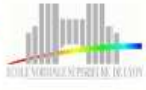
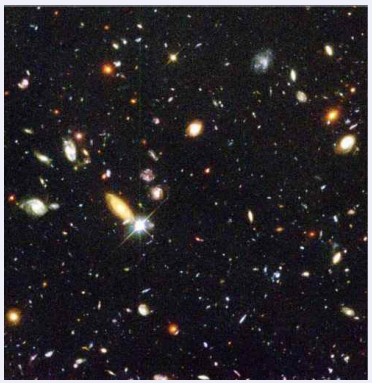
Peer-to-Peer Prefix Tree for Large Scale Service Discovery

Cédric Tedeschi

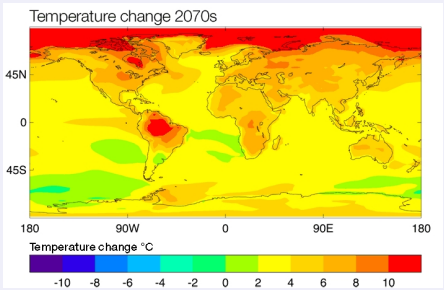
October 2, 2008



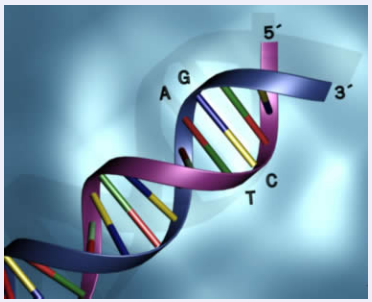

Computing Needs



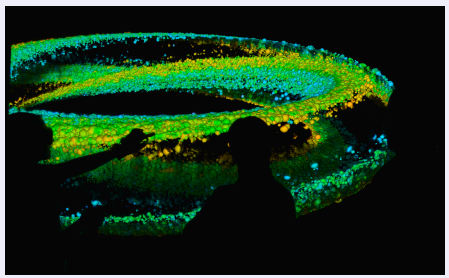
Cosmology



Climate prediction



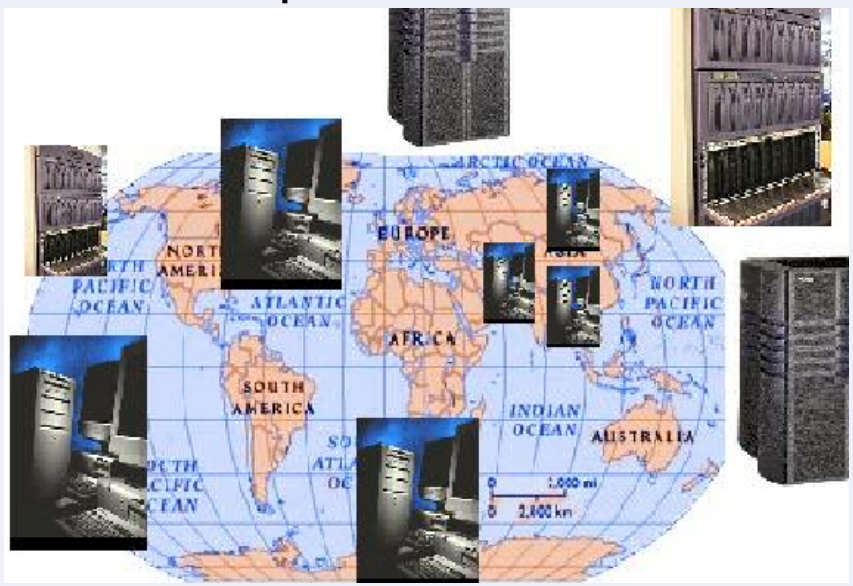
Genomics

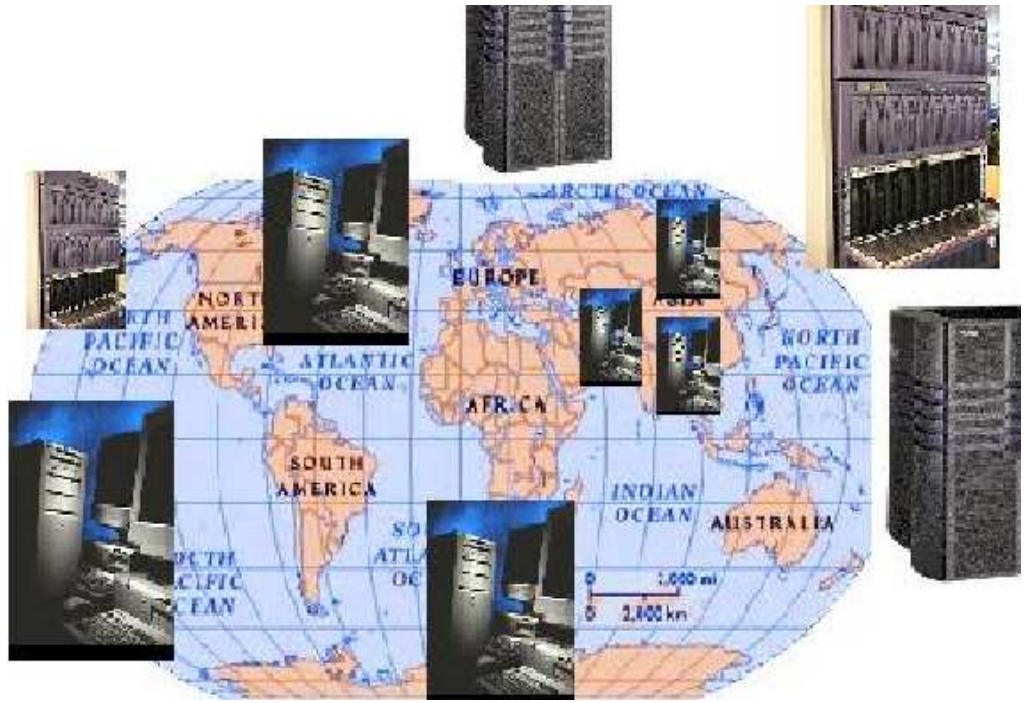


Nuclear security

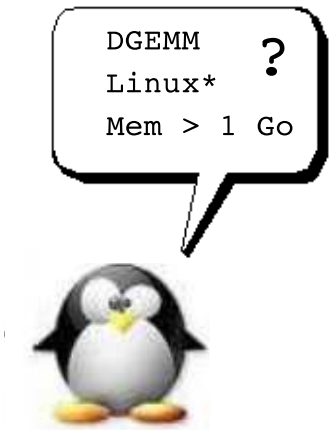
Computing Power

Computational Grids

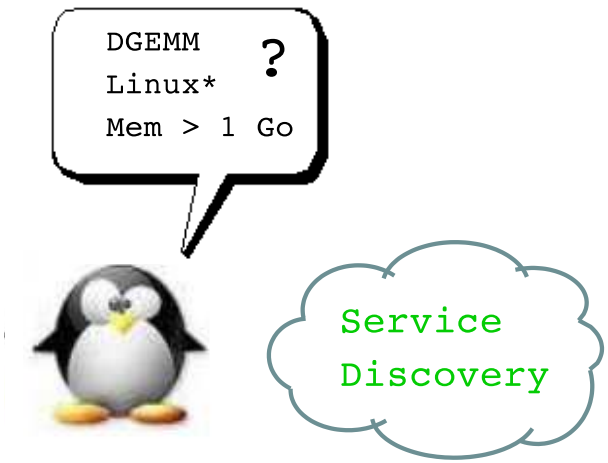


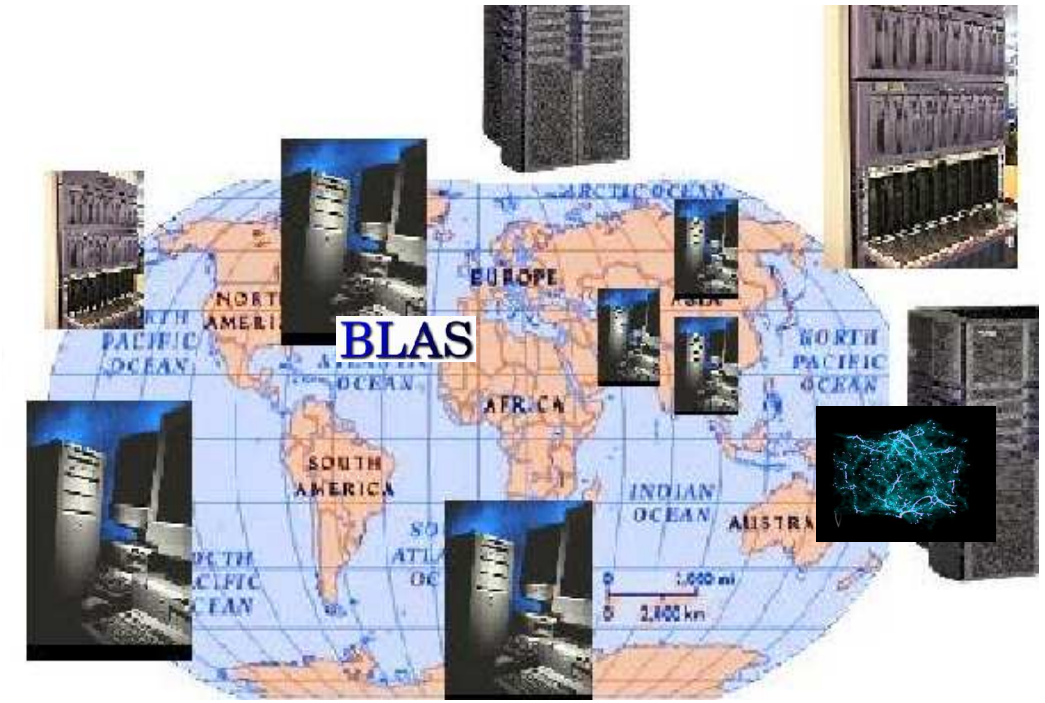

EUROPE
AFRICA
SOUTH AMERICA
INDIAN OCEAN
AUSTRALIA
NORTH PACIFIC OCEAN
ATLANTIC OCEAN

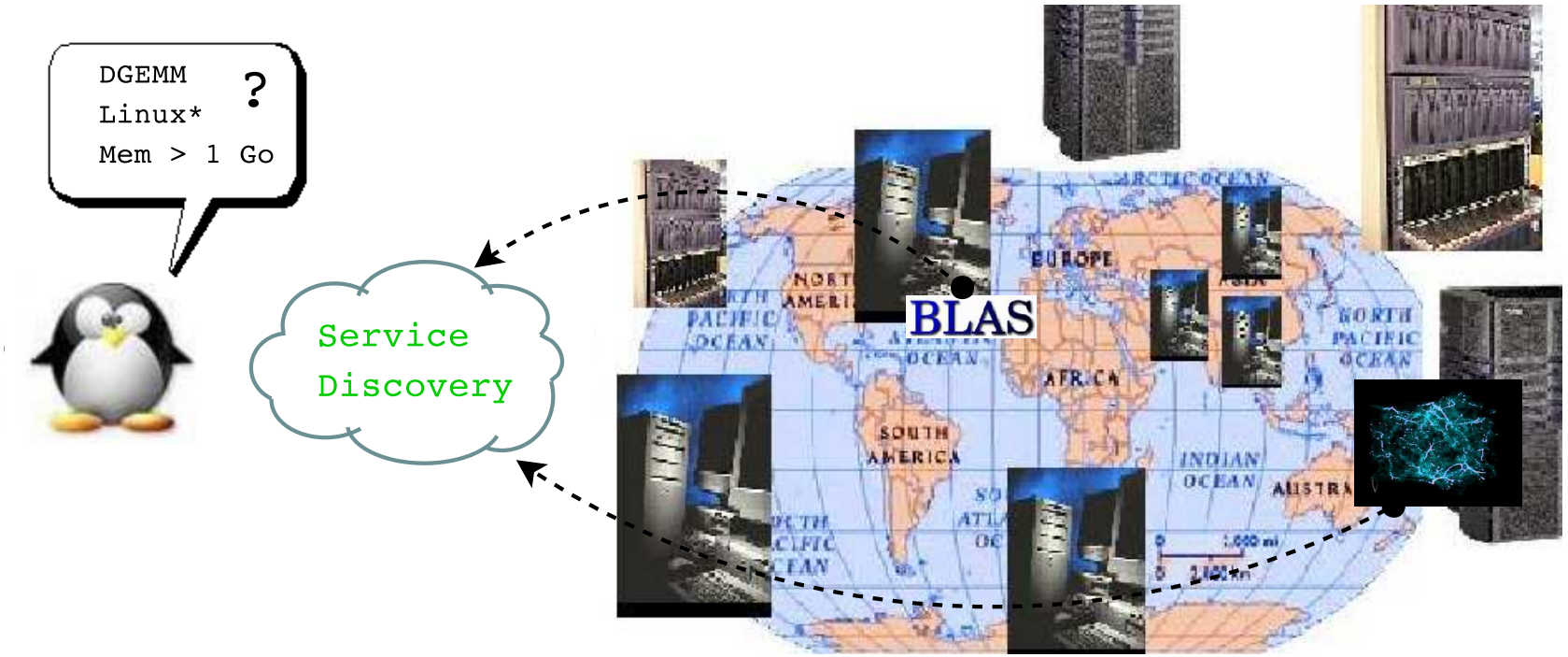

BLAS
EUROPE
AFRICA
SOUTH AMERICA
INDIAN OCEAN
NORTH PACIFIC OCEAN

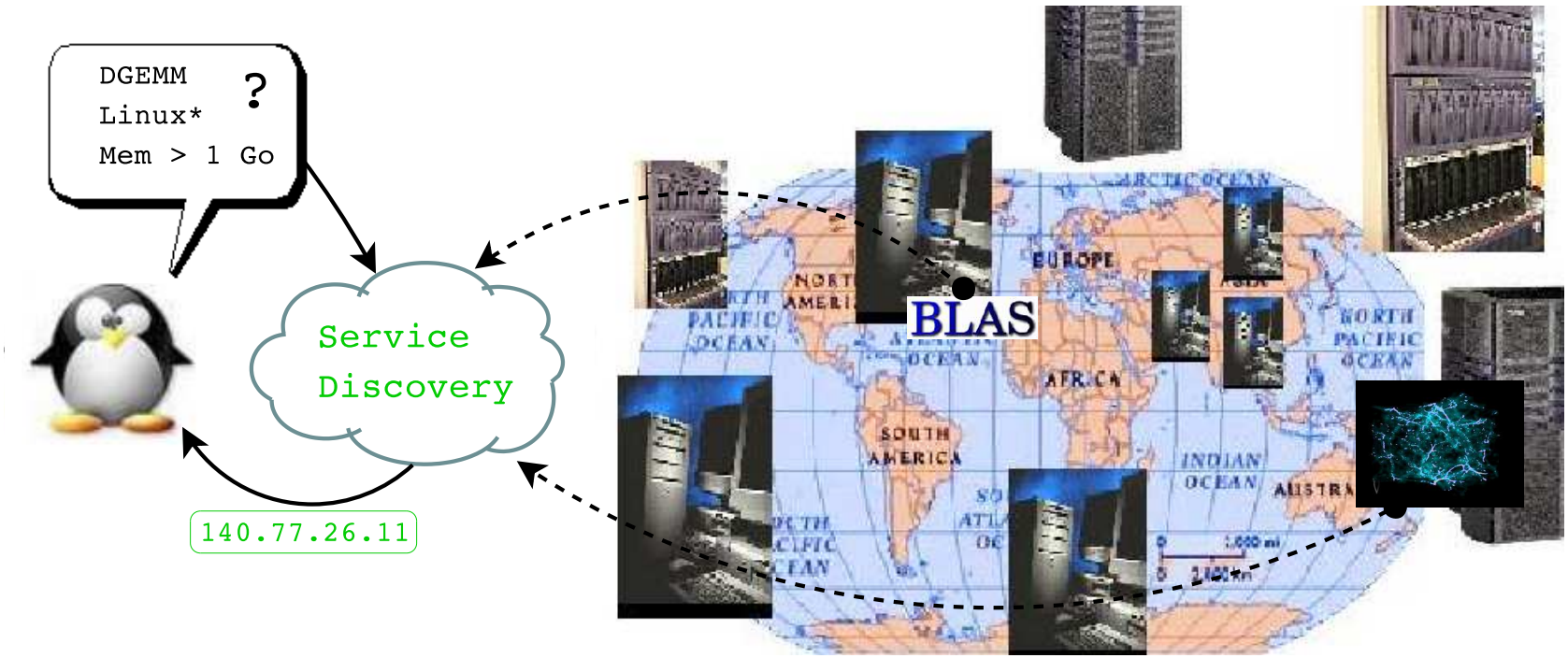

DGEMM
Linux*
Mem > 1 Go
?


BLAS
EUROPE
AFRICA
SOUTH AMERICA
INDIAN OCEAN
NORTH PACIFIC OCEAN

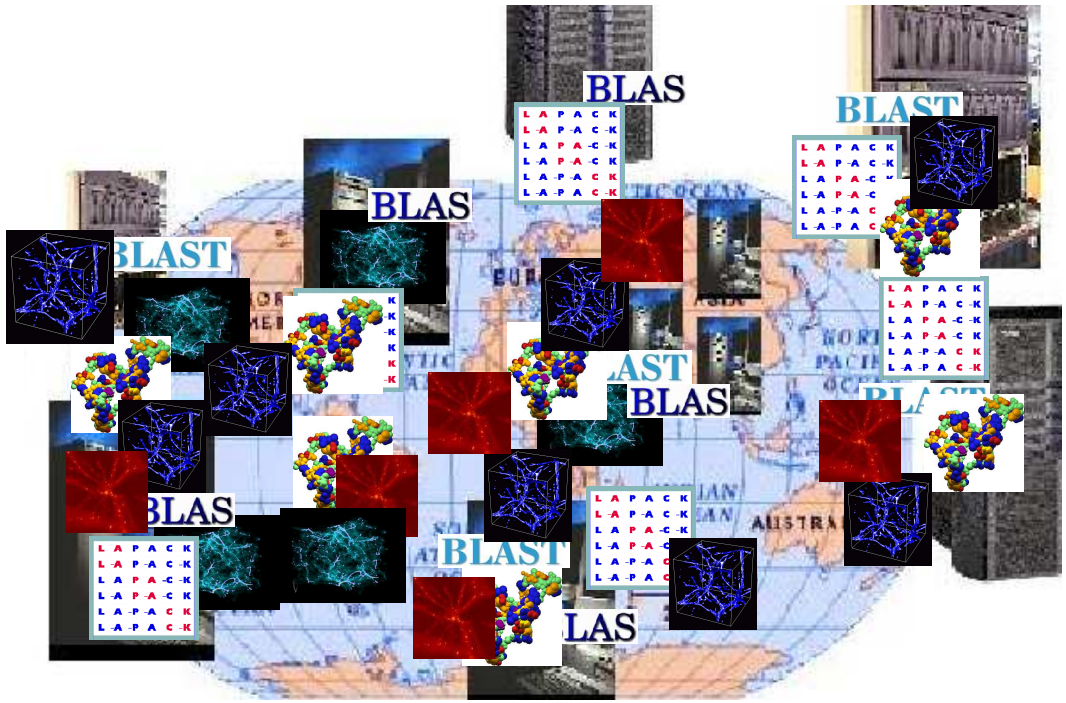

DGEMM
Linux*
Mem > 1 Go
?
Service
Discovery

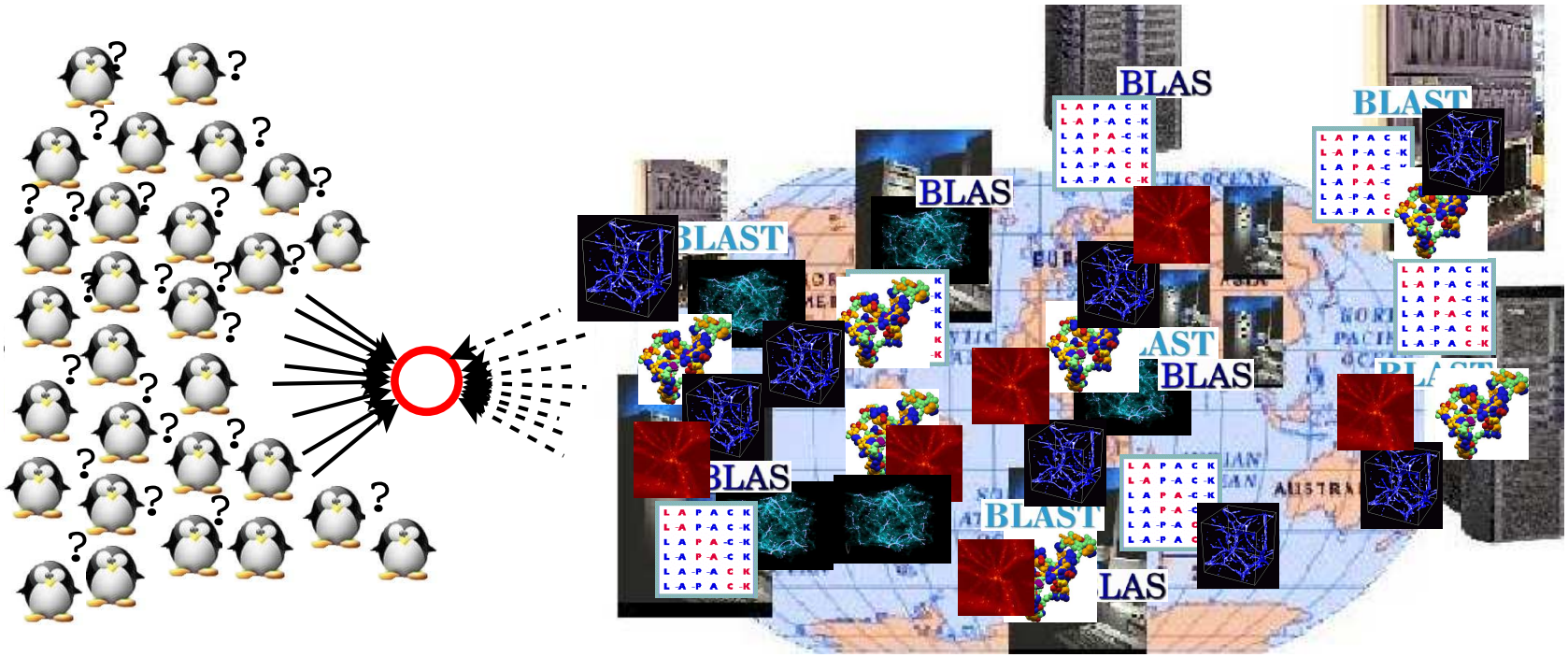

BLAS
EUROPE
AFRICA
SOUTH
AMERICA
INDIAN
OCEAN
PACIFIC
OCEAN


DGEMM
Linux*
Mem > 1 Go
?
Service
Discovery
BLAS
EUROPE
AFRICA
SOUTH
AMERICA
INDIAN
OCEAN

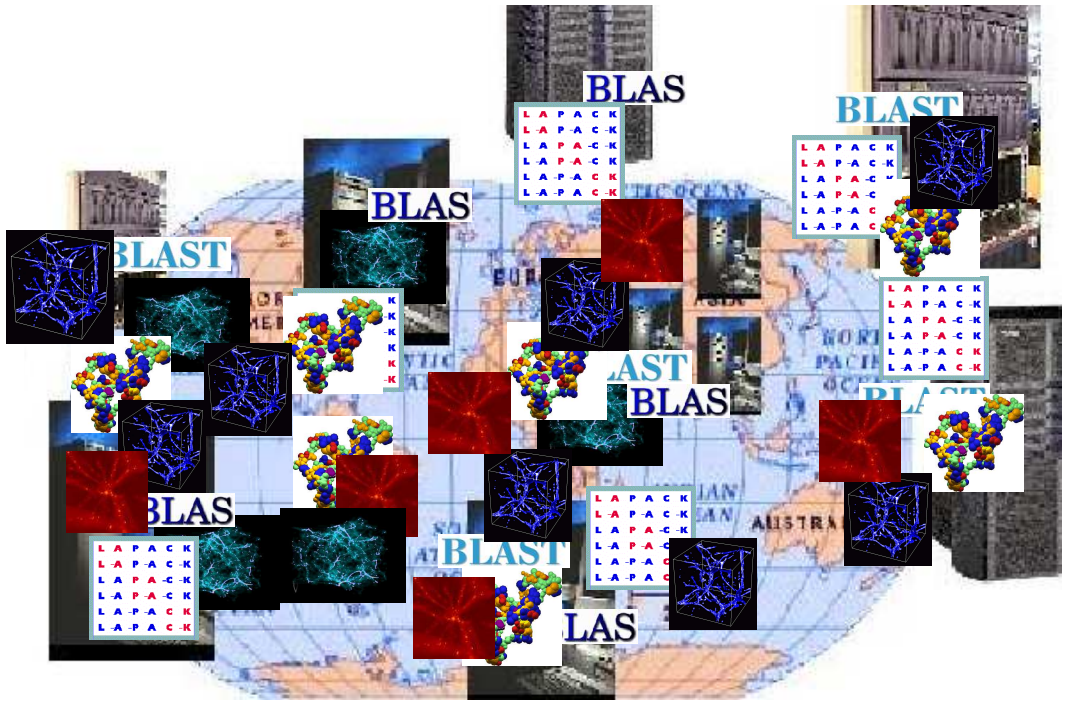

DGEMM
Linux*
Mem > 1 Go
?
Service
Discovery
140.77.26.11
BLAS
EUROPE
AFRICA
SOUTH
AMERICA
INDIAN
OCEAN


Service
Discovery


BLAS
BLAST
BLAS
BLAST
BLAS
BLAS
BLAST
LAS


BLAS
BLAST
BLAS
BLAST
BLAS
BLAS
BLAST
LAS


P2P


BLAS
BLAST
BLAS
BLAST
BLAS
BLAS
BLAST
LAS

P2P Information Retrieval (review)

Unstructured (Gnutella)

- Flooding
- Cost
- Non-exhaustiveness

Structured (DHTs)

- Routing
- Exhaustiveness
- Scalability
  - Logarithmic state
  - Logarithmic path
- Drawbacks
  - Exact queries only

P2P Information Retrieval (review)

Unstructured (Gnutella)

- Flooding
- Cost
- Non-exhaustiveness

Structured (DHTs)

- Routing
- Exhaustiveness
- Scalability
  - Logarithmic state
  - Logarithmic path
- Drawbacks
  - Exact queries only

P2P Information Retrieval (review)

Unstructured (Gnutella)

- Flooding
- Cost
- Non-exhaustiveness

Structured (DHTs)

- Routing
- Exhaustiveness
- Scalability
  - Logarithmic state
  - Logarithmic path
- Drawbacks
  - Exact queries only

Objectives

A P2P structured solution with

- Expressiveness
  - Range queries
  - Automatic completion
  - Multi-attribute queries
- Efficiency in a P2P environment
  - Heterogeneous $\Rightarrow$ Load balancing
  - Dynamic $\Rightarrow$ Fault-tolerance

Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

DLPT : A trie-based indexing system

Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height

Definition

Each node is the greatest common prefix of any pair of its children

DLPT : A trie-based indexing system

Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height

DLPT : A trie-based indexing system

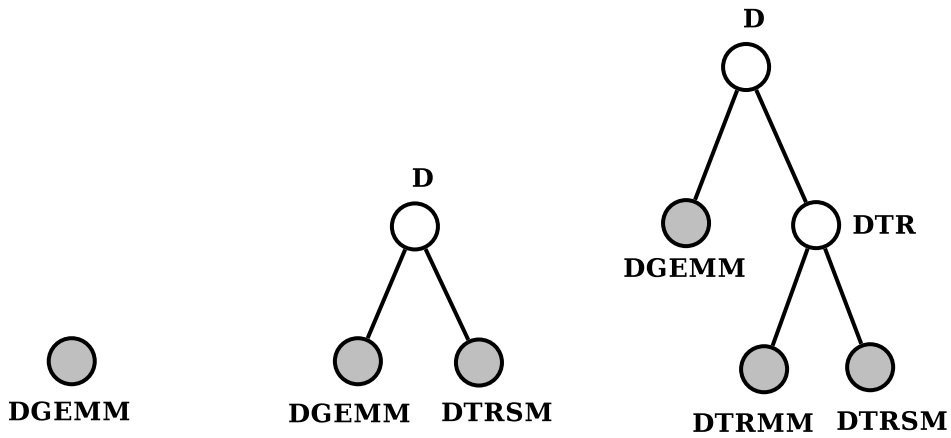
Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height

DLPT : A trie-based indexing system

Distributed Logical Structure

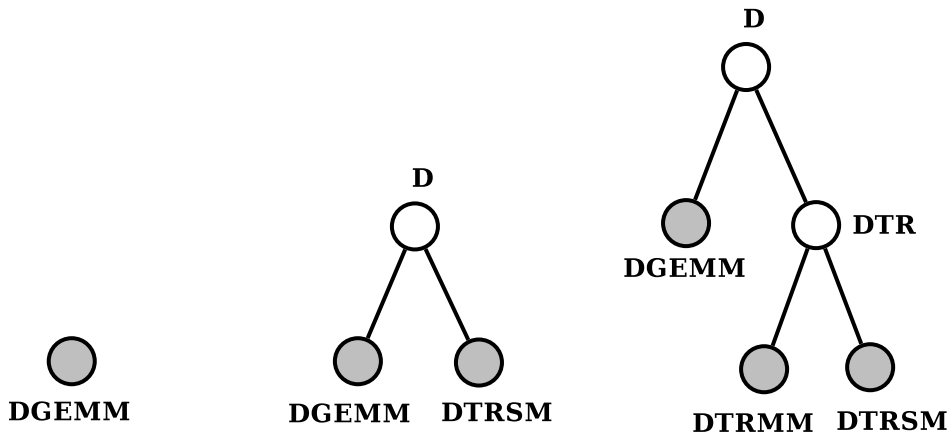
- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height



DLPT : A trie-based indexing system

Distributed Logical Structure

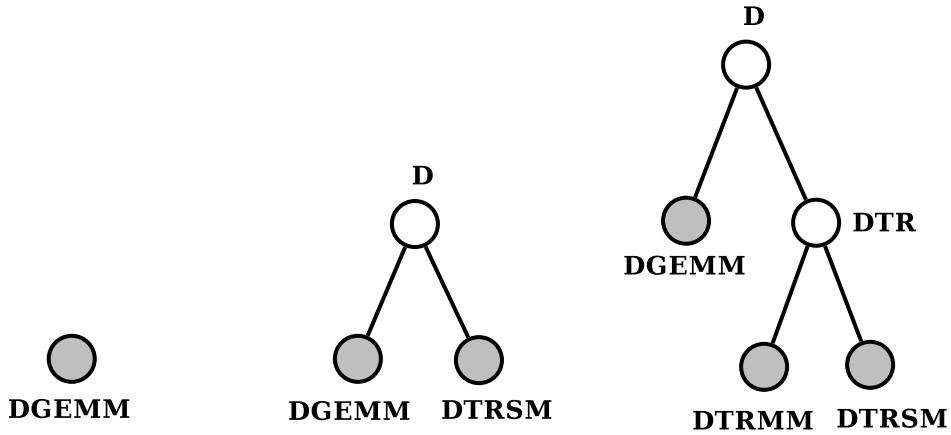
- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height



DLPT : A trie-based indexing system

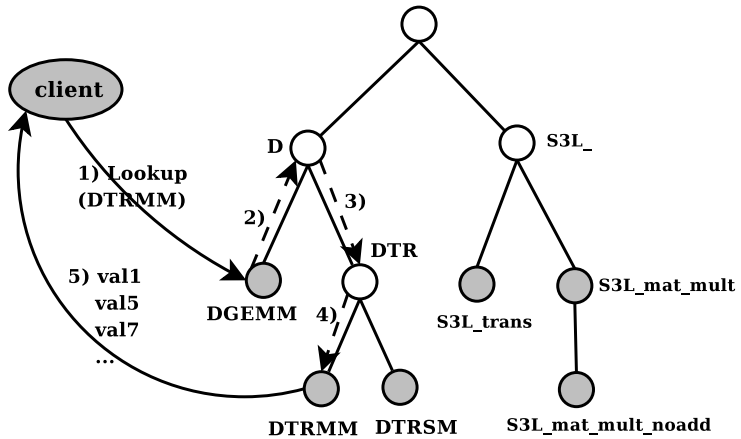
Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height



Lookup

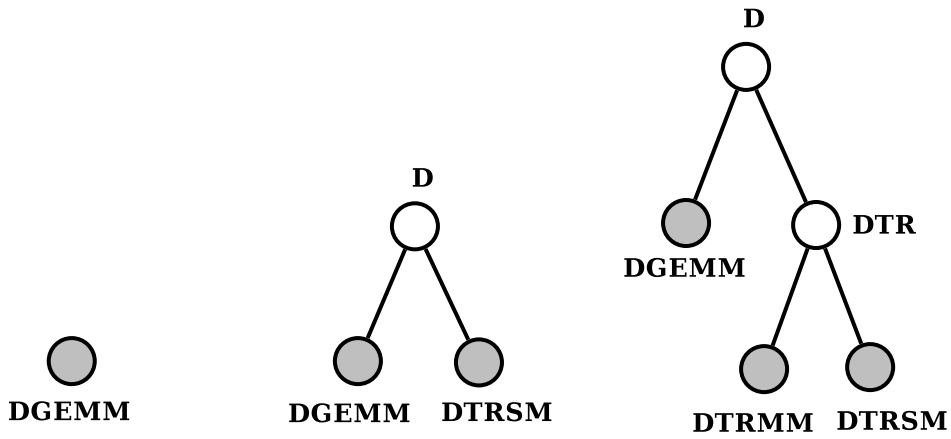
- Exact match
- Autocompetion
- Range queries



DLPT : A trie-based indexing system

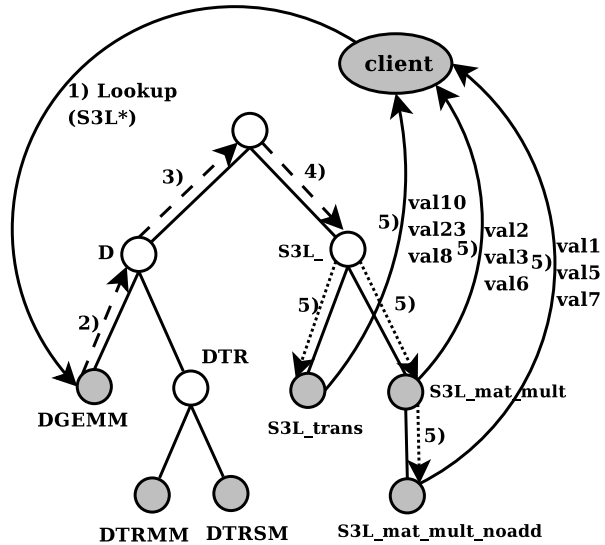
Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height



