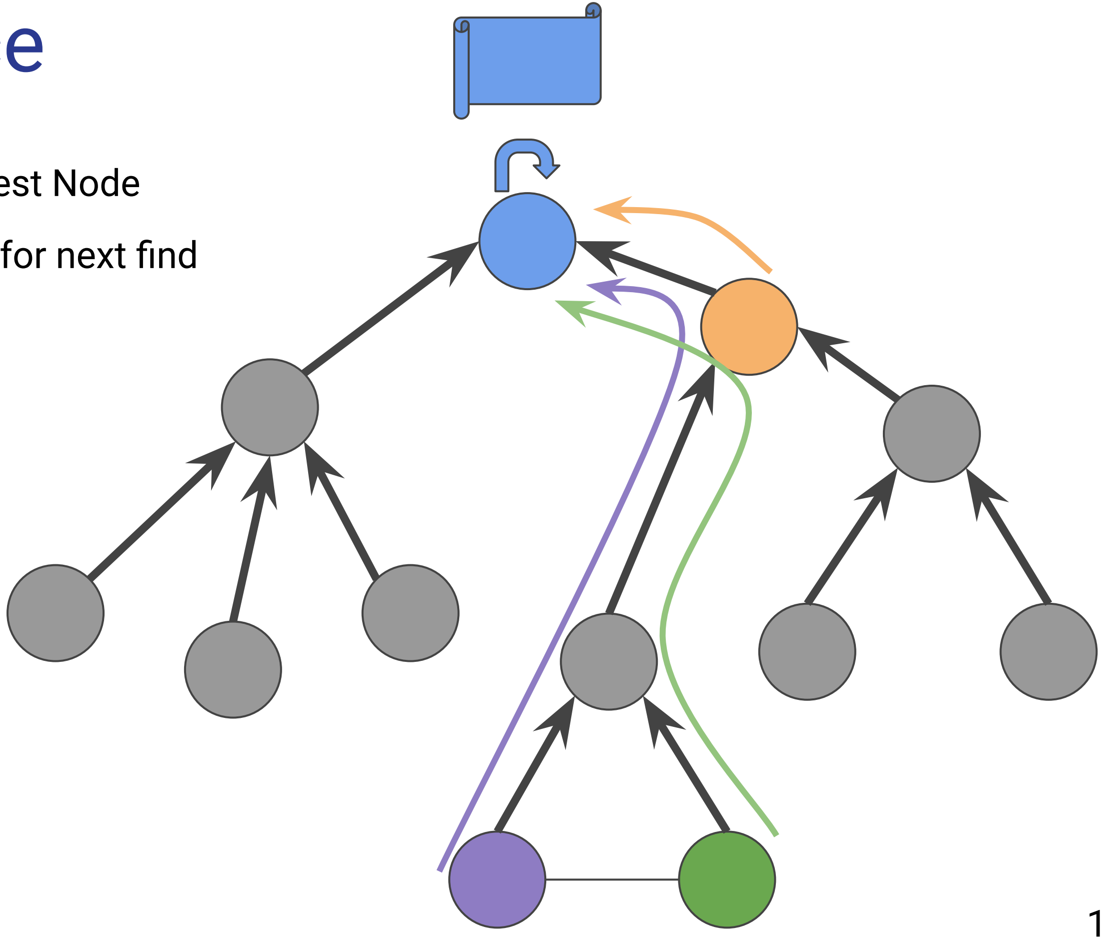
Arrow: Correctness

- Each edge in just one of the four states
- "Find" requests will go through static tree
- If an edge "e" be traversed twice:
  - Two traversals must be subsequent,
  - But at the end of state "2", find request will go out of "v", since each node has always have an outgoing edge



