Fault-Tolerant Distributed Systems

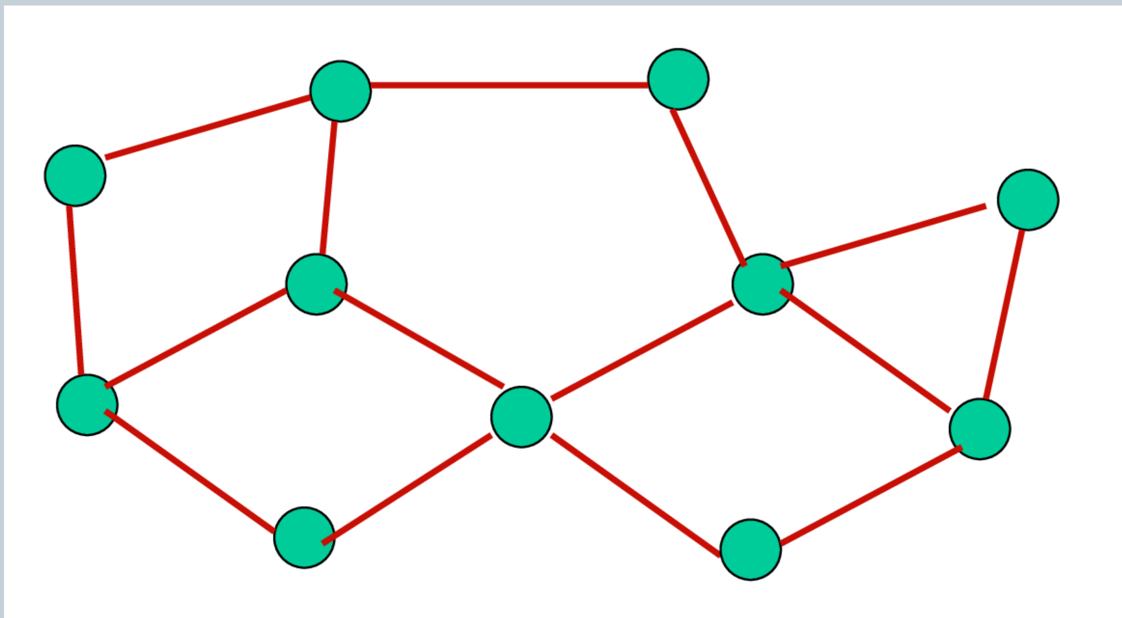
DR. SUSHANTA KARMAKAR
ASSISTANT PROFESSOR
DEPT. OF COMPUTER SC. AND ENGG.
IIT GUWAHATI

Outline

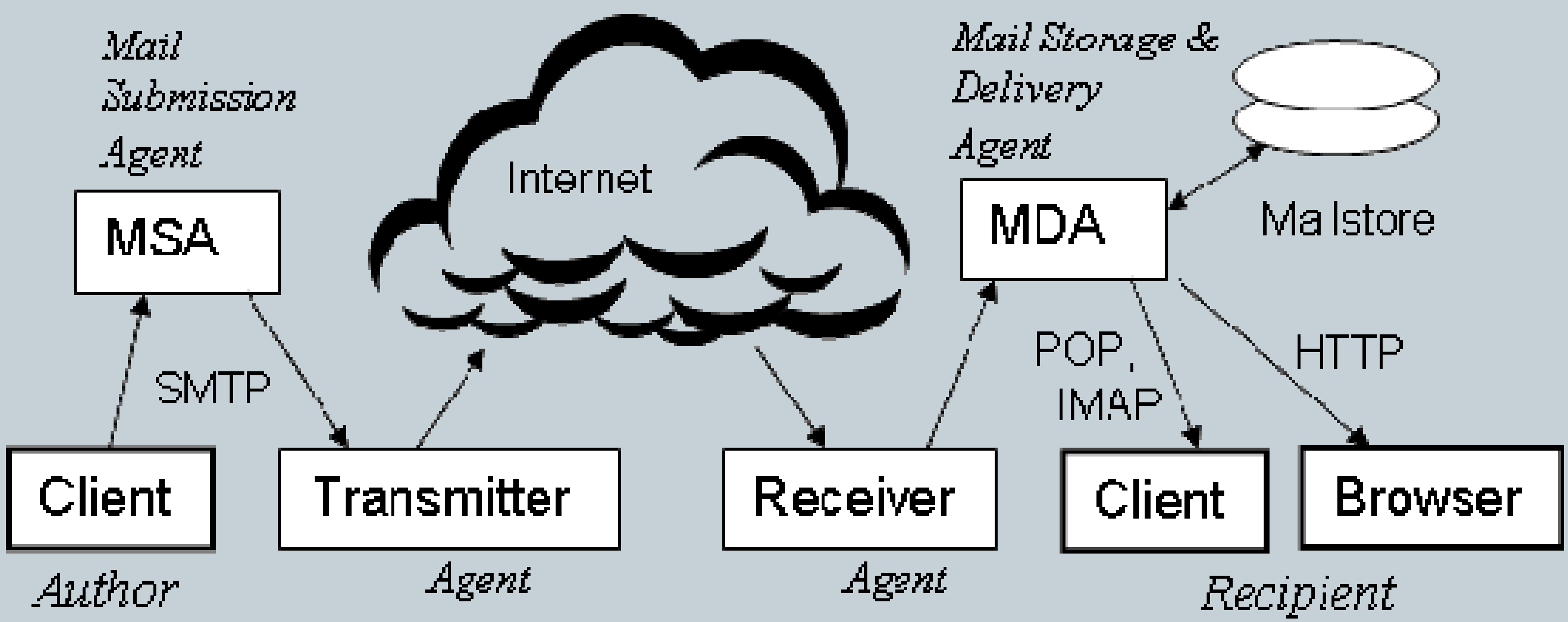
- Introduction
- Importance of Fault-Tolerance in DS
- Classification of Faults
- Fault-Tolerant Algorithms: A few case study
- Conclusion and future directions

Introduction

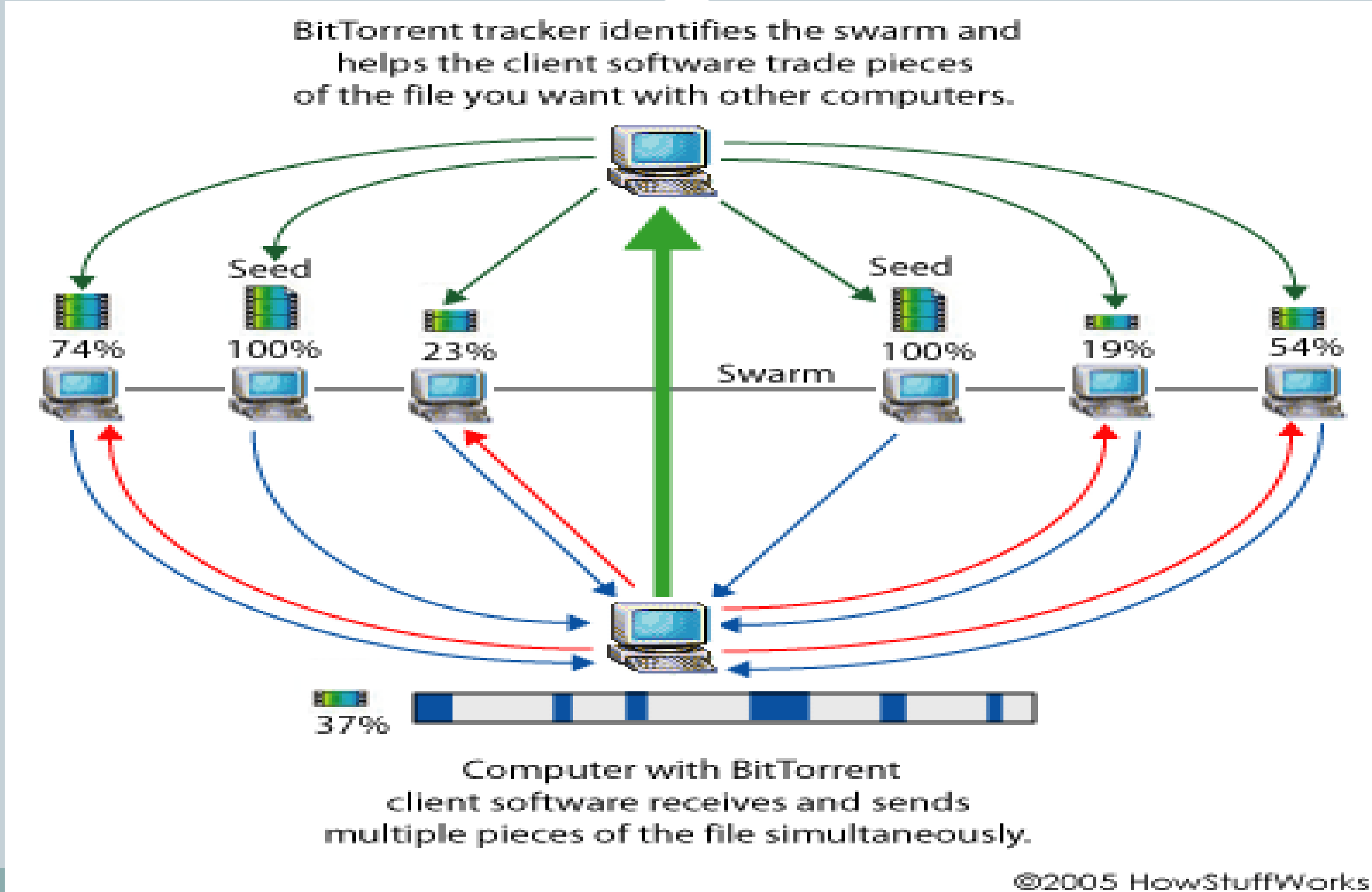
- Def1 **[Distributed System]:** It is a collection of autonomous nodes (process, computer, sensor etc) communicating with each other to achieve a **common goal**



Email System



Content Distribution Networks (CDN)



Other examples

- World wide web
- Network File Server
- Banking Network
- Railway/Airline Reservation
- P2P networks, sensor networks, SETI@home

Motivations of Distributed System

- Speed up/concurrency
- Resource Sharing
- Scalability
- Fault-tolerance

Fault-Tolerance in DS

- A fault is the manifestation of an unexpected behavior
- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults
- Fault-tolerance is important
  - Computers today perform critical tasks (GSLV launch, nuclear reactor control, air traffic control, patient monitoring system)
  - Cost of failure is high

Some Stories

- Sep 23, 1999, NASA lost the $125 million Mars orbiter spacecraft because one team used metric units while another used English units

- 15 April 2010, GSLV MK II, First flight test of the ISRO designed and built Cryogenic Upper Stage. Failed to reach orbit due to malfunction of the Fuel Booster Turbo Pump (FBTP) of the cryogenic upper stage

Fault Tolerance

- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults

- Fault tolerance is related to **dependability**

Dependability

Dependability Includes

- Availability
- Reliability
- Safety
- Maintainability

Availability & Reliability (1)

- **Availability**: A measurement of whether a system is *ready to be used immediately*
  - System is up and running at any given moment
- **Reliability**: A measurement of whether a system can *run continuously without failure*
  - System continues to function for a long period of time

Availability & Reliability (2)

- A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable
- A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

Safety & Maintainability

- **Safety**: A measurement of *how safe failures are*
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability**: A measurement of *how easy it is to repair a system*
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically? Self-healing systems?

Classification of Faults

- **Crash Failure** : node ceases to execute its actions
  - Synchronous system -> crash can be detected using timeout
  - Asynchronous system -> hard to detect crash
- **Omission Failures**: message sent, but not received
  - Core are of research in networking
- **Transient Failure:** state of a node becomes corrupted (by hardware/software failure)

Fault classification (contd.)