Lookup

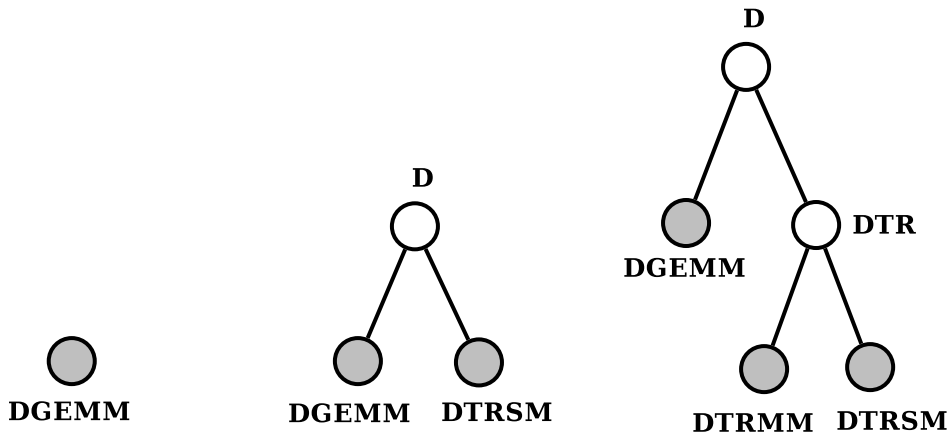
- Exact match
- Autocompetion
- Range queries



DLPT : A trie-based indexing system

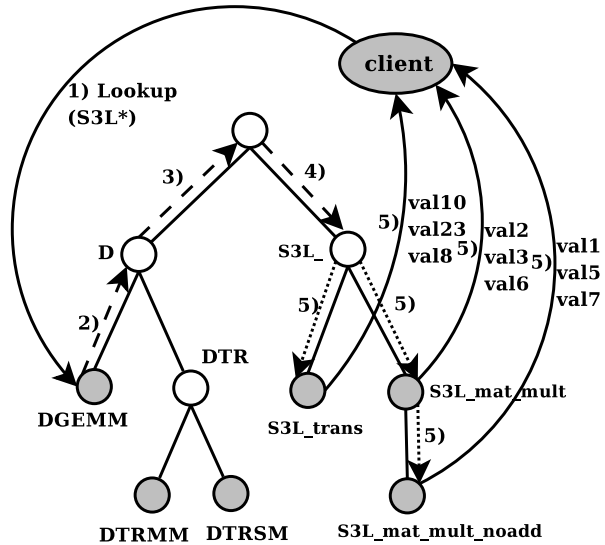
Distributed Logical Structure

- Greatest Common Prefix Tree
- Dynamically constructed
- Bounded degree and height



Lookup

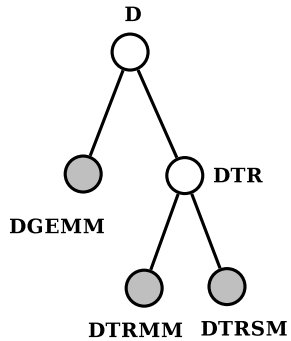
- Exact match
- Autocompetion
- Range queries



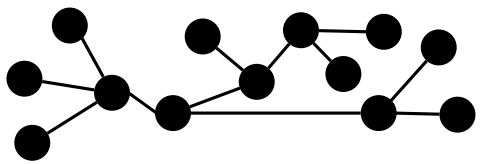
Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

The Mapping Problem

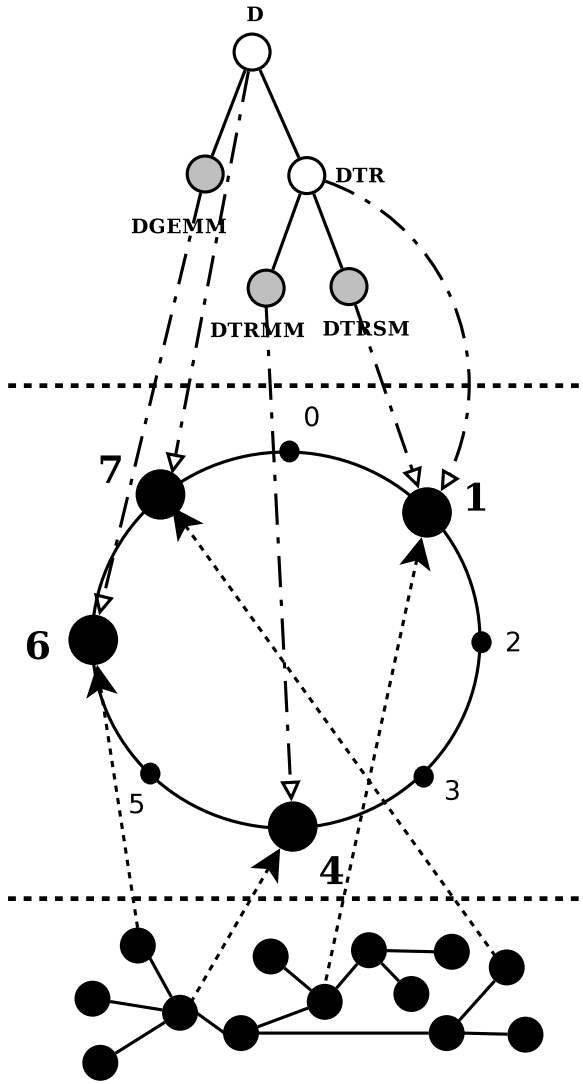


- Associate a **Node** to a **Peer**
- An underlying DHT



The Mapping Problem

- Associate a **Node** to a **Peer**
- An underlying DHT

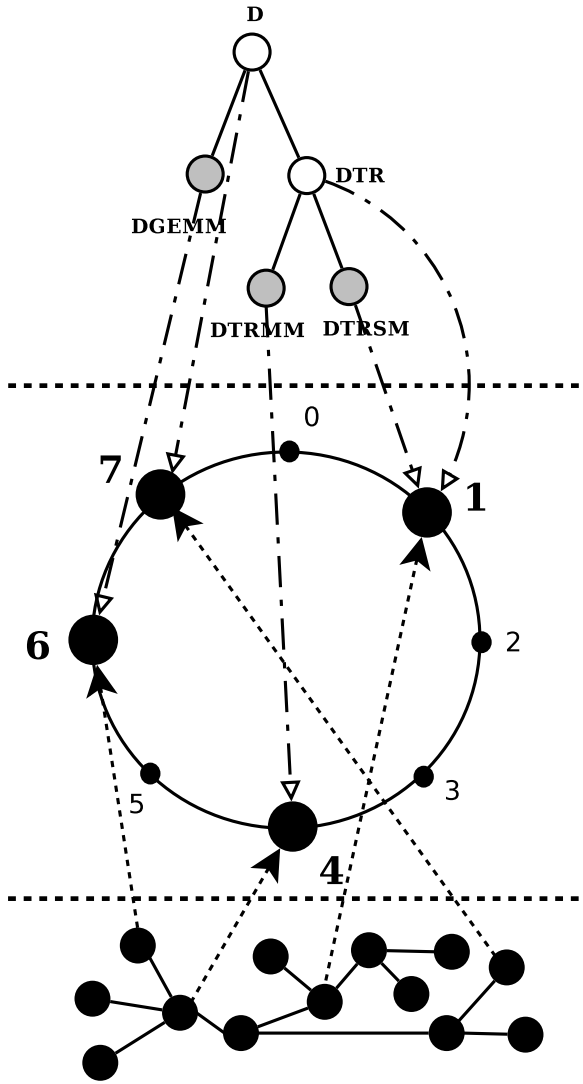


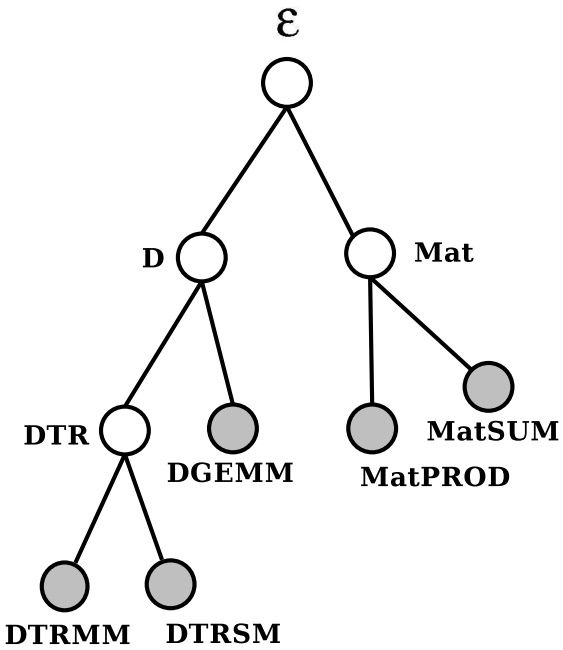
The Mapping Problem

Drawbacks

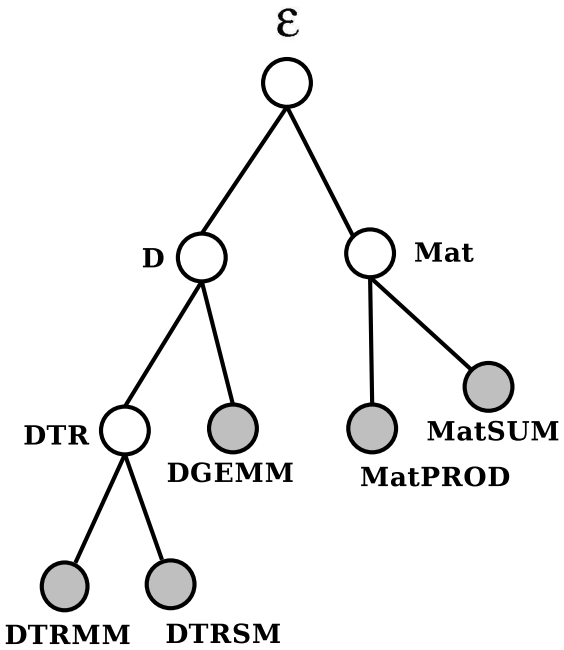
Need for a DHT

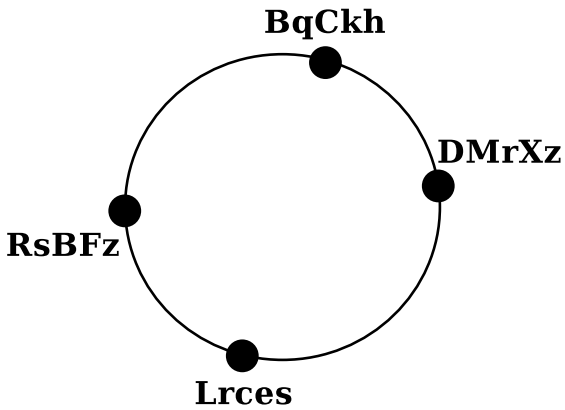
- Costly
- Random mapping (No clustering)



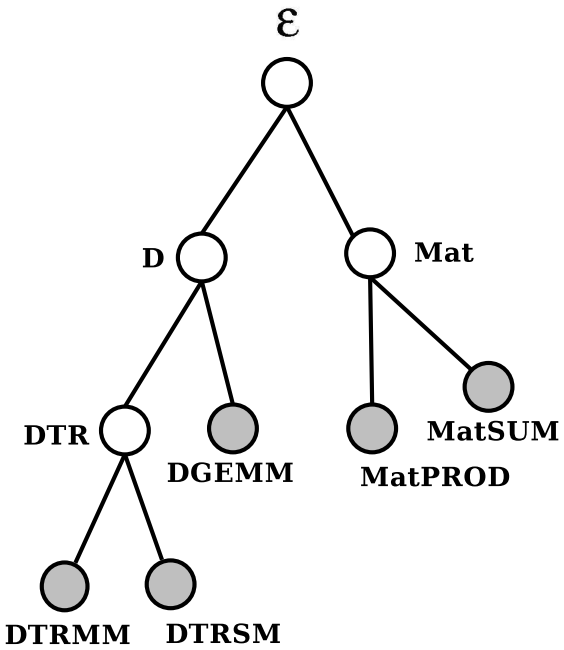


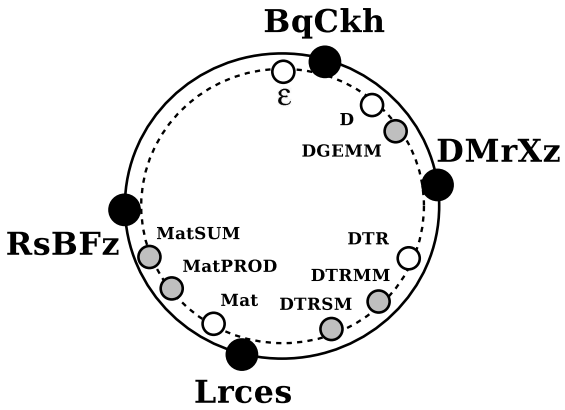
- Build a ring over peers
- Each peer maintains at least one node
- Map using **consistent hashing**
- Clustering



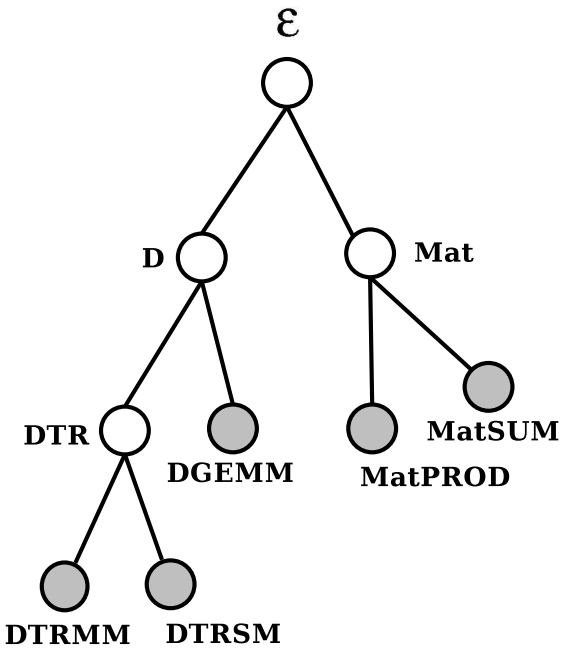


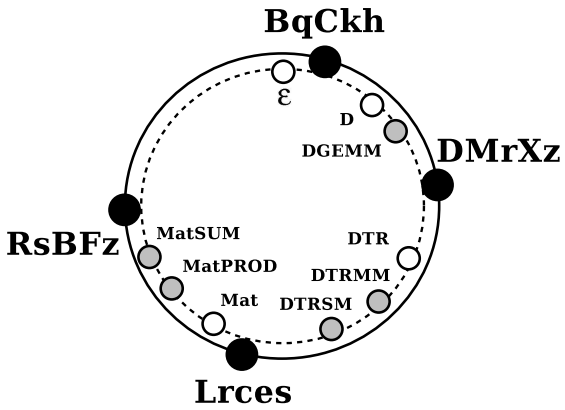
- Build a ring over peers
- Each peer maintains at least one node
- Map using **consistent hashing**
- Clustering

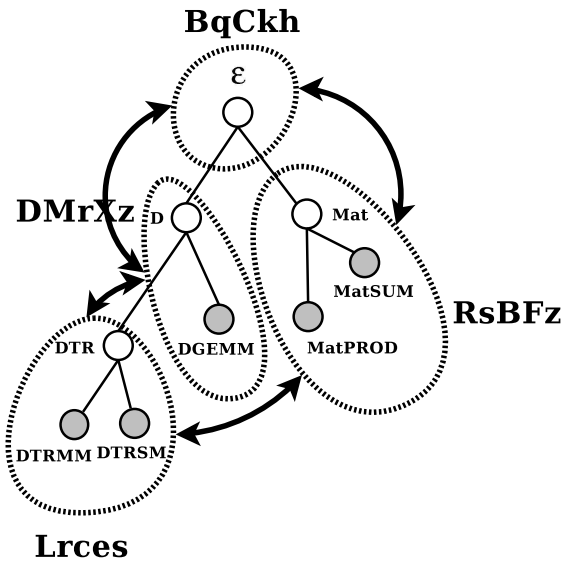




- Build a ring over peers
- Each peer maintains at least one node
- Map using **consistent hashing**
- Clustering

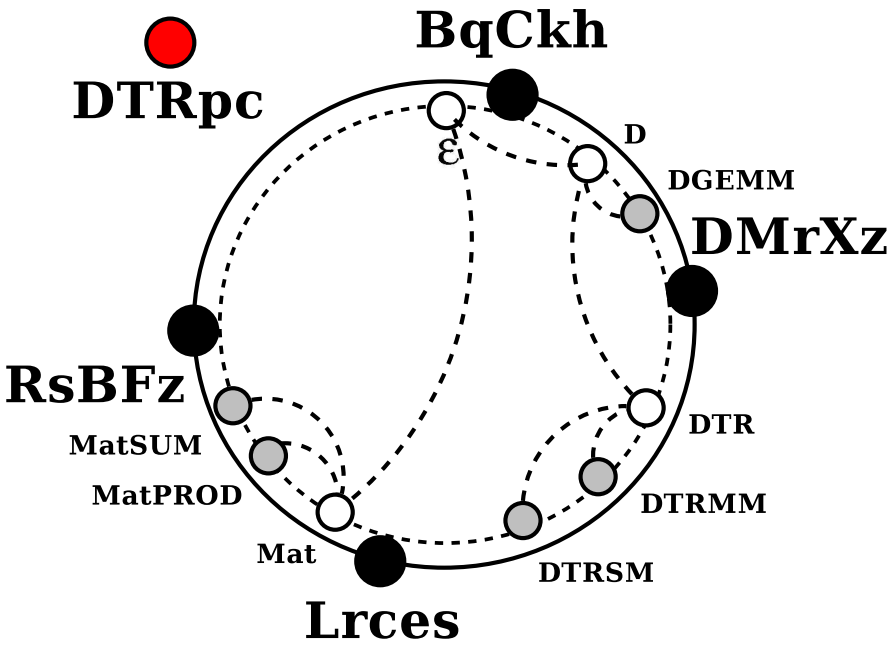






- Build a ring over peers
- Each peer maintains at least one node
- Map using **consistent hashing**
- Clustering

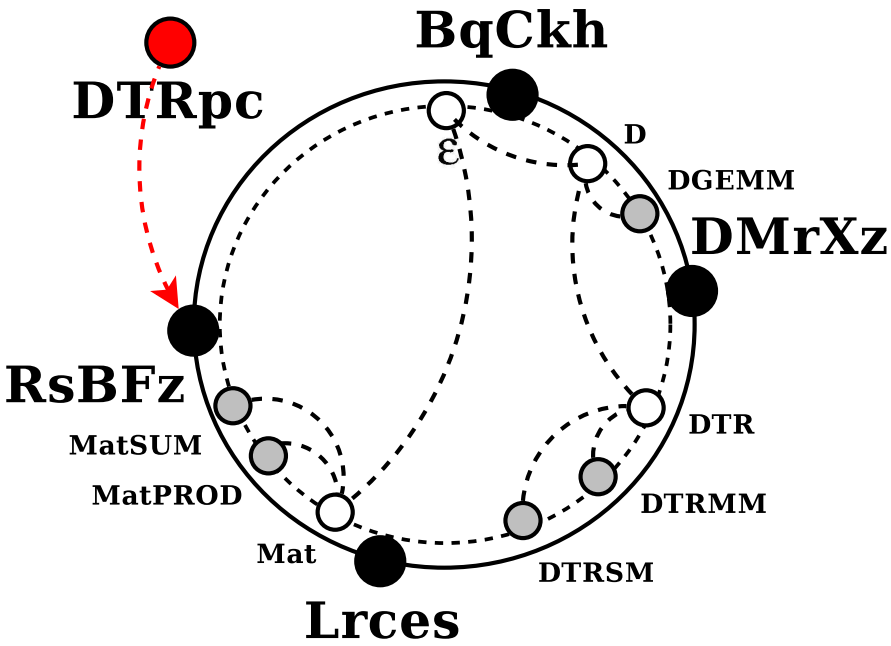
Insertion of a New Peer



Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

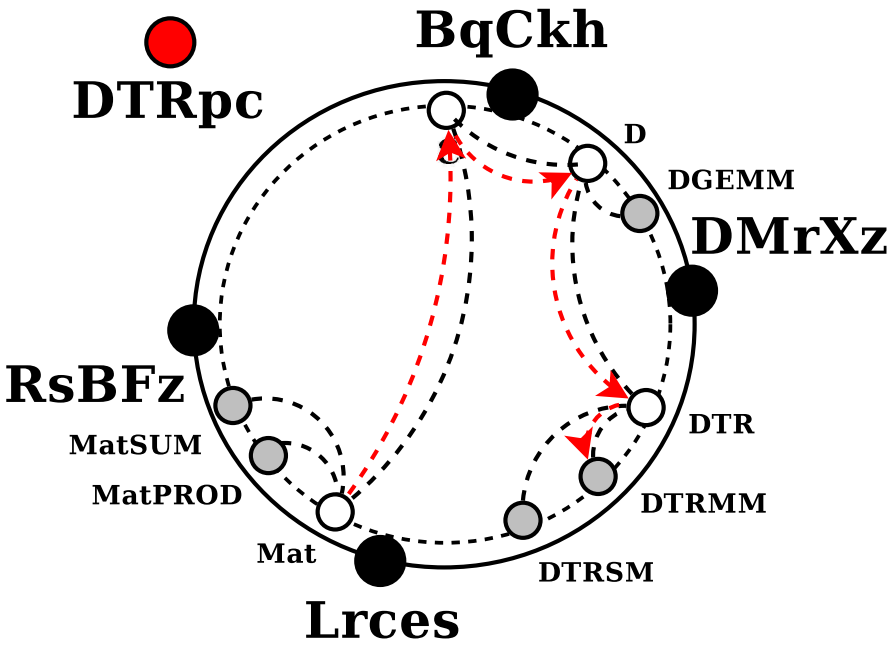
Insertion of a New Peer



Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

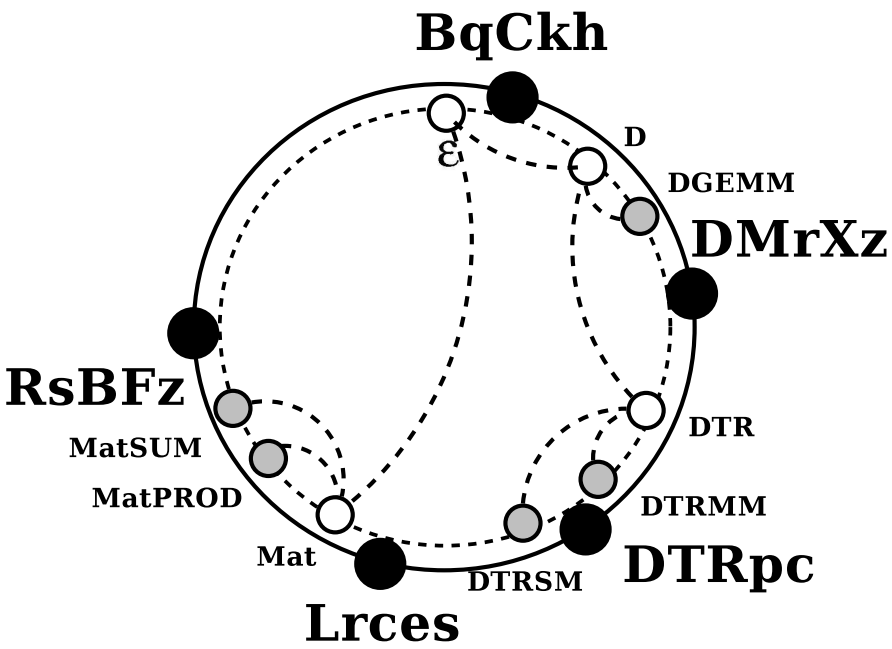
Insertion of a New Peer



Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

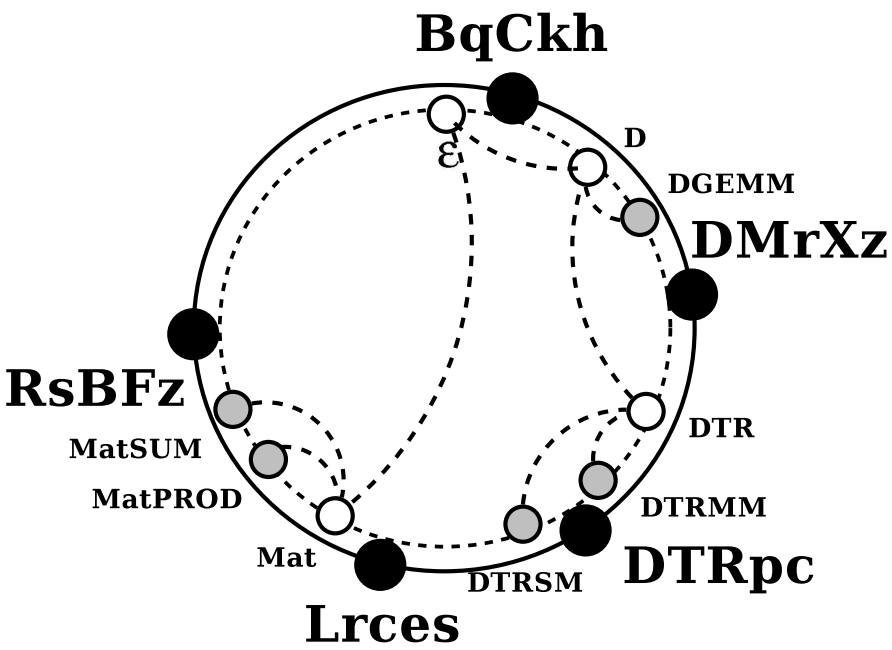
Insertion of a New Peer



Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

Insertion of a New Peer



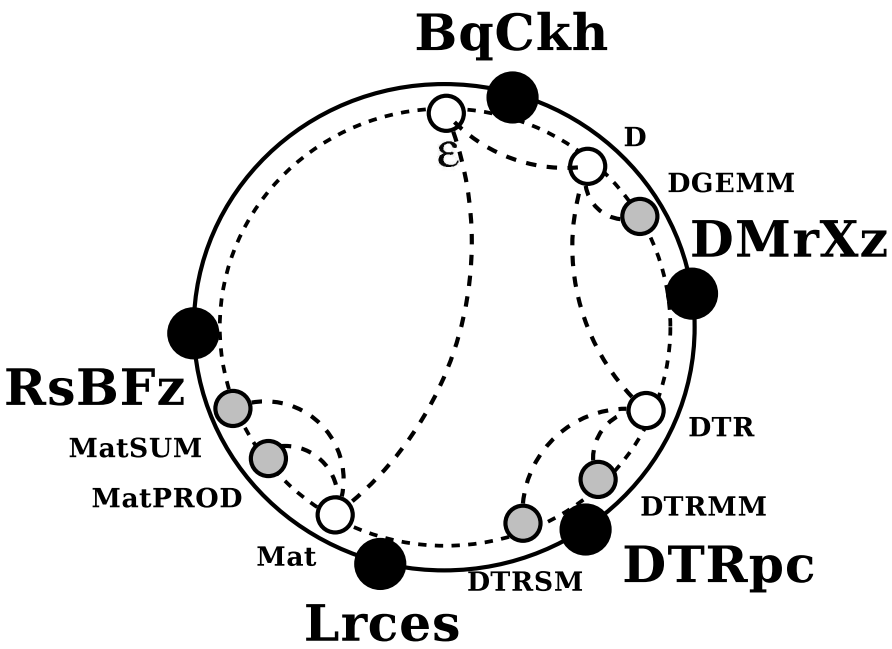
Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

Join Complexities

$O(Trie\ complexities)$

Insertion of a New Peer



Join Algorithm

1. Random contact peer
2. Initiating the tree routing
3. Routing in the tree
4. Successor of P is P_n or $succ(P_n)$

Join Complexities

$O(Trie\ complexities)$

Load balancing

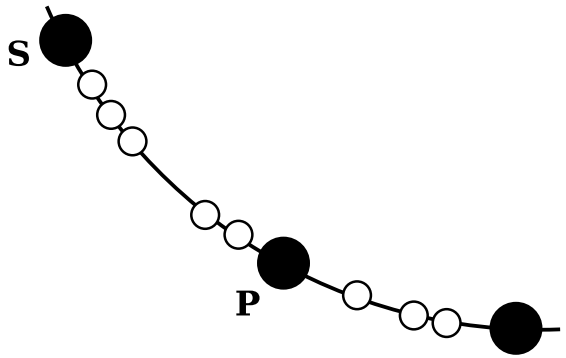
- Depth of nodes
- Popularity of services
- Capacity of peers

Load balancing heuristics - related work (DHTs)

Each peer is assigned a set of *items*

- Karger and Ruhl, 2001
  - Periodic random item balancing
  - Homogeneity of peer capacities
- Godfrey *et al.*, 2003
  - Periodic item redistribution
  - Semi-centralized
- Ledlie and Seltzer, 2005
  - Chooses the best location for a joining peer among k
  - Heterogeneity of peer capacities and data popularity

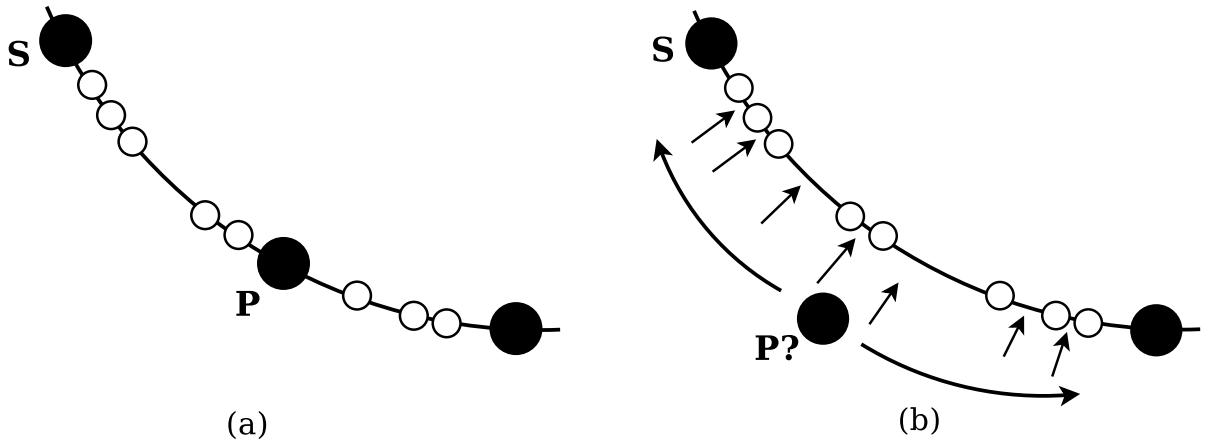
A novel heuristic : **Max Local Throughput**



(a)

End of period τ
S get load information
from P

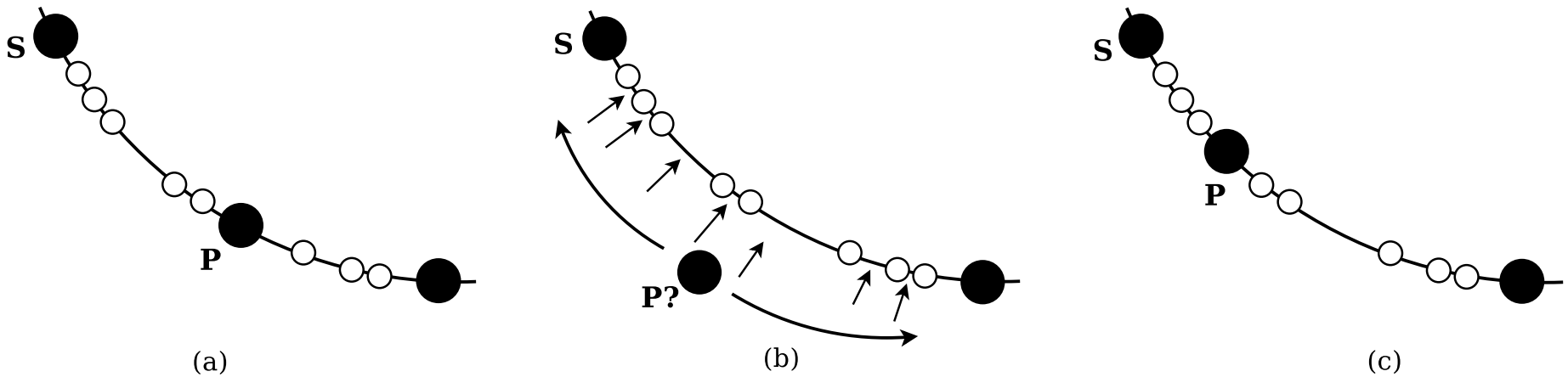
A novel heuristic : **Max Local Throughput**



End of period τ
S get load information from P

Search for a new locally optimal distribution

A novel heuristic : **Max Local Throughput**



End of period τ
S get load information from P

Search for a new locally optimal distribution

Starting the next period $\tau + 1$

A novel heuristic : **Max Local Throughput**

- At the end of period τ
- Two adjacent peers S and P with capacity C_S and C_P
- ν_S^τ and ν_P^τ the sets of nodes currently managed by S and P
- $L_S^\tau = \sum_{n \in \nu_S^\tau} l_n$
- $L_P^\tau = \sum_{n \in \nu_P^\tau} l_n$
- $T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P)$
- Maximize the throughput of $\tau + 1$ based on information of τ
- Find $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes $T_{S,P}^{\tau+1}$.

A novel heuristic : **Max Local Throughput**

- At the end of period τ
- Two adjacent peers S and P with capacity C_S and C_P
- ν_S^τ and ν_P^τ the sets of nodes currently managed by S and P
- $L_S^\tau = \sum_{n \in \nu_S^\tau} l_n$
- $L_P^\tau = \sum_{n \in \nu_P^\tau} l_n$
- $T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P)$
- Maximize the throughput of $\tau + 1$ based on information of τ
- Find $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes $T_{S,P}^{\tau+1}$.

A novel heuristic : **Max Local Throughput**

- At the end of period τ
- Two adjacent peers S and P with capacity C_S and C_P
- ν_S^τ and ν_P^τ the sets of nodes currently managed by S and P
- $L_S^\tau = \sum_{n \in \nu_S^\tau} l_n$
- $L_P^\tau = \sum_{n \in \nu_P^\tau} l_n$
- $T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P)$
- Maximize the throughput of $\tau + 1$ based on information of τ
- Find $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes $T_{S,P}^{\tau+1}$.