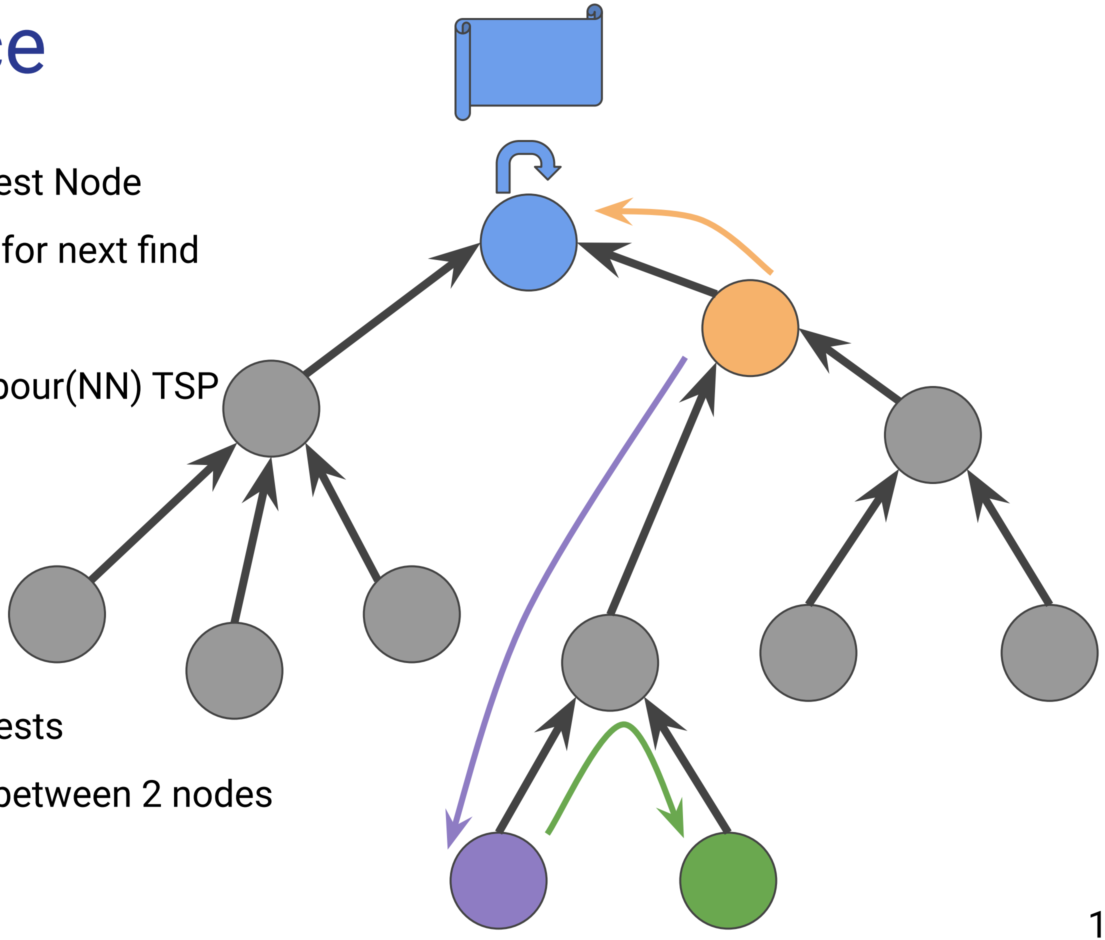
Arrow: Performance

- First “Find” reaches root <-> Nearest Node
- Everything “seems” to be reseted for next find



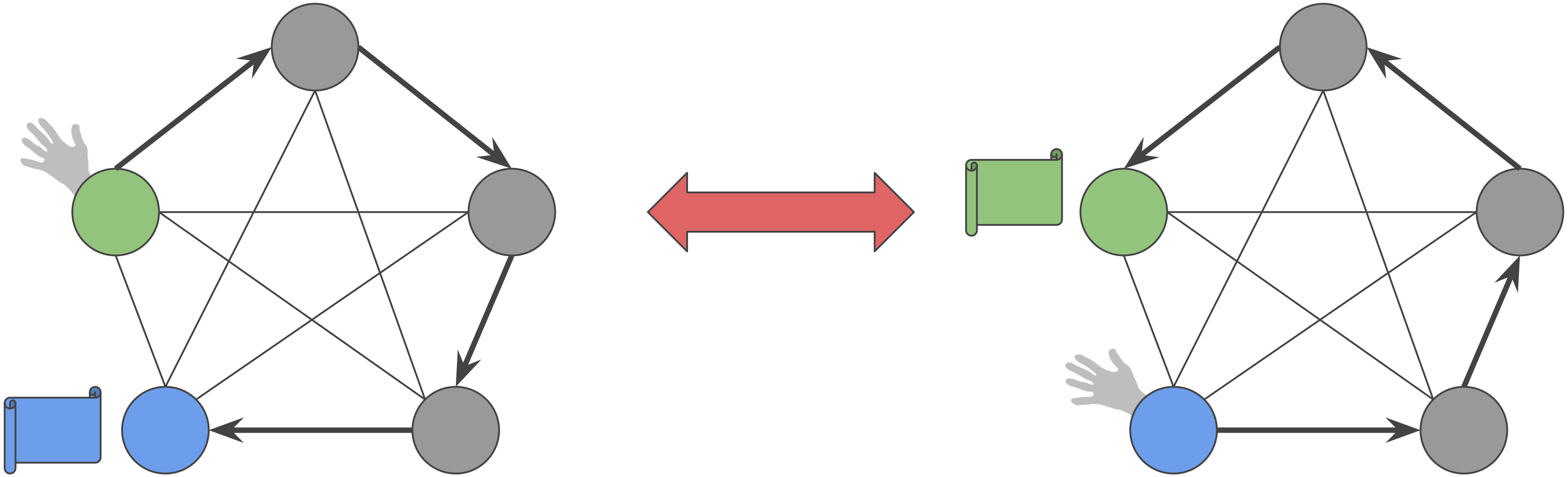
Arrow: Performance

- First “Find” reaches root <-> Nearest Node
- Everything “seems” to be reseted for next find
- Algorithm mimics Nearest Neighbour(NN) TSP
- $C_{NN} \leq \frac{3}{2}\left[\log_2 D\right] C_{Optimal}$
- Ordering Cost:
  - For “find” operations
  - latency? -> Well-space requests
  - |messages|? -> Alternating between 2 nodes
  - latency + |messages|