- **Byzantine Failure:** unpredictable behavior of a node (complete arbitrary)
- **Temporal failure**
- **Security failure**

Case Studies in the tutorial

- Crash Tolerance
- Transient Failure Tolerance
- Byzantine Failure Tolerance

Types of Fault-tolerance (1)

- Masking tolerance
  - Triple-modular redundancy
  - N-modular redundancy
  - Preserves both **safety** and **liveness** properties

- Non-masking tolerance
  - Safety may be violated, liveness is not compromised
  - Backward error recovery (check-point based)
  - Forward error recovery (self-stabilization*)

Types of Fault-tolerance (2)

- Fail-safe tolerance
  - Safety is surely preserved
  - No guarantee on liveness
  - e.g. mask single fault, stop at double or more faults

- Graceful degradation
  - Neither masking nor full recovery
  - Exhibits degraded behavior
  - e.g. Shortest-path computation in faulty-environment

Design of Adaptive Distributed Systems by Protocol Switching

Adaptive Distributed Algorithms

- Performance of a distributed algorithm depends on environment.
  - ex. load, mobility etc
- Environment may change with time
- Need for distributed algorithms that can cope with changing environment

Adaptation Techniques

- Modify runtime parameters
  - Example – adjusting buffer size in routers with load
- Adaptation by nature
  - Adaptive mutual exclusion by Anderson et al. [1]
- Adaptation as a protocol layer
  - **Snoop** protocol by H. Balakrishnan et al. [3]

Motivation of the Work

Existing approaches not sufficient in many cases, may need to run different algorithms in different conditions

- An Example
  - Routing in ad-hoc networks
    - AODV
    - DSR

Protocol Switching

- P_1 and P_2 are two protocols for the same problem, E_1 and E_2 are two environments, and M is the performance evaluation metric
  - P_1 is better than P_2 under E_1
  - P_2 is better than P_1 under E_2
- Dynamically switch from P_1 to P_2 as environment changes from E_1 to E_2 and vice-versa

Additional Criteria

- Maintain desirable properties during switching
- Examples of desired properties
  - Mutual exclusion
    - No more than one process can enter the critical section during switching
  - Routing
    - No loss of packet during switching

Components of Distributed Protocol Switching

- When to switch
  - May require global coordination
- **How to switch**
  - Switching algorithm

Solution Approaches

- Centralized switching by two-phase-commit
  - Simple and easy to implement
  - Large switching overhead
  - Global freeze
  - Not scalable

- **Localized distributed switching**
  - Switching is based on local information
  - Low overhead per node
  - Local freeze
  - scalable

Overall Motivation

- Proposing localized distributed algorithms for dynamic switching from one protocol to another
- Maintaining some desirable property of the system during switching

Related Work

- Ted Herman [11] → design of adaptive programs from self-stabilizing components.

- Bar-Noy et al. [4] → shifting between different algorithms on the fly to solve byzantine agreement

Related Work (contd.)

- Arora et al. [2] → fault-tolerant method to switch from one state to another without requiring global freeze.
- Liu et al. [15] → adaptation by dynamically mapping the state of a process in one protocol to the state in another.
- Liu et al. [16] → overview of the communication properties for correct functioning of the protocol in [15]
- Mocito and Rodrigues [19] → switching between different total order algorithms.

Objective of the Work

- Design of adaptive algorithm by protocol switching for **single source broadcast** problem
  - Tolerating node failure
    - Crash fault
    - Transient fault

Adaptive Broadcast by Switching from a BFS tree to a DFS tree

BFS to DFS switching

- Non-fault-tolerant algorithm for dynamic switching from a BFS tree to a DFS tree
- Fault-tolerant algorithm for dynamic switching from a BFS tree to a DFS tree

System Model

- Asynchronous message passing system
- Reliable and FIFO channels
- Crash fault
- Connected Graph
- The single source *r* does not fail

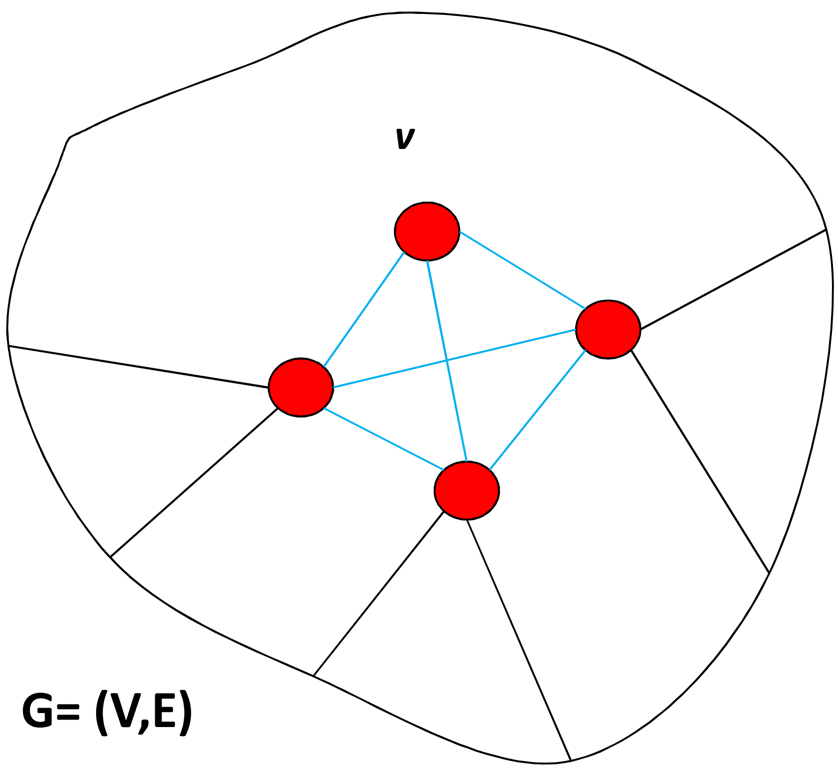
Solution Approach

- Non-fault-tolerant switching algorithms
- Local repair of BFS and DFS
  - Faults may happen when no switching is in progress
- Fault-tolerant actions that help tolerate arbitrary crash faults during switching

Switching from a BFS tree to a DFS tree

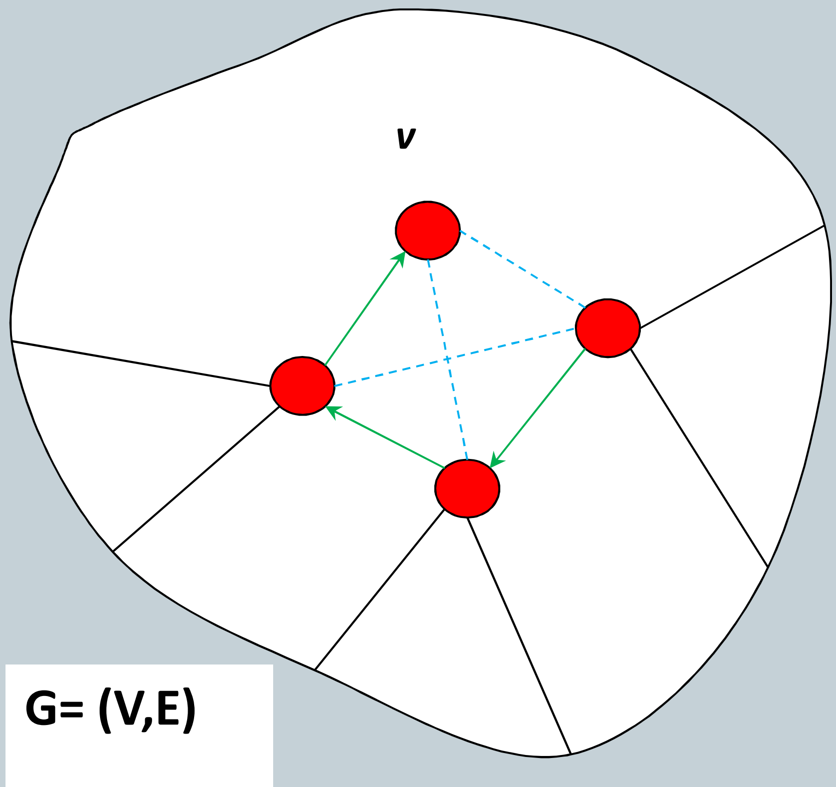
- G = (V, E) is the graph
- T is a BFS tree of G rooted at *r*
- T' is a DFS tree of G rooted at *r*
- Switch from T to T'

Definitions



- Let G_v= (V', E') be some subgraph of graph G such that
  - V' $\subseteq$ {v}$\cup$ N(v)
  - E' $\subseteq$ {(u,v)| u,v $\in$ V' }
- If T_v is a DFS spanning tree of G_v rooted at v then T_v is defined as the ***local DFS subtree* of G_v** rooted at v

Local DFS Subtree



Switching Algorithm for BFS to DFS

- TOKEN based local switching from BFS to DFS
- The root of the BFS tree, *r*, gets the TOKEN first
- For a node v, CSet(v) = N(v) – [TSet(v) ∪ {p(v)}]
- On receiving the TOKEN for the first time, a node v builds a **local DFS subtree**, rooted at itself, of the graph induced by CSet(v) ∪ {v}.

Switching Algorithm for BFS to DFS (contd.)

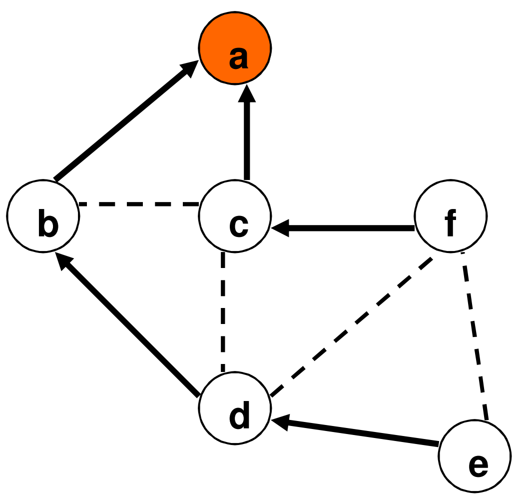
- After v builds **local DFS subtree**, it sends the TOKEN to some $u \in Child(v)$ if u has not already got the TOKEN
- $\forall u \in Child(v)$, if u has got the TOKEN, v sends the TOKEN to p(v)
- If v has already got the TOKEN then it forwards the TOKEN to some u using the same rule

Partial DFS Tree

- The nodes that have received the TOKEN at least once form a DFS tree.
  - **partial DFS tree**