A novel heuristic : **Max Local Throughput**

- At the end of period τ
- Two adjacent peers S and P with capacity C_S and C_P
- ν_S^τ and ν_P^τ the sets of nodes currently managed by S and P
- $L_S^\tau = \sum_{n \in \nu_S^\tau} l_n$
- $L_P^\tau = \sum_{n \in \nu_P^\tau} l_n$
- $T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P)$
- Maximize the throughput of $\tau + 1$ based on information of τ
- Find $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes $T_{S,P}^{\tau+1}$.

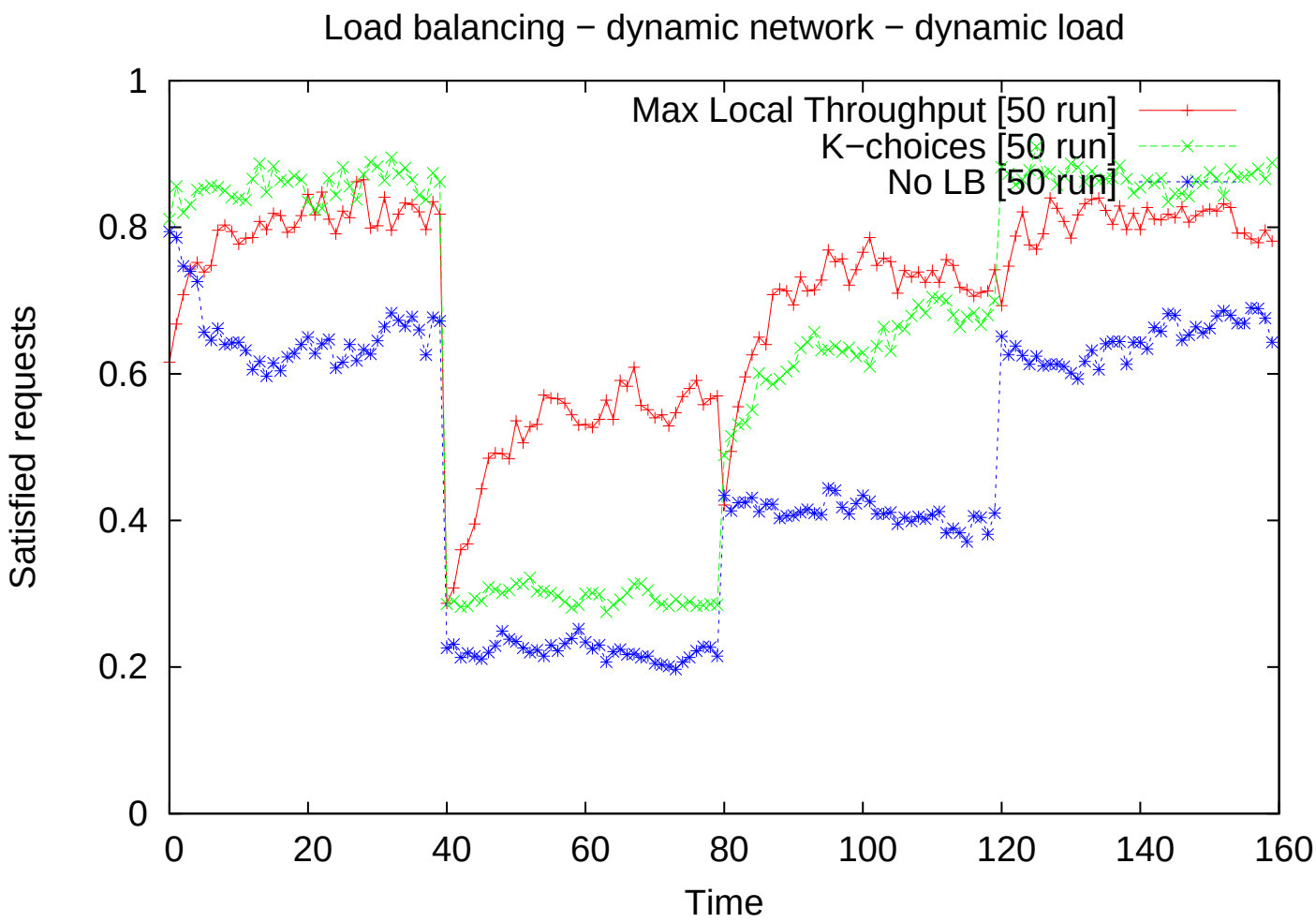
A novel heuristic : **Max Local Throughput**

- At the end of period τ
- Two adjacent peers S and P with capacity C_S and C_P
- ν_S^τ and ν_P^τ the sets of nodes currently managed by S and P
- $L_S^\tau = \sum_{n \in \nu_S^\tau} l_n$
- $L_P^\tau = \sum_{n \in \nu_P^\tau} l_n$
- $T_{S,P}^\tau = min(L_S^\tau, C_S) + min(L_P^\tau, C_P)$
- Maximize the throughput of $\tau + 1$ based on information of τ
- Find $\nu_S^{\tau+1}$ and $\nu_P^{\tau+1}$ that maximizes $T_{S,P}^{\tau+1}$.

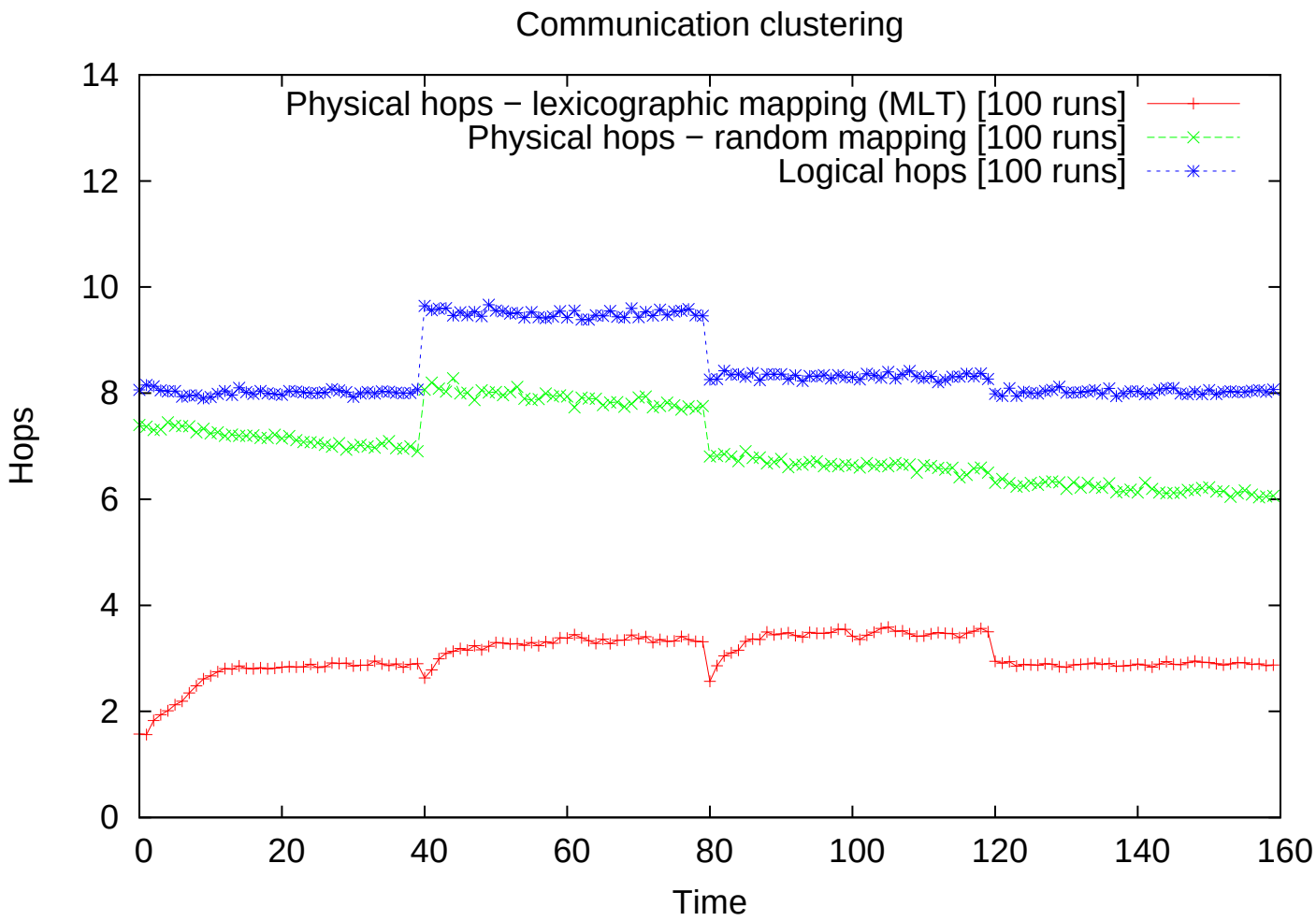
Simulation results

Load	Stable network		Dynamic network	
	Max local th.	K-choices	Max local th.	K-choices
5%	39,62%	38,58%	18.25%	32,47%
10%	103,41%	58,95%	46,16%	51,00%
16%	147,07%	64,97%	65,90%	59,11%
24%	165,25%	59,27%	71,26%	60,01%
40%	206,90%	68,16%	97,71%	67,18%
80%	230,51%	76,99%	90,59%	71,93%

Simulation results



Simulation results



Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

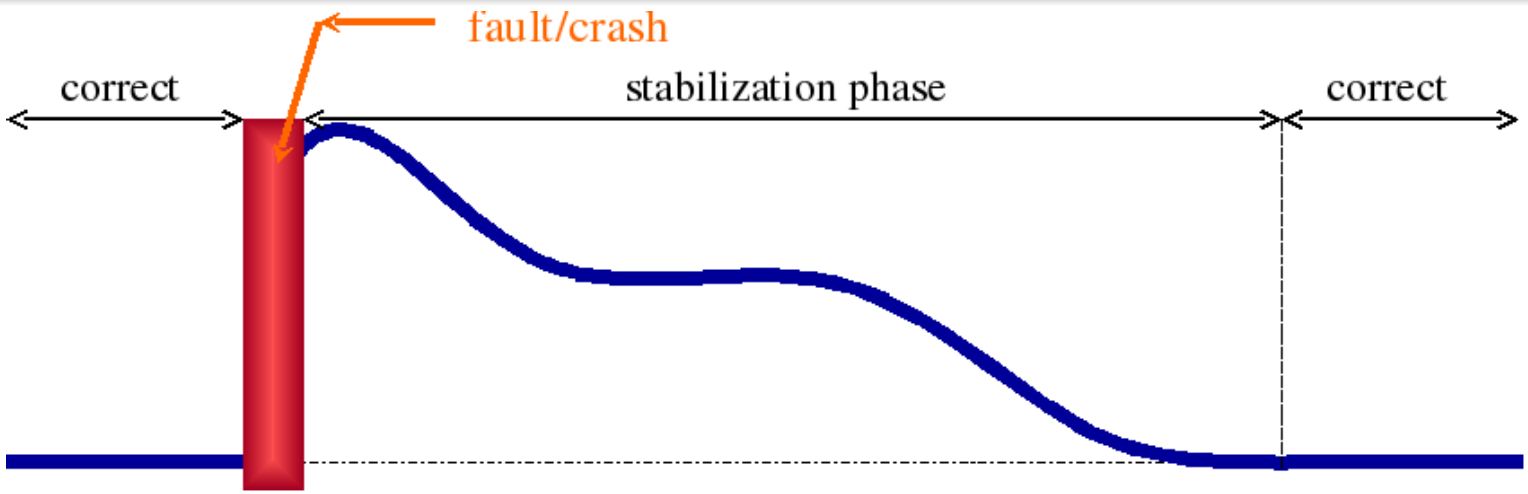
- Creation and maintenance of the tree
- Fault-tolerance
  - Usually, based on replication
    - Costly
    - Unable to recover after arbitrary transient failures
  - Best-effort ?

- Creation and maintenance of the tree
- Fault-tolerance
  - Usually, based on replication
    - Costly
    - Unable to recover after arbitrary transient failures
  - Best-effort ?

- Creation and maintenance of the tree
- Fault-tolerance
  - Usually, based on replication
    - Costly
    - Unable to recover after arbitrary transient failures
  - **Best-effort ?**

Self-Stabilization (Dijkstra, 74)

- General technique to tolerate transient faults
- Guaranteed to converge to the intended behavior
  - Regardless of the initial state of the system
  - In finite time



Architecture Model

Physical Network

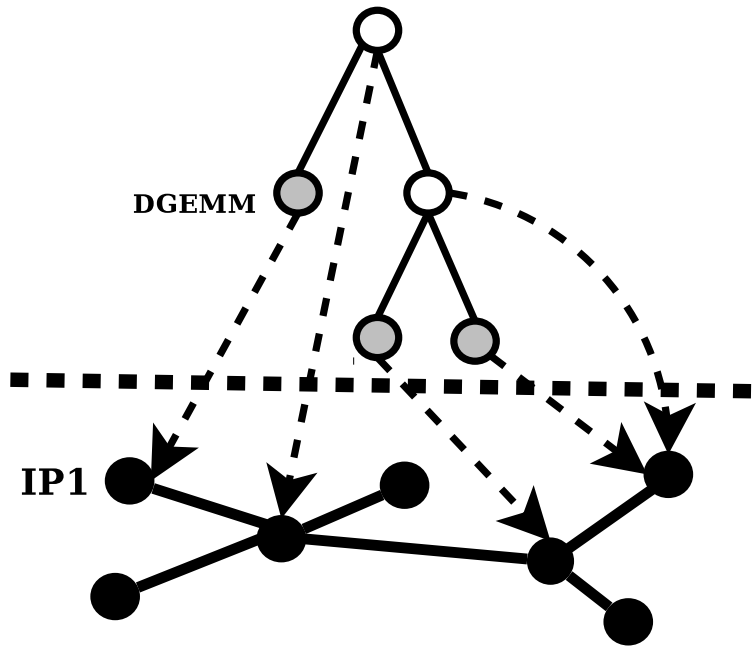
Basic entity : **peer** (processor)

- P_1 can communicate with P_2 if P_1 *knows* P_2.
- Runs some logical *nodes*

Logical Tree

Basic entity : (tree) **node**

- Distributed among peers
- Mapping details abstracted out
- Protocol run inside nodes



State Model

- Message exchanges modeled by the ability to read variables of other nodes (**Neighbors**)
- A node can only write its own variables

- The **state** of a node is defined by the values of its variables
- The **configuration** of the system is the product of the states

- Actions : $< guard > \rightarrow < statement >$
  - $< guard >$ bool. expr. of variables of p and its neighbors
  - $< statement >$
    - Executed only if its $< guard > = true$
    - Updates one or more variables of p

- Distributed and unfair scheduler daemon

State Model

- Message exchanges modeled by the ability to read variables of other nodes (**Neighbors**)
- A node can only write its own variables

- The **state** of a node is defined by the values of its variables
- The **configuration** of the system is the product of the states

- Actions : $< guard > \rightarrow < statement >$
  - $< guard >$ bool. expr. of variables of p and its neighbors
  - $< statement >$
    - Executed only if its $< guard > = true$
    - Updates one or more variables of p

- Distributed and unfair scheduler daemon

State Model

- Message exchanges modeled by the ability to read variables of other nodes (**Neighbors**)
- A node can only write its own variables

- The **state** of a node is defined by the values of its variables
- The **configuration** of the system is the product of the states

- Actions : $< guard > \rightarrow < statement >$
  - $< guard >$ bool. expr. of variables of p and its neighbors
  - $< statement >$
    - Executed only if its $< guard > = true$
    - Updates one or more variables of p

- Distributed and unfair scheduler daemon

State Model

- Message exchanges modeled by the ability to read variables of other nodes (**Neighbors**)
- A node can only write its own variables

- The **state** of a node is defined by the values of its variables
- The **configuration** of the system is the product of the states

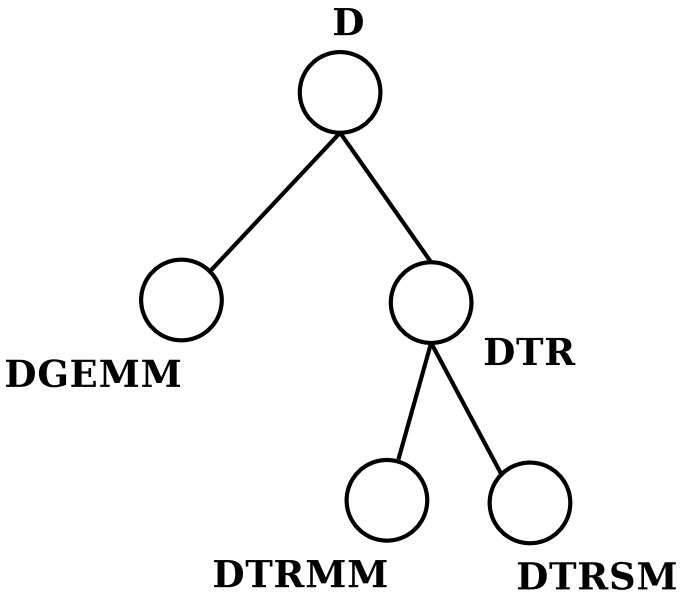
- Actions : $< guard > \rightarrow < statement >$
  - $< guard >$ bool. expr. of variables of p and its neighbors
  - $< statement >$
    - Executed only if its $< guard > = true$
    - Updates one or more variables of p

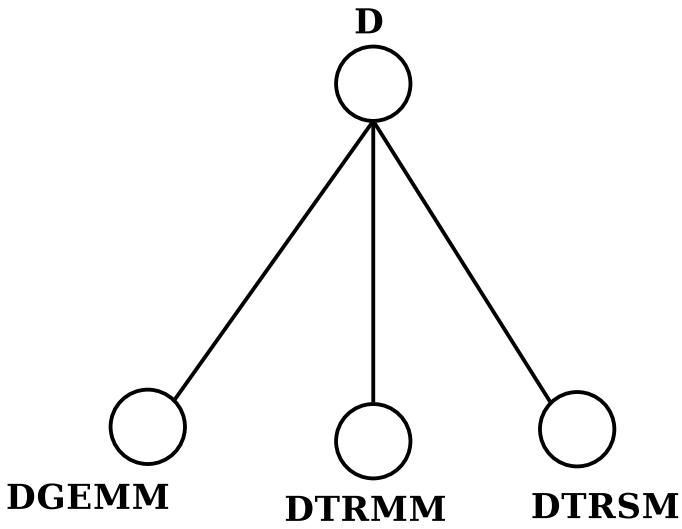
- Distributed and unfair scheduler daemon

Snap-stabilization

Let P be a protocol designed to solve a task T. P is said **snap-stabilizing** if and only if, starting from any configuration, any execution e of P always satisfies the specification of T.

The Distributed Structures Maintained





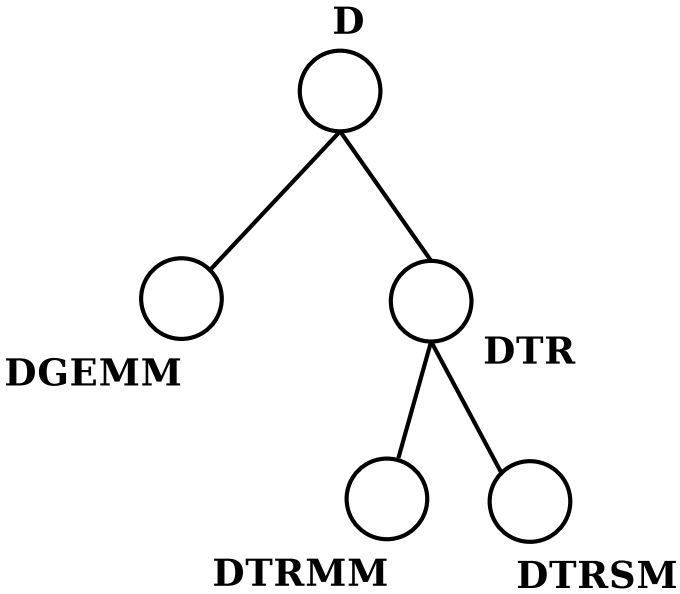
PGCP Tree

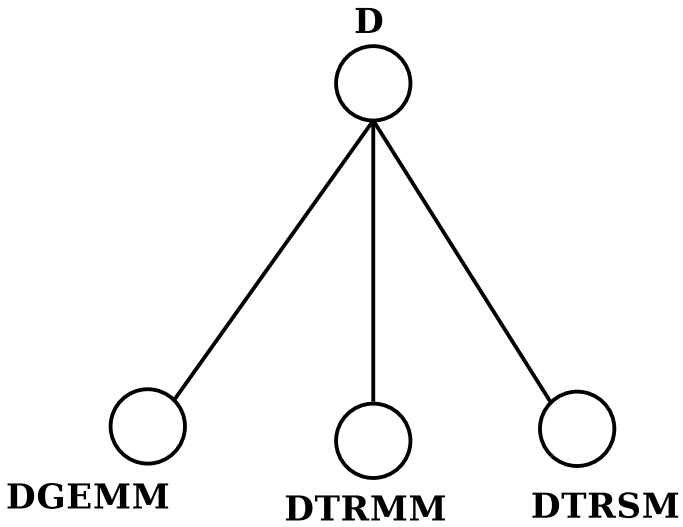
A labeled rooted tree s.t. each node label is the **PGCP** of any pair of its children labels.

Prefix Heap

A labeled rooted tree s.t. each node label is the **PGCP** of all its children labels.

The Distributed Structures Maintained





A Prefix Heap s.t. for each node p, every pair of children (q, r) :

$$\begin{cases} (1) & l_q \neq l_r \\ (2) & l_q(\text{resp. } l_r) \text{ is not a prefix of } l_r(\text{resp. } l_q) \\ (3) & |GCP(l_q, l_r)| = |l_p| \end{cases}$$

is a PGCP Tree.

Snap-stabilizing PGCP Tree construction

Topology Assumptions

The initial graph is a rooted connected tree.

Snap-stabilizing PGCP Tree

A protocol P is considered as a snap-stabilizing PGCP tree construction if and only if, any execution initiated by the root terminates in finite time and when the execution terminated, the tree is a PGCP tree.

The Protocol

Functions

- `newNode(lbl, st, chldn)`
- `destroy (p)`
- `heapify()` locally creates a heap
- `repair()` locally builds a PGCP Tree from a heap

Phases

- From the snap-stabilizing PIF [Bui, Datta, Petit, Villain - 99]
- **Broadcast** phase : top-down wave initiating the algorithm
- **Heapify** phase : down-top traversal building a prefix heap
- **Repair** phase : final top-down wave building a PGCP tree

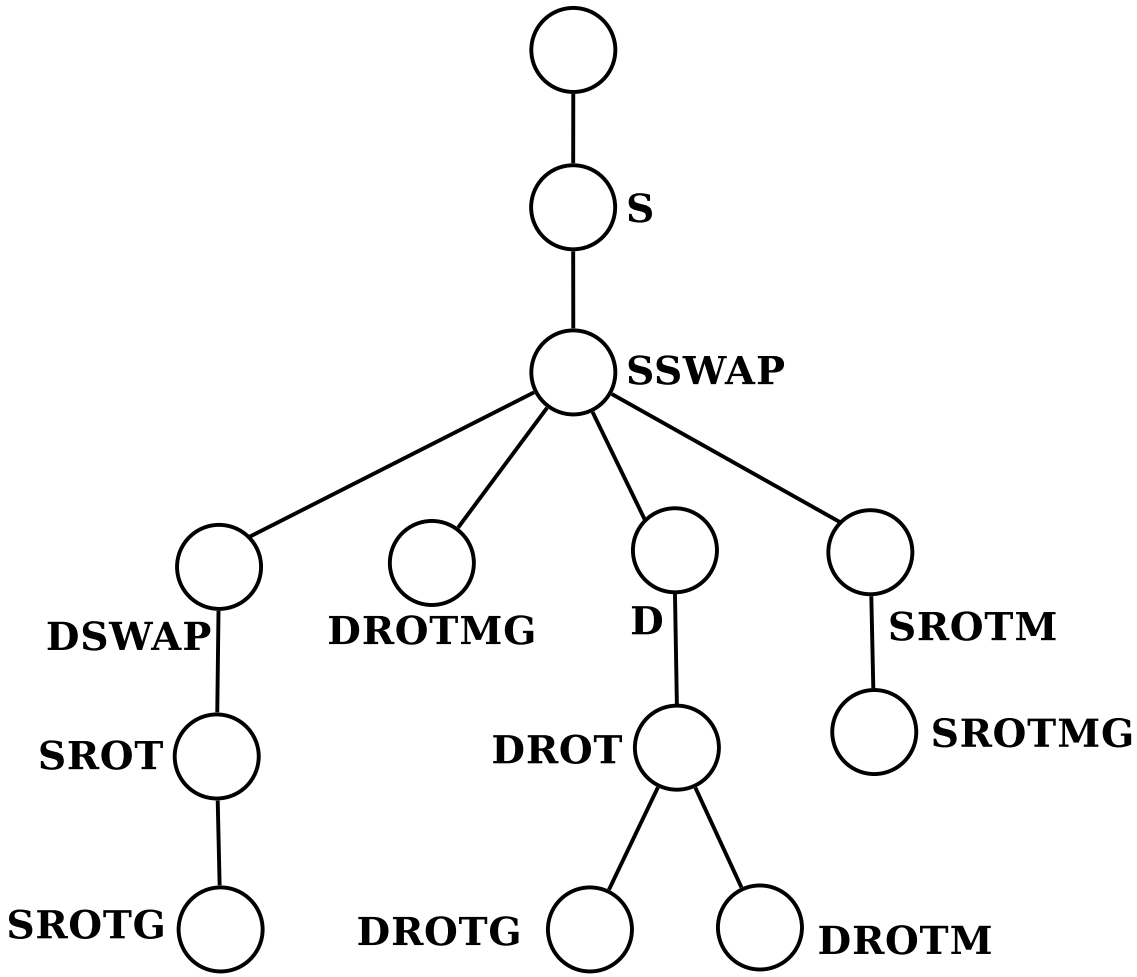
The Protocol

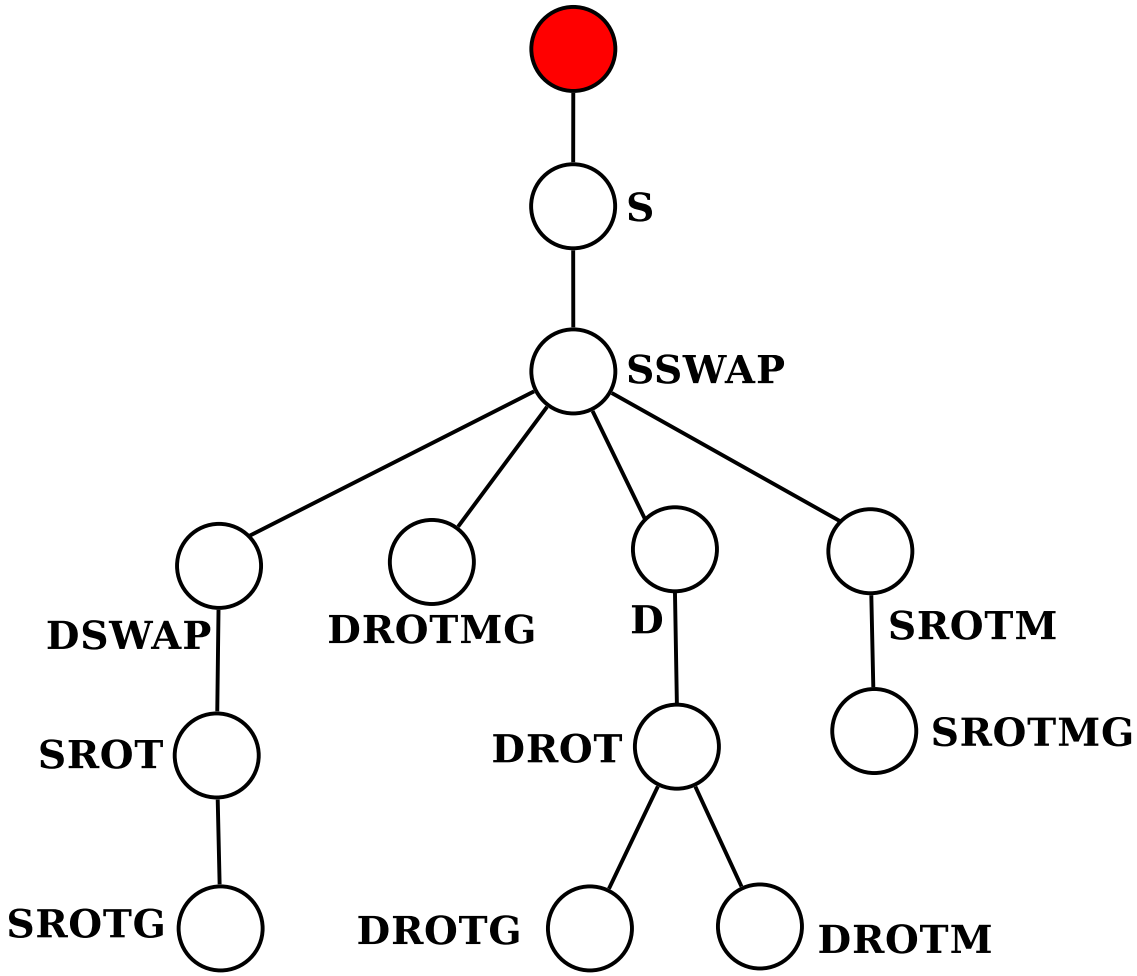
Functions

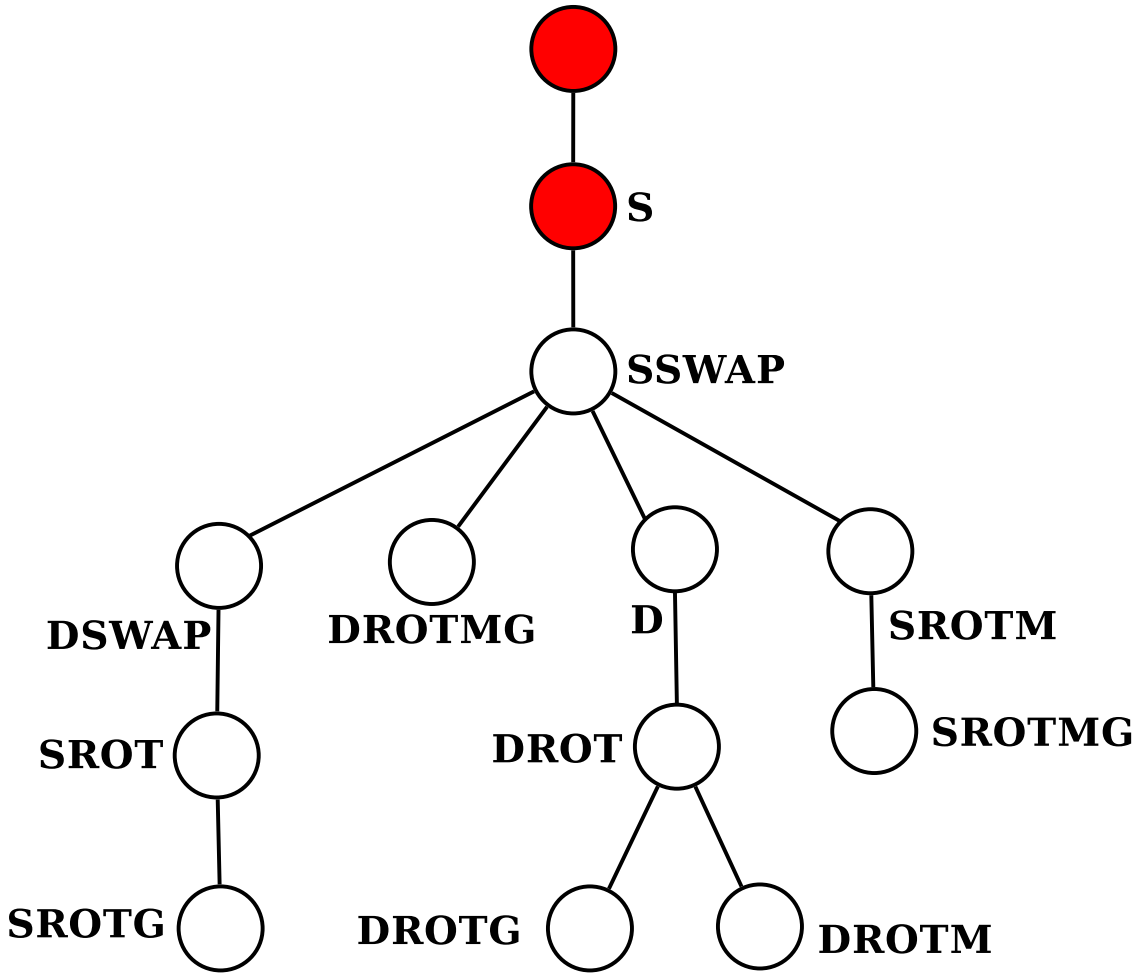
- `newNode(lbl, st, chldn)`
- `destroy (p)`
- `heapify()` locally creates a heap
- `repair()` locally builds a PGCP Tree from a heap