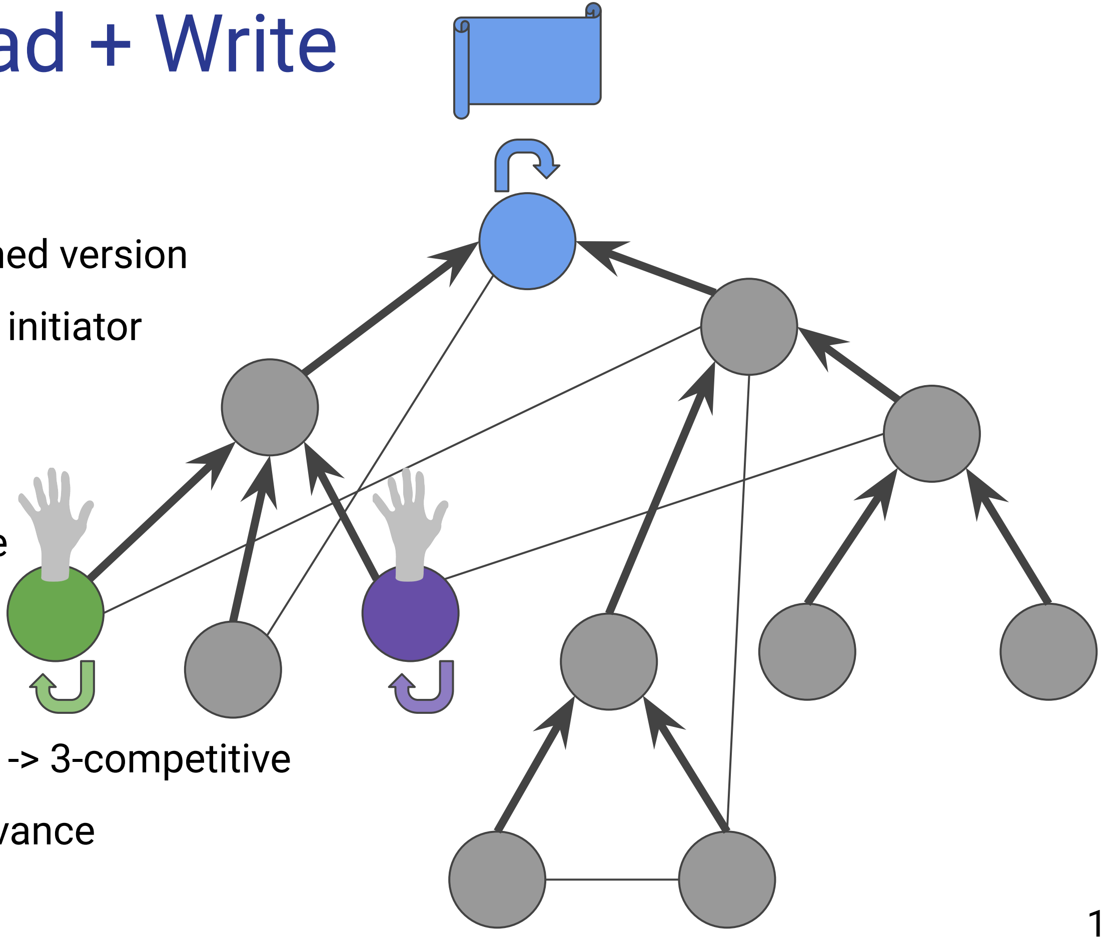
Arrow: Bad example

- Dependent on the choice of spanning tree
- Lack of auto-adjustments



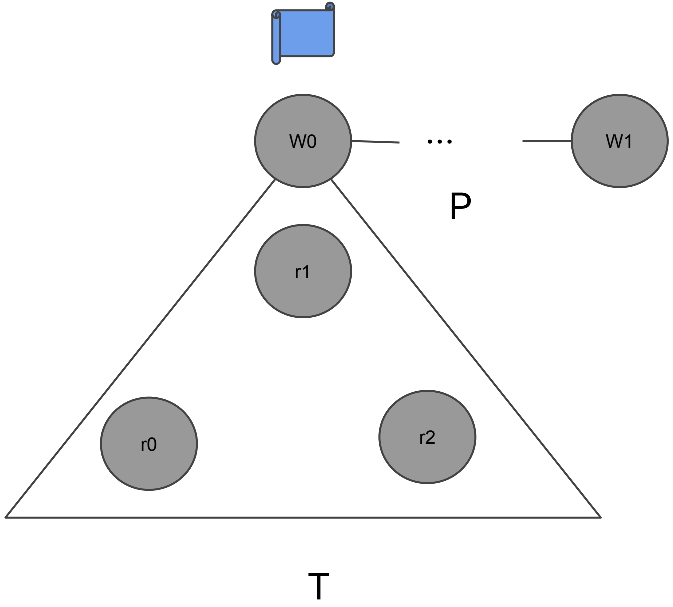
Arrow: Multiple Read + Write

- Read:
  - Follow arrows until a cached version
  - Cache on the way back to initiator
- Write:
  - Change the value locally
  - Follow arrows and reverse
  - Flag cached as obsolete
- Performance:
  - Only message complexity -> 3-competitive
  - Time -> Opt cach all in advance