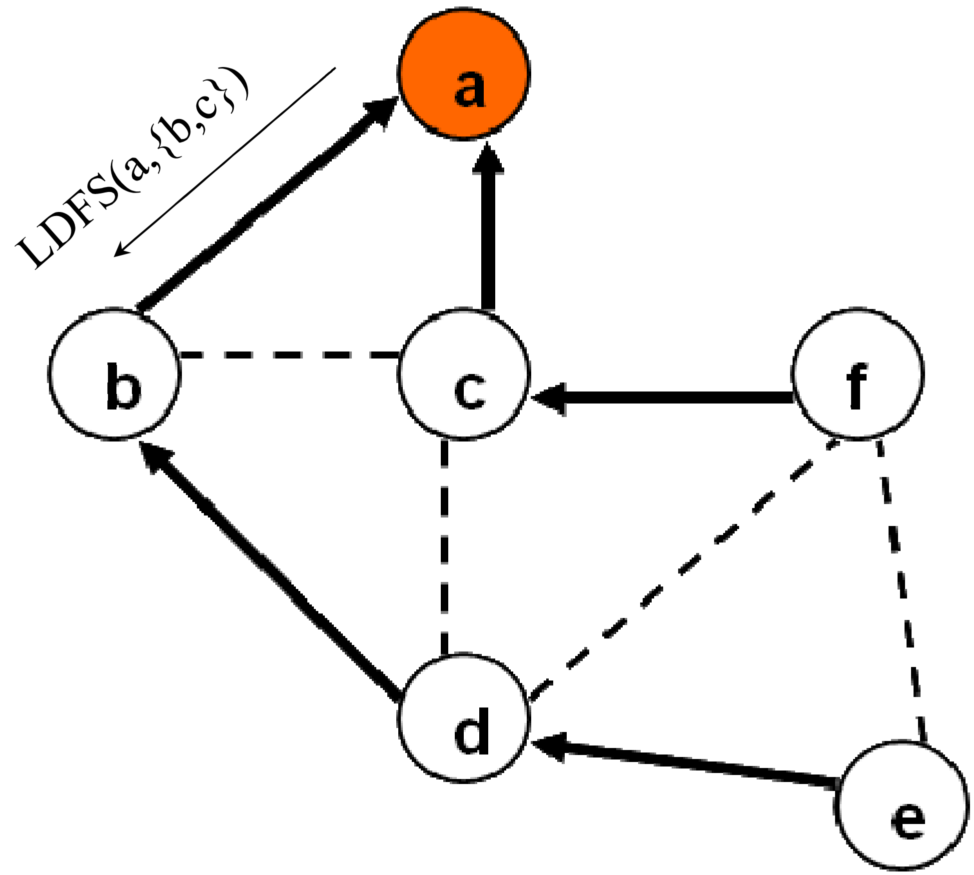
Tree edges of a **local DFS subtree** may change with time but that of **partial DFS tree** will not change

An Example (BFS to DFS)

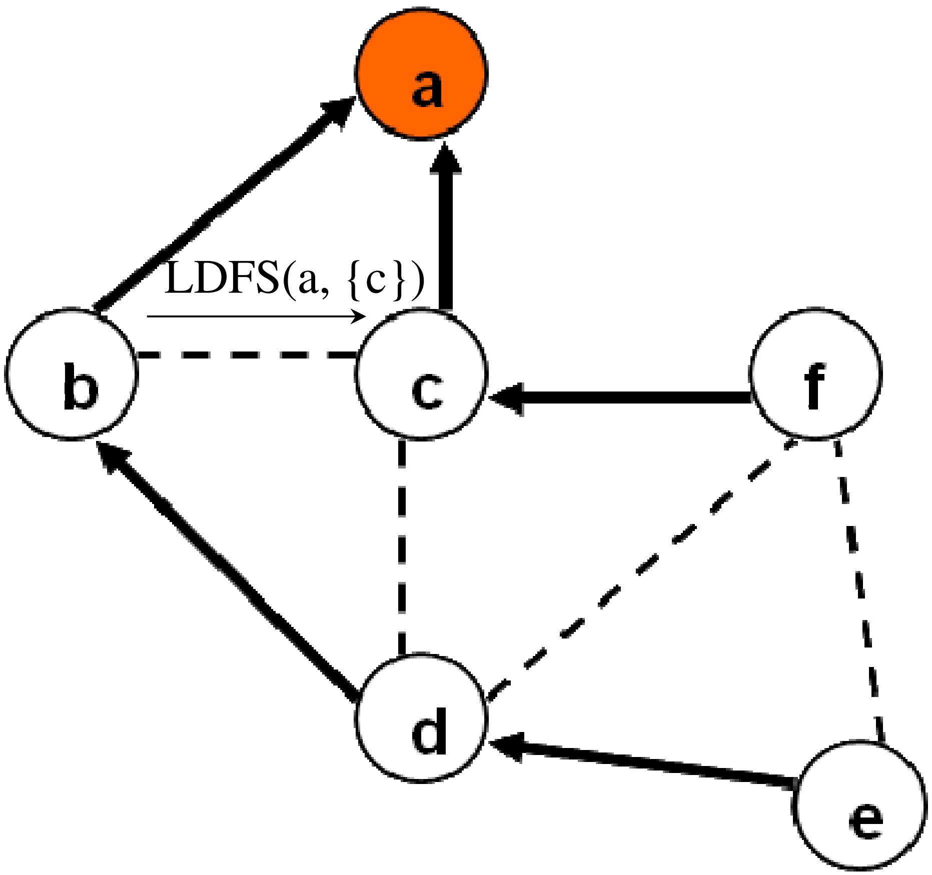


- Initially for each node v, CSet(v)=TSet(v)=ϕ
- ‘a’ has got the TOKEN
- CSet(a) = {b, c}
- ‘a’ builds a **local DFS subtree**, rooted at ‘a’, of the graph induced by the set of nodes {a, b, c}

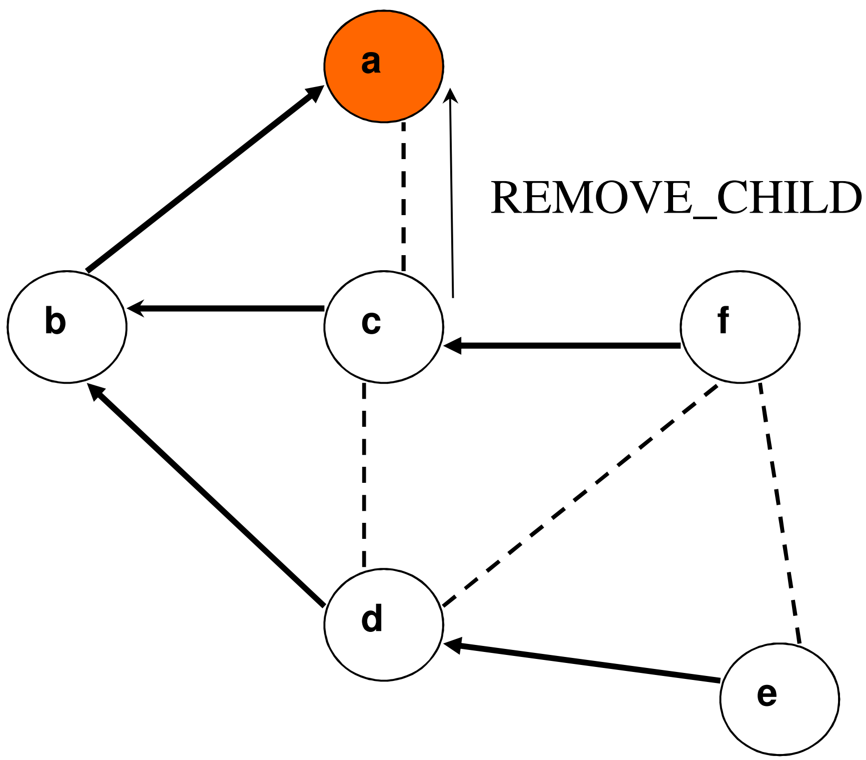
Steps



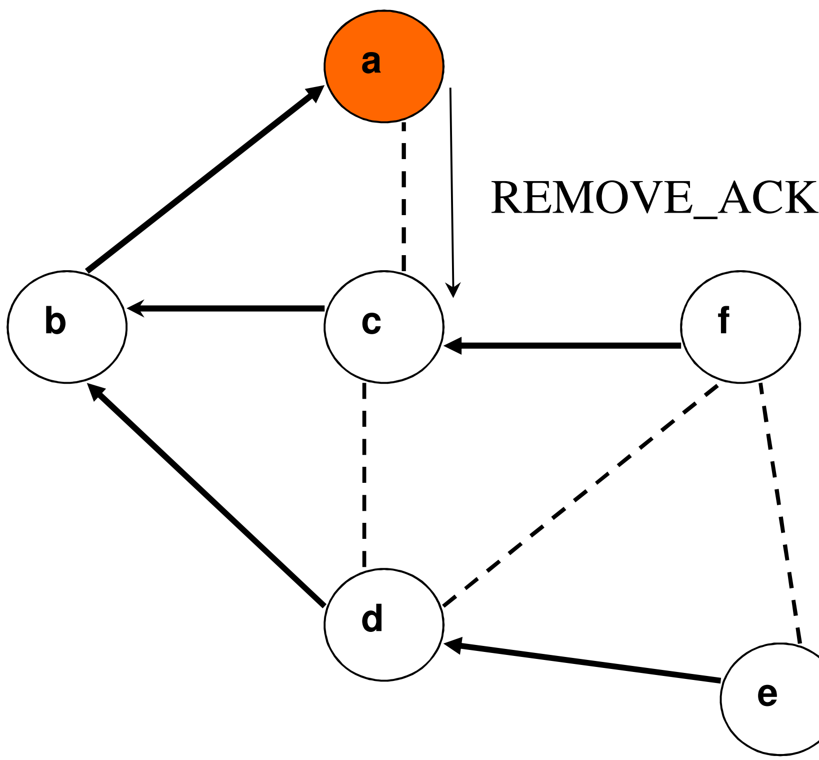
Steps (contd.)



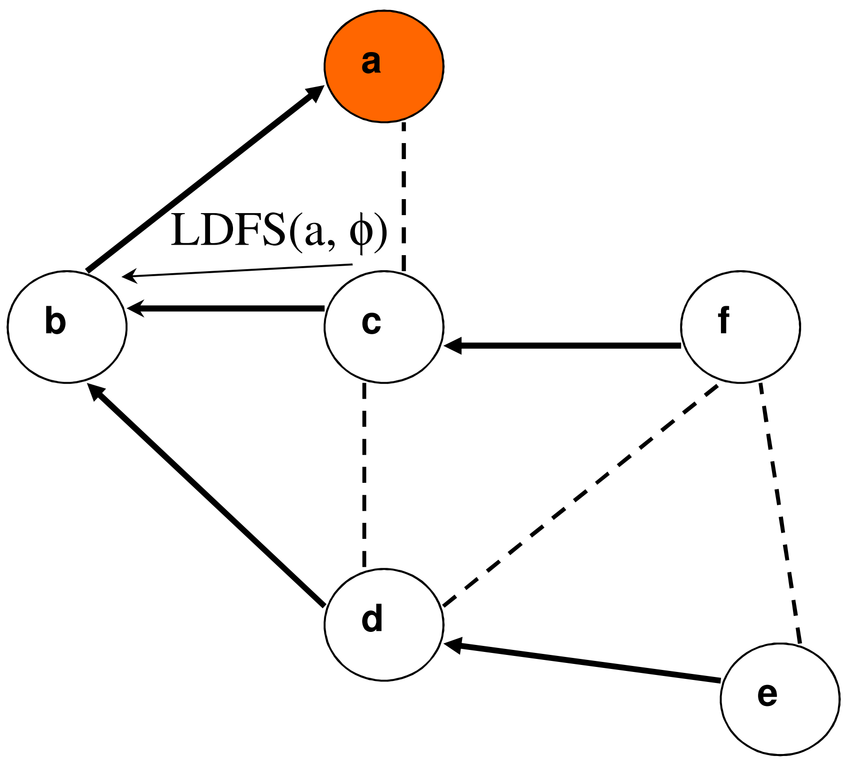
Steps (contd.)