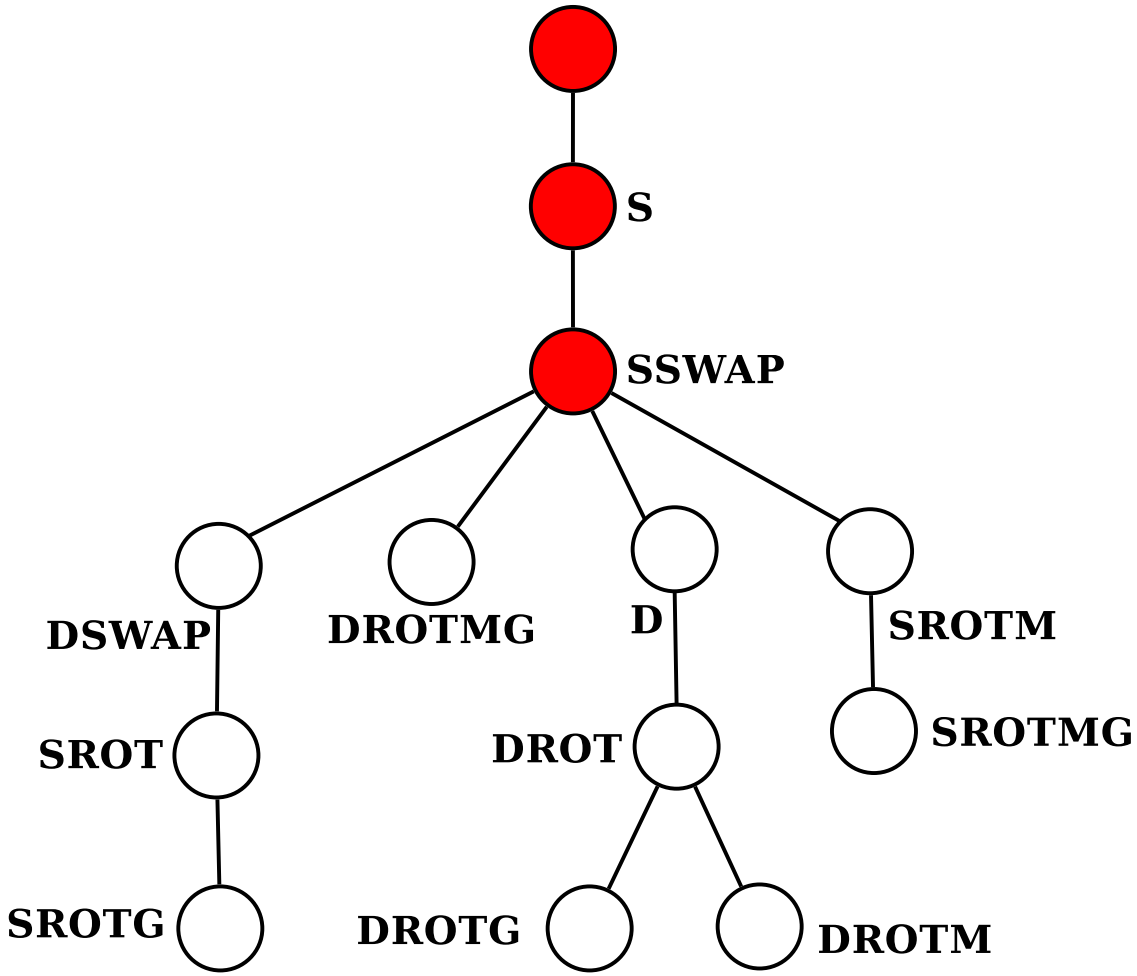
Phases

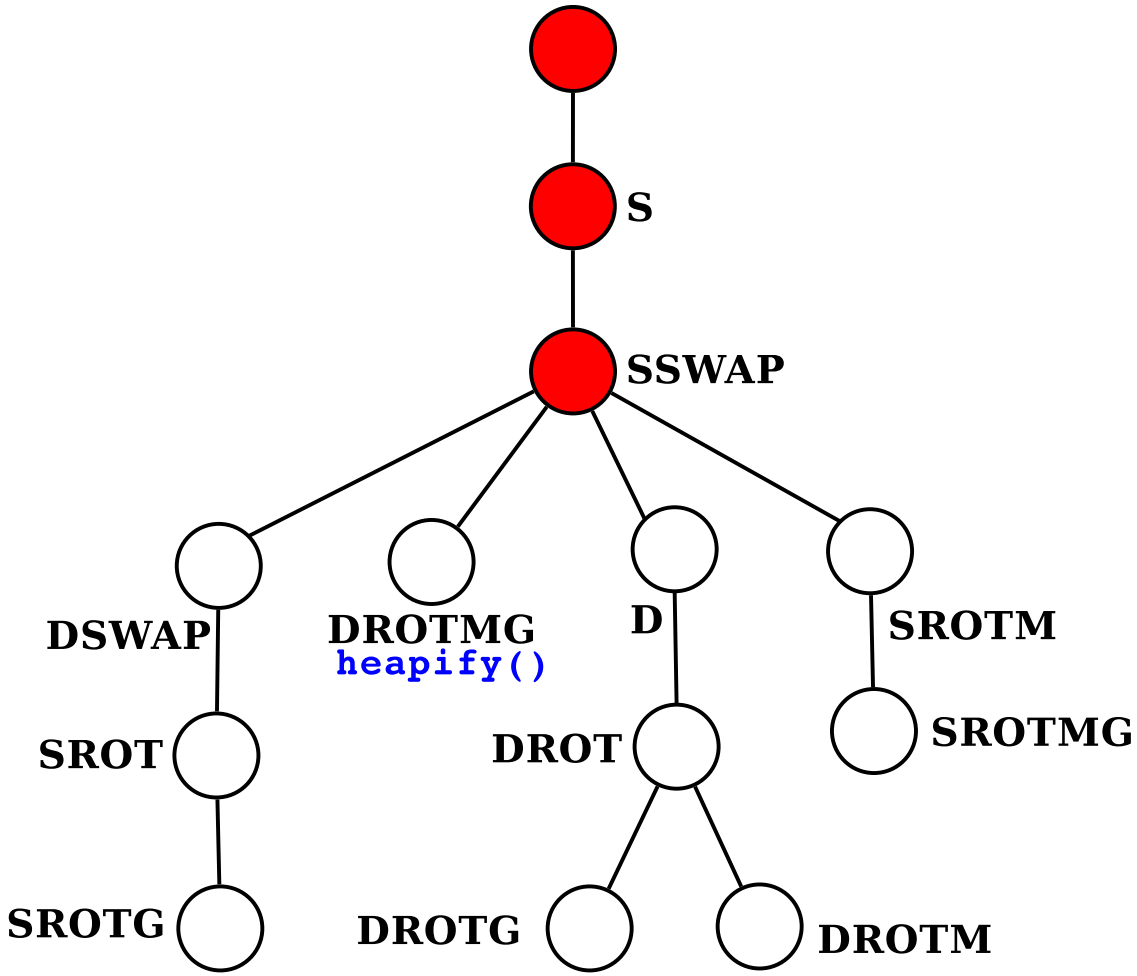
- From the snap-stabilizing PIF [Bui, Datta, Petit, Villain - 99]
- **Broadcast** phase : top-down wave initiating the algorithm
- **Heapify** phase : down-top traversal building a prefix heap
- **Repair** phase : final top-down wave building a PGCP tree

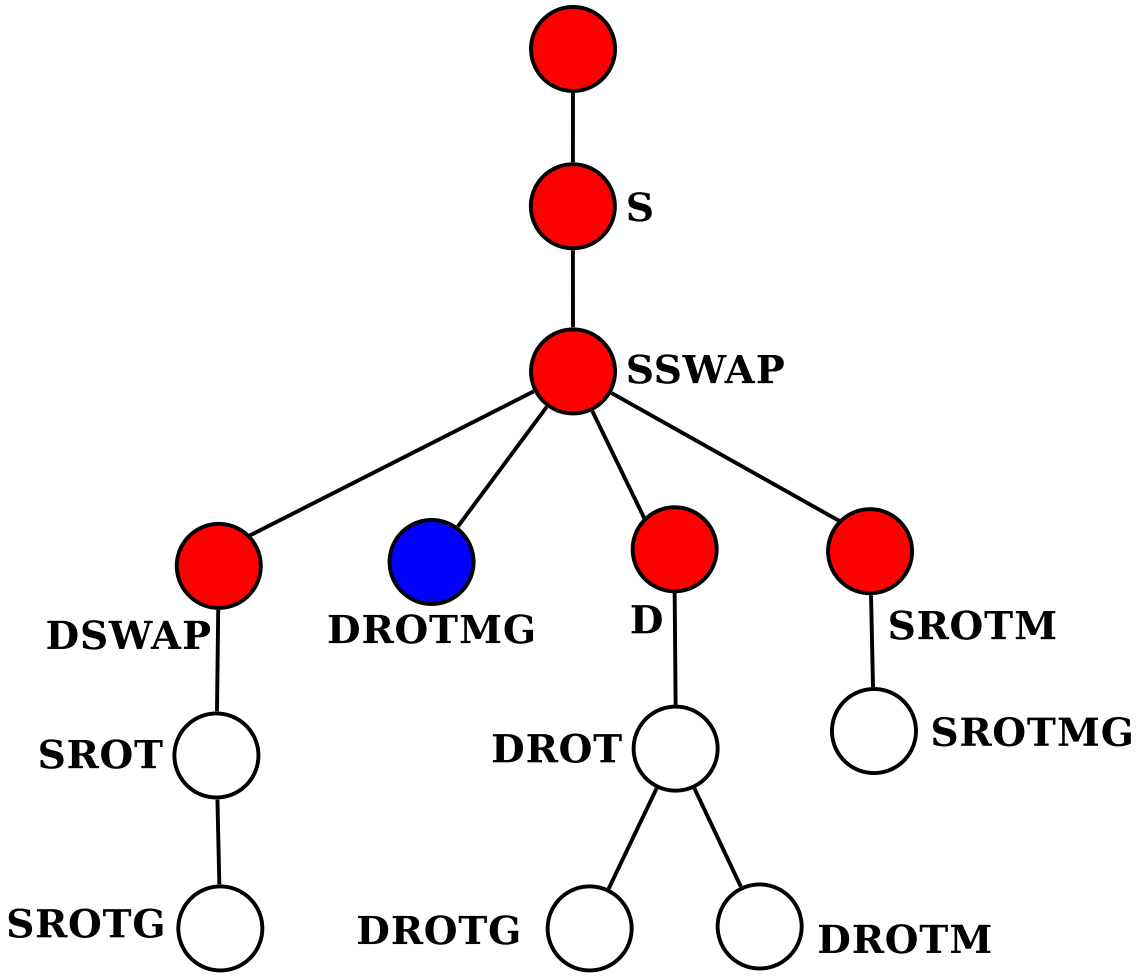

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

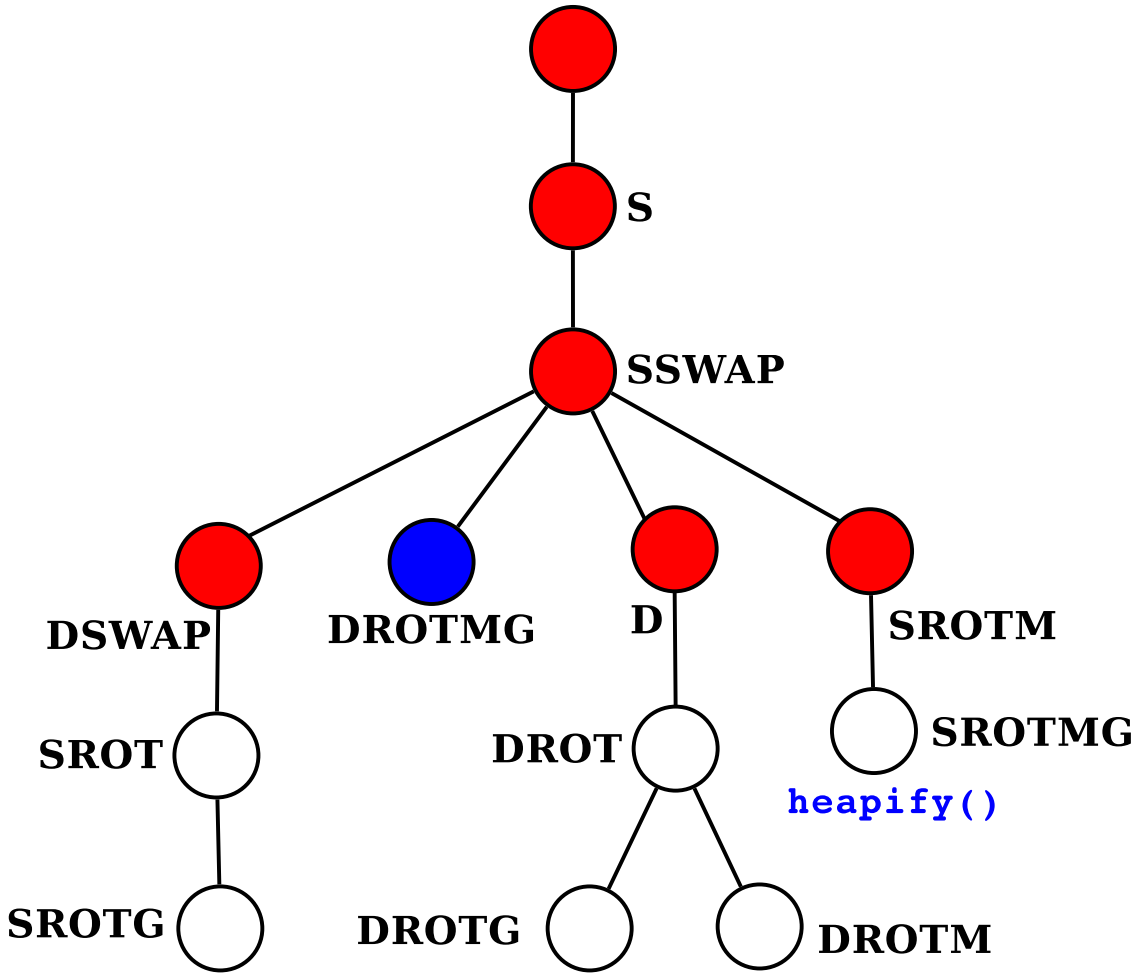

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

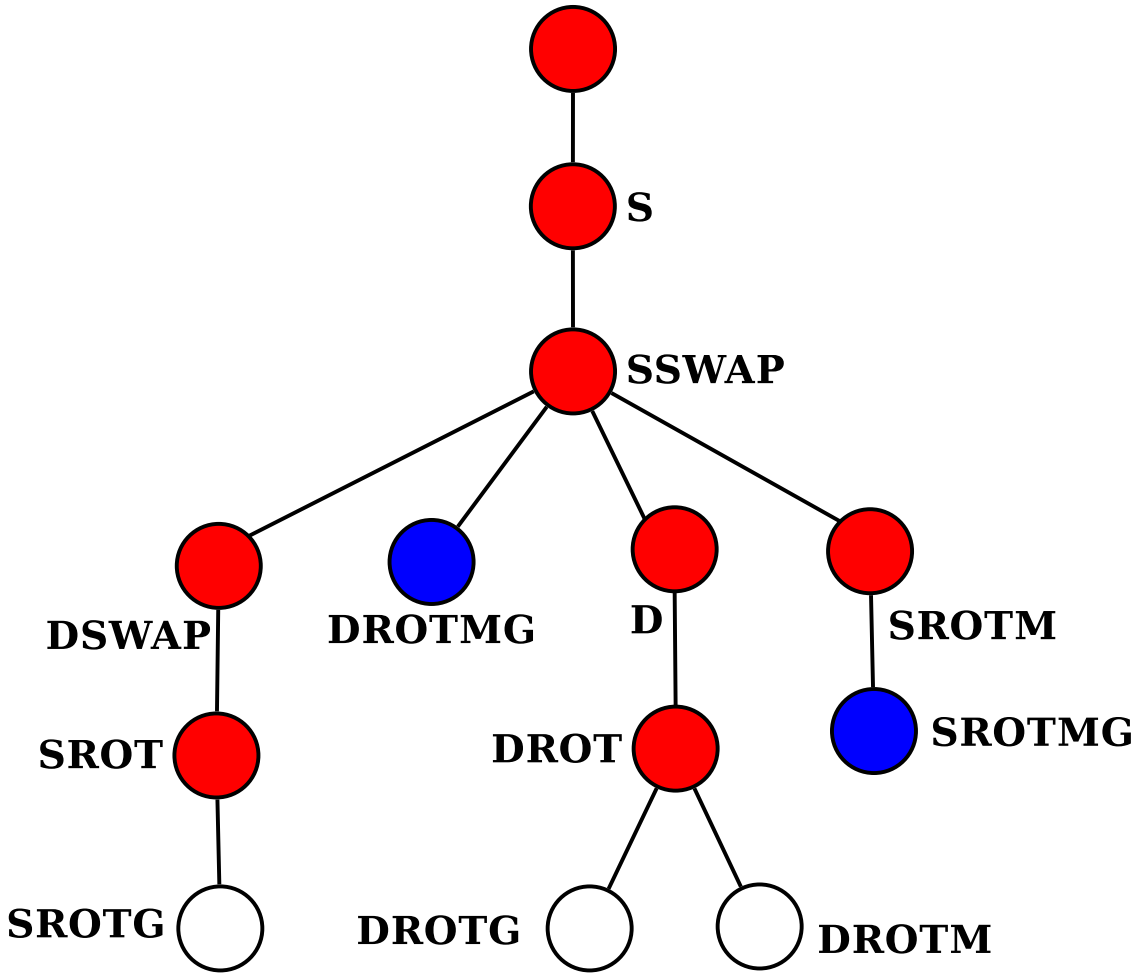

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

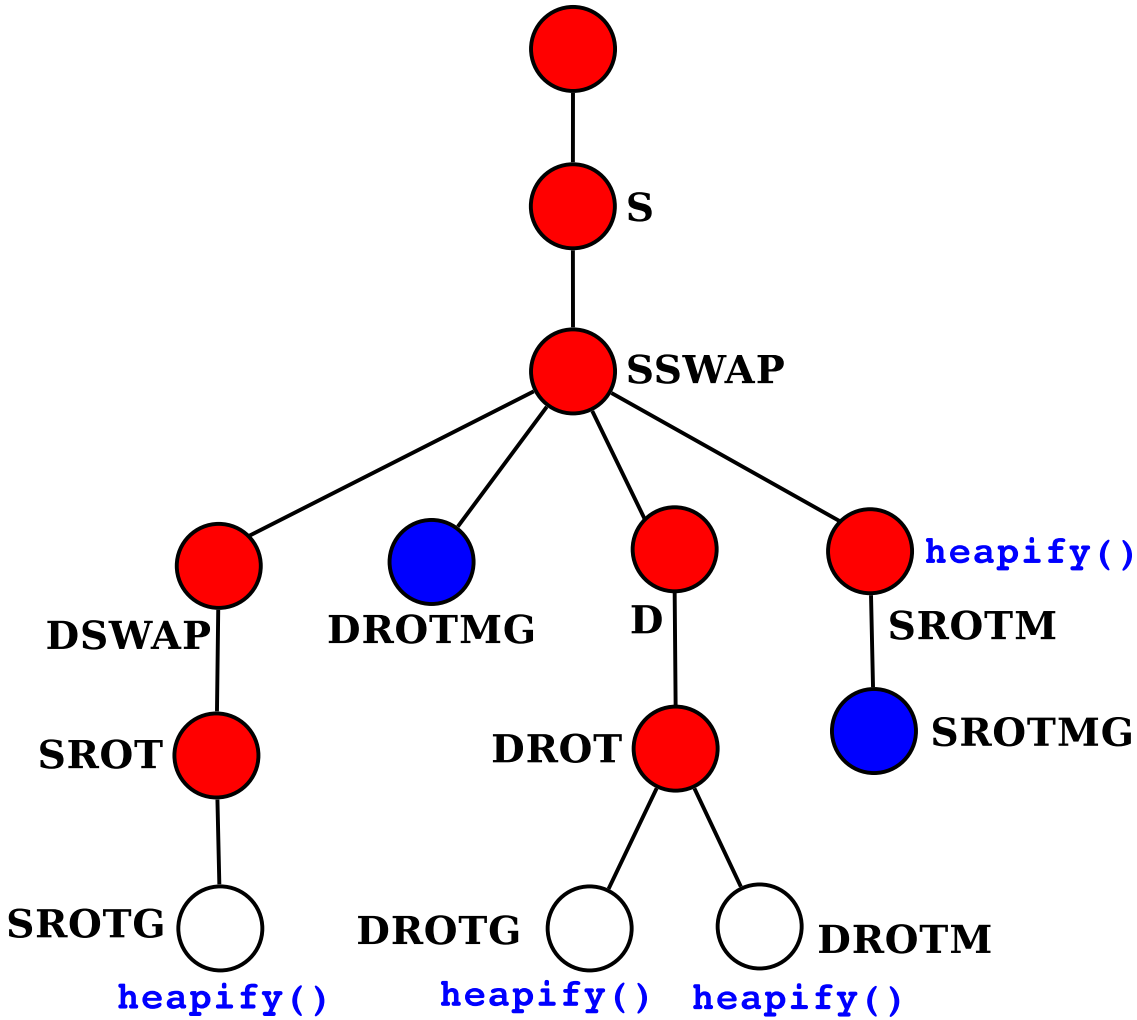

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

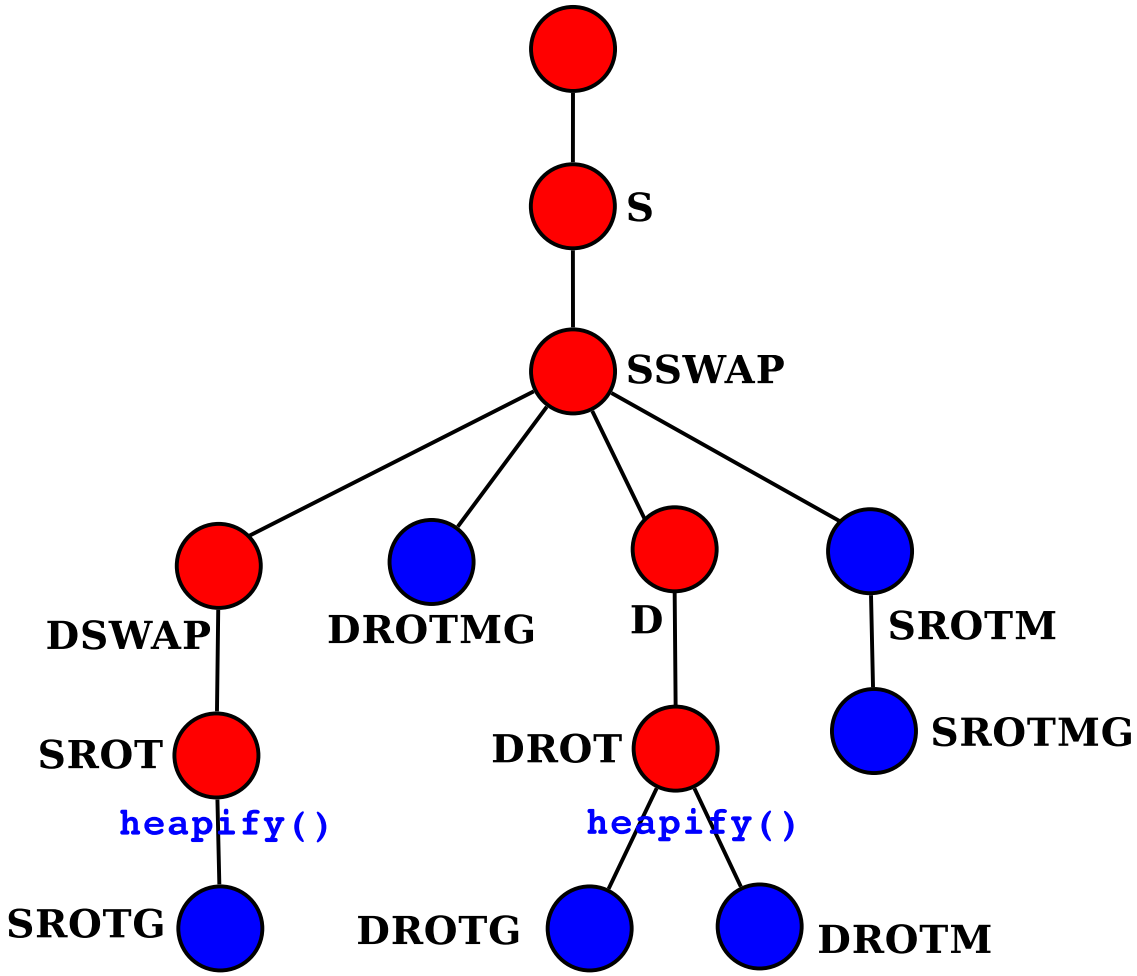

S
SSWAP
DSWAP
DROTMG
heapify()
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

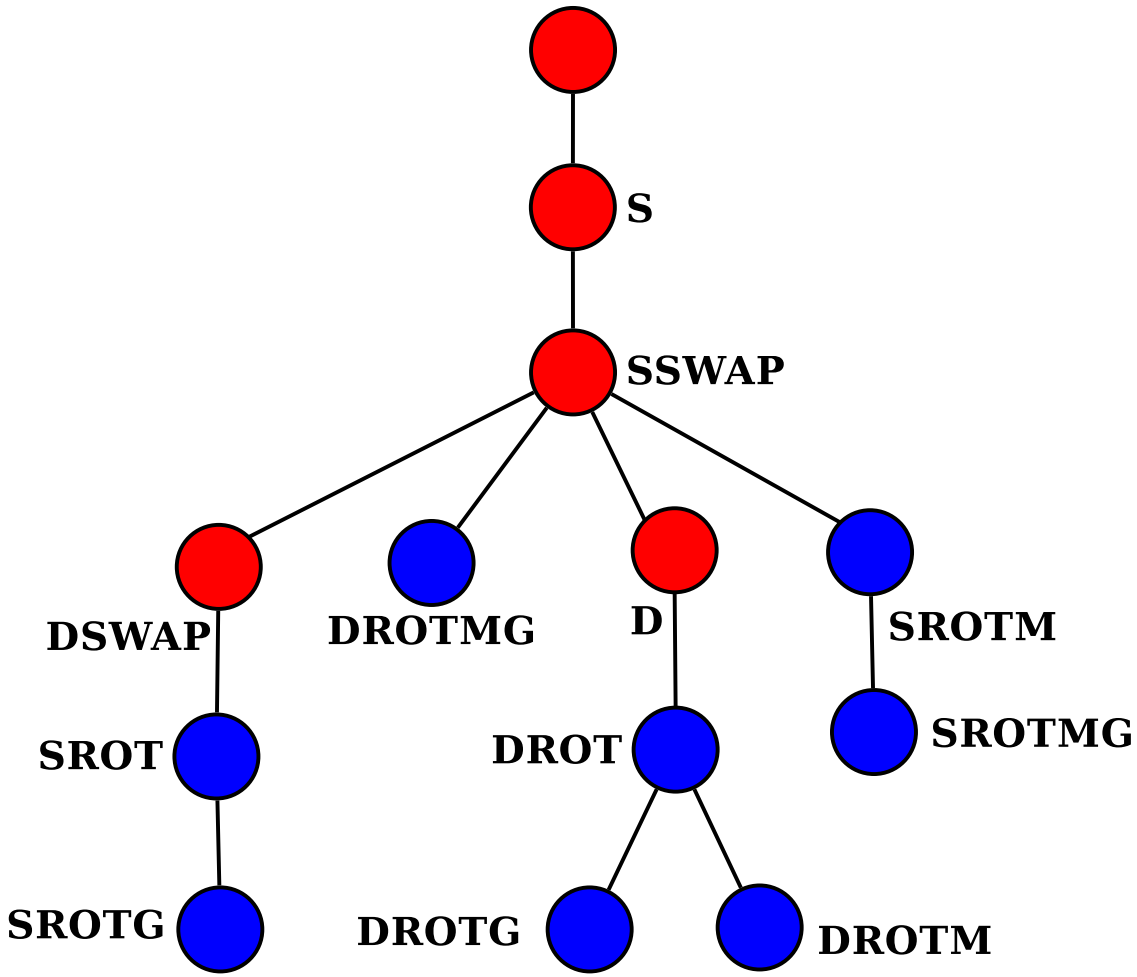

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

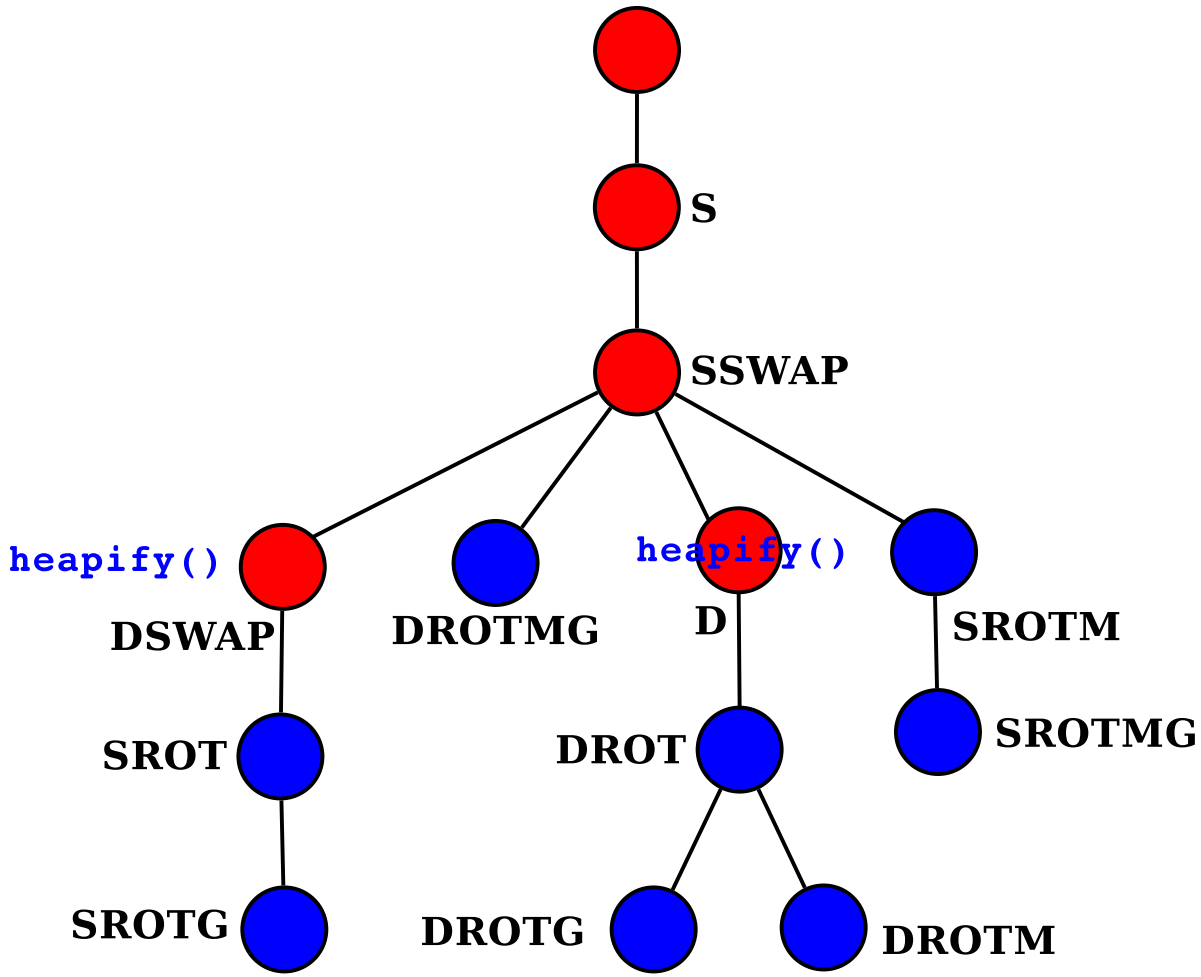

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
heapify()
SROTG
DROTG
DROTM

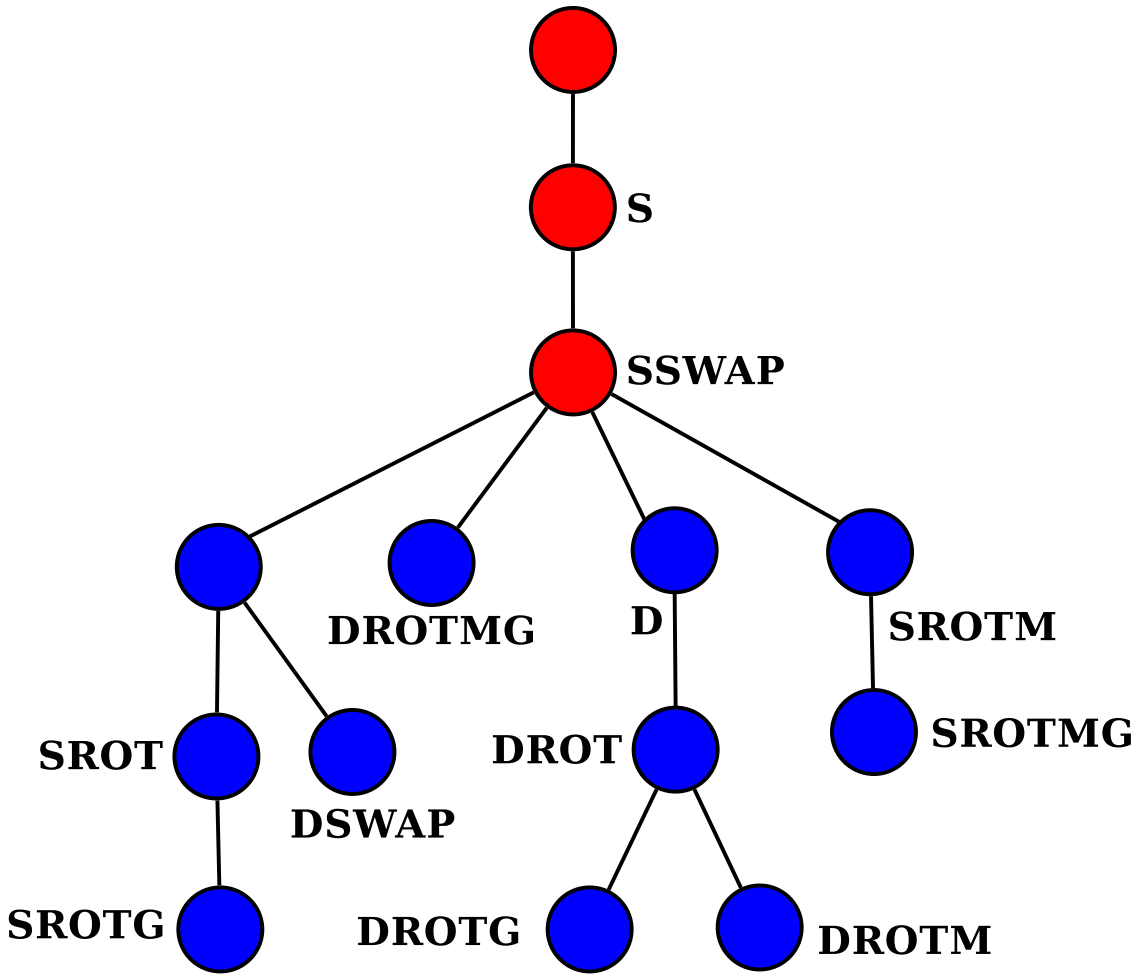

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

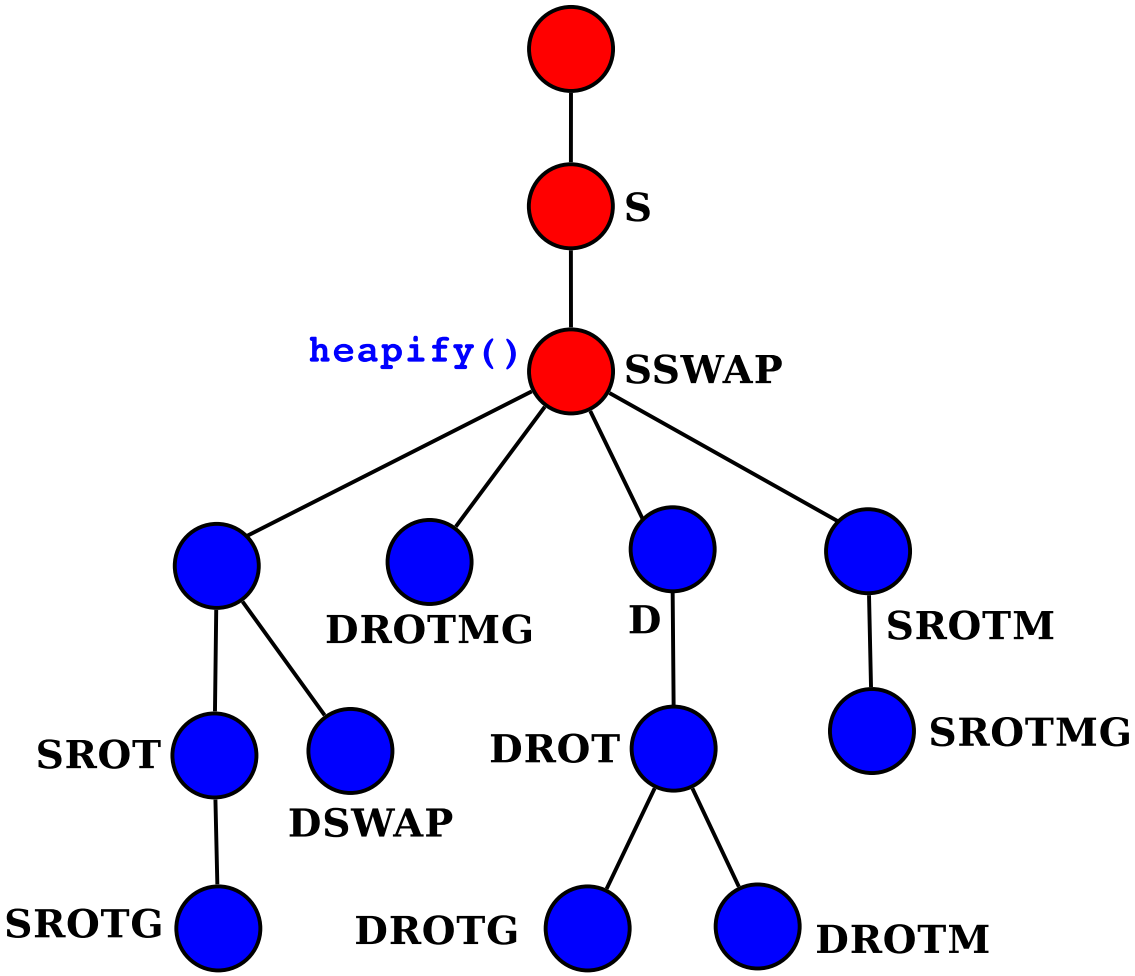

S
SSWAP
DSWAP
DROTMG
D
heapify()
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM
heapify()
heapify()
heapify()