Arrow: 3-competitiveness

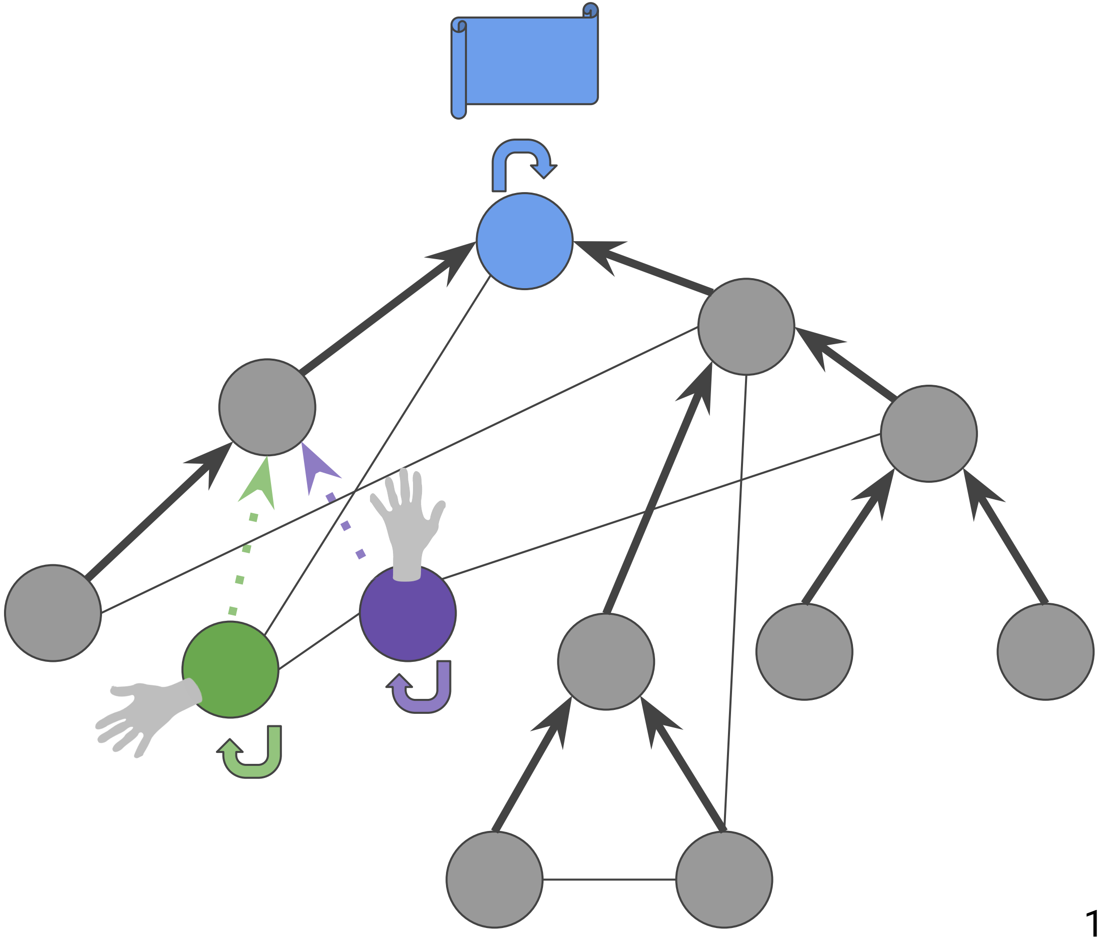
- w0,r0,r1,....w1
- Alg :
  - Read -> 2|T|
  - Write -> |T|+|P|
- OPT:
  - |T|+|P|