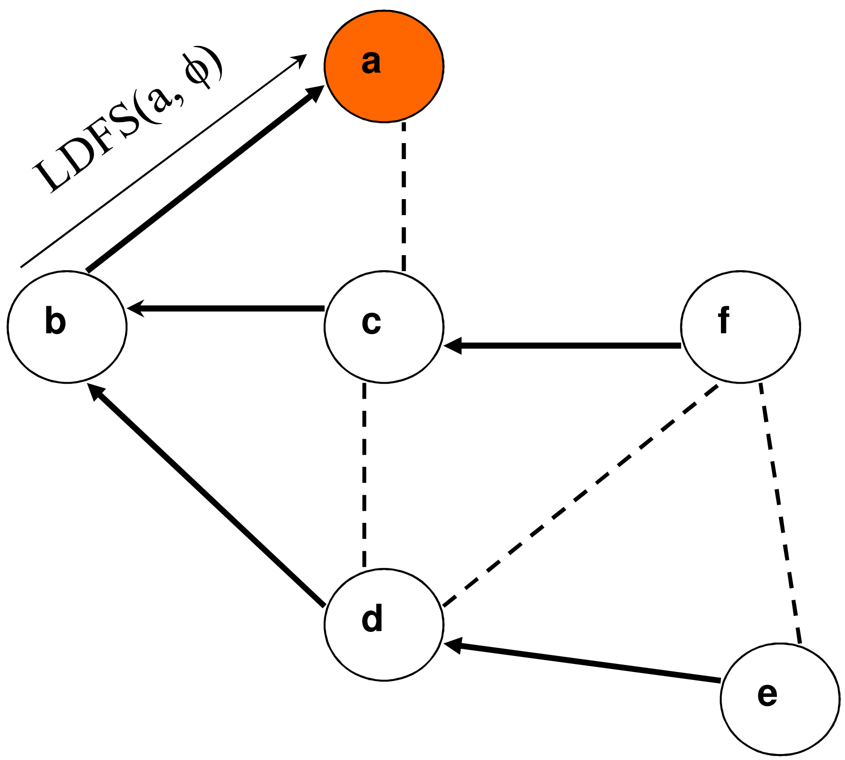
Steps (contd.)



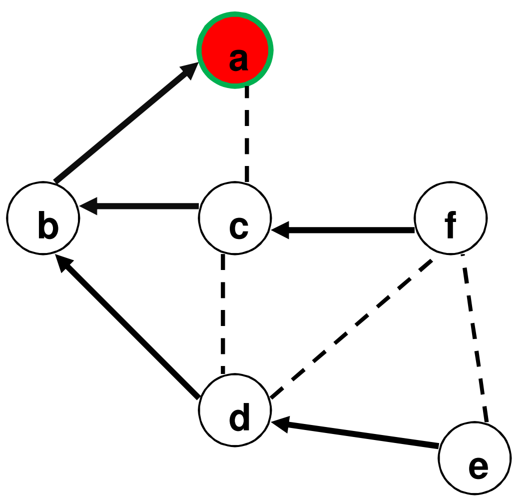
Steps (contd.)



Steps (contd.)



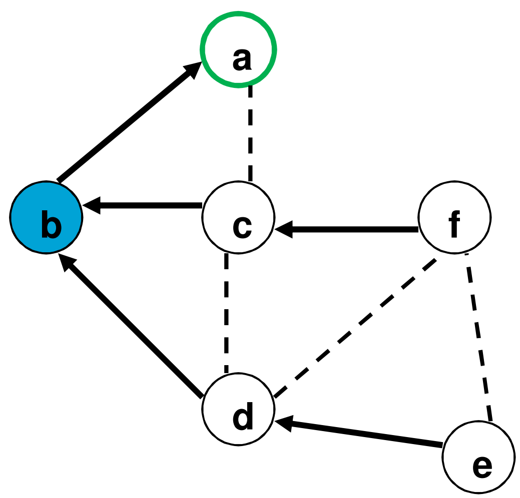
The topology after local switching at node **a**



- Nodes having a green outline belong to the **partial DFS tree** of G.

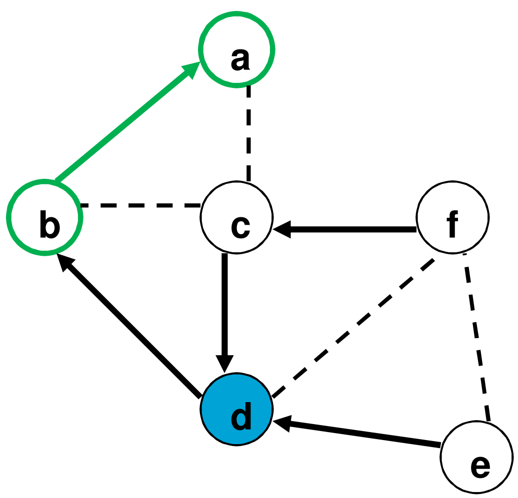
- Node **a** sends the TOKEN to its only child **b**

Example (contd.)



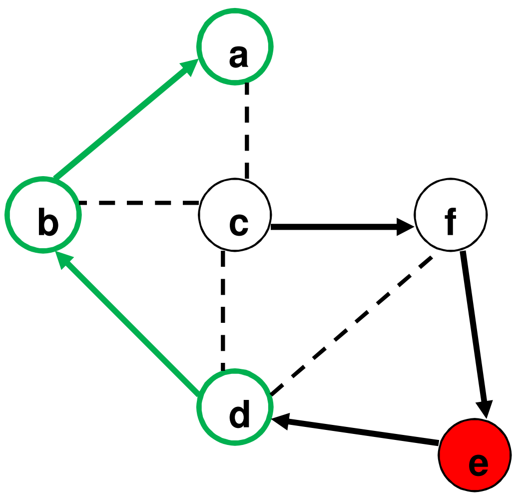
- TSet(b) = {a}, TSet(c)={a}
- ‘b’ got the TOKEN
- CSet(b) = {c, d}
- ‘b’ builds a local DFS subtree, rooted at ‘b’, of the graph induced by the set of nodes {b, c, d }

Example (contd.)



- TSet(c)={a, b}, TSet(d)={b}
- 'd' has got the TOKEN
- CSet(d) = {c, e, f}
- 'd' builds a local DFS subtree, rooted at 'd', of the graph induced by {c, d, e, f}

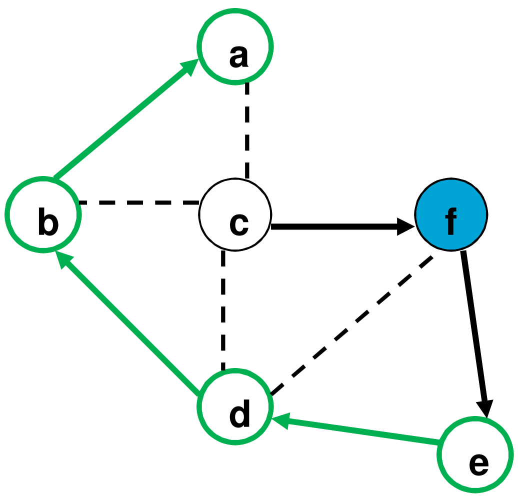
Example (contd.)



- TSet(c)={a, b, d}, TSet(e)={d}, TSet(f)={d}
- 'e' has got the TOKEN
- CSet(e) = {f}
- 'e' builds a local DFS subtree, rooted at 'e', of the graph induced by {e, f}

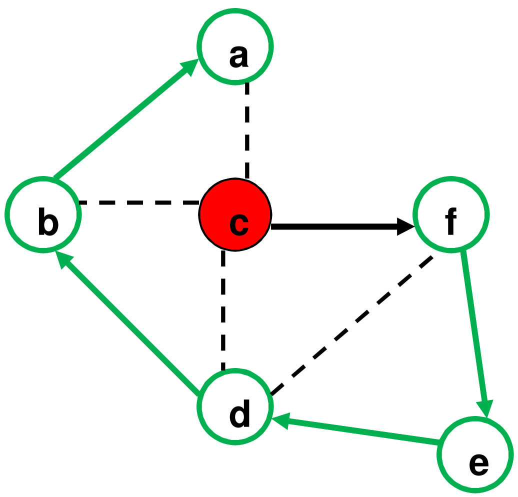
After this there will be no change in the spanning tree

Example (contd.)



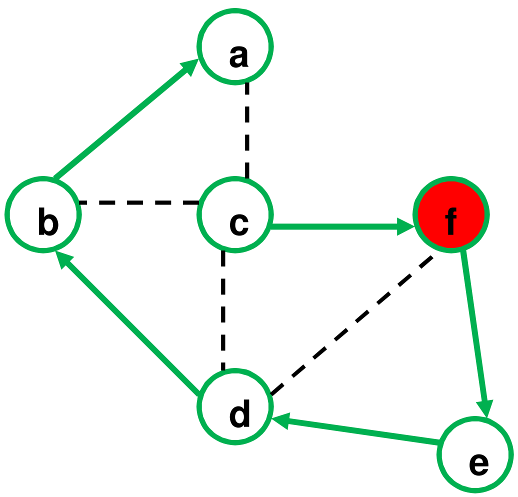
- TSet(f)={d, e}
- ‘f’ has got the TOKEN
- CSet(f) = {c}
- ‘f’ builds a local DFS subtree, rooted at ‘f’, of the graph induced by {c, f}

Example (contd.)



- TSet(c) = {a, b, d, f}
- CSet(c) = φ
- ‘c’ builds a local DFS subtree, rooted at ‘c’, of the graph induced by {c}
- Now ‘c’ sends the TOKEN back to ‘f’

Example (contd.)



- Now ‘f’ sends the TOKEN back to ‘e’ and so on.
- Algorithm stops when TOKEN comes to ‘a’

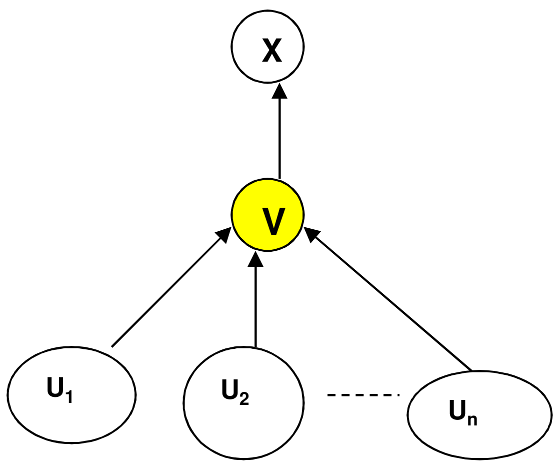
Properties

- Switching eventually completes.
- The algorithm terminates with a DFS tree topology
- The message complexity of the switching algorithm is O(|E|) for no fault case.
- Each broadcast message is eventually correctly delivered in spite of switching provided no failure occurs.

Fault-tolerant Switching from a BFS Tree to a DFS Tree

- When a node fails?
  - No switching in progress
  - Switching in progress

Fault in No Switching

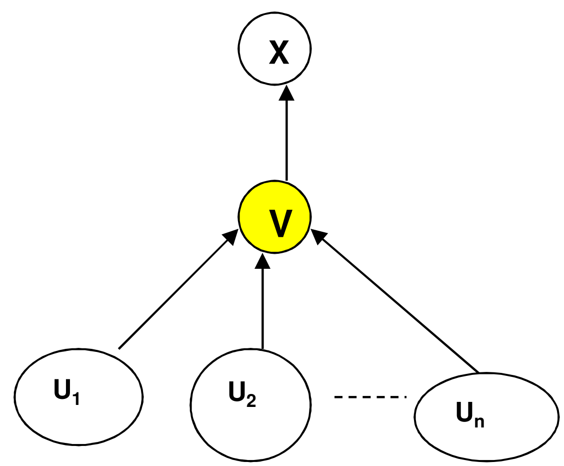


- A node **v** in a tree (BFS/DFS) may crash when no switching is in progresss
- The tree must be repaired to continue the broadcast
- We do local repair of trees as it is attractive for limited failures in terms of time and message complexity

Local Repair of BFS

- Let node **v** crash
- Each of **u_1, u_2, … , u_n** and node **x** executes BfsCrashAction(**v**)

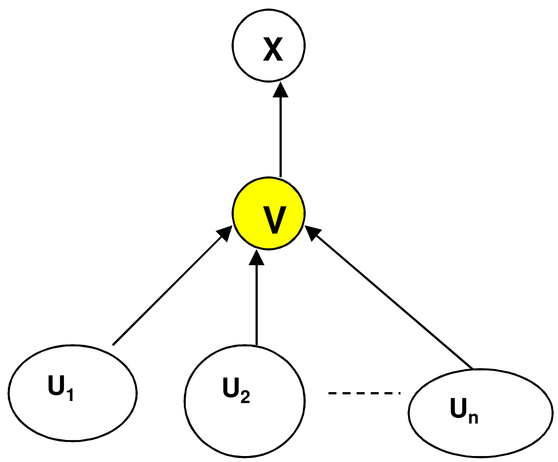
BfsCrashAction(v)
$N(u) = N(u) - \{v\}$
if $p(u) = v$ **then**
  ResetLevelAction(v)