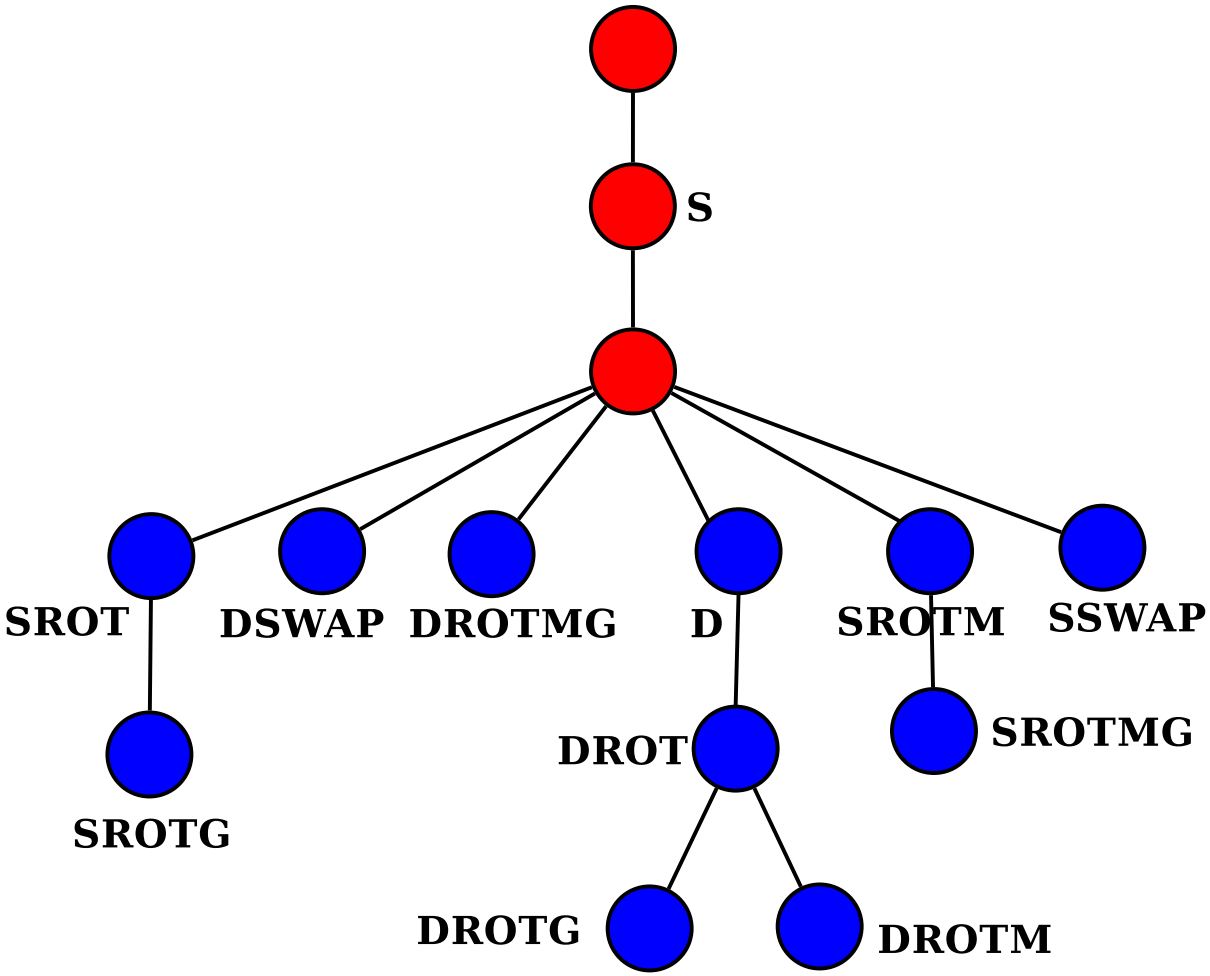

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
heapify()
heapify()
SROTG
DROTG
DROTM

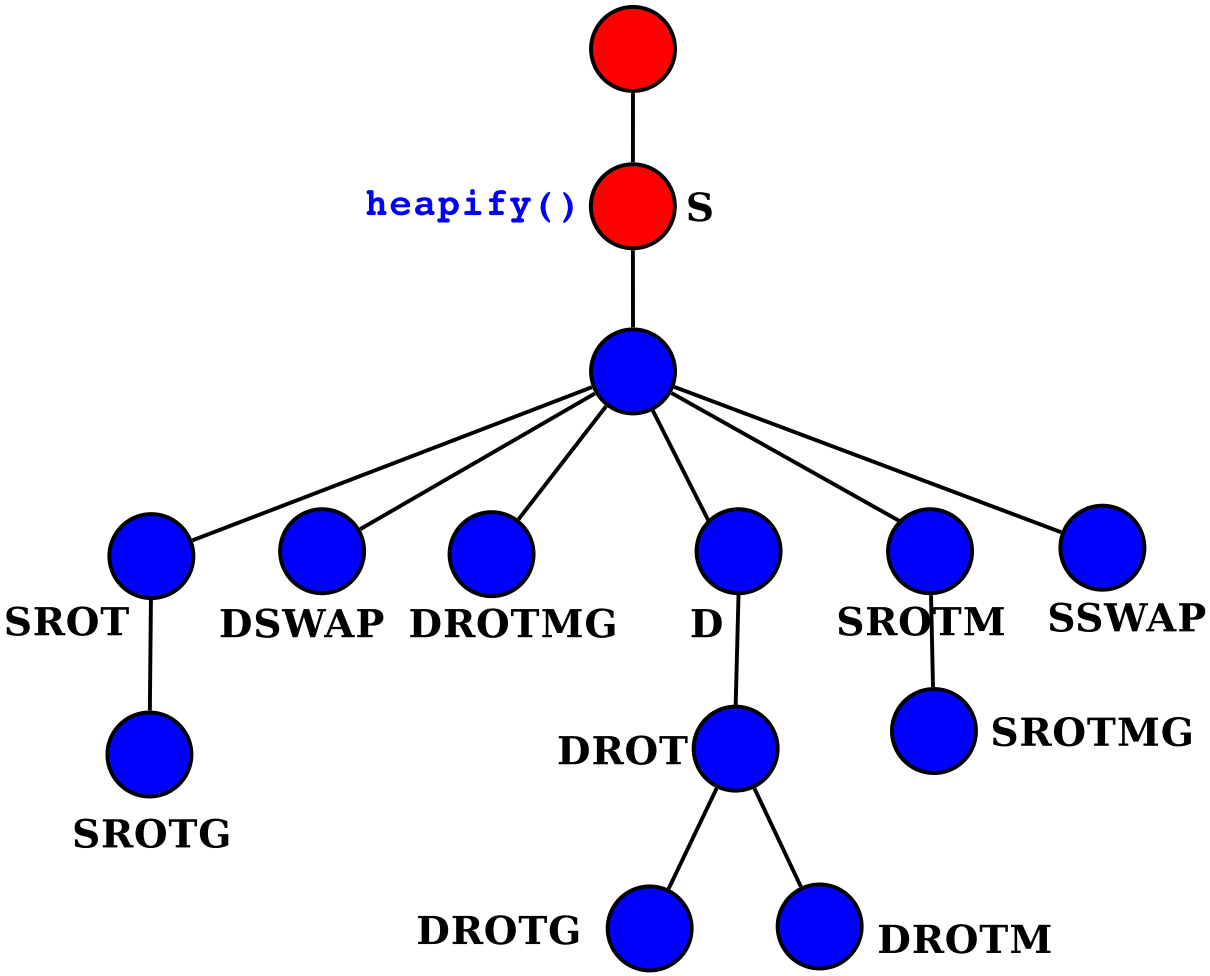

S
SSWAP
DSWAP
DROTMG
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

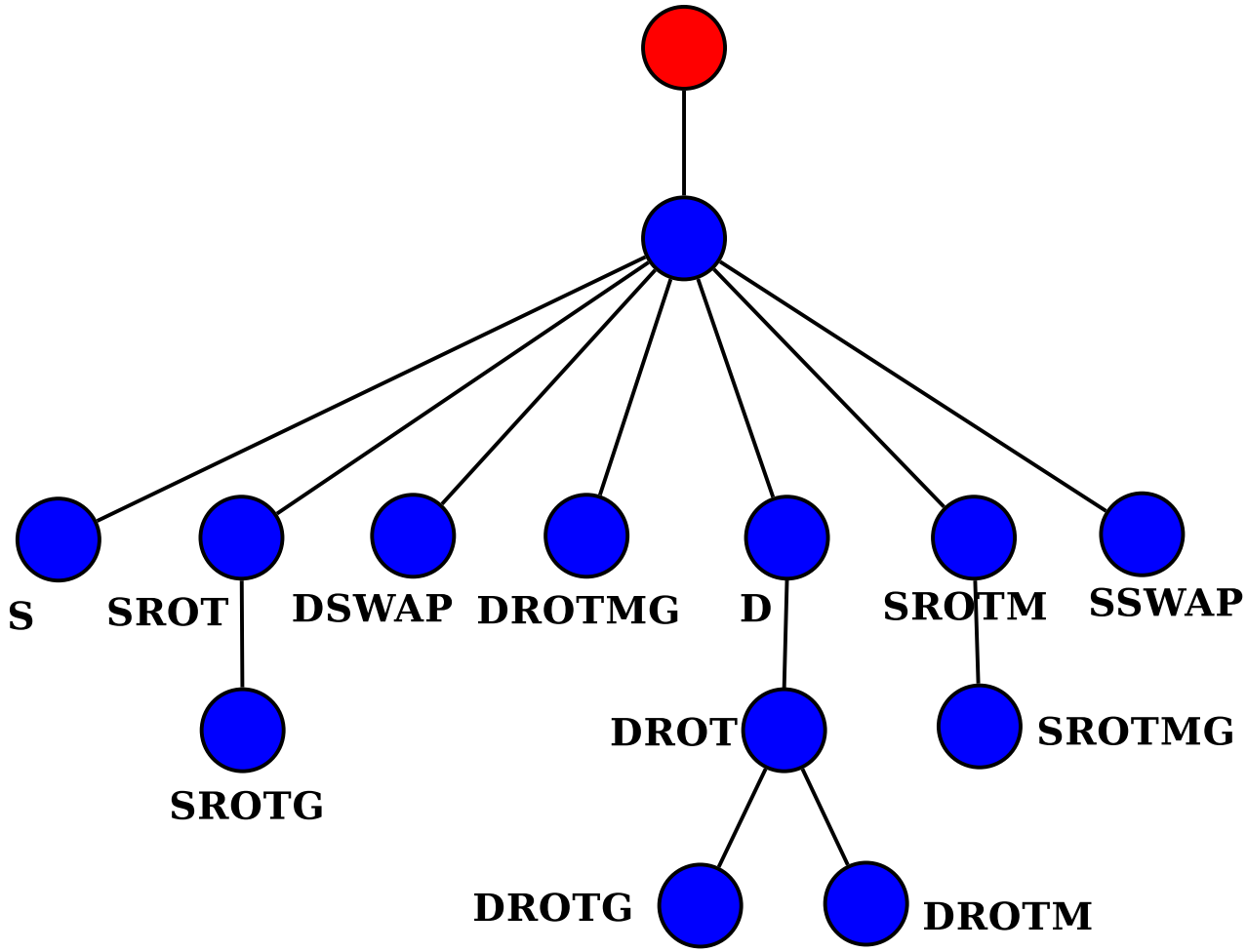

S
SSWAP
heapify()
DSWAP
DROTMG
heapify()
D
SROTM
SROT
DROT
SROTMG
SROTG
DROTG
DROTM