E2

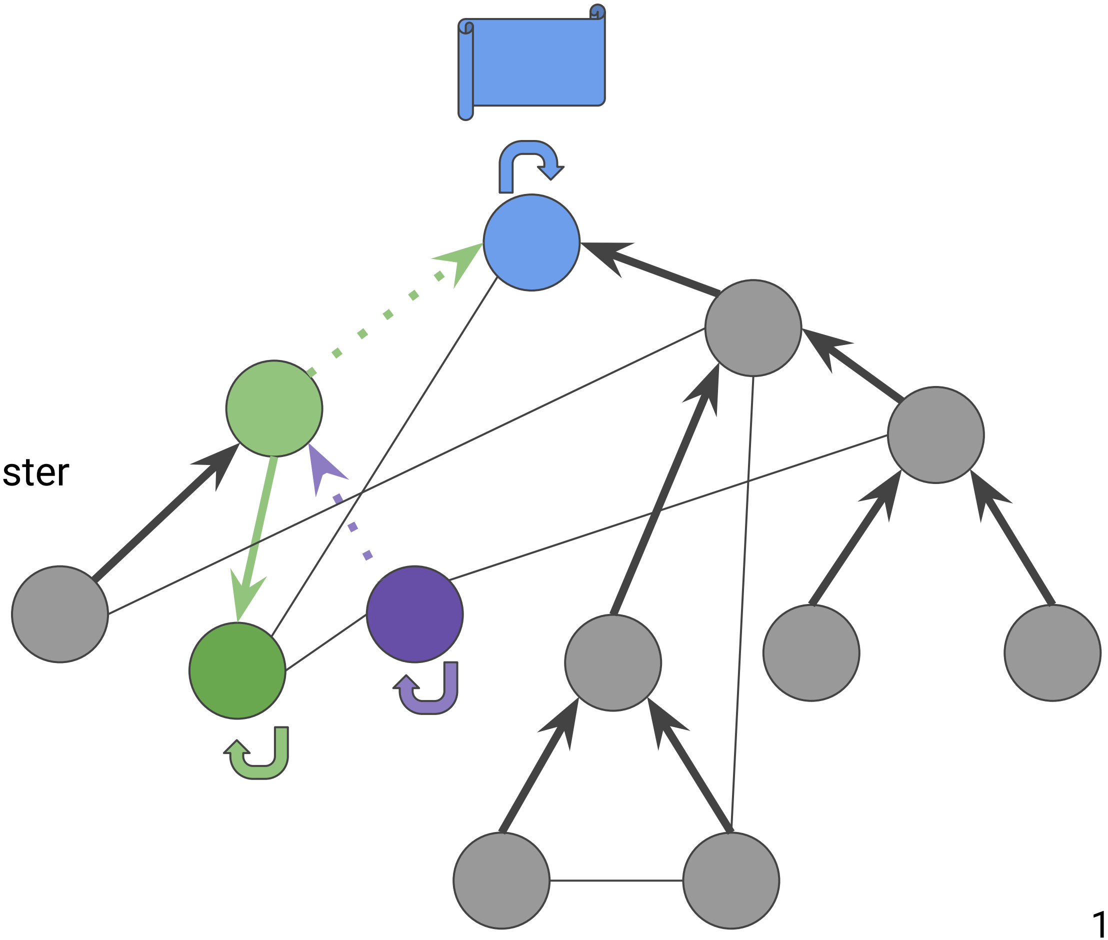
Ivy: Protocol

- Assume complete graph
- Start as Arrow



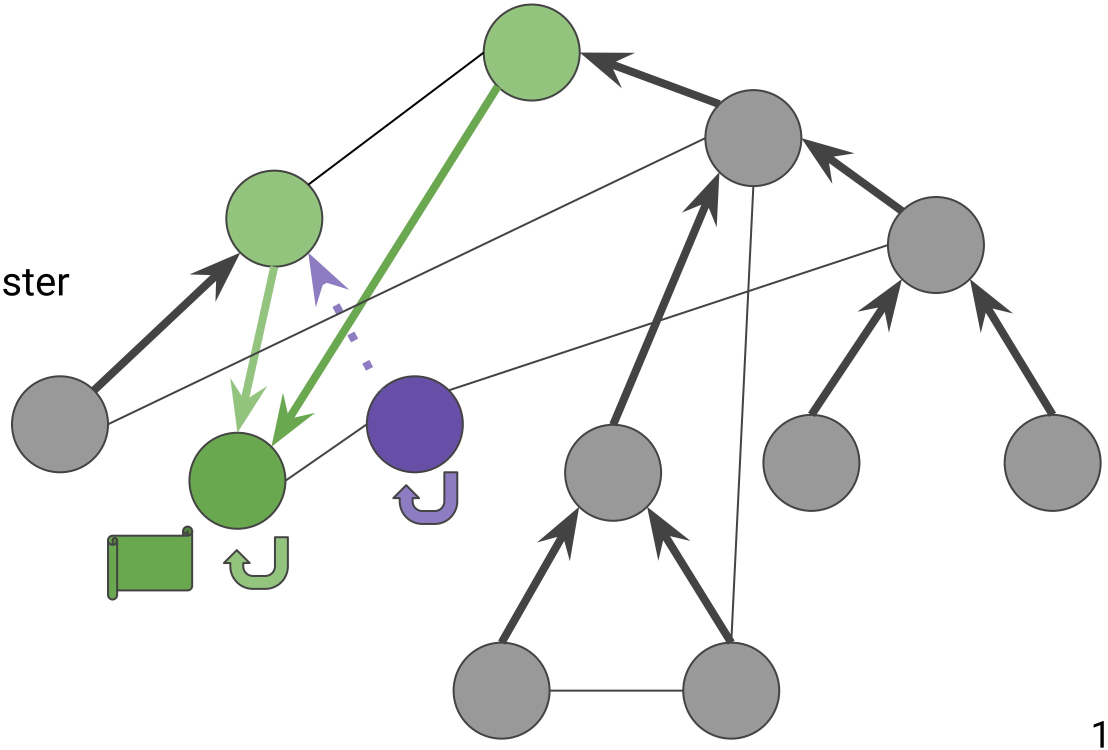
Ivy: Protocol

- Assume complete graph
- Start as Arrow
- Arrival:
  - Point directly to the requester



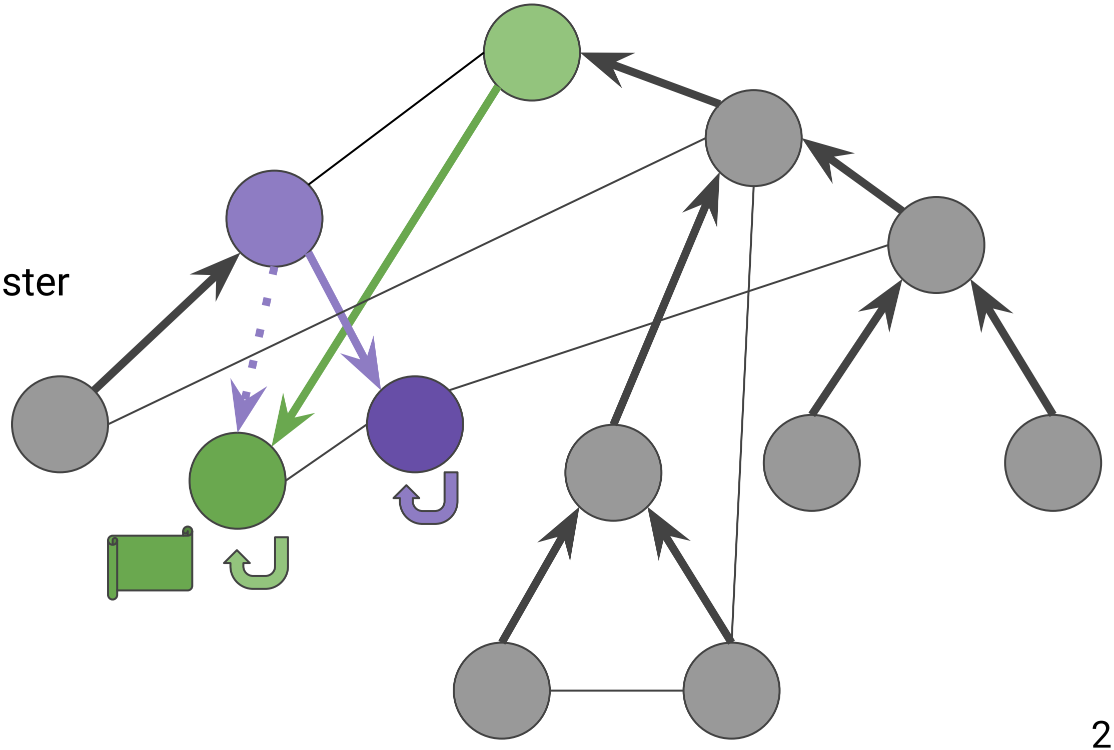
Ivy: Protocol

- Assume complete graph
- Start as Arrow
- Arrival:
  - Point directly to the requester



