IPFRR WITH FAST NOTIFICATION

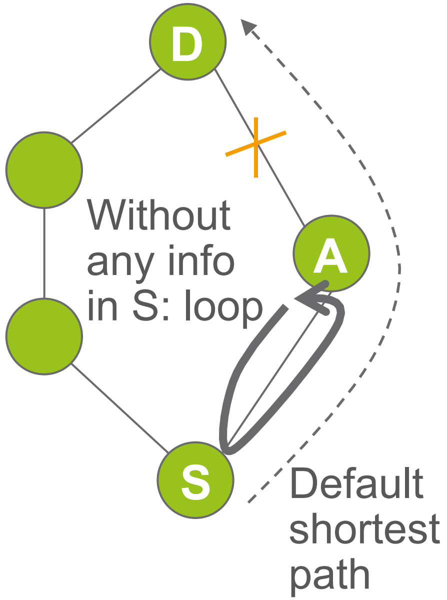
András CSÁSZÁR,
Gábor ENYEDI,
Sriganesh KINI

Outline

- Conceptual idea
- Details
  - Preparation phase
  - Fail-over operation
  - Packet contents
  - Bypassing legacy nodes

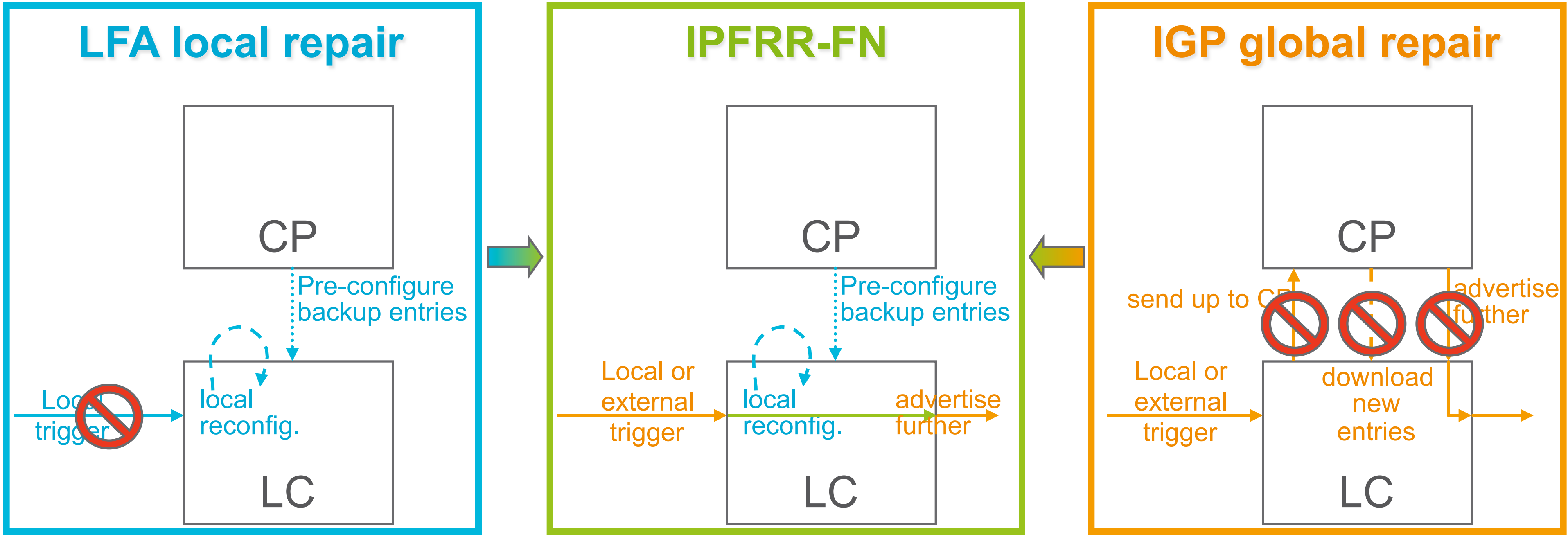
IPFRR – A Fresh Perspective

- LFA is not perfect because not all failures can be repaired purely locally
- IGP can repair all failures but it is slow
- Can we combine them and get the best of both worlds?
- YES, WE CAN



IPFRR with Fast Notification