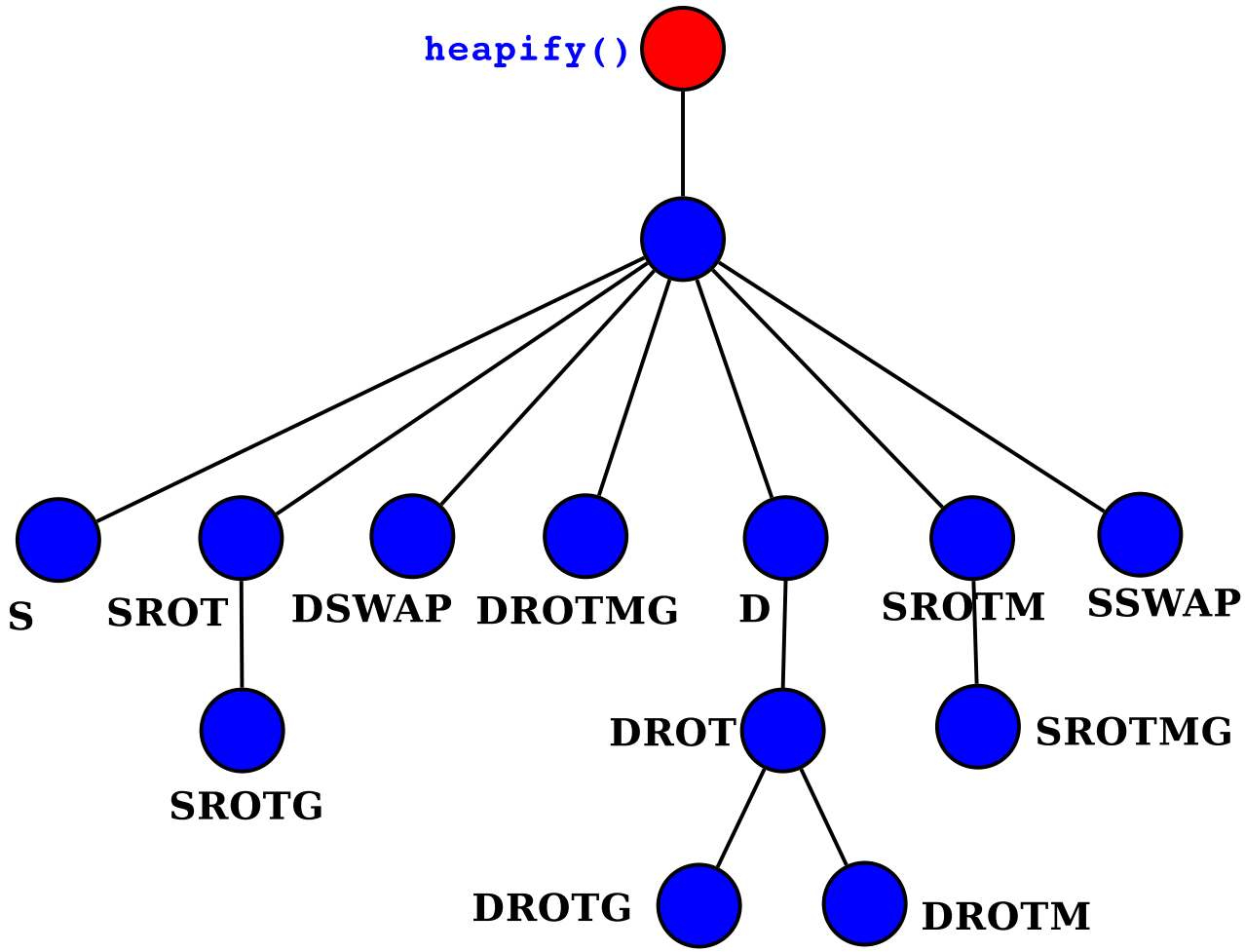

S
SSWAP
DROTMG
D
SROTM
SROT
DSWAP
DROT
SROTMG
SROTG
DROTG
DROTM

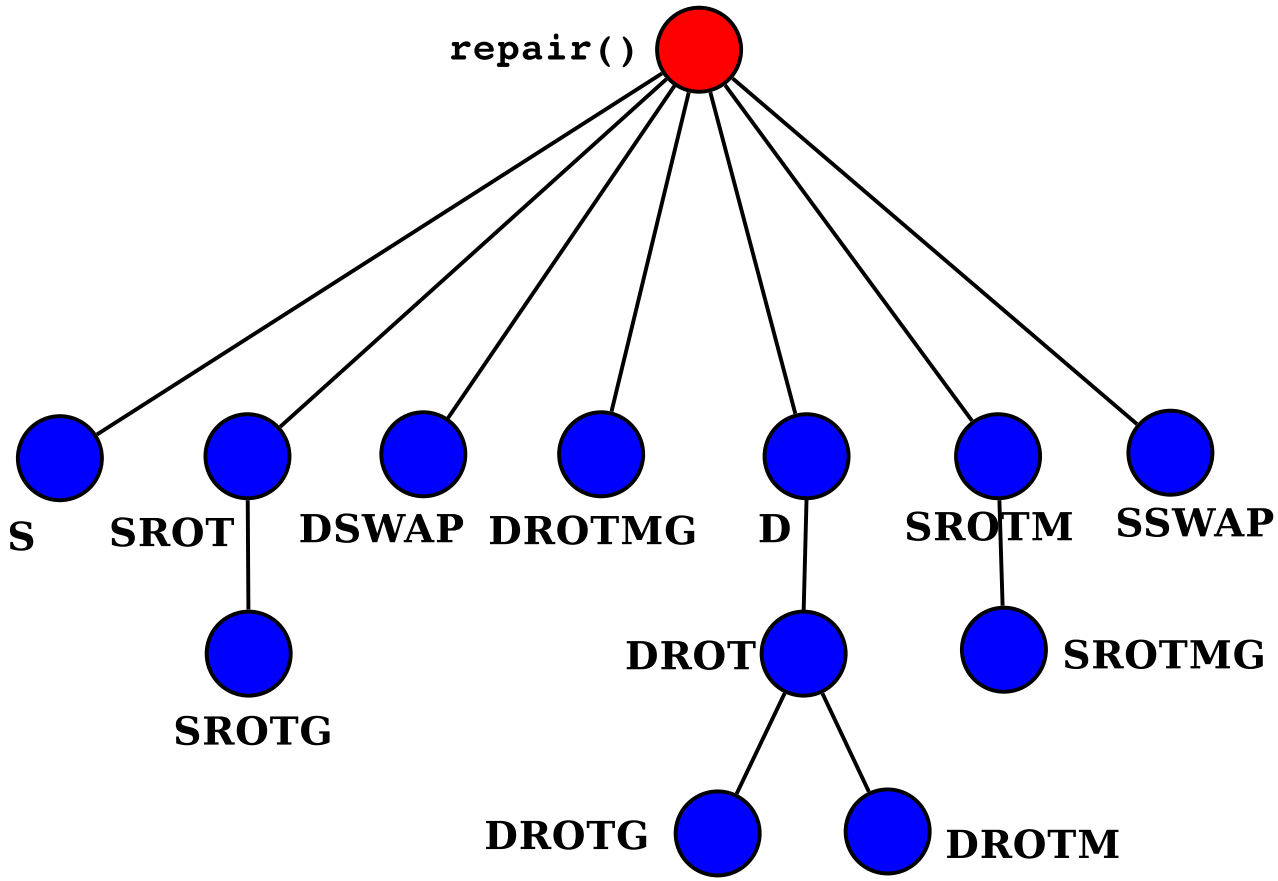

S
heapify()
SSWAP
DROTMG
D
SROTM
SROT
DSWAP
DROT
SROTMG
SROTG
DROTG
DROTM

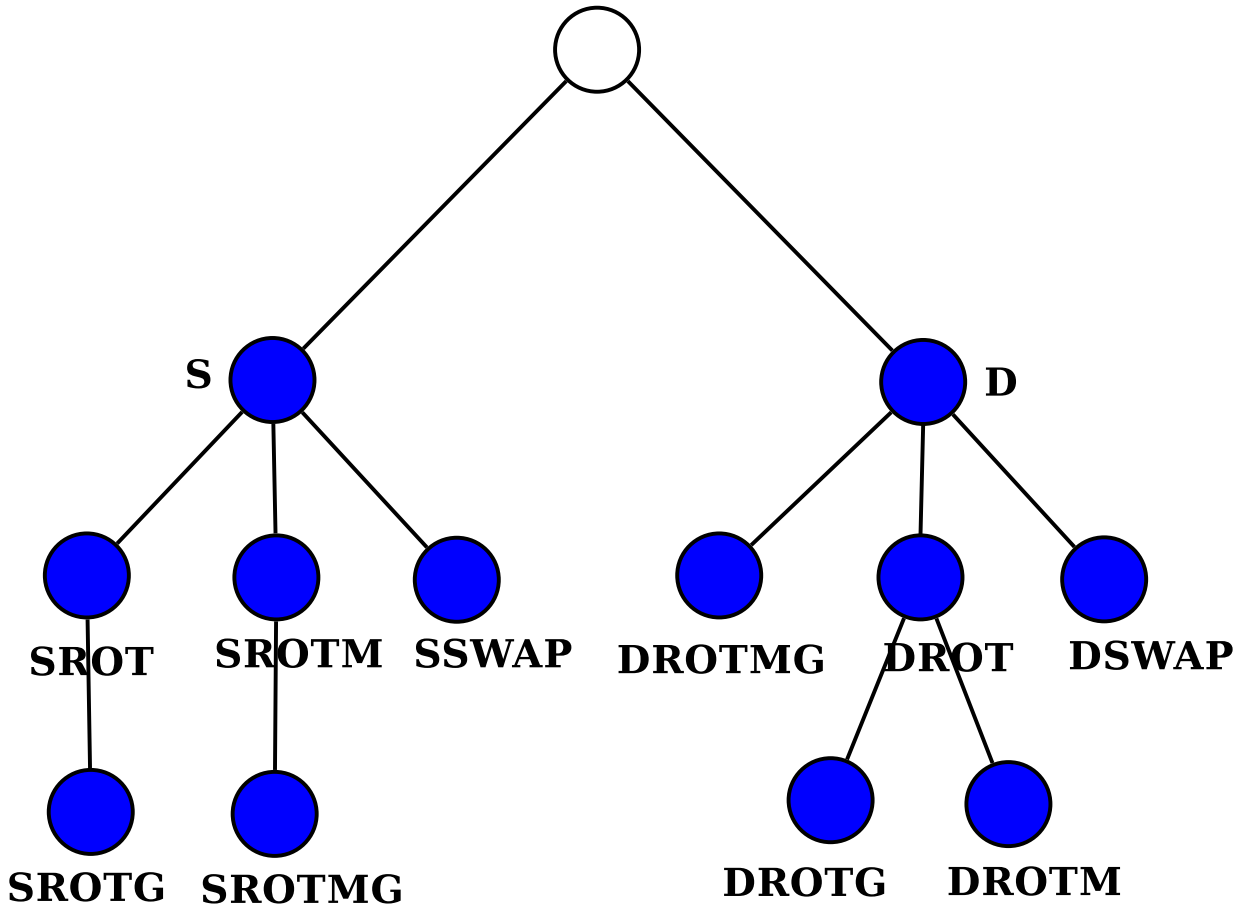

S
SROT
DSWAP
DROTMG
D
SROTM
SSWAP
SROTG
DROT
SROTMG
DROTG
DROTM

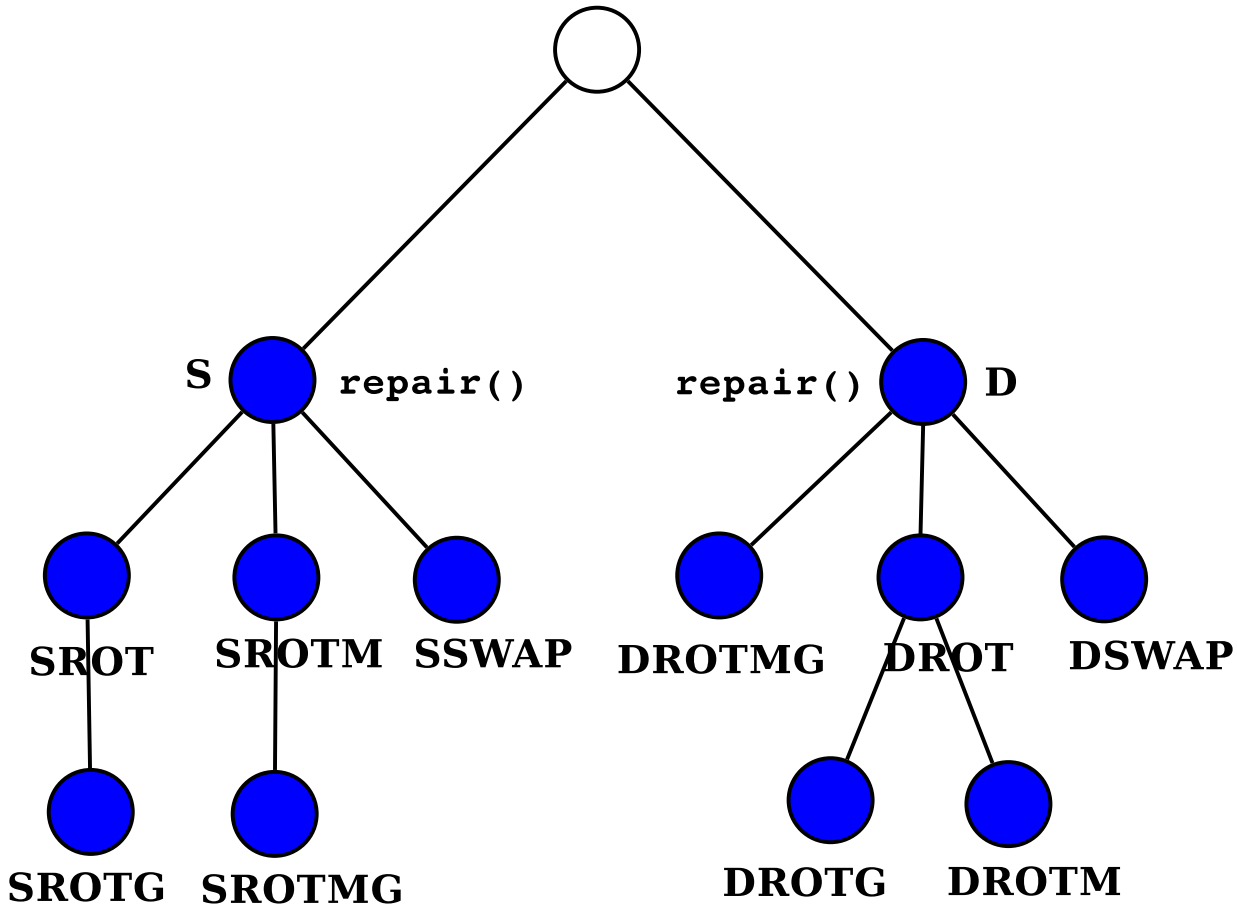

heapify()
S
SROT
DSWAP
DROTMG
D
SROTM
SSWAP
SROTG
DROT
SROTMG
DROTG
DROTM

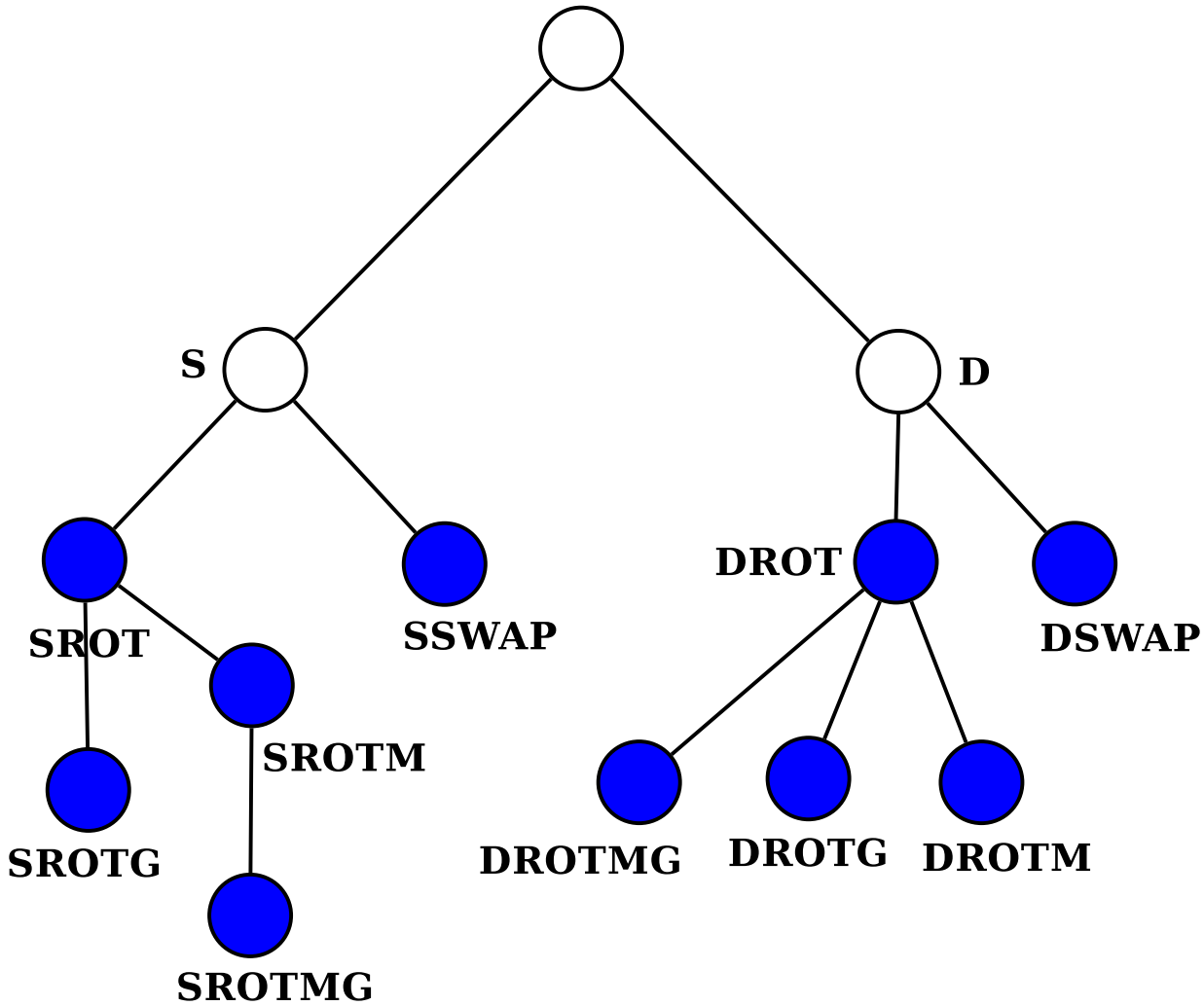

S
SROT
DSWAP
DROTMG
D
SROTM
SSWAP
SROTG
DROT
SROTMG
DROTG
DROTM

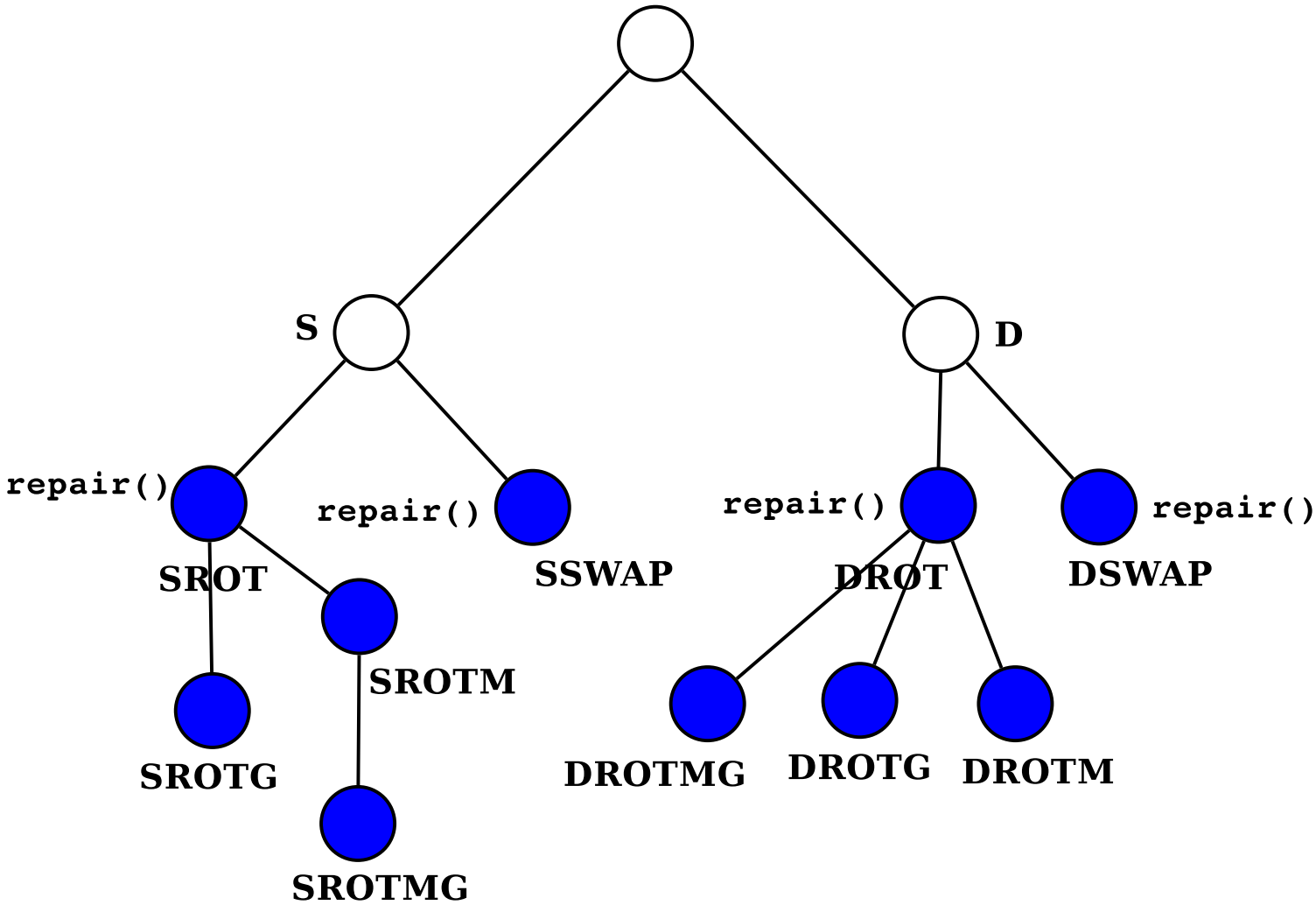

heapify()
S
SROT
DSWAP
DROTMG
D
SROTM
SSWAP
SROTG
DROT
SROTMG
DROTG
DROTM

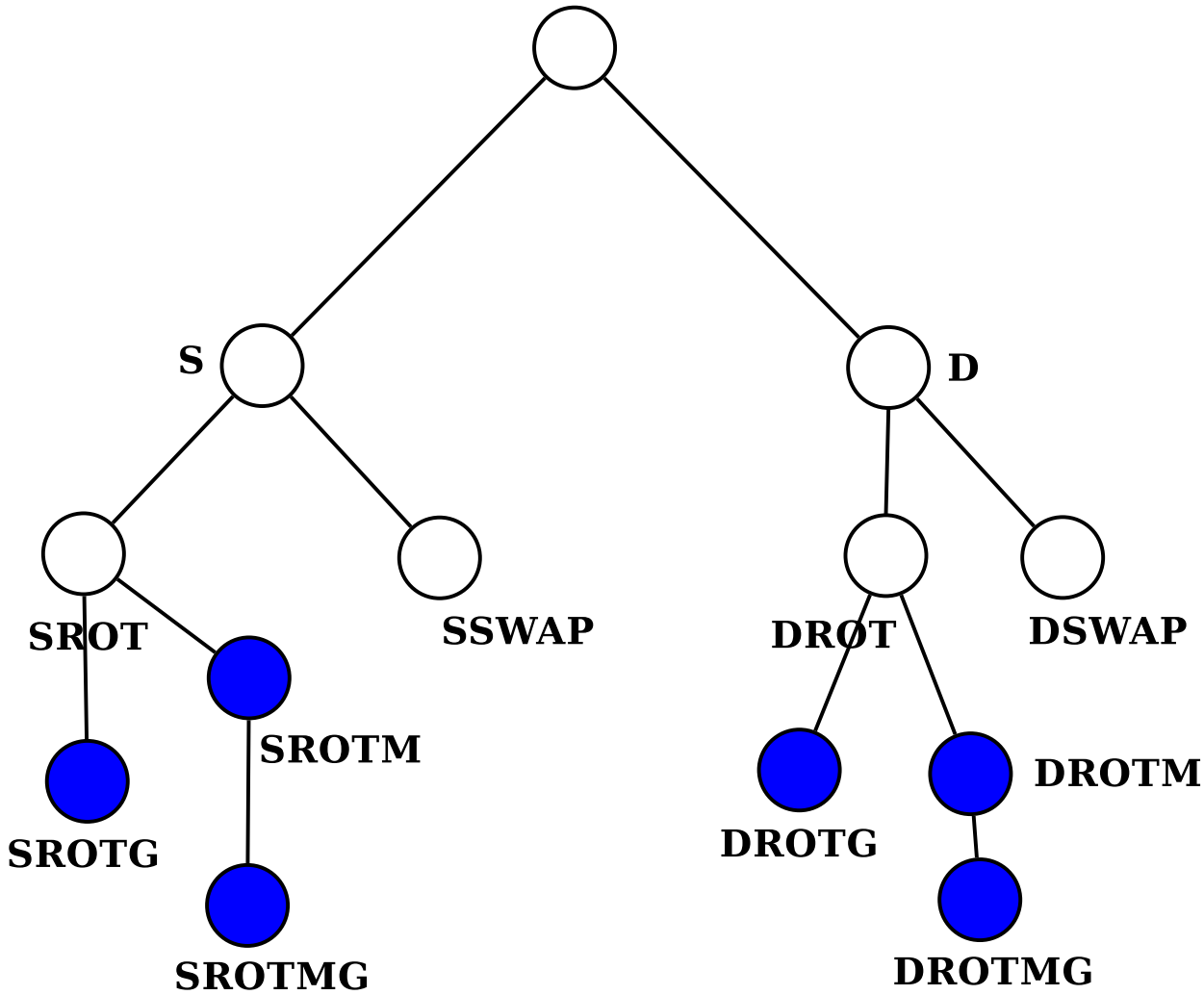

repair()
S
SROT
DSWAP
DROTMG
D
SROTM
SSWAP
SROTG
DROT
SROTMG
DROTG
DROTM

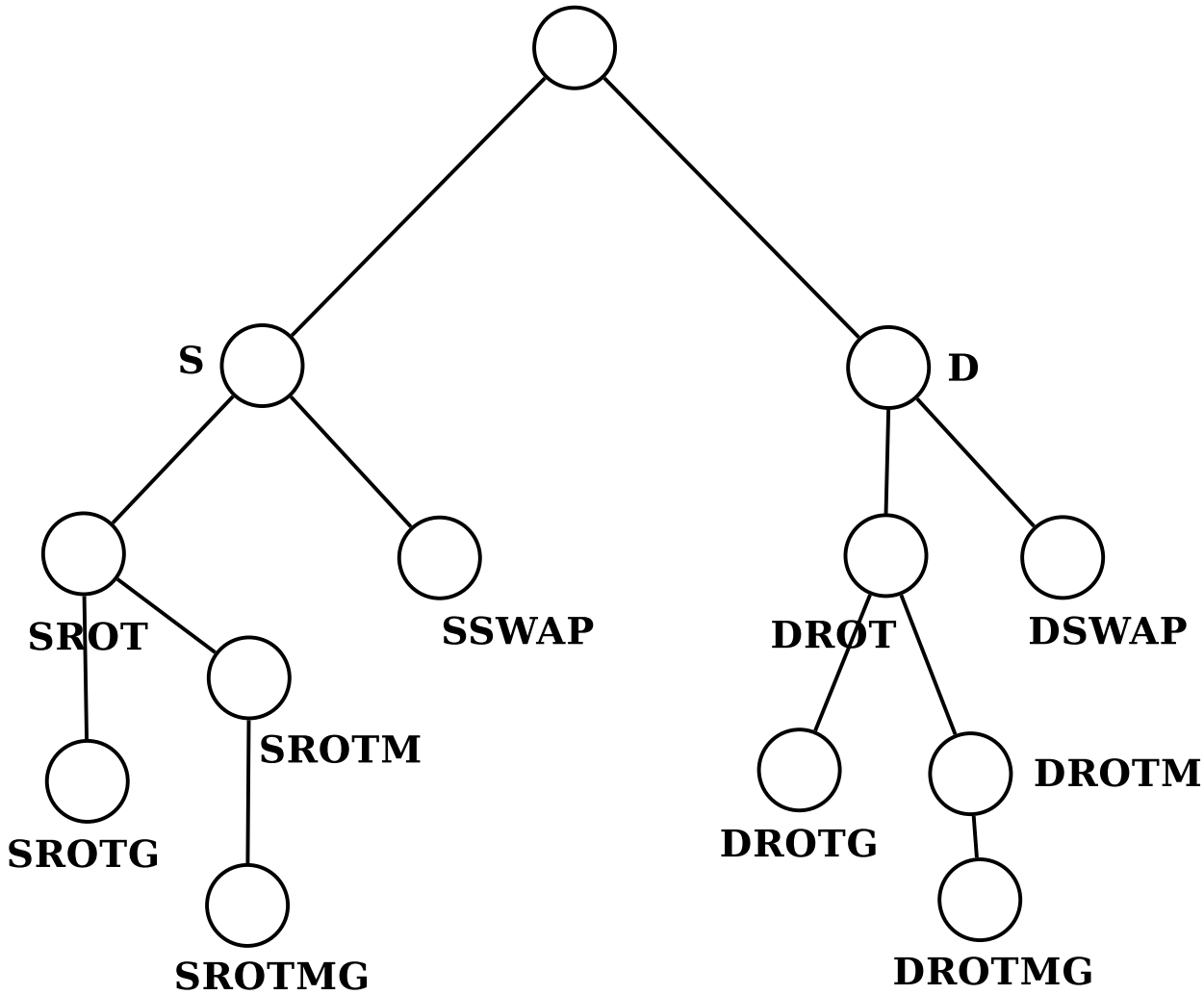

S
D
SROT
SROTM
SSWAP
DROTMG
DROT
DSWAP
SROTG
SROTMG
DROTG
DROTM


S
repair()
repair()
D
SROT
SROTM
SSWAP
DROTMG
DROT
DSWAP
SROTG
SROTMG
DROTG
DROTM


S
D
SROT
SSWAP
DROT
DSWAP
SROTM
SROTG
SROTMG
DROTMG
DROTG
DROTM


S
D
repair()
repair()
repair()
repair()
SROT
SSWAP
DROT
DSWAP
SROTM
SROTG
SROTMG
DROTMG
DROTG
DROTM

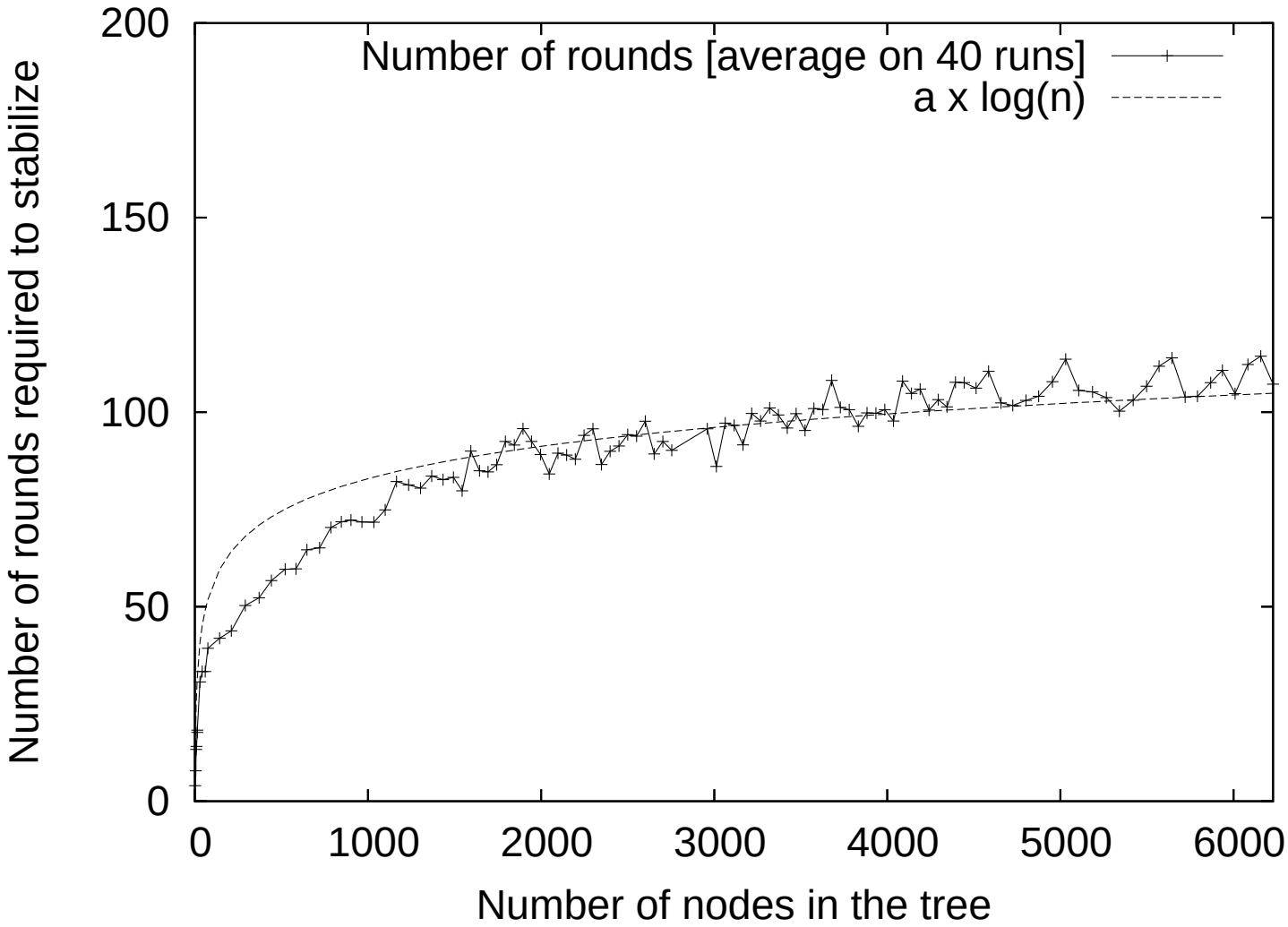

S
D
SROT
SSWAP
DROT
DSWAP
SROTM
SROTG
DROTM
DROTG
SROTMG
DROTMG


S
D
SROT
SSWAP
DROT
DSWAP
SROTM
DROTM
SROTG
DROTG
SROTMG
DROTMG

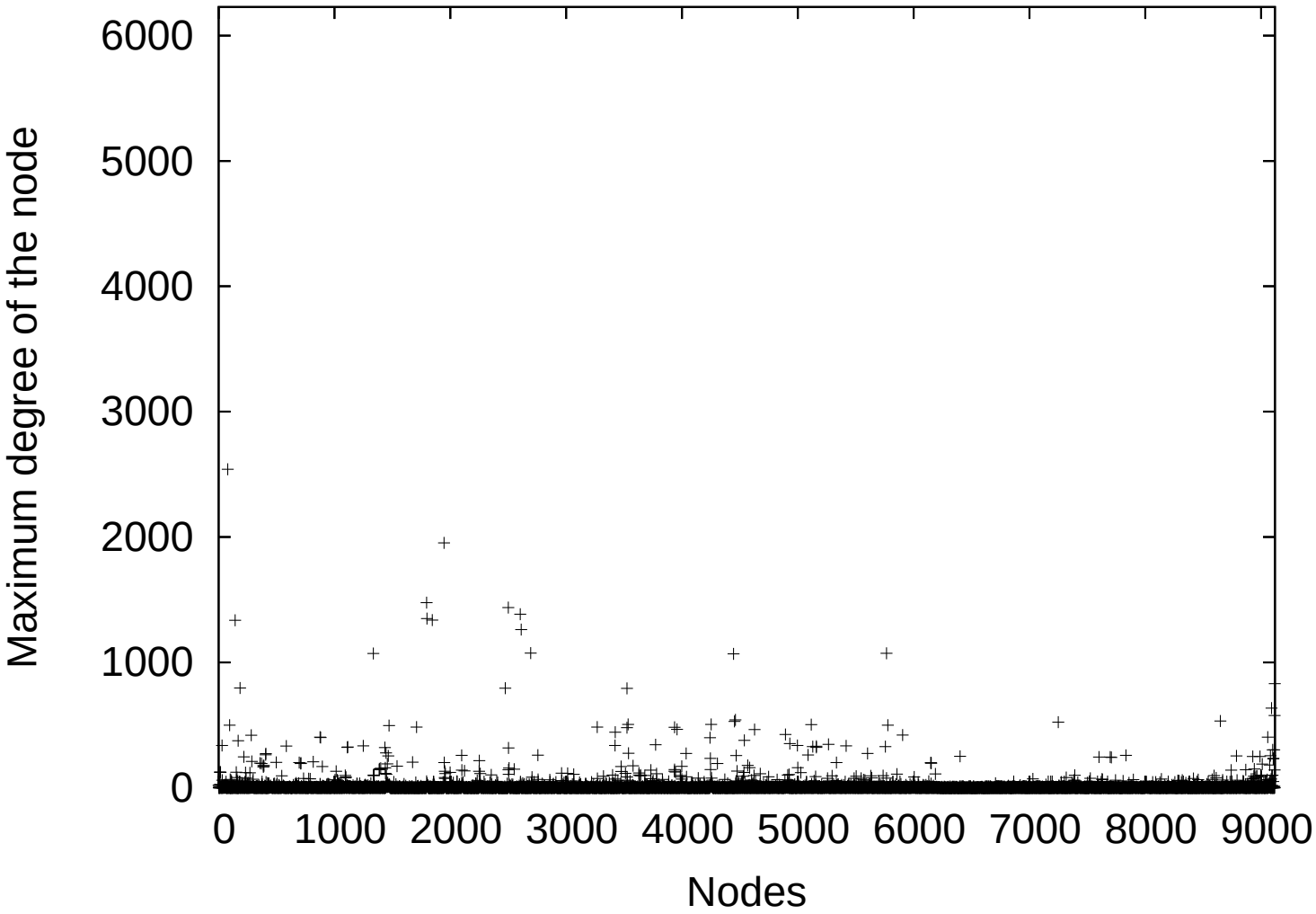
Complexities

- Average convergence time to PGCP tree : $O(h + h')$ rounds
- Worst case :
  - $O(n)$ rounds
  - $O(n^2)$ operations
  - $O(n)$ extra space (degree)

Simulation : Time Complexity



Simulation : Extra Space



Snap in State Model *vs* Self in Message Passing

Snap : Drawbacks

- Assumes a rooted connected tree
- Written in a coarse grain communication model

A New Protocol

- Work with any initial topology
- Designed in the message passing model

Snap in State Model *vs* Self in Message Passing

Snap : Drawbacks

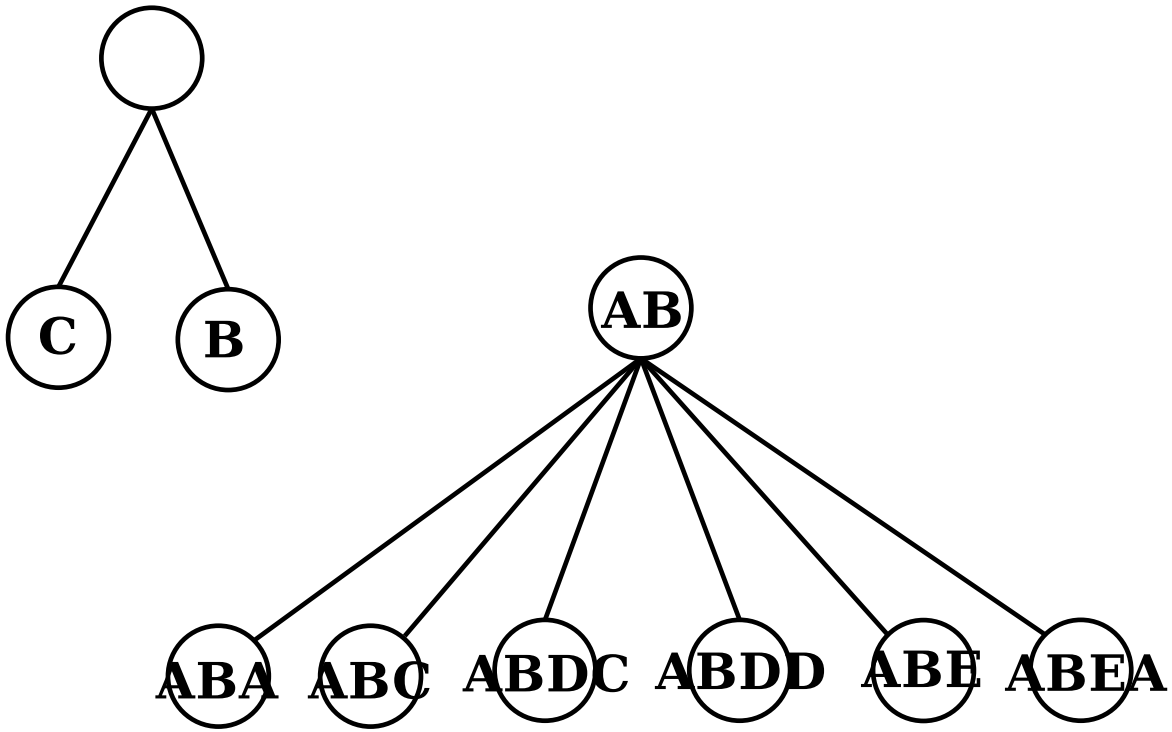
- Assumes a rooted connected tree
- Written in a coarse grain communication model

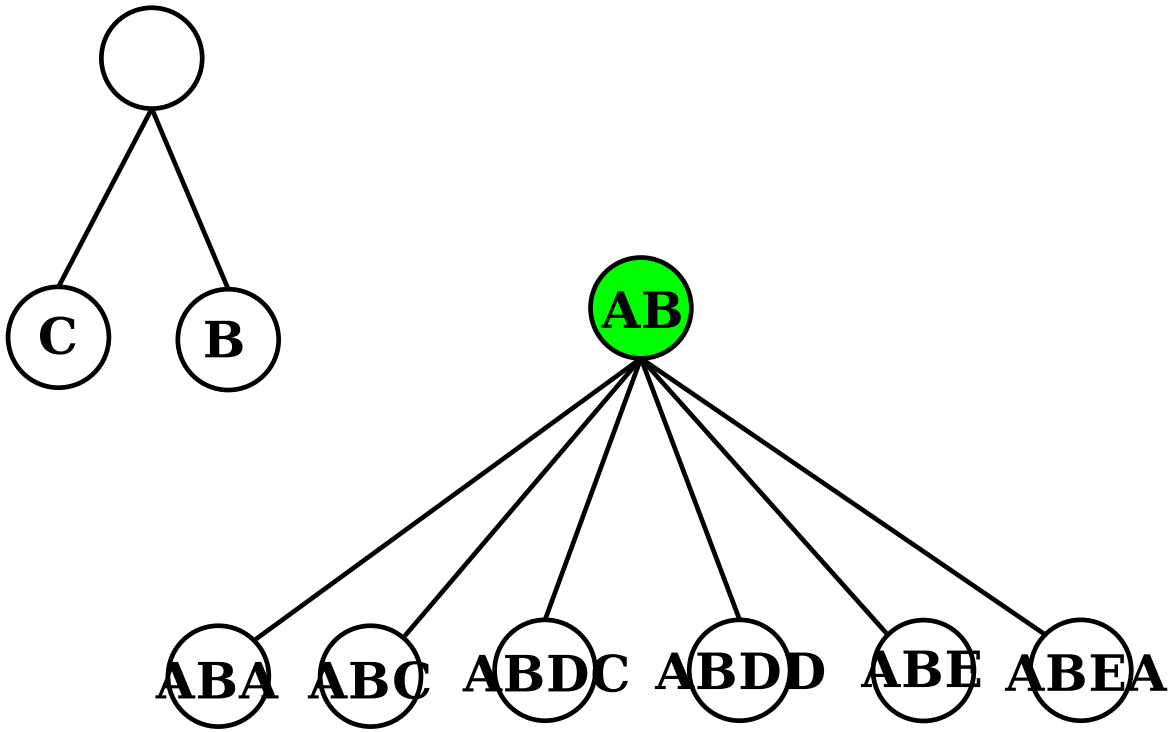
A New Protocol

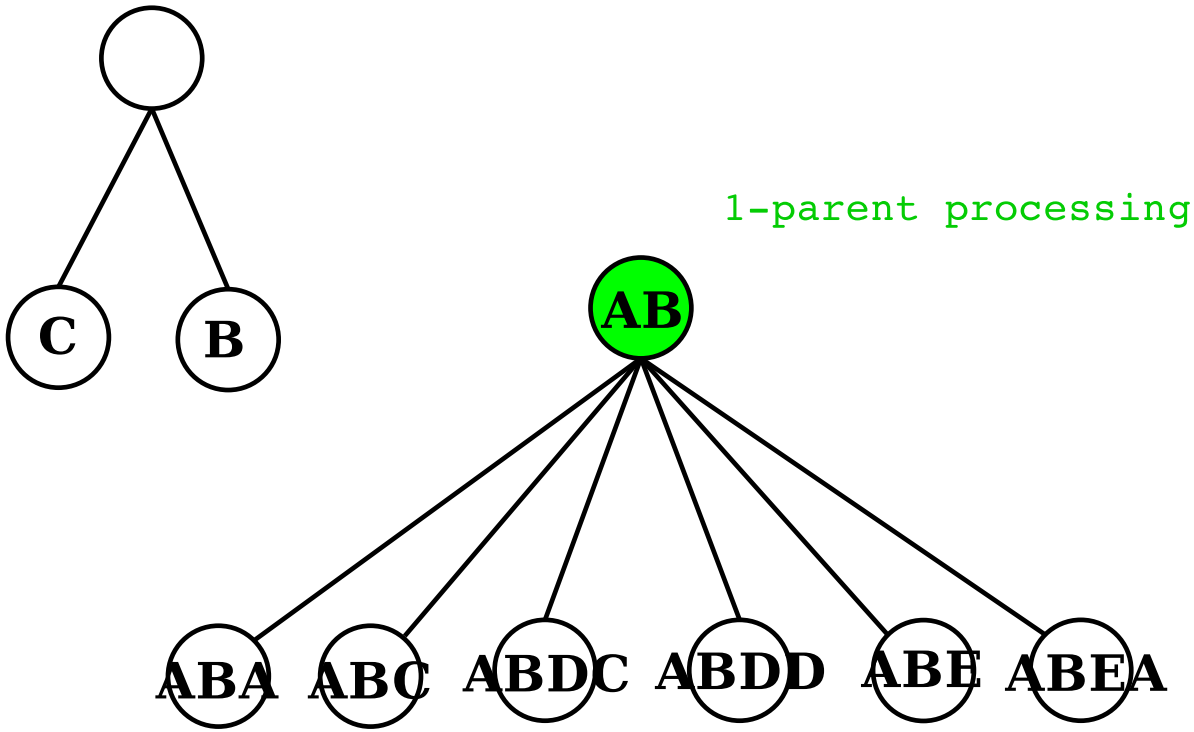
- Work with any initial topology
- Designed in the message passing model

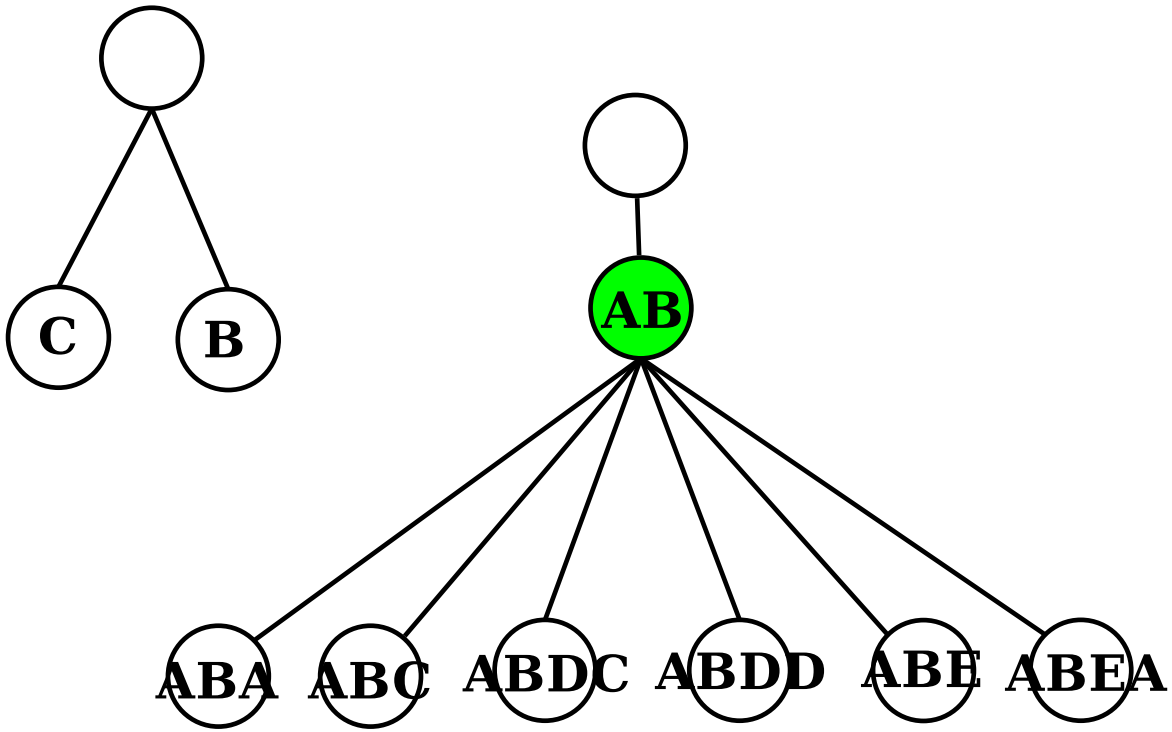
Steps

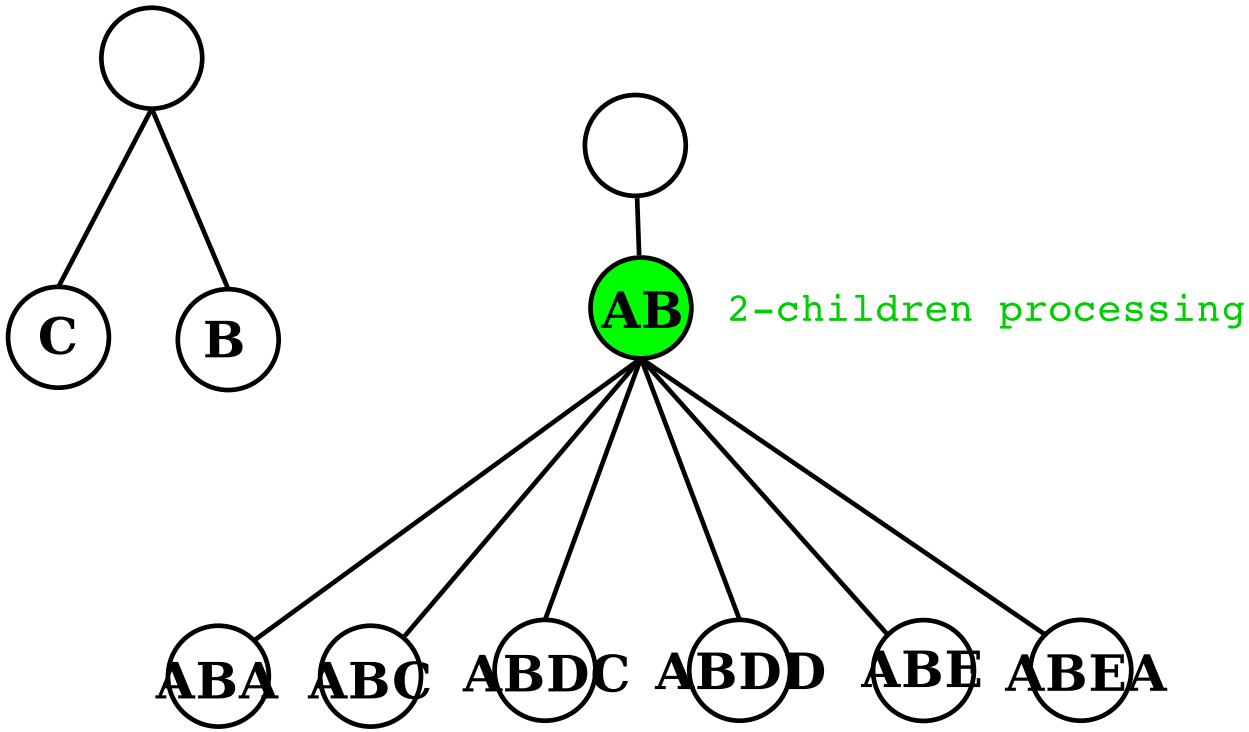
- Launched periodically on each node
- 1 - Parent processing
- 2 - Children processing