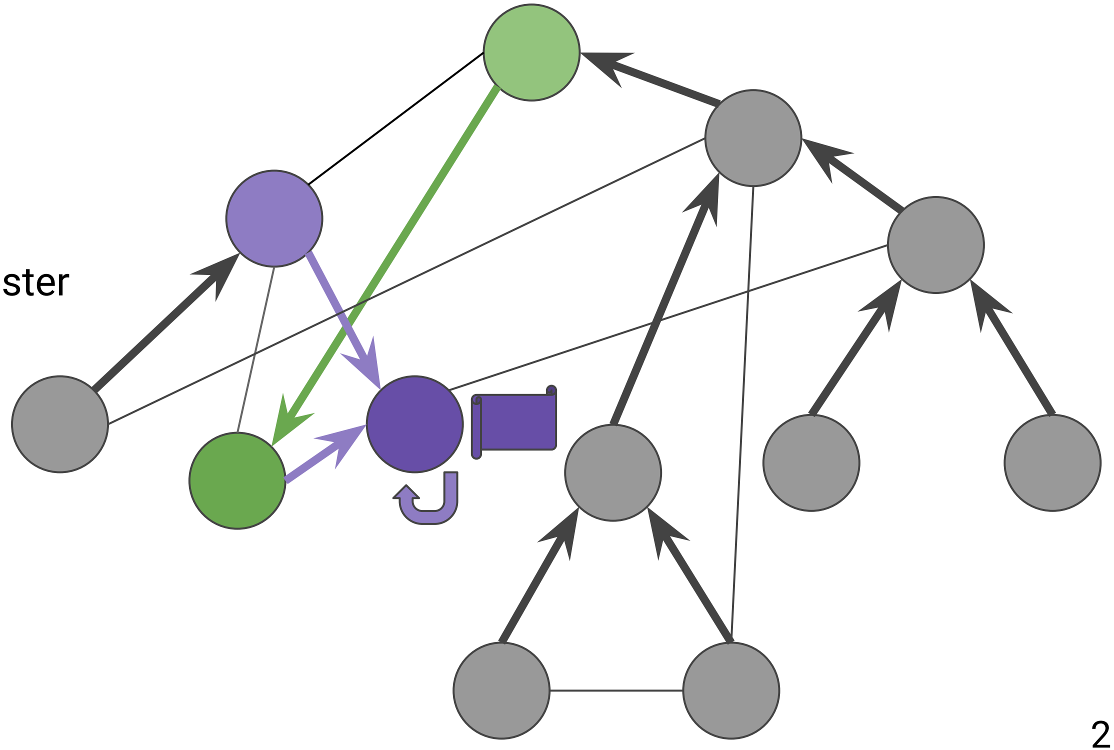
Ivy: Protocol

- Assume complete graph
- Start as Arrow
- Arrival:
    - Point directly to the requester

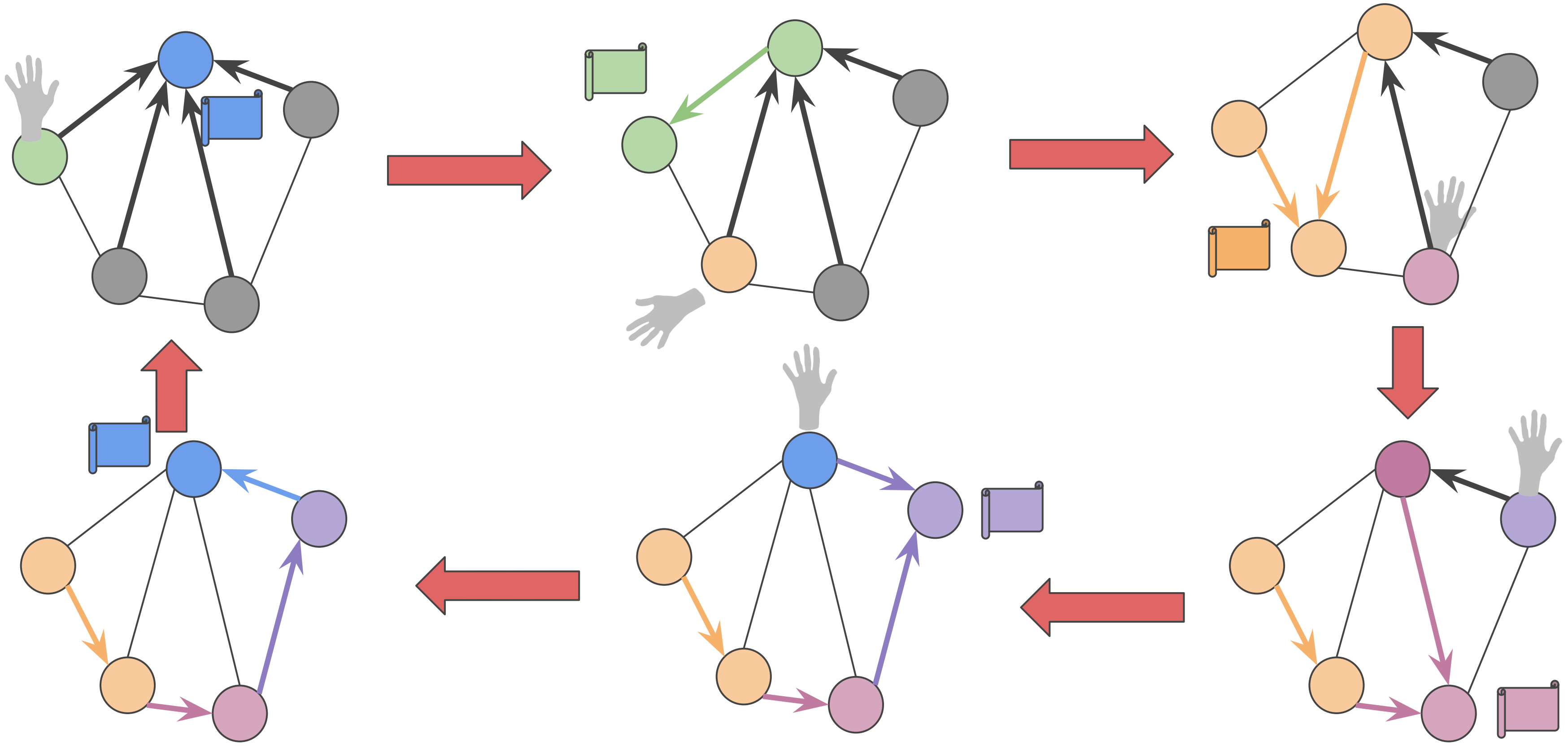


Ivy: Protocol

- Assume complete graph
- Start as Arrow
- Arrival:
  - Point directly to the requester