Local Repair of DFS

- Let node **v** crash
- Each of $\mathbf{u_1, u_2, \ldots, u_n}$ and node **x** executes DfsCrashAction(**v**)

DfsCrashAction(v)
$N(u) = N(u) - \{v\}$
if $p(u) = v$ **then**
  ChangePathAction(v)



ResetLevelAction(v) and ChangePathAction(v)

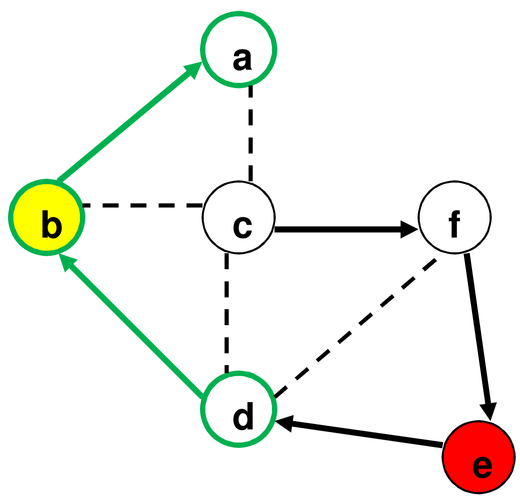
RESETLEVELACTION(V) or
upon receiving $ResetLevel(v) \wedge \tau(u) = \mathbb{T}_u$
GetParamBFS()
$\mathbb{N}_u = \{x : x \in N(u) \wedge v \notin \mathcal{P}_{x \rightsquigarrow r}\}$
if $\mathbb{N}_u = \phi$ **then**
    $\forall w \in N(u) : p(w) = u$, send $ResetLevel(v)$ to w
else
    $\mathcal{P}'_{u \rightsquigarrow r} = \mathcal{P}_{u \rightsquigarrow r}$
    $\exists y \in \mathbb{N}_u : L_y = min_{z \in \mathbb{N}_u}\{L_z\}$
    if $v \in \mathcal{P}_{u \rightsquigarrow r}$ **then**
        $L_u = L_y + 1$
        send $REMOVE$ to $p(u)$
        $p(u) = y$
        send ADD to $p(u)$
        $\mathcal{P}_{u \rightsquigarrow r} = \mathcal{P}_{y \rightsquigarrow r} \odot u$
    else if $L_u > L_y + 1$ **then**
        $L_u = L_y + 1$
        send $REMOVE$ to $p(u)$
        $p(u) = y$
        send ADD to $p(u)$
        $\mathcal{P}_{u \rightsquigarrow r} = \mathcal{P}_{y \rightsquigarrow r} \odot u$
    endif
    if $(\mathcal{P}_{u \rightsquigarrow r} \neq \mathcal{P}'_{u \rightsquigarrow r})$ **then**
        $\forall w \in N(u) - \{p(u)\}$, send $ResetLevel(v)$ to w
    endif
endif

CHANGEPATHACTION(V) or
upon receiving $ChangePath(v) \wedge \tau(u) = \mathbb{T}_u$
GetParamDFS()
$\mathbb{N}_u = \{x : x \in N(u) \wedge v \notin \mathcal{P}_{x \rightsquigarrow r}\}$
if $\mathbb{N}_u = \phi$ **then**
    $\forall w \in N(u) : p(w) = u$, send $ChangePath(v)$ to w
else
    $\mathcal{P}'_{u \rightsquigarrow r} = \mathcal{P}_{u \rightsquigarrow r}$
    $path_u = min_{\prec}\{path_x \odot \beta_x(u) : x \in \mathbb{N}_u\}$
    send $REMOVE$ to $p(u)$
    $p(u) = f(path_u)$
    send ADD to $p(u)$
    $\mathcal{P}_{u \rightsquigarrow r} = \mathcal{P}_{p(u) \rightsquigarrow r} \odot u$
    if $(\mathcal{P}_{u \rightsquigarrow r} \neq \mathcal{P}'_{u \rightsquigarrow r})$ **then**
        $\forall w \in N(u) - \{p(u)\}$, send $ChangePath(v)$ to w
    endif
endif

upon receiving $REMOVE$ from $w \rightarrow$
    $Child(u) = Child(u) - \{w\}$

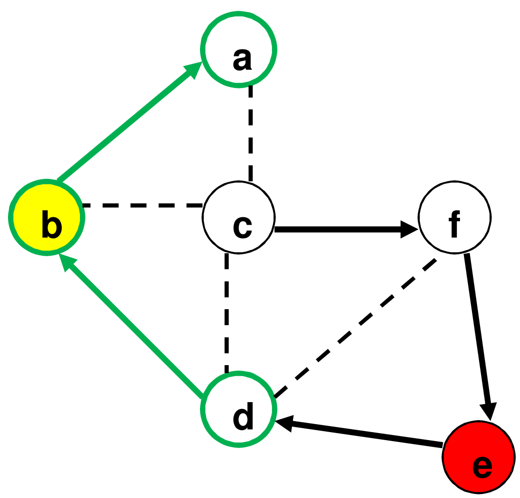
upon receiving ADD from $w \rightarrow$
    $Child(u) = Child(u) \cup \{w\}$

Fault during Switching



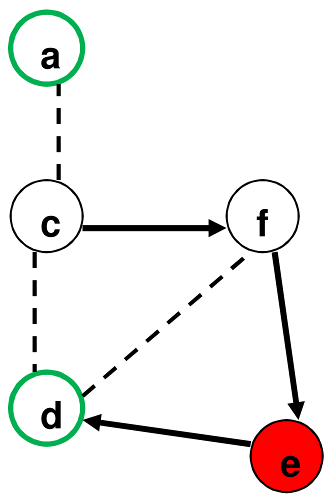
- At any intermediate state during switching, there is a **partial DFS tree** and a **partial BFS tree** of the graph **G**
- **TOKEN** holding node may belong to either **partial DFS** or **partial BFS**
- A fault may occur in
  - **partial BFS tree**
  - **partial DFS tree**

Case Study



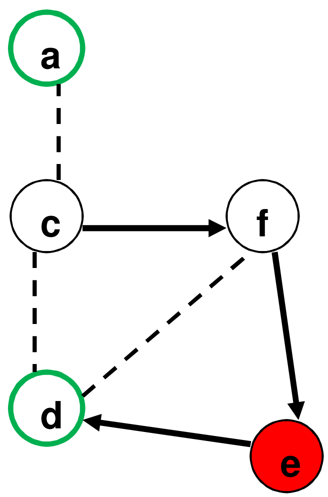
- Suppose a node **b** belonging to partial DFS tree crashes.

Case Study (contd.)



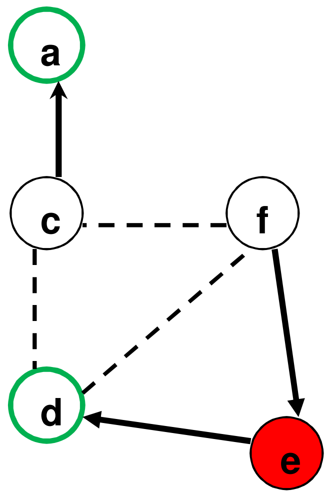
- Resultant structure after the crash of **b**
- Note that node **a, c, d** have detected the crash of **b**
- **TOKEN** is at **e**
- Node **a, c, d** remove **b** from theirs' neighborhood

Case Study (contd.)



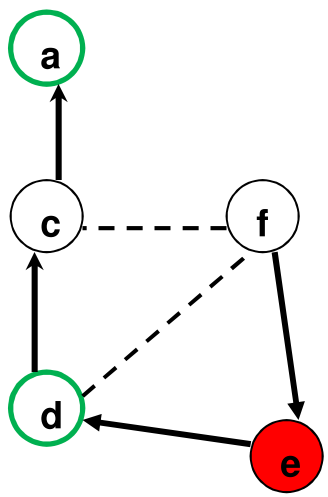
- Node **d** execute *DfsCrashAction(**b**)*
- Node **a** may generate another TOKEN at **a** to restart switching at **a**
- Another switching due to **TOKEN** at **e**
- Node **c, e, f** may execute **ChangePathAction(b)**

Case Study (contd.)



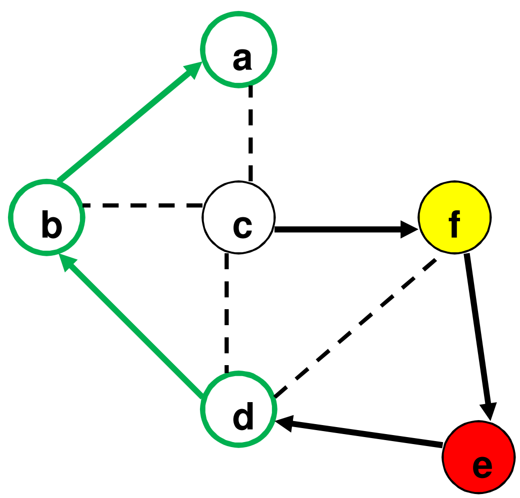
- Node **c** eventually changes it parent to **a** either by **ChangePathAction(b)** or due to fresh switching from **a**

Case Study (contd.)



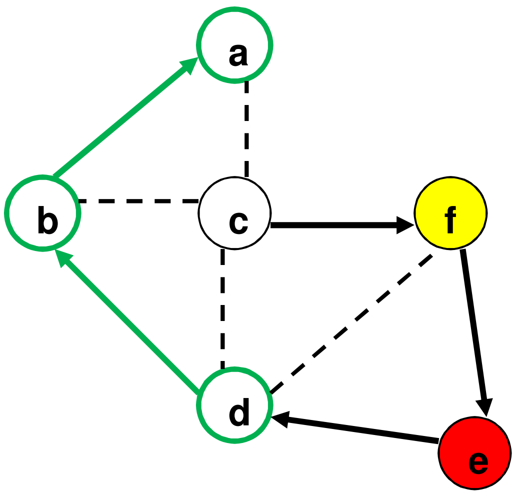
- Eventually **d, e, f** reassign theirs' parents as shown in figure due to **ChangePath(b)** messages.
- TOKENs at **e** may perish or may result in switching
- Overall, a correct DFS tree of G rooted at **a** results

Crash of a Node in Partial BFS Tree



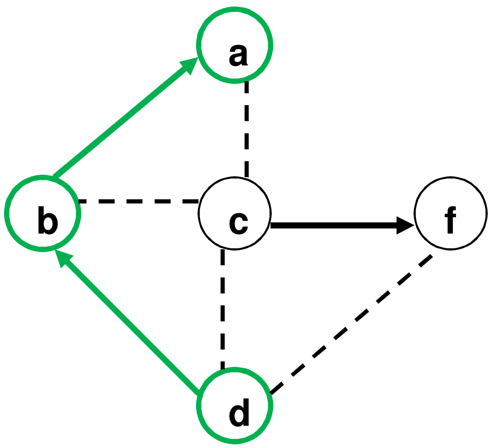
- If **f** crashes then each neighbor belonging to **partial BFS tree** should execute *BfsCrashAction(**f**)*
- Each neighbor belonging to **partial BFS tree** should execute *ResetLevelAction(**f**)* on receiving a *ResetLevel(**f**)* message

Crash of TOKEN Holder



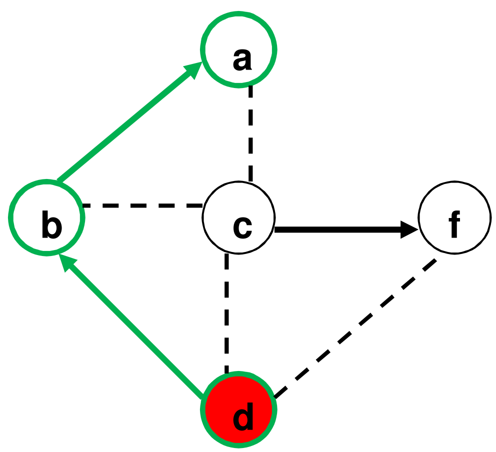
- What happens if TOKEN holding node **e** crashes?