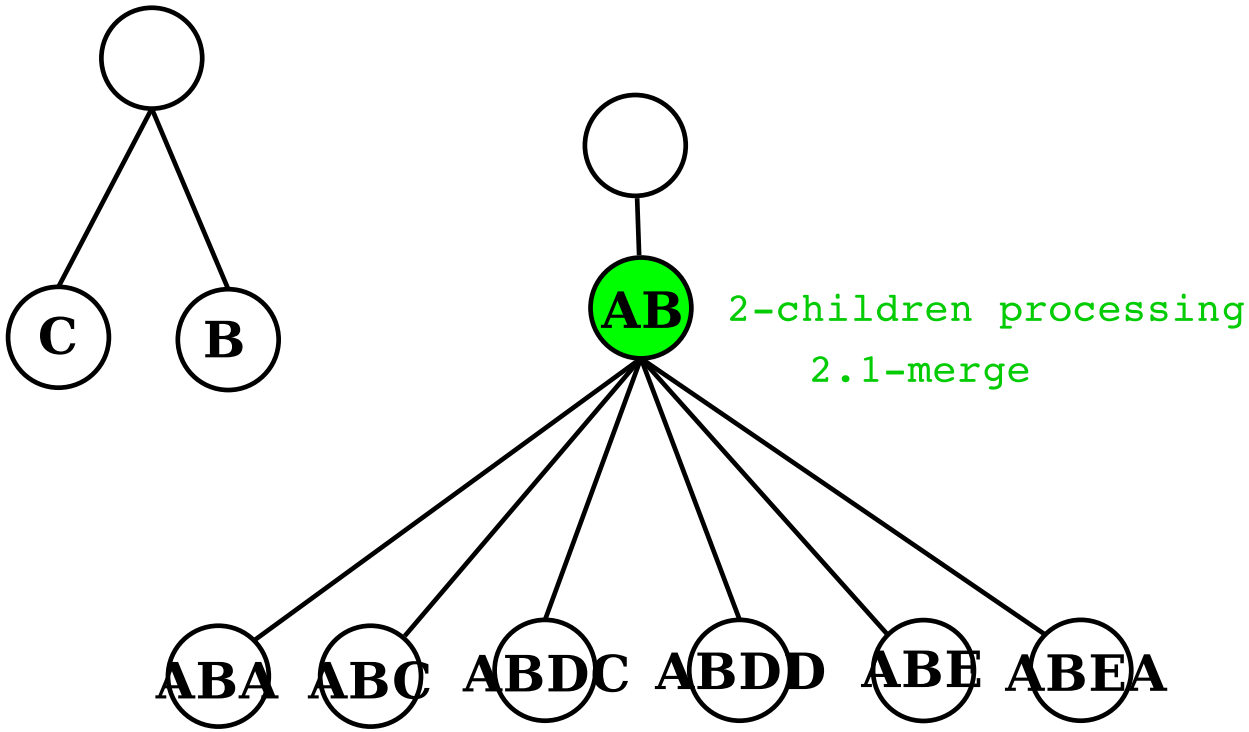

C
B
AB
ABA
ABC
ABDC
ABDD
ABE
ABEA

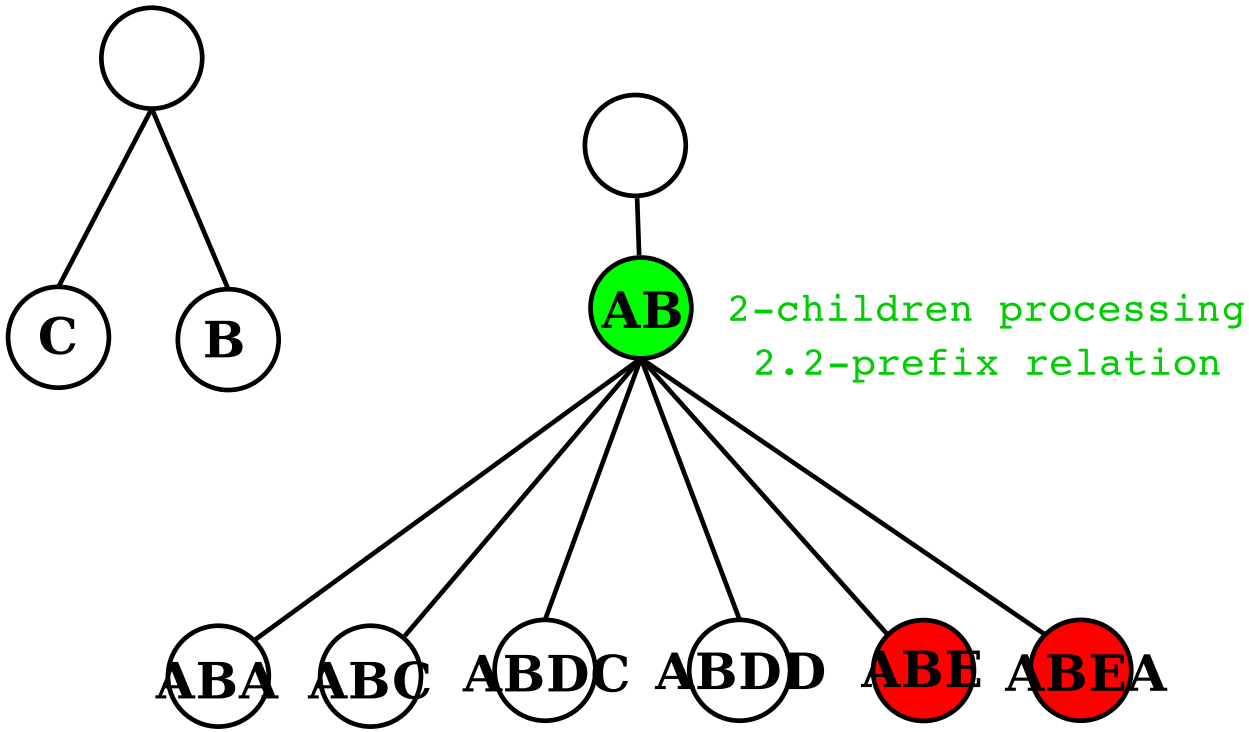

C
B
AB
ABA
ABC
ABDC
ABDD
ABE
ABEA

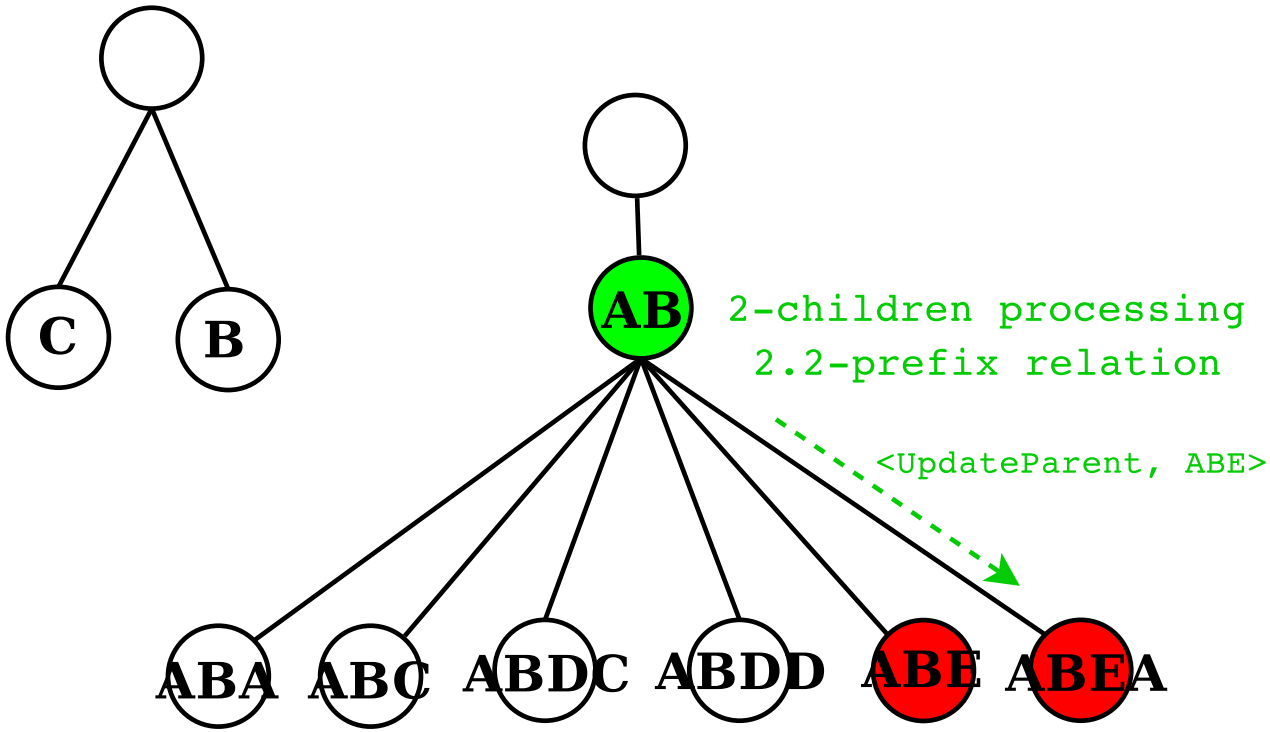

C
B
1-parent processing
AB
ABA
ABC
ABDC
ABDD
ABE
ABEA

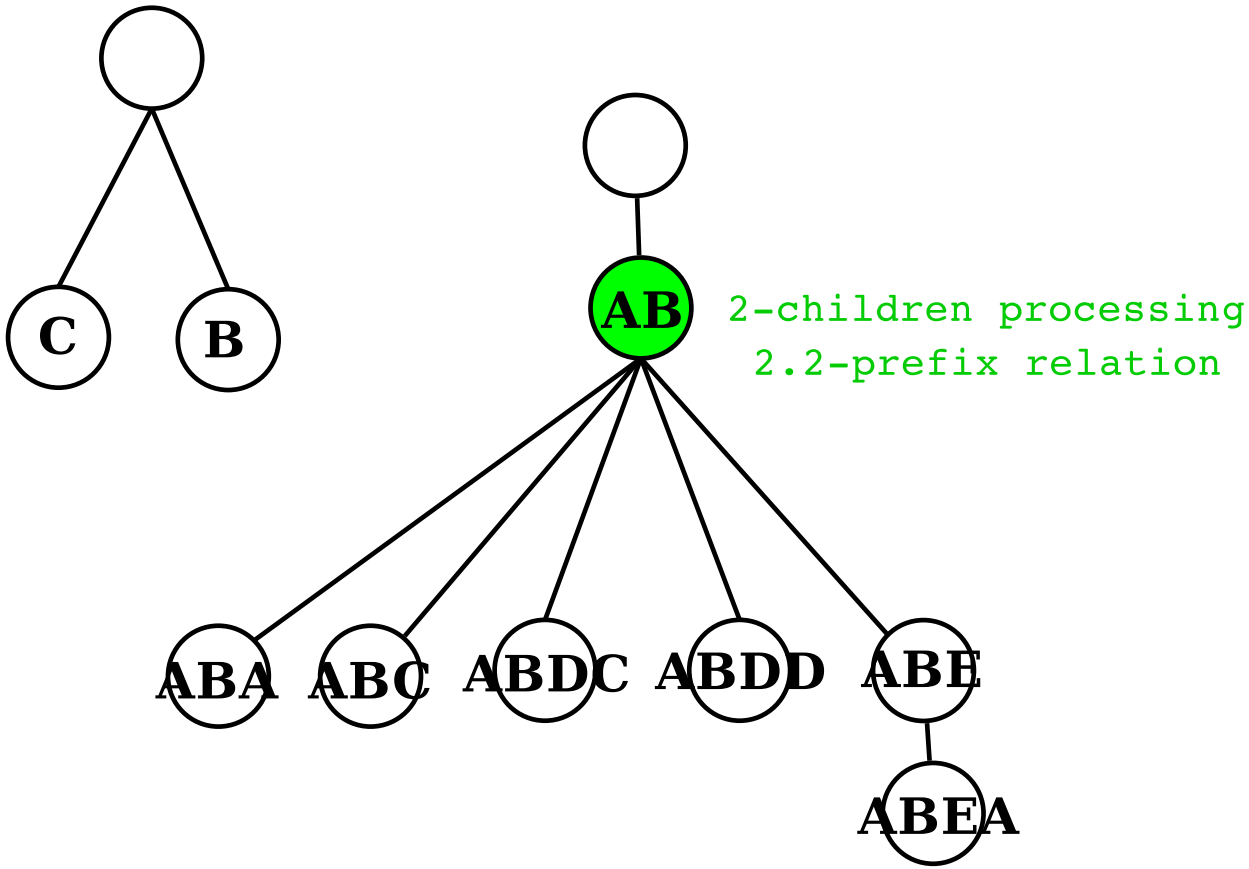

C
B
AB
ABA
ABC
ABDC
ABDD
ABE
ABEA

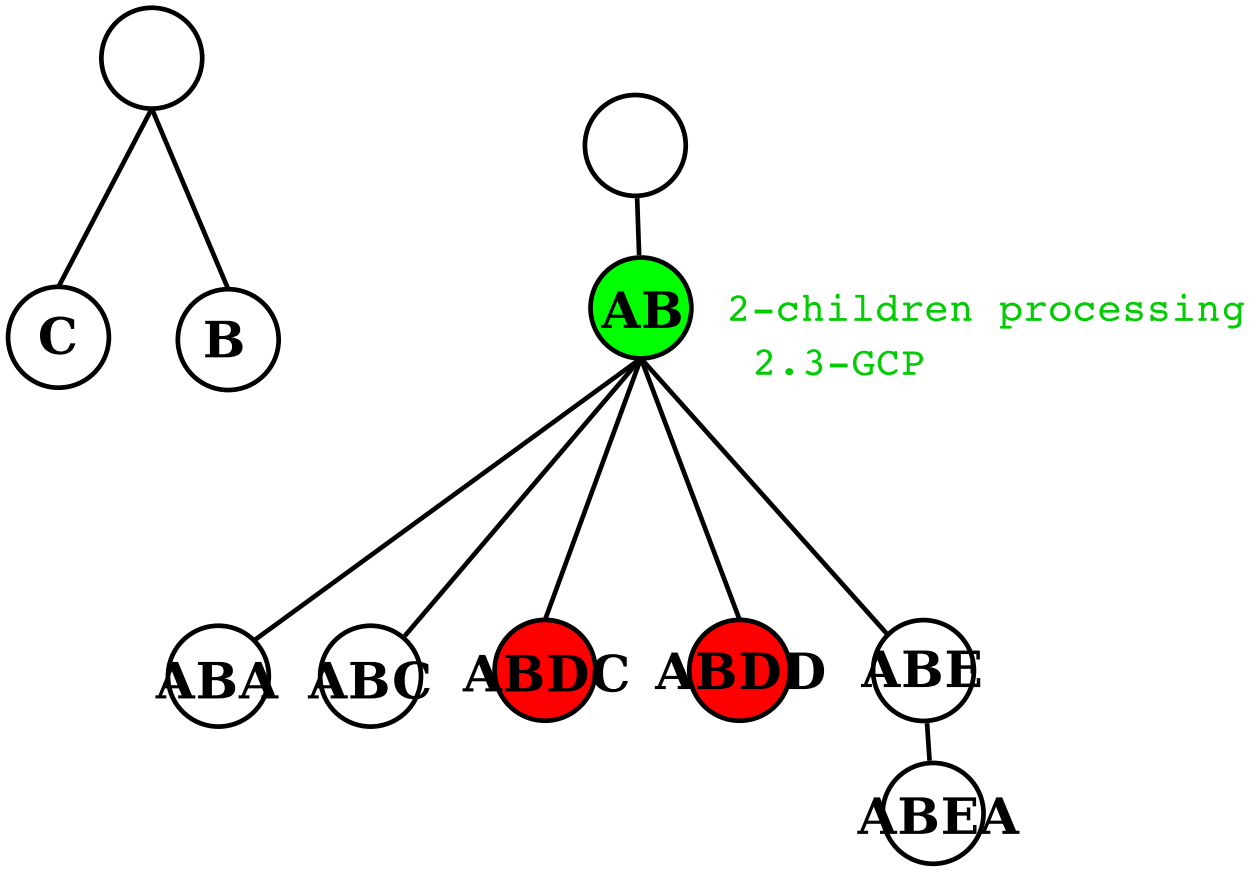

C
B
AB
2-children processing
ABA
ABC
ABDC
ABDD
ABE
ABEA

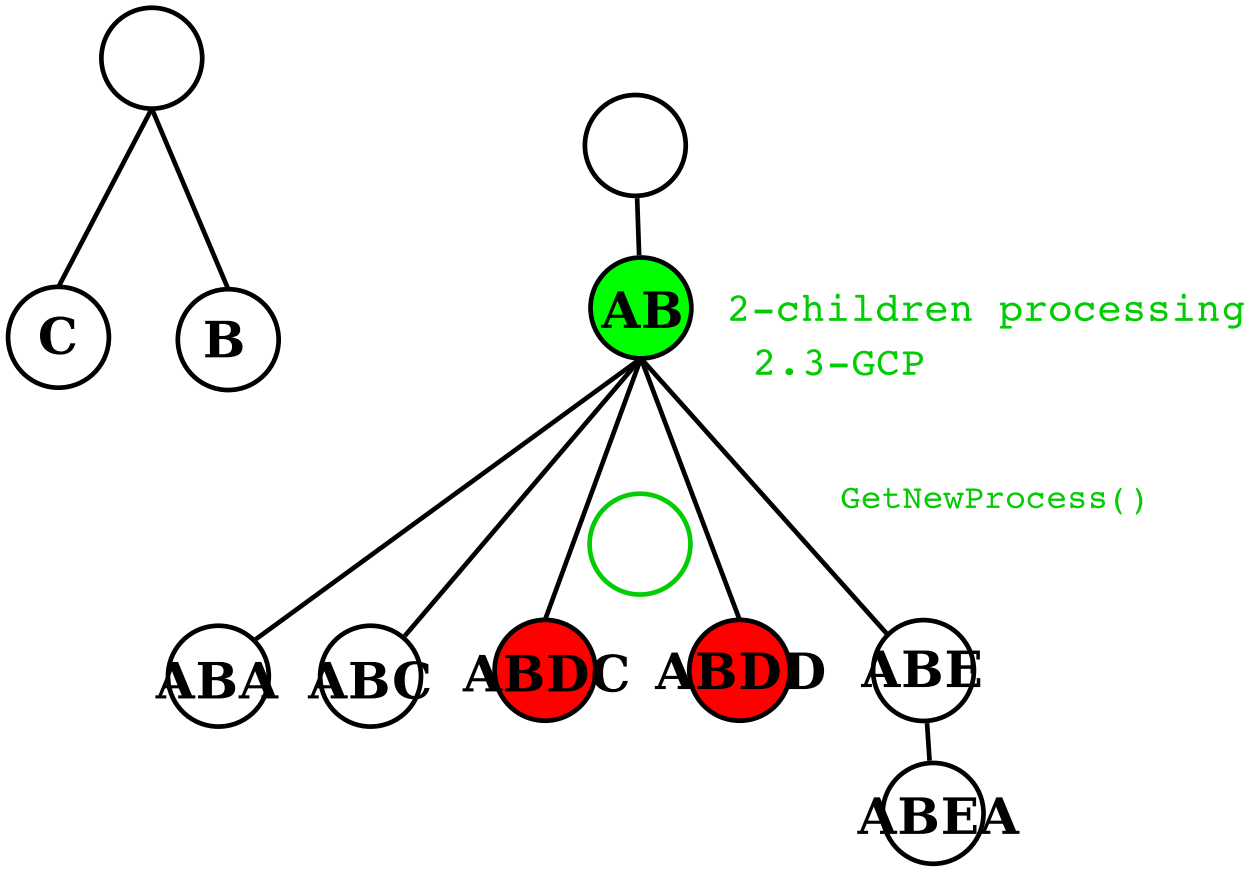

C
B
AB
2-children processing
2.1-merge
ABA
ABC
ABDC
ABDD
ABE
ABEA

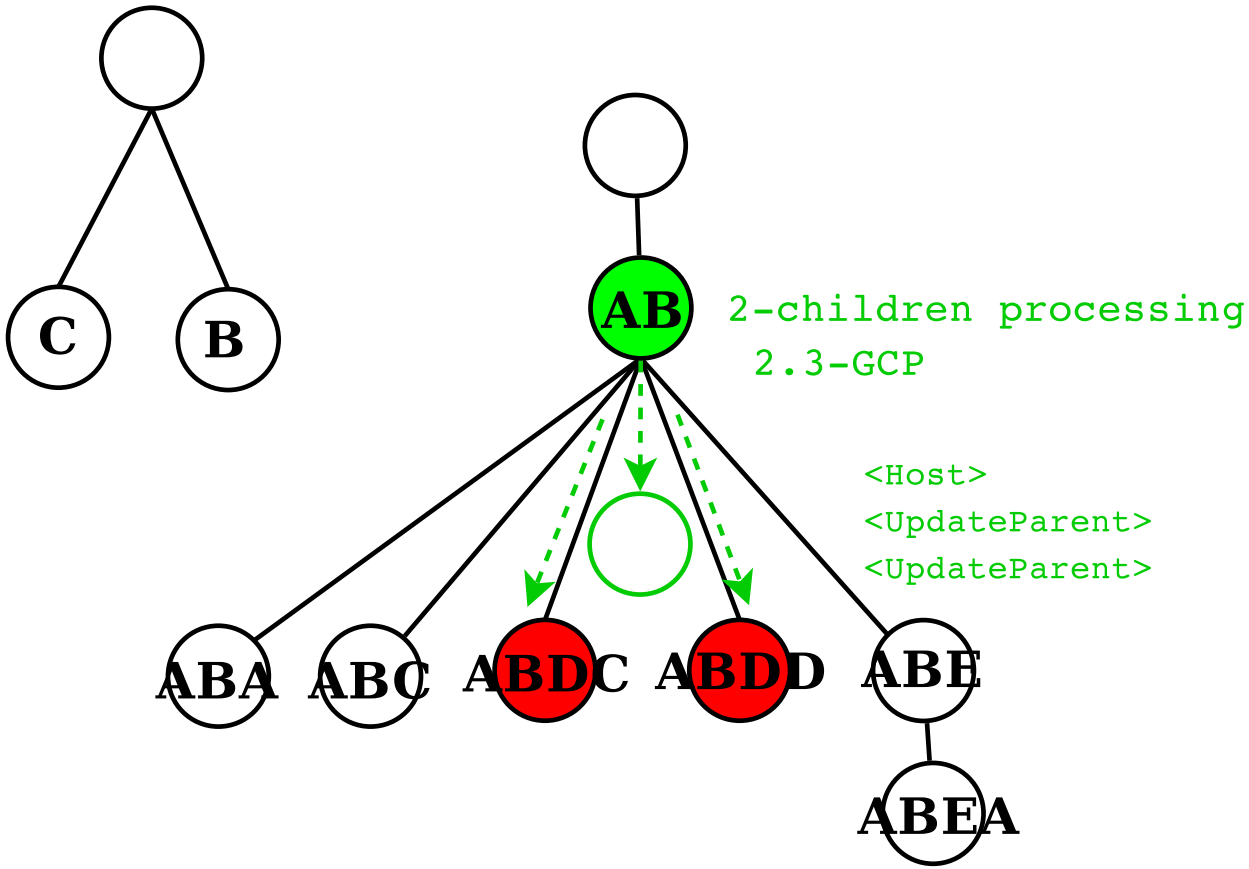

C
B
AB
2-children processing
2.2-prefix relation
ABA
ABC
ABDC
ABDD
ABE
ABEA

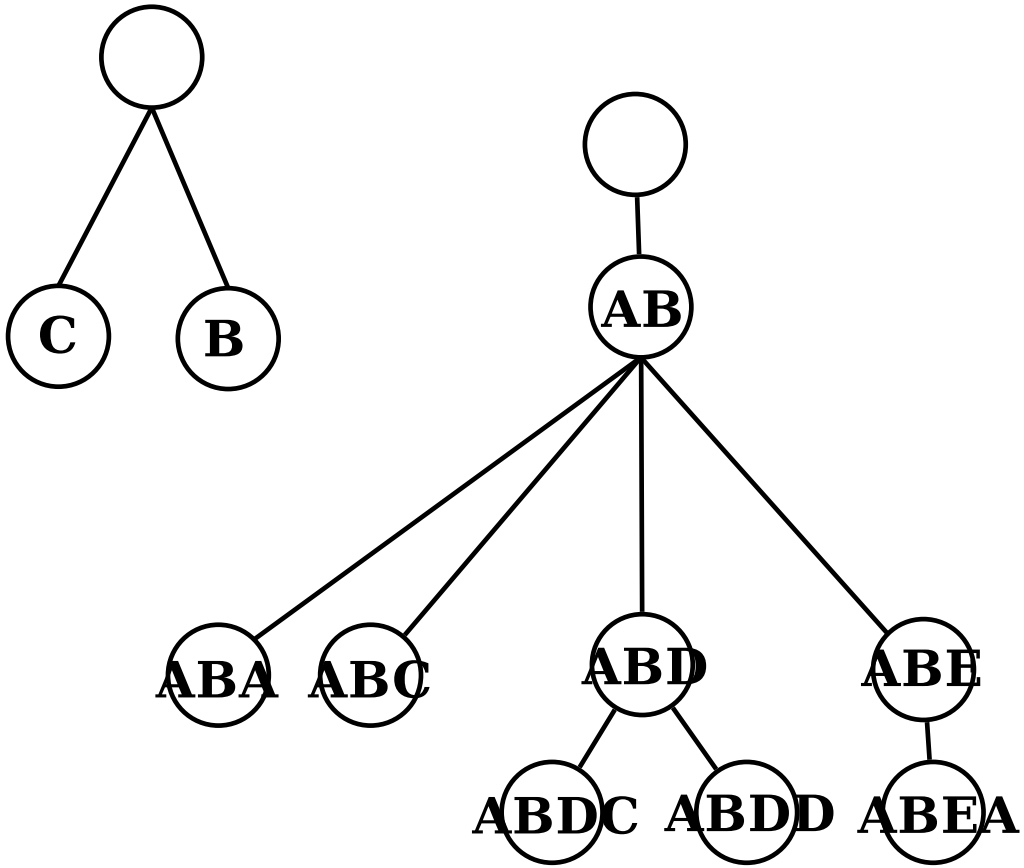

C
B
AB
2-children processing
2.2-prefix relation
<UpdateParent, ABE>
ABA
ABC
ABDC
ABDD
ABE
ABEA

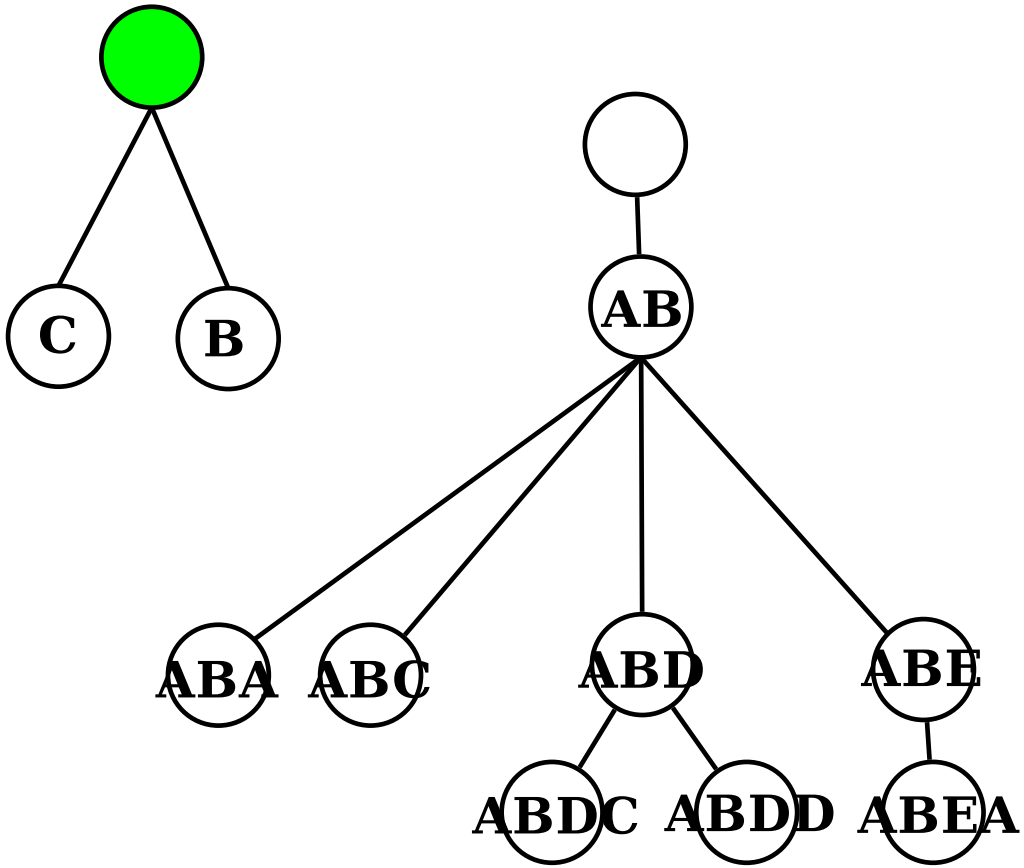

C
B
AB
2-children processing
2.2-prefix relation
ABA
ABC
ABDC
ABDD
ABE
ABEA

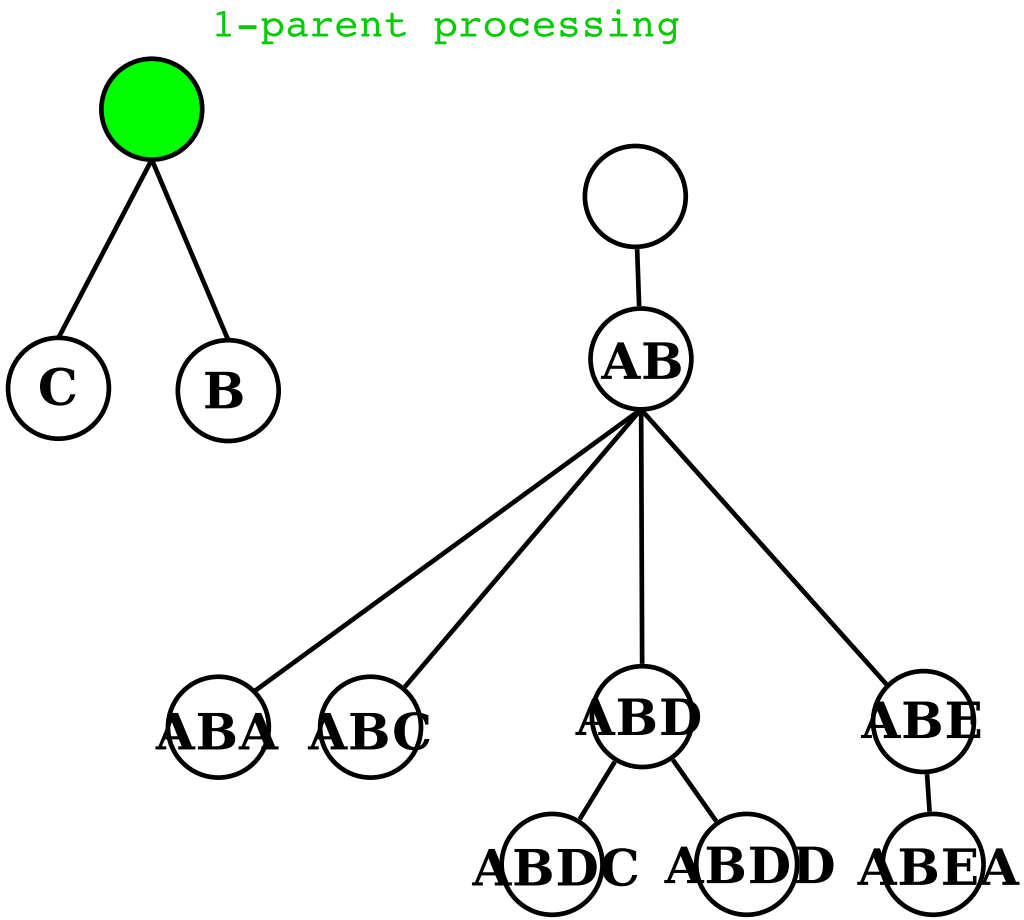

C
B
AB
2-children processing
2.3-GCP
ABA
ABC
ABDC
ABDD
ABE
ABEA

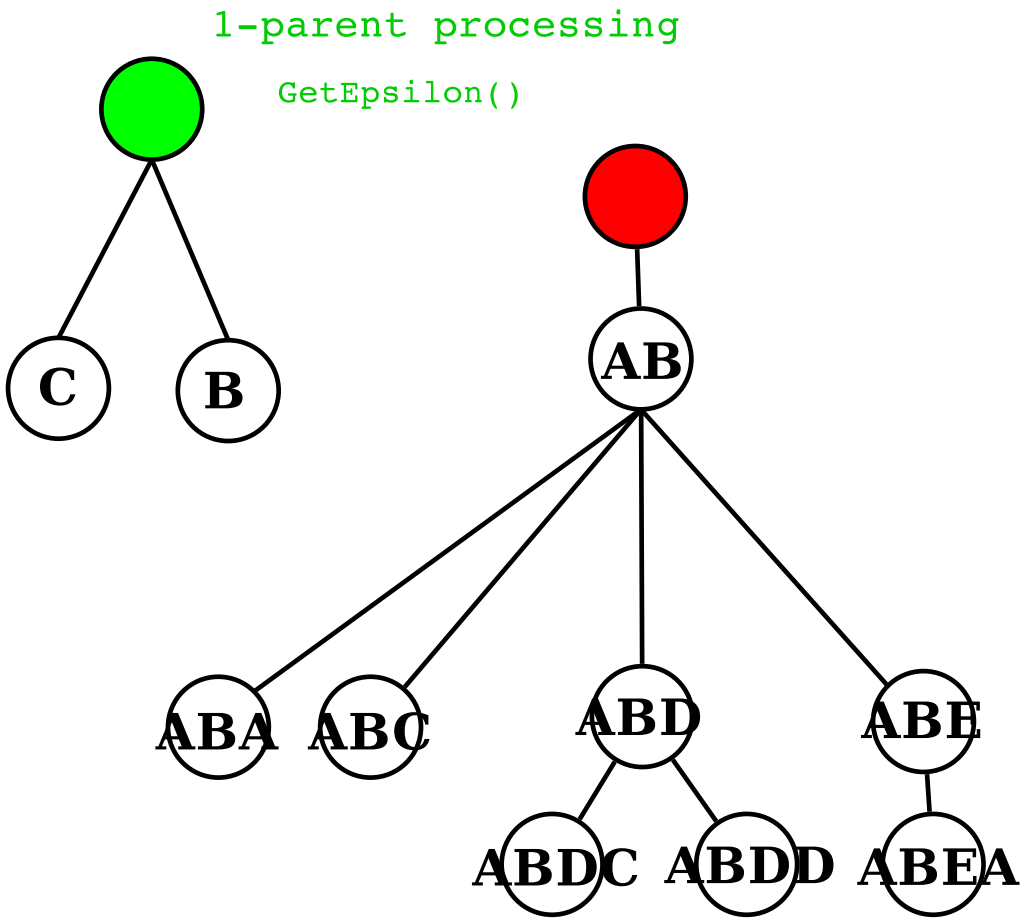

C
B
AB
2-children processing
2.3-GCP
GetNewProcess()
ABA
ABC
ABDC
ABDD
ABE
ABEA

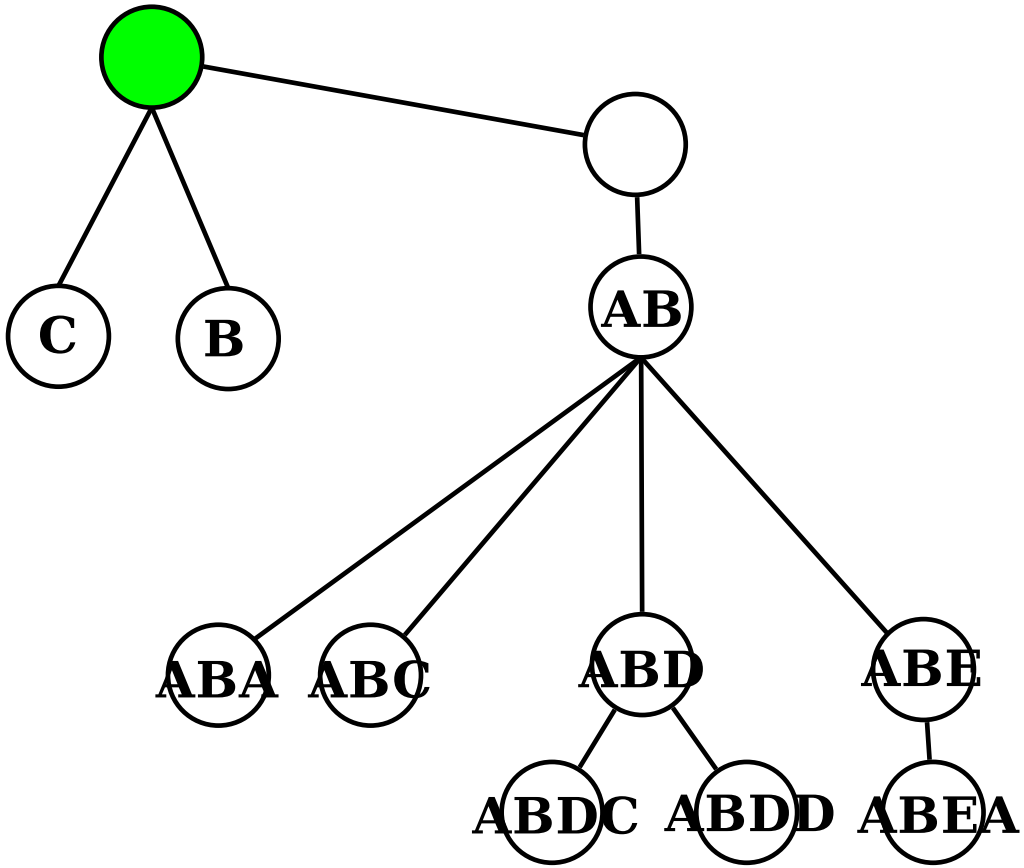

C
B
AB
2-children processing
2.3-GCP
<Host>
<UpdateParent>
<UpdateParent>
ABA
ABC
ABDC
ABDD
ABE
ABEA