Ivy: Bad example



Ivy: Analysis

$$\Phi(T) = \sum_{u \in V} \frac{\log s(u)}{2}.$$

$$a_i = k_i - \Phi(T_{i-1}) + \Phi(T_i)$$

Cost of “i”th request

$$1 + \log(\alpha - 1)/2 \le \log \alpha$$

$$a_i = k_i + \frac{1}{2} \cdot \sum_{j=0}^{k_i - 1} \log \left(\frac{s_{j+1} - s_j}{s_j} \right)$$

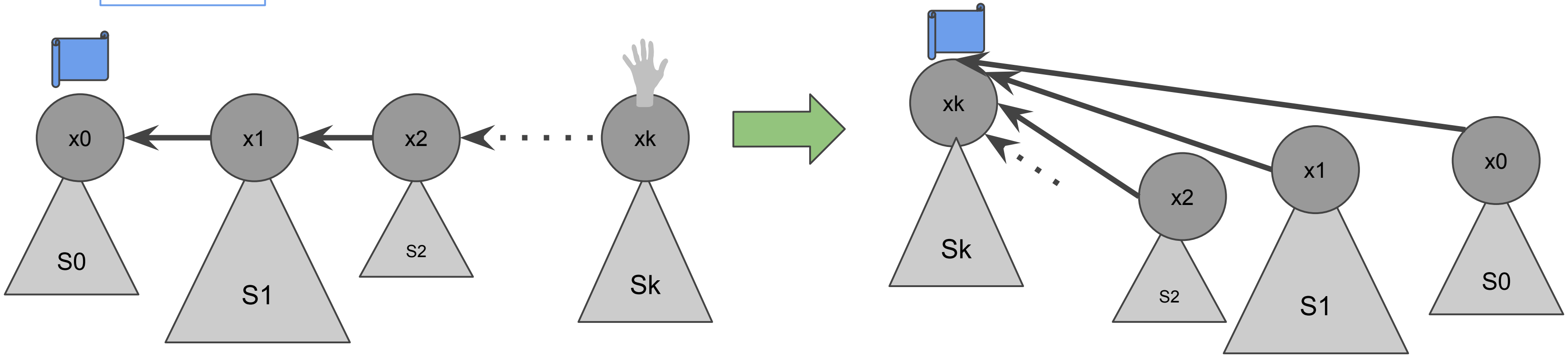
$$\sum_{i=1}^{m} a_i \ge \sum_{i=1}^{m} k_i.$$

$$a_i \le \sum_{j=0}^{k_i - 1} \log \frac{s_{j+1}}{s_j}$$

Size of “j”th subtree

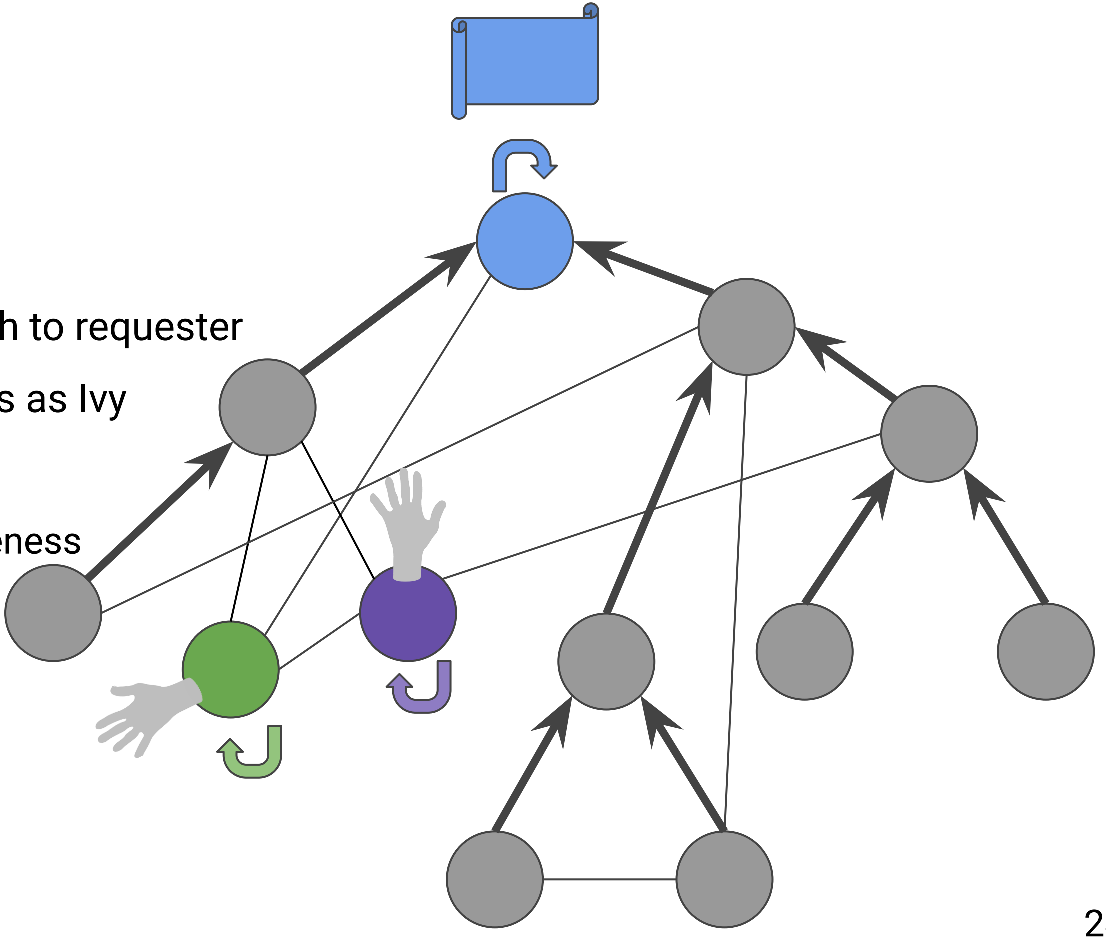
$$= \log s_{k_i} - \log s_0 \le \log n,$$