Crash of TOKEN Holder (contd.)



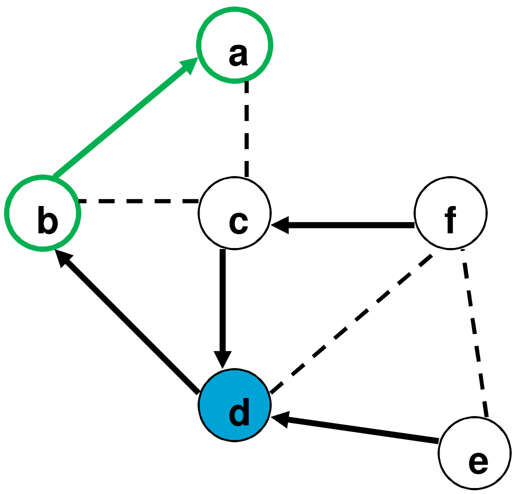
- Suppose the TOKEN holding node **e** crashes.
- TOKEN can be generated at any of **a, b, d** to continue the switching
- TOKEN is actually generated at the nearest ancestor **d**

Crash of TOKEN Holder (contd.)



- A fresh **local DFS subtree** formation starts at **d**
- Eventually a DFS tree rooted at **a** results.

Special Case



- Suppose **d** is currently doing the local switching.
  - **d** creahses, already covered
  - Some member of **Cset(d)** crashes

- Cset(d) = {c,e,f}
- In this case the local switching is just restarted at **d**

Fault-tolerant Actions for Switching from BFS to DFS

(S_1) $tokenVisited(u) \wedge Crash(v) \rightarrow DfsCrashAction(v)$

(S_2) $tokenVisited(u) \wedge received\ ChangePath(v) \rightarrow ChangePathAction(v)$

(S_3) $\neg tokenVisited(u) \wedge Crash(v) \rightarrow BfsCrashAction(v)$

(S_4) $\neg tokenVisited(u) \wedge received\ ResetLevel(v) \rightarrow ResetLevelAction(v)$

(S_5) $\neg tokenVisited(u) \wedge received\ ChangePath(v) \rightarrow ChangePathFlag(u) = 1; ID(u) = v$

(S_6) $tokenVisited(u) \wedge ChangePathFlag(u) = 1 \rightarrow ChangePathFlag(u) = 0; ChangePathAction(ID(u))$

(S_7) $tokenVisited(u) \wedge \neg tokenHolder(u) \wedge Crash(v) \wedge tdir(u) = v \wedge tdir(u) \neq p(u) \rightarrow tokenHolder(u) = true$
$tokenVisited(u) = false$
$\forall w \in CSet, Reset(w)$

(S_8) $tokenVisited(u) \wedge tokenHolder(u) \wedge Crash(v) \wedge v \in CSet(u) \rightarrow tokenVisited(u) = false$
$\forall w \in CSet, Reset(w)$

Properties

- Under arbitrary crash failures, the BFS to DFS switching algorithm eventually terminates with a DFS tree as the broadcast topology. No specific broadcast delivery guarantee in this case.

Broadcast Properties under Single Crash Fault

- Under single crash fault, each broadcast message having timestamp less than or equal to Υ is eventually correctly delivered to all the non-faulty nodes where

$\Upsilon = min\{\mathbb{T}_{u_1}, \mathbb{T}_{u_2}, \ldots, \mathbb{T}_{u_m}\}$ and $\mathbb{T}_{u_i}$ is the timestamp of the last message received by u_i before it detects the crash of *v*.

Broadcast Properties under Single Crash Fault (contd.)

- Under single crash failure, each message broadcast by the single source r after the system reaches a state of Z is eventually correctly delivered to all the nodes where Z is the set of states of the system where any node $w \in V$ does not change p(w) anymore due to receipt of **ChangePath(v)** or **ResetLevel(v)** message, but w may change p(w) due to the receipt of an **LDFS** messages.

Adaptive Broadcast by switching from a DFS tree to a BFS tree

DFS to BFS switching

- Non-fault-tolerant algorithm for dynamic switching from a DFS tree to a BFS tree

- Fault-tolerant algorithm for dynamic switching from a DFS tree to a BFS tree

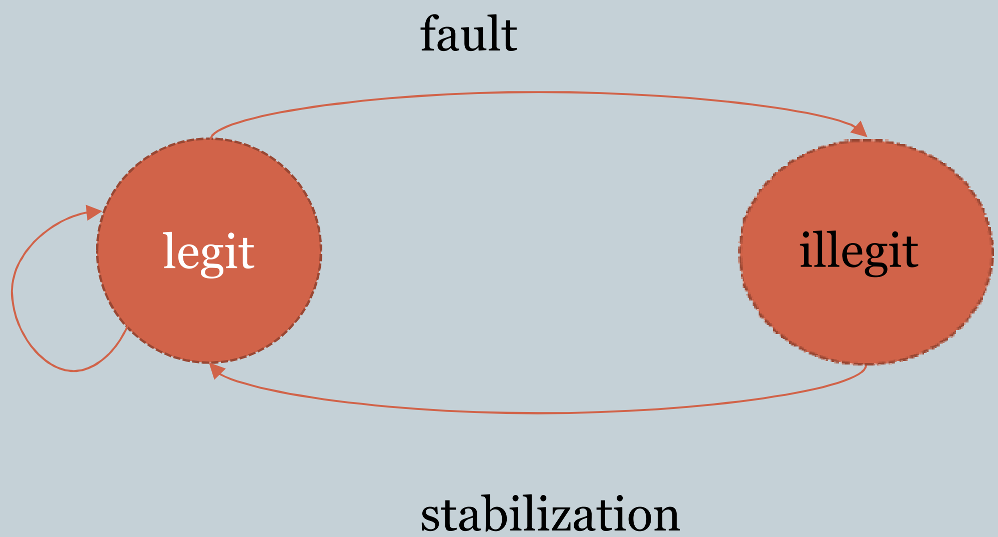
Approach is similar to BFS to DFS case but algorithms are different

Adaptive broadcast by Self-Stabilizing Spanning Tree Switching

Self-Stabilization

- Automatic handling of transient failure in a distributed environment
  - **Convergence**: a system, in an *illegitimate state*, eventually reaches a legitimate state
  - **Closure**: once in a *legitimate state*, the system remains in some legitimate state until further failure

State Transition



Dijkstra's Self-stabilization in a Ring

- Data may become corrupted
- Code is not corrupted
- **Problem:** Given a unidirectional ring, design a scheme so that exactly one node has the privilege eventually in spite of arbitrary initial state
  - Safety: Number of nodes with enabled guard is exactly one
  - Liveness: Each node gets its guard enabled infinitely often

Pseudocode

S[0] = S[n-1] -> S[0] = S[0]+1 mod k // process 0

S[i] ≠ S[i-1] -> S[i] = S[i-1] // process i≠0

Self-Stabilizing Spanning Tree Switching

- A self-stabilizing distributed algorithm for dynamic switching between arbitrary trees T and T'
  - Under no failure, each broadcast message is correctly delivered to all the nodes, in spite of switching
  - Under arbitrary failure, switching eventually completes with the desired tree as output
  - Investigate the broadcast properties under single transient failure

System Model

- System as connected graph G = (V, E)
- Shared memory model; each node has unique ID
- Each node reads from 2-hop neighborhood but writes only in local memory (**relaxed later**)
- Local FIFO buffer B_i for broadcast
- Transient failure
- Both T and T' are pre-computed (**relaxed later**)

Definitions

- N_i is the neighborhood of node i
- b_i as protocol variable
- T^i_M as message timestamp
- *max*(B_i) denote index of last message placed in B_i
- *read*(B_j, *n*) function executed by i

Definitions (contd.)

- p_i is the parent of node i in **T**
- C_i is the children of i in **T**

Illustration of Broadcast

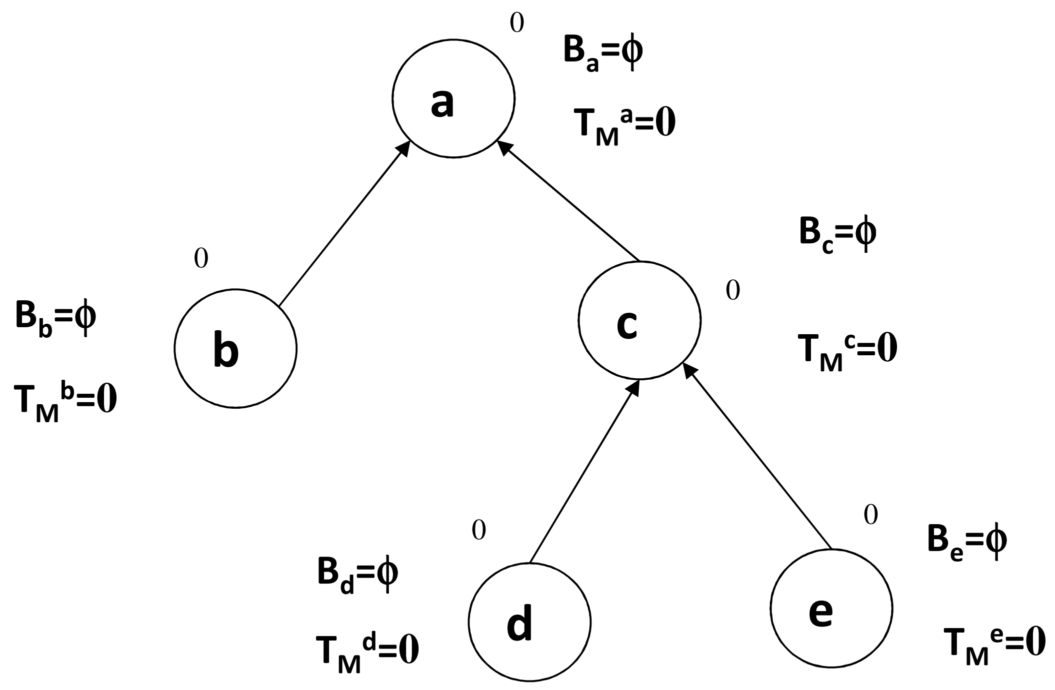


Illustration of Broadcast (contd.)

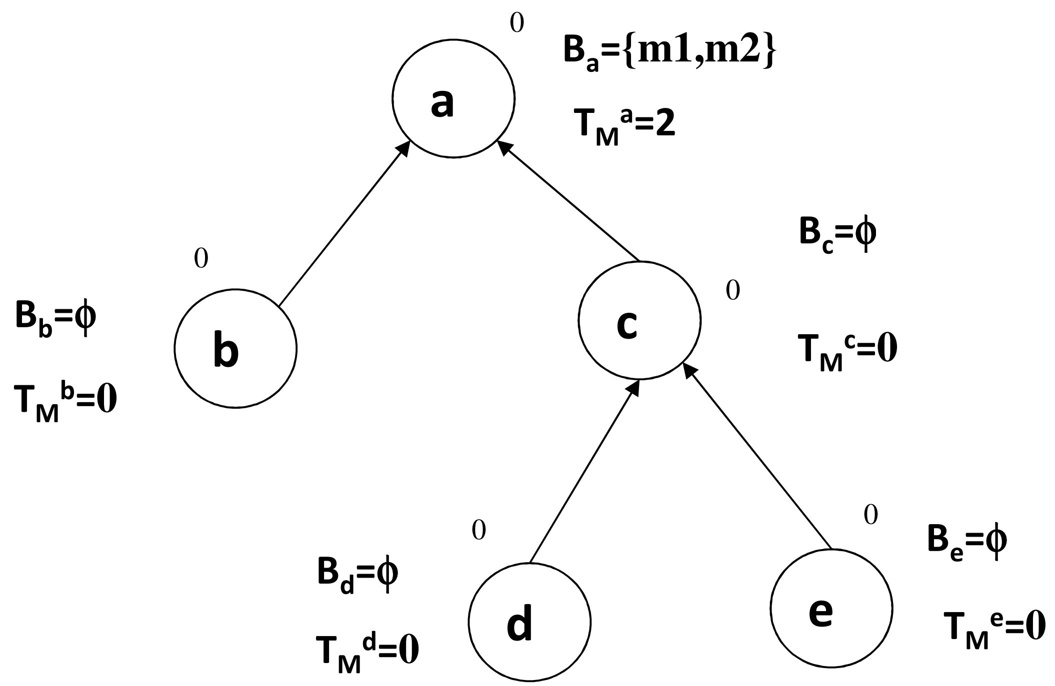


Illustration of Broadcast (contd.)

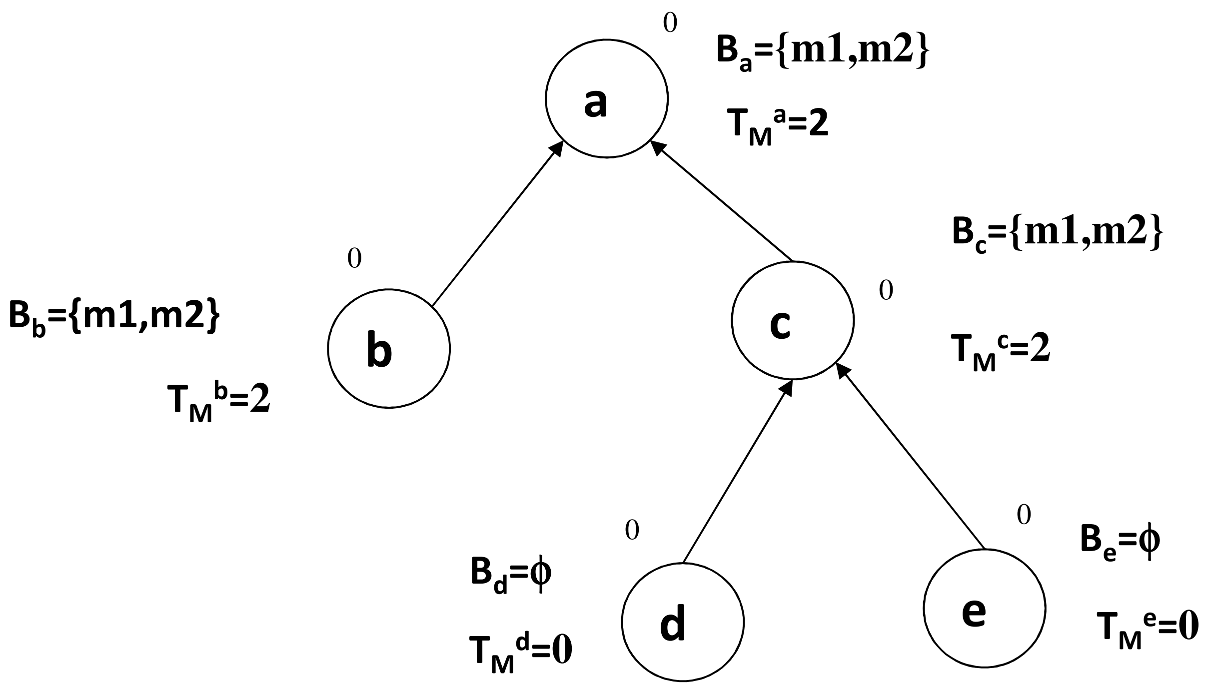
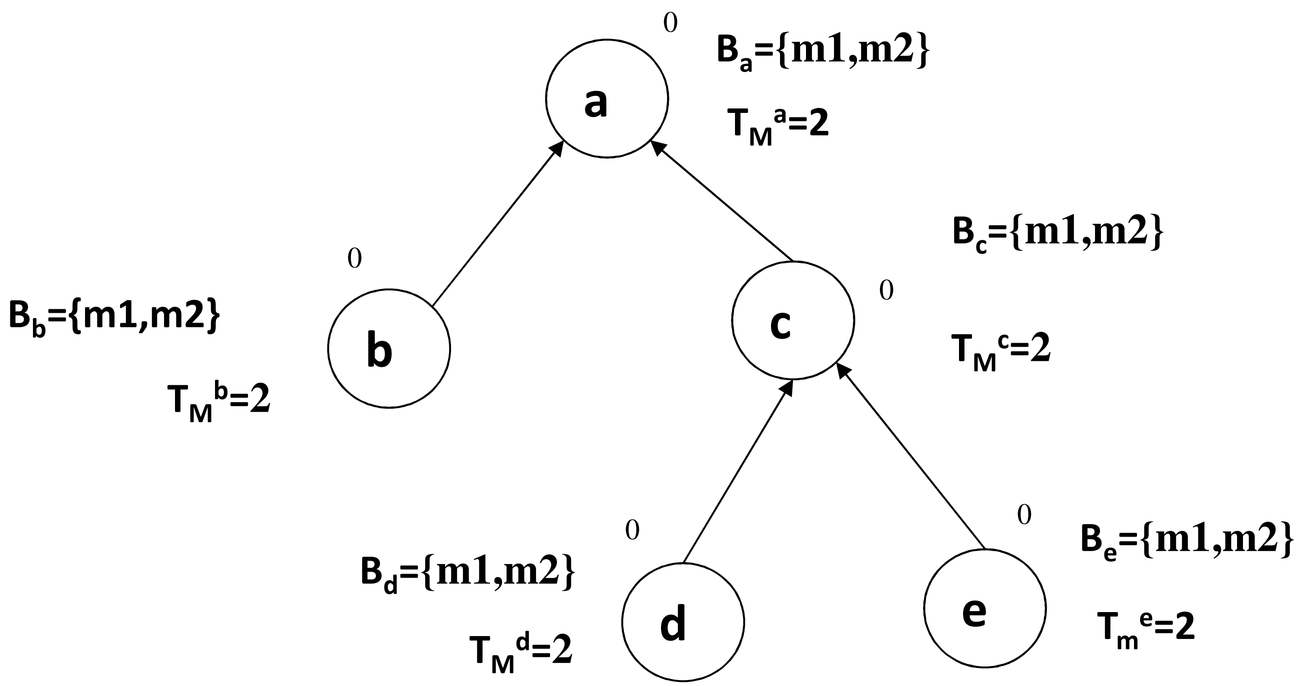


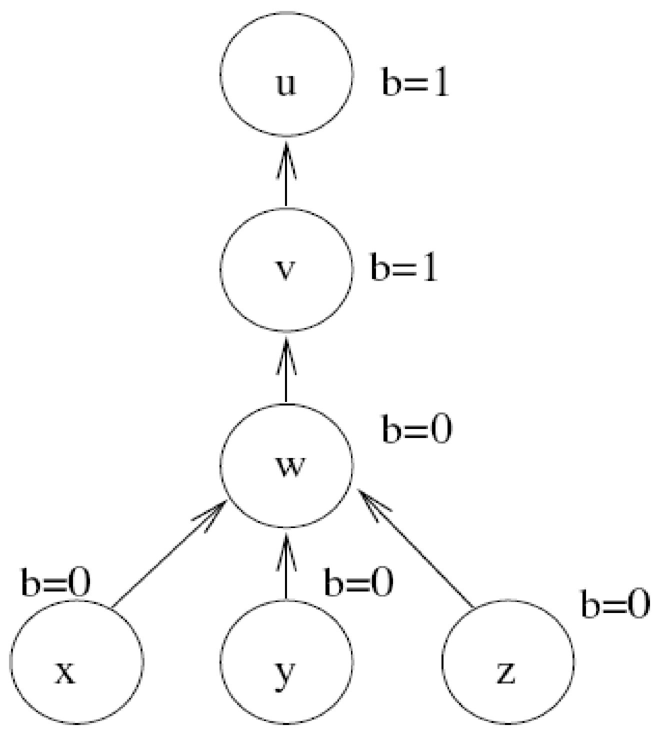
Illustration of Broadcast (contd.)



Self-Stabilizing Switching from T to T'

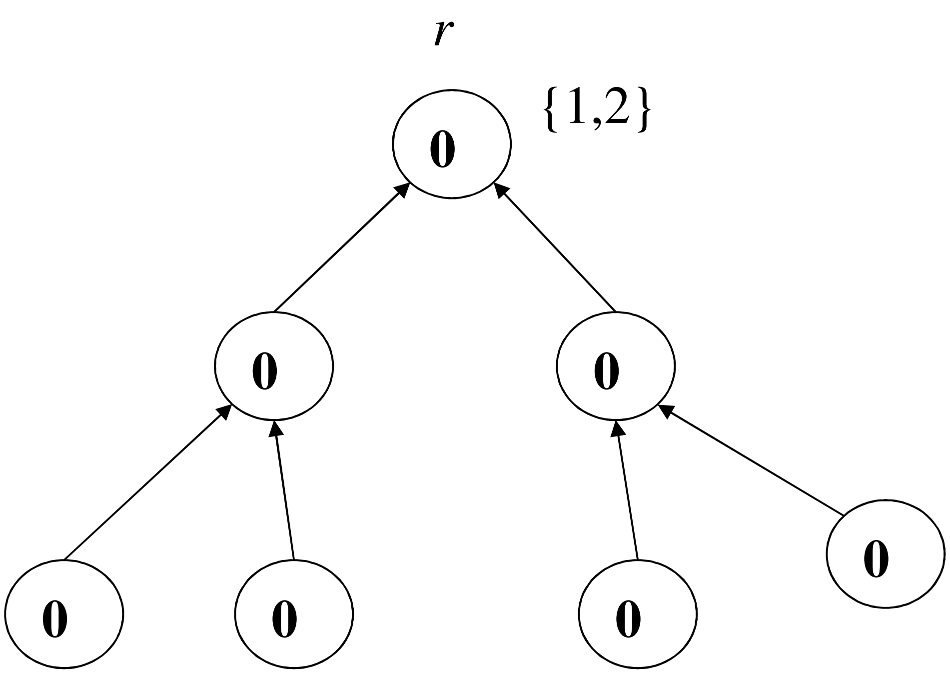
- b_i=0 → i uses T for broadcast
- b_i=1 → i uses T' for broadcast
- At **r** , $b_i = f(L)$ where L is load of the network monitored by T or T'
- Let $p(b_i)$ denote the parent of i as per the current value of b_i

Switching (contd.)



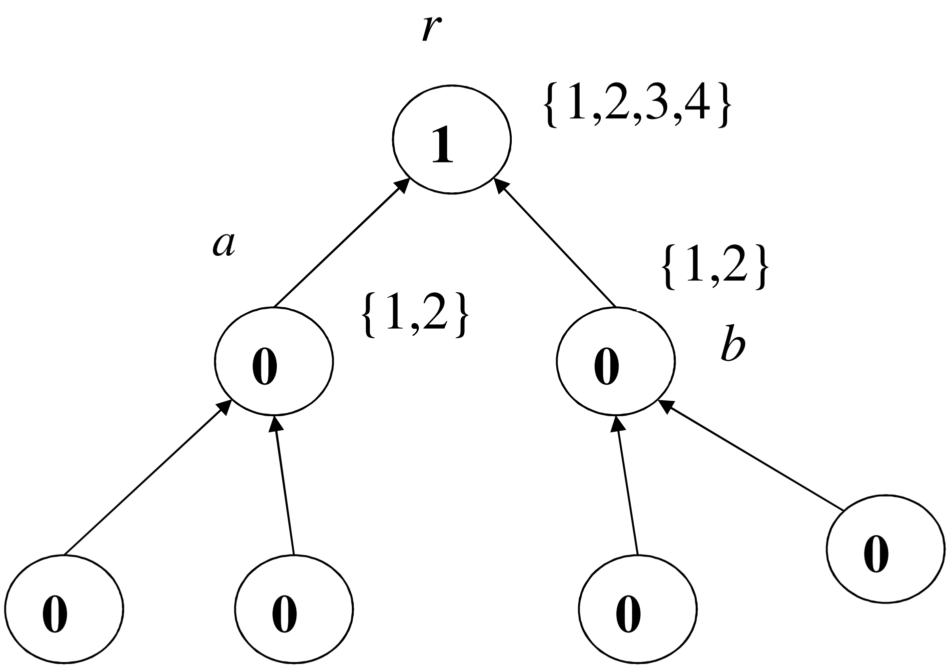
Illustrating a situation when a node w is about to switch

Switching (contd.)



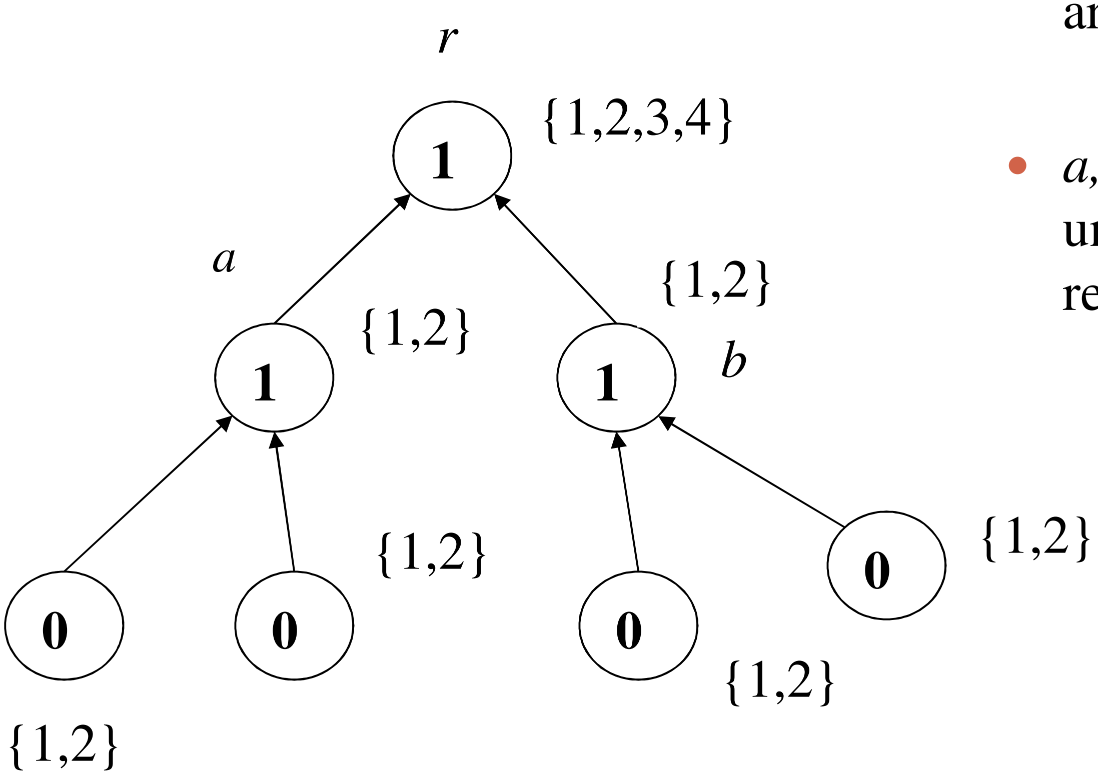
- Suppose *r* is ready to switch and has two messages 1,2 in buffer.
- *r* is not allowed to switch until its all neighbors has read those messages

Switching (contd.)



- Now *r* has switched. Node *a,b* are ready to switch.
- *a,b* are not allowed to switch until its theirs' neighbors have read those messages

Switching (contd.)



- Now *r* has switched. Node *a,b* are ready to switch.
- *a,b* are not allowed to switch until its theirs' neighbors have read those messages

Switching (contd.)

- Let $U(i) \equiv b_i \neq b_{p_i}$
- Let $X(i) \equiv U(i) \wedge [(\forall j \in C_i) \neg U(j)] \wedge \neg U(p_i)$
- Let $Y(i) \equiv (\forall j \in N_i)\ (\mathrm{T}^{\mathrm{j}}_{\mathrm{M}} \geq \mathrm{T}^{\mathrm{i}}_{\mathrm{M}})$

Switching Protocol

$$(S_1) \quad X(r) \wedge Y(r) \longrightarrow b_r = f(L)$$

$$(S_2) \quad X(i) \wedge Y(i) \longrightarrow b_i = b_{p_i}$$

$$(S_3) \quad \neg X(r) \wedge b_r \neq f(L) \longrightarrow b_r = f(L)$$

$$(S_4) \quad \neg X(i) \wedge U(i) \wedge \neg U(p_i) \longrightarrow b_i = b_{p_i}$$

$$(S_5) \quad \neg U(i) \wedge T_M^i \neq max(B_i) \longrightarrow T_M^i = max(B_i)$$

$$(S_6) \quad \neg U(i) \wedge (\forall j \in N_i)(b_j = b_i) \wedge T_M^i = max(B_i) \wedge T_M^{p(b_i)} = max(B_{p(b_i)}) \wedge (T_M^i < T_M^{p(b_i)}) \longrightarrow read(B_{p(b_i)}, T_M^{p(b_i)} - T_M^i)$$

Legitimate State

- $U(i)$=*false* at each node i
- $Y(i)$=*true* at each node i

Properties

- Under no failure, each broadcast message *m* is eventually correctly *read* by all the nodes
- Under arbitrary transient failure, switching eventually terminates and each node eventually uses either T and T'

Important Parameters for Single Transient Fault

- **T_s** is the time when the faulty behavior of a node starts
- **T_{ss}** is the time when the faulty behavior of a node stops
- **T_r** is the time when a node recovers from its faulty behavior (i.e. legitimate state is reached)

Properties

- Under single transient failure, each broadcast message *m* read by each child of the faulty node *i* before time T_s is eventually correctly read by all the non-faulty nodes

- Under single transient failure, each broadcast message *m* that has not yet been read by the faulty node *i* before time T_{ss} is eventually correctly read by all the nodes

Self-Stabilizing Switching with Self-Stabilizing T and T’

- What happens if T and T’ are not pre-computed but obtained by some self-stabilizing algorithm
- A general self-stabilizing algorithm is given for dynamic switching between T and T’

Properties

- From any arbitrary state the algorithm eventually terminates.

- On termination the system uses either T or T’ for broadcast.

Consensus

- When to switch
- How to agree on that
- Every node proposes a value (0/1)
- The problem is to agree on a particular value
- It is fault that makes it challenging

Fundamental Properties of Consensus

- **Termination:** every non-faulty process must eventually decide
- **Agreement:** the final value decided by every non-faulty process must be same
- **Validity:** if every non-faulty process starts with a value **v** then the final decision must be **v**

[Note: no one knows who is behaving bad, but it is known that some are bad]

Consensus in asynchronous system

- Impossibility result by FLP (JACM 1985, 32(2))
- Concept of Failure Detectors for asynchronous consensus

Consensus in synchronous system

- Byzantine Genrals Problem
- **Result1:** If simple message passing is used then there is no solution to the byzantine generals problem with 3 generals out of whom 1 is traitor
- **Result2:** There exist a solution for 4 generals out of whom 1 is traitor
- **General solution with n >= 3f+1**

Future Work

- Distributed switching in time-varying networks
- Designing multiple initiator based distributed switching algorithms

References

- [1] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In 14th International Symposium on Distributed Computing (DISC), Toledo, Spain, October 2000.
- [2] A. Arora and M. Gouda. Distributed reset. IEEE Transactions on Computers, 43(9):1026–1038,September 1994.
- [3] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance overwireless networks. In The ACM Annual International Conference on Mobile Computing and Networking (MobiCom), Berkeley, California, USA, November 1995.
- [4] A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong. Shifting gears: Changing algorithms on the fly to expedite byzantine agreement. Information and Computation, 97(2):205–233, 1992.

References (contd.)

- [5] W. K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In 21st International Conference on Distributed Computing Systems (ICDCS), Phoenix (Mesa), Arizona, USA, April 2001.
- [6] Z. Collin and S. Dolev. Self-stabilizing depth-first search. Information Processing Letters, 49(6):297–301, March 1994.
- [7] S. R. Das, C. E. Perkins, and E. M. Royer. Performance comparison of two on-demand routing protocols for ad hoc networks. In 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Tel-Aviv, Israel, March 2000.
- [8] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. Communications of the ACM, 17(11):643–644, November 1974.

References (contd.)

- [9] Shiwa S. Fu, Nian-Feng Tzeng, and Zhiyuan Li. Empirical evaluation of distributed mutual exclusion algorithms. In 11th International Symposium on Parallel Processing (IPPS), Geneva,Switzerland, April 1997.
- [10] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), Seattle, Washington, USA, August 1999.
- [11] T. Herman. Adaptivity through distributed convergence. Ph.D. Thesis, Department of Computer Science, University of Texas at Austin, 1991.
- [12] S. T. Huang and N. S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. Information Processing Letters, 41(2):109–117, February 1992.

References (contd.)

- [13] V. Jacobson. Congestion avoidance and control. In ACM SIGCOMM Symposium on Communications Architectures and Protocols, Stanford, California, USA, August 1988.
- [14] A. Jain, S. Karmakar, and A. Gupta. Adaptive connected dominating set – an exercise in distributed output switching. In 8th International Conference on Distributed Computing and Networking (ICDCN), Guwahati, India, December 2006.
- [15] X. Liu and R. van Renesse. Brief announcement: Fast protocol transition in a distributed environment. In 19th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Portland, Oregon, USA, July 2000.
- [16] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In IEEE International Workshop on Applied Reliable Group Communication, Phoenix, Arizona, April 2001.

References (contd.)

- [17] A. J. Martin. Distributed mutual exclusion on a ring of processes. Science of Computer Programming, 5(3):265–276, October 1985.
- [18] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. Communications of the ACM, 26(1):90–95, January 1983.
- [19] J. Mocito and L. Rodrigues. Run-time switching between total order algorithms. In 12th European Conference on Parallel Computing (Euro-Par), Dresden, Germany, August 2006.
- [20] Venugopalan Ramasubramanian, Zygmunt J. Haas, and Emin G¨un Sirer. SHARP: a hybrid adaptive routing protocol for mobile ad hoc networks. In 4th ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc), Annapolis, Maryland, USA, June 2003.

References (contd.)

- [21] K. Raymond. A tree-based algorithm for distributed mutual exclusion. ACM Transactions on Computer Systems, 7(1):61–77, February 1989.
- [22] O. Rutti, P.Wojciechowski, and A. Schiper. Structural and algorithmic issues of dynamic protocol update. In 20th International Parallel and Distributed Processing Symposium (IPDPS), Rhodes Island, Greece, April 2006.
- [23] Sang H. Son. An adaptive checkpointing scheme for distributed databases with mixed types of transactions. IEEE Transactions on Knowledge and Data Engineering, 1(4):450–458, December 1989.
- [24] B. Williams and T. Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In 3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc), Lausanne, Switzerland, June 2002.

Thank You