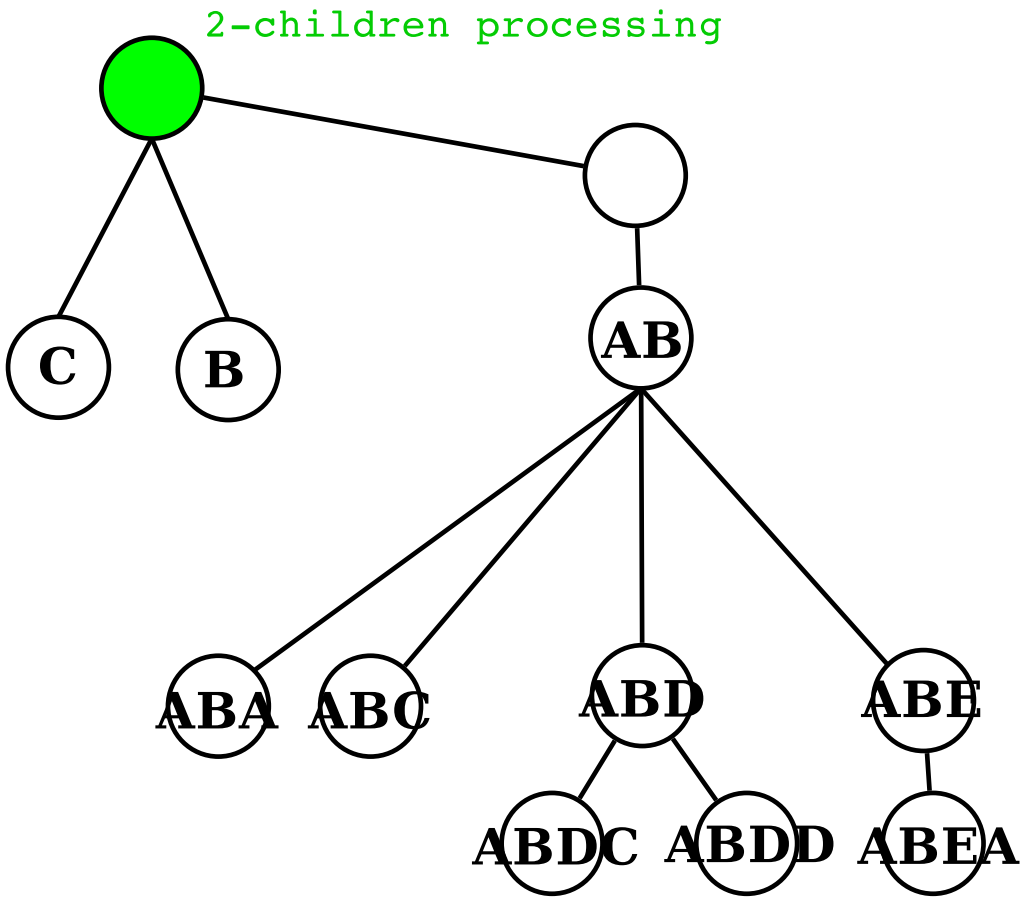

C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA

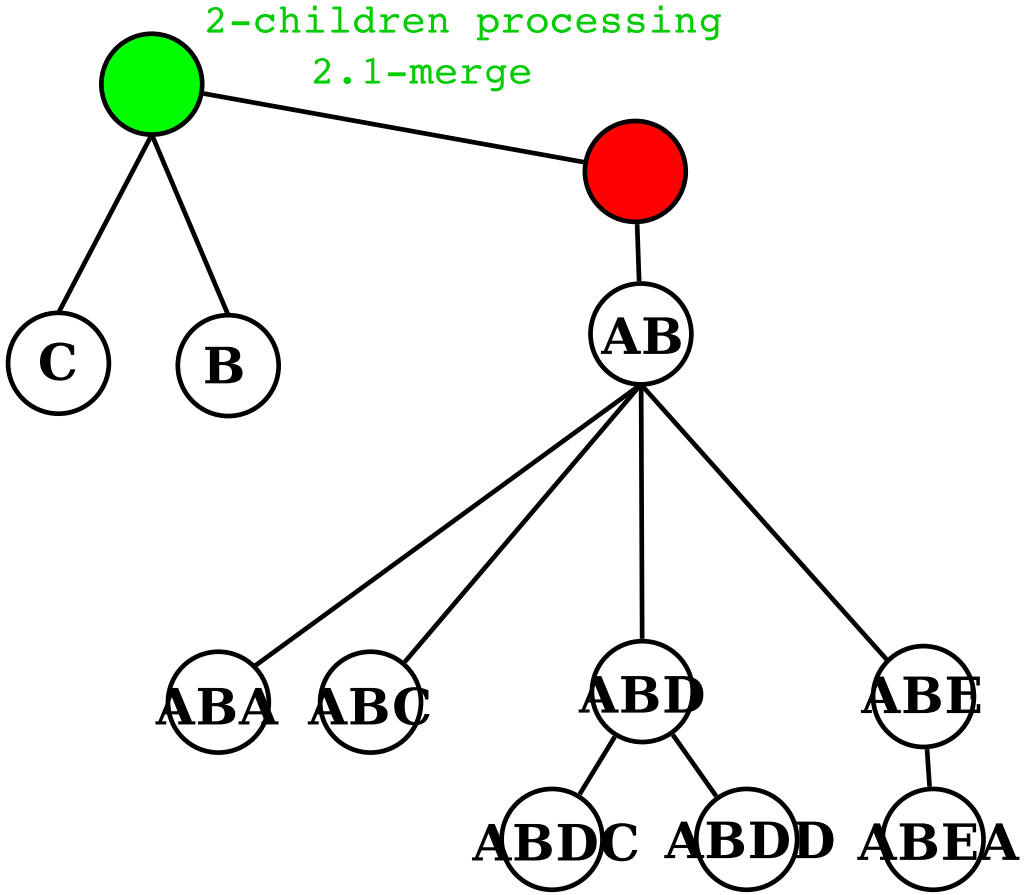

C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA

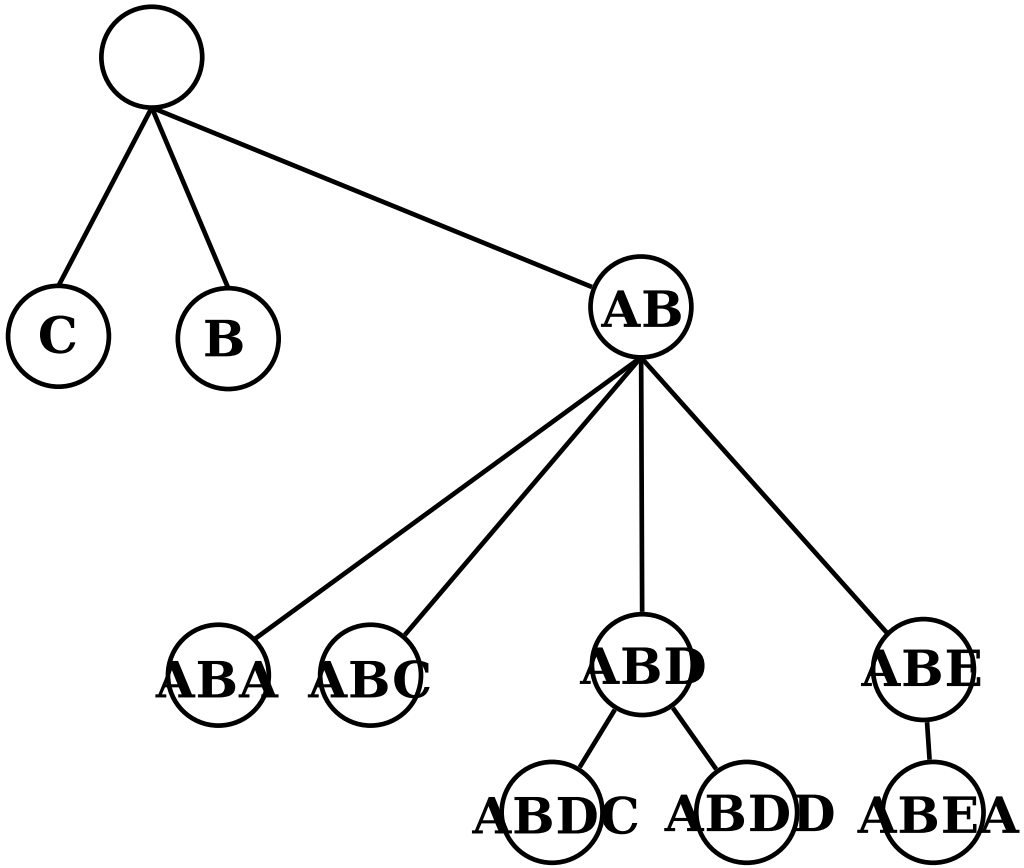

1-parent processing
C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA


1-parent processing
GetEpsilon()
C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA

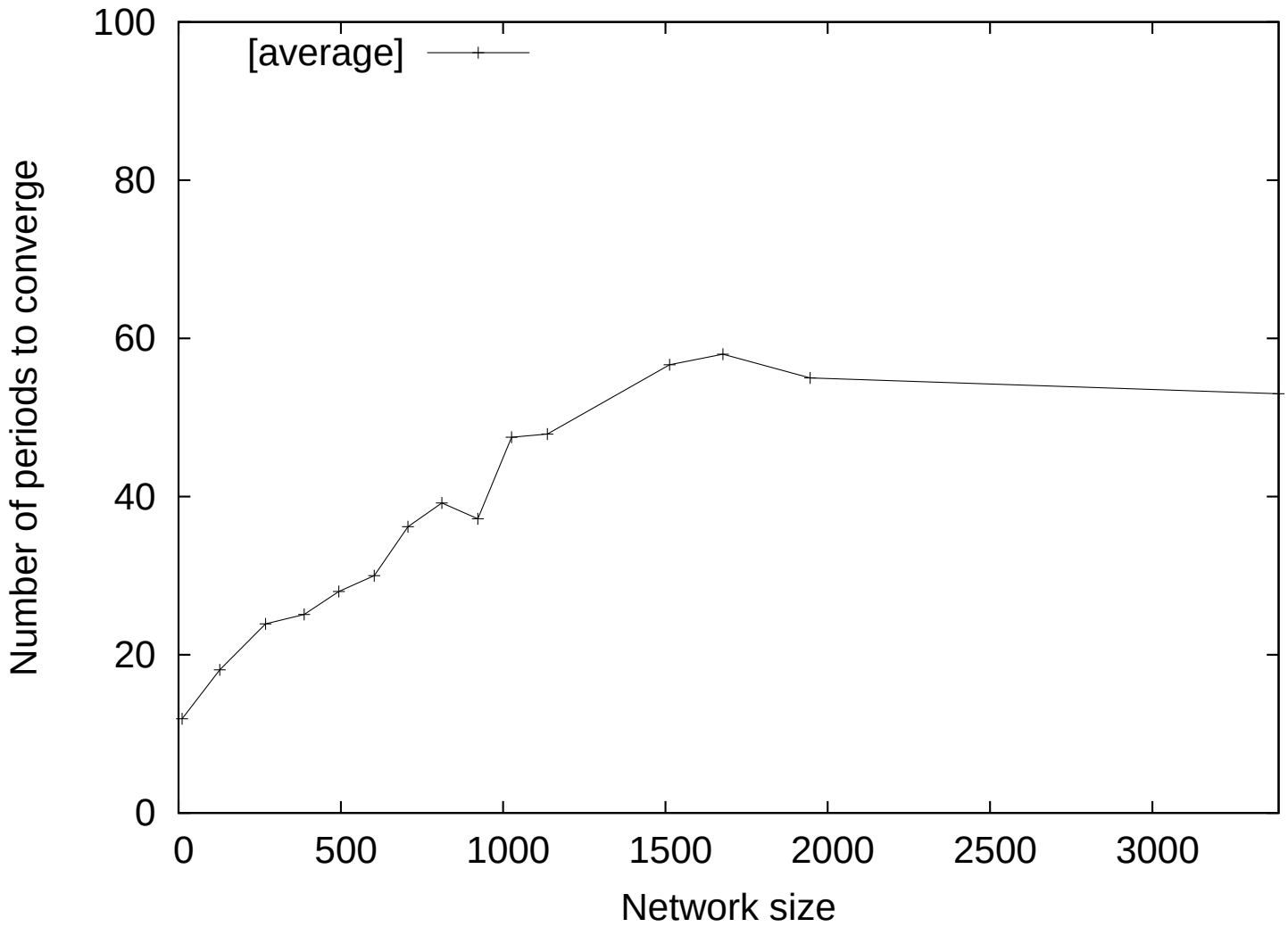

C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA


2-children processing
C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA

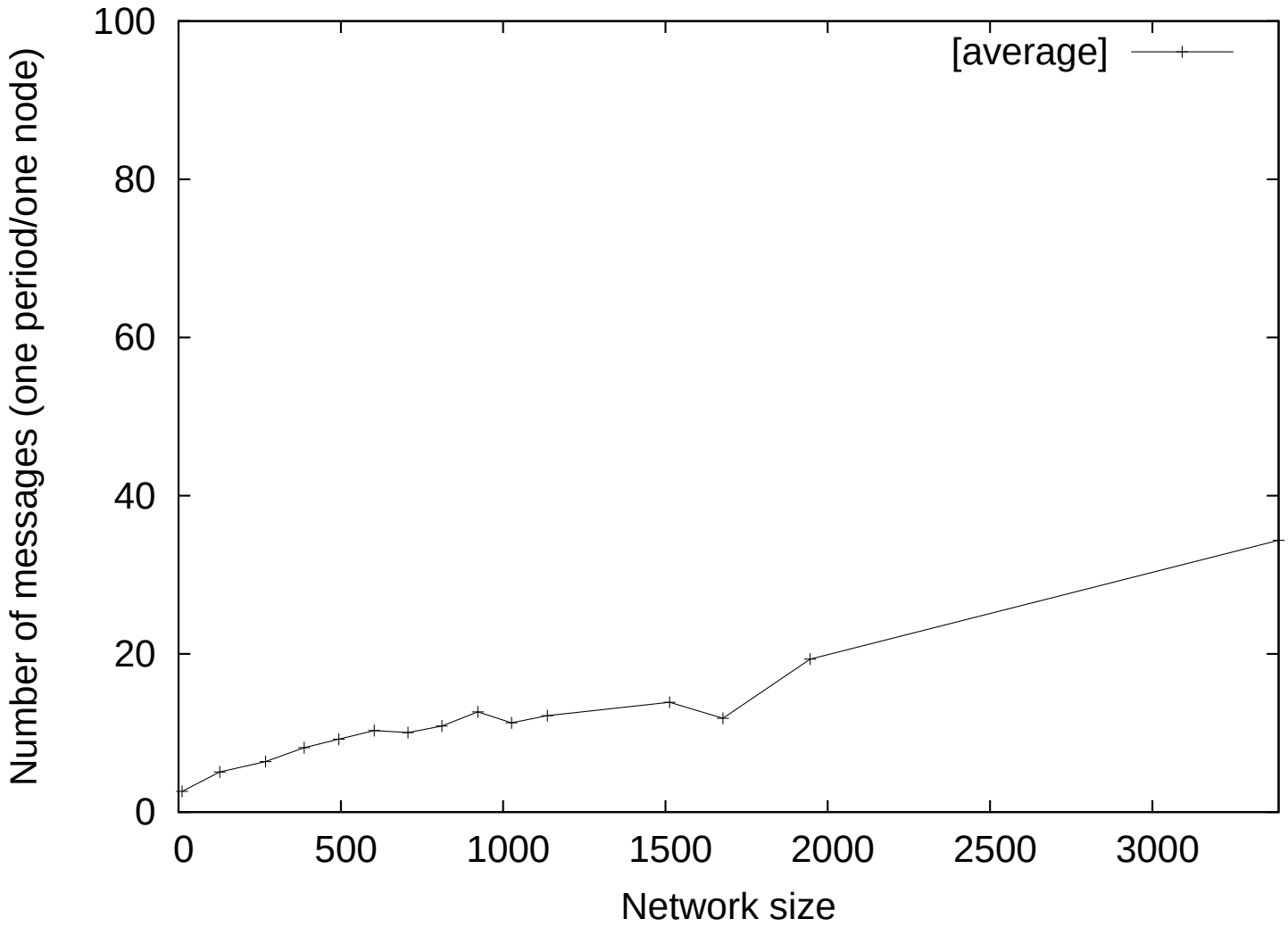

2-children processing
2.1-merge
C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA


C
B
AB
ABA
ABC
ABD
ABE
ABDC
ABDD
ABEA

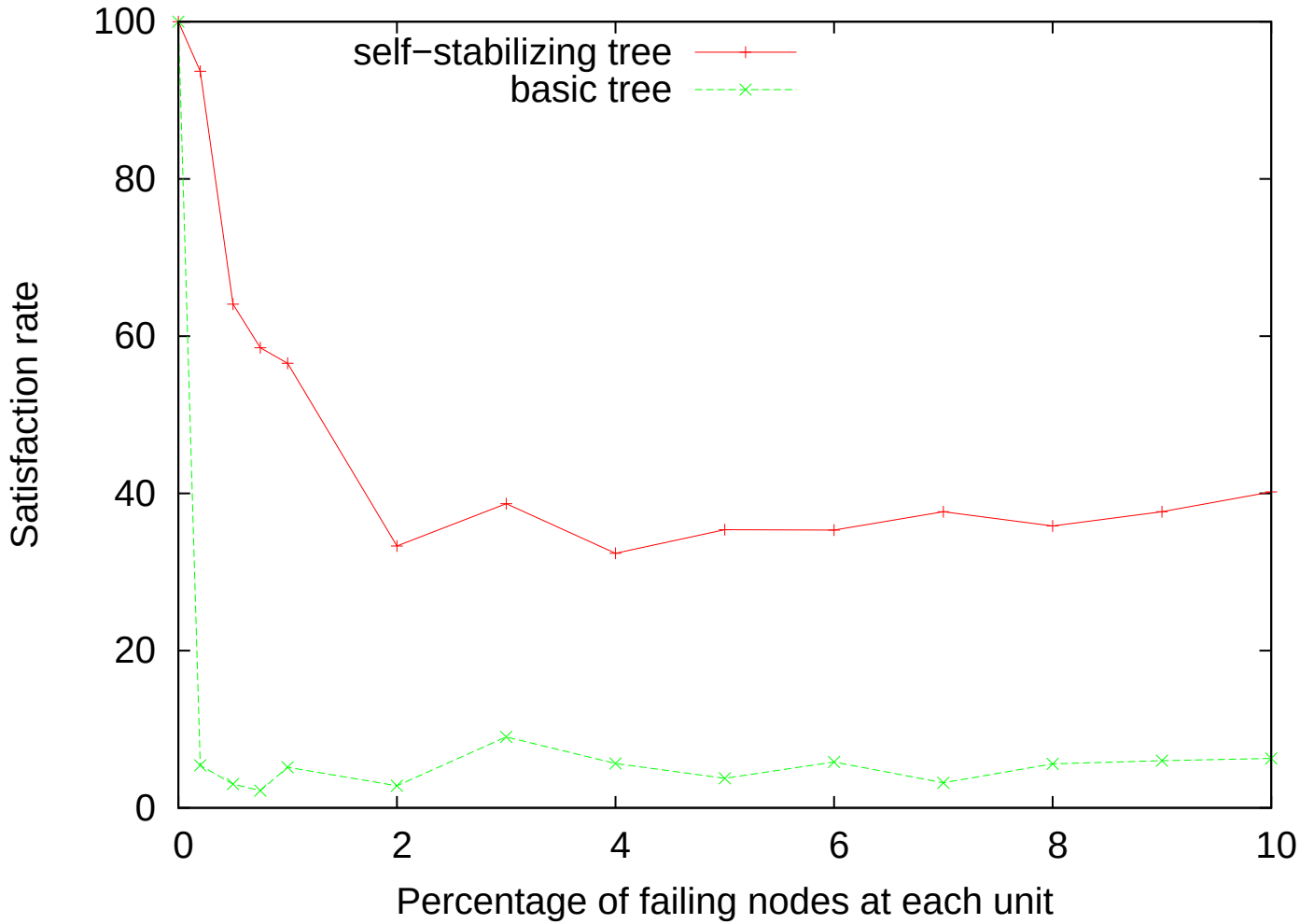
Simulation : Convergence Time



Simulation : Communication Amount



Simulation : Degradation

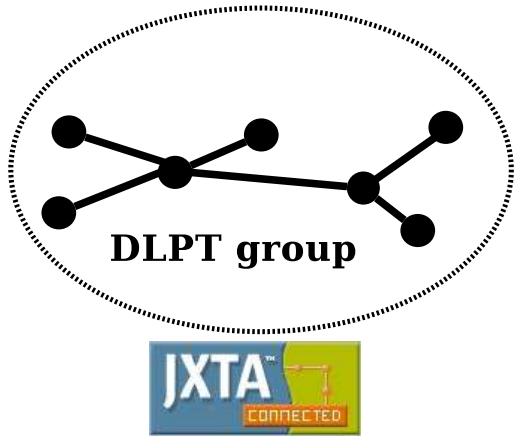


Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

Physical Layer

- JXTA-based
- JXTA discovery
- "**DLPT**" JXTA-group

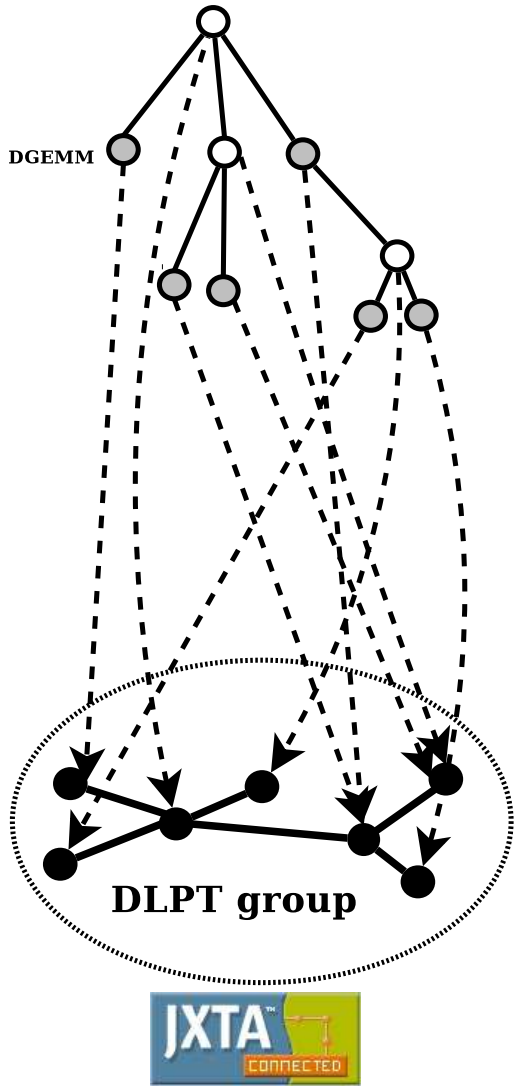


Logical Layer

- Explicit tree topology maintenance
- Random distribution of nodes on peers
- Message in the tree :
  - Serialized communication between JXTA peers
  - JXTA communications

Physical Layer

- JXTA-based
- JXTA discovery
- "**DLPT**" JXTA-group



Experiments : Set-Up

Platform used

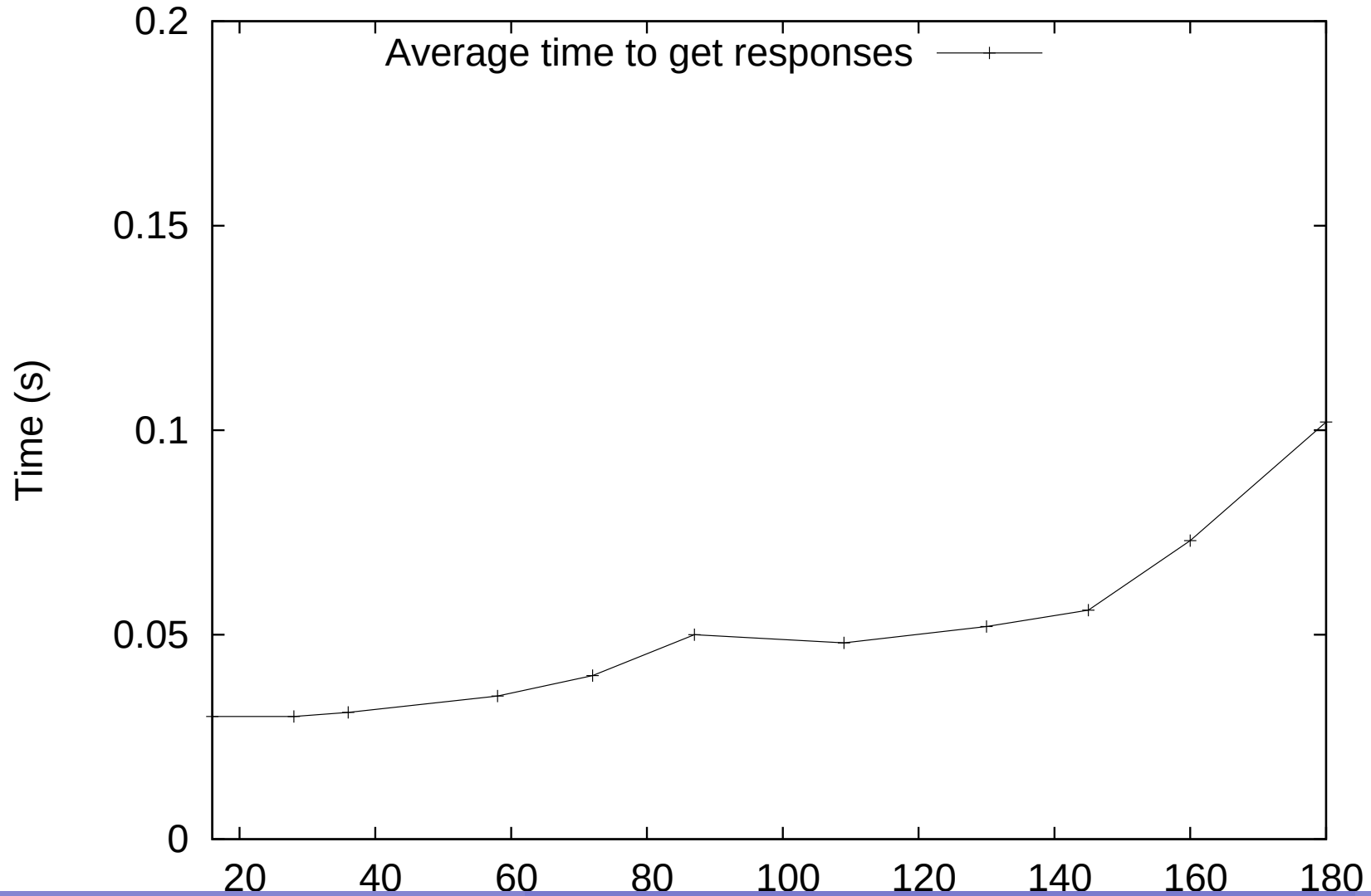
Grid'5000 platform

- `Capricorne` cluster in Lyon
  - 25 nodes (2 x AMD Opteron 246)
  - Gigabit Ethernet (Myrinet-2000 cards)
- `Grelon` cluster in Nancy
  - 60 nodes (2 x Intel Xeon 5110)
  - Gigabit Ethernet (Broadcom BCM5721 cards)

Steps

- Physical deployment
- Simulation of servers (insertion requests)
- Simulation of clients (discovery requests)

Experiments : Capricorne Results



Experiments : `Grelon` Results

Nodes (Services)	Peers	Response time (s)
432 (300)	60	0.105

Outline

1. DLPT : a Trie-Based Solution
2. Mapping & Load Balancing
   - Protocol
   - Load Balancing
   - Simulation
3. Fault-tolerance
   - Snap-Stabilizing PGCP Tree (State Model)
   - Self-Stabilizing PGCP Tree (Message-Passing Model)
4. Prototype Implementation
   - Design
   - Early Experiments
5. Summary & Open Problems

Summary

- DLPT
  - A novel P2P-fashioned approach for service discovery
  - Structured and Expressive
- Mapping and load balancing
  - Self-contained architecture
  - Load balancing heuristics
- Fault-tolerance
  - Best-effort alternatives
  - Snap-stabilizing protocol in the state model
  - Self-stabilizing protocols in message-passing
- Software Prototype
  - Two layer architecture implemented
  - Promising first experiments

Summary

- DLPT
  - A novel P2P-fashioned approach for service discovery
  - Structured and Expressive
- Mapping and load balancing
  - Self-contained architecture
  - Load balancing heuristics
- Fault-tolerance
  - Best-effort alternatives
  - Snap-stabilizing protocol in the state model
  - Self-stabilizing protocols in message-passing
- Software Prototype
  - Two layer architecture implemented
  - Promising first experiments

Summary

- DLPT
  - A novel P2P-fashioned approach for service discovery
  - Structured and Expressive
- Mapping and load balancing
  - Self-contained architecture
  - Load balancing heuristics
- Fault-tolerance
  - Best-effort alternatives
  - Snap-stabilizing protocol in the state model
  - Self-stabilizing protocols in message-passing
- Software Prototype
  - Two layer architecture implemented
  - Promising first experiments

Summary

- DLPT
  - A novel P2P-fashioned approach for service discovery
  - Structured and Expressive
- Mapping and load balancing
  - Self-contained architecture
  - Load balancing heuristics
- Fault-tolerance
  - Best-effort alternatives
  - Snap-stabilizing protocol in the state model
  - Self-stabilizing protocols in message-passing
- Software Prototype
  - Two layer architecture implemented
  - Promising first experiments

Hints for Future Work

- Design
  - Topology awareness
  - Optimizations (cache, shortcuts)
- Mapping & Load balancing
  - Coupling our objective function with *K-choices*
  - Analysis results on the clustering obtained
  - Implementation
- Fault-tolerance
  - Snap-stabilization and message-passing
  - Self-stabilization and availability
  - Implementation
- Experiments
  - Load balancing
  - Self-stabilization
  - Larger scale

Hints for Future Work

- Design
  - Topology awareness
  - Optimizations (cache, shortcuts)
- Mapping & Load balancing
  - Coupling our objective function with *K-choices*
  - Analysis results on the clustering obtained
  - Implementation
- Fault-tolerance
  - Snap-stabilization and message-passing
  - Self-stabilization and availability
  - Implementation
- Experiments
  - Load balancing
  - Self-stabilization
  - Larger scale

Hints for Future Work

- Design
  - Topology awareness
  - Optimizations (cache, shortcuts)
- Mapping & Load balancing
  - Coupling our objective function with *K-choices*
  - Analysis results on the clustering obtained
  - Implementation
- Fault-tolerance
  - Snap-stabilization and message-passing
  - Self-stabilization and availability
  - Implementation
- Experiments
  - Load balancing
  - Self-stabilization
  - Larger scale

Hints for Future Work

- Design
  - Topology awareness
  - Optimizations (cache, shortcuts)
- Mapping & Load balancing
  - Coupling our objective function with *K-choices*
  - Analysis results on the clustering obtained
  - Implementation
- Fault-tolerance
  - Snap-stabilization and message-passing
  - Self-stabilization and availability
  - Implementation
- Experiments
  - Load balancing
  - Self-stabilization
  - Larger scale

Service Discovery : Open Problems

Different kinds of platforms

- Petascale computing
- Research grids
- Desktop computing

Different kinds of volatility

- Failures
- Crashes
- Resources sharing

Thx !