$$\sum_{i=1}^{m} k_i \le m \log n$$



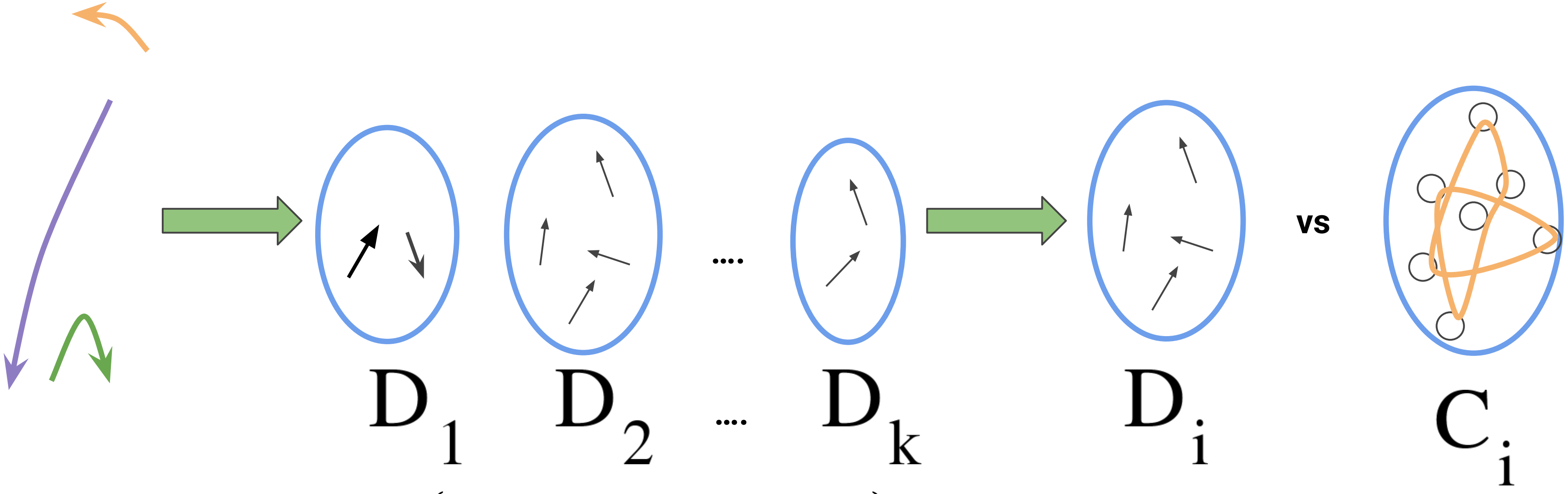
Arvy

- Start as Arrow
- On arrival:
  - Point to a node on the path to requester
- Same bad example and analysis as Ivy
- On ring:
  - Using bridges 5-competitiveness



Thank you

Arrow: Nearest Neighbour TSP



$D_i = \{(u,v) \mid 2^{i-1} d_s \le d(u,v) < 2^i d_s\}$

Smallest edge

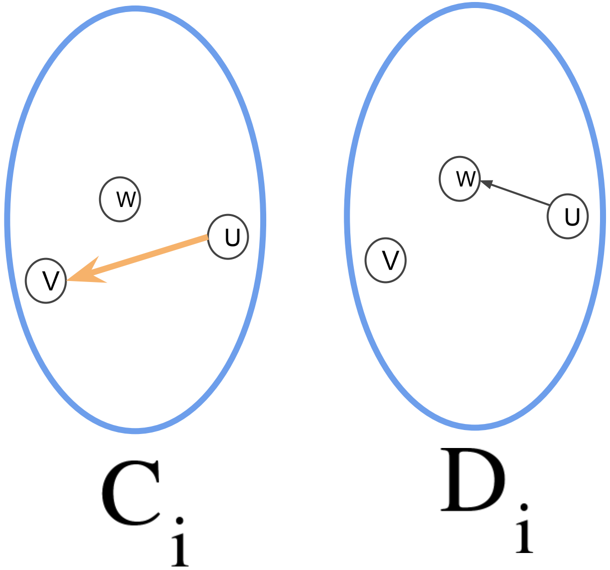
$V_i = \{v \mid v \in D_i\}$

C_i : Optimal TCP on V_i

$C_i < C_{opt}$

Triangle Inequality Assumption

Arrow: Nearest Neighbour TSP



$$\forall\, (u, w) \in D_i : d(u, w) \leq 2ds_i$$

$$\sum_{e \in D_i} d(e) \leq \frac{3}{2} \sum_{e \in C_i} d(e)$$

$$\forall (u, v) \in C_i \, \exists \, (u, w) \in D_i .\ d(u, w) < d(u, v)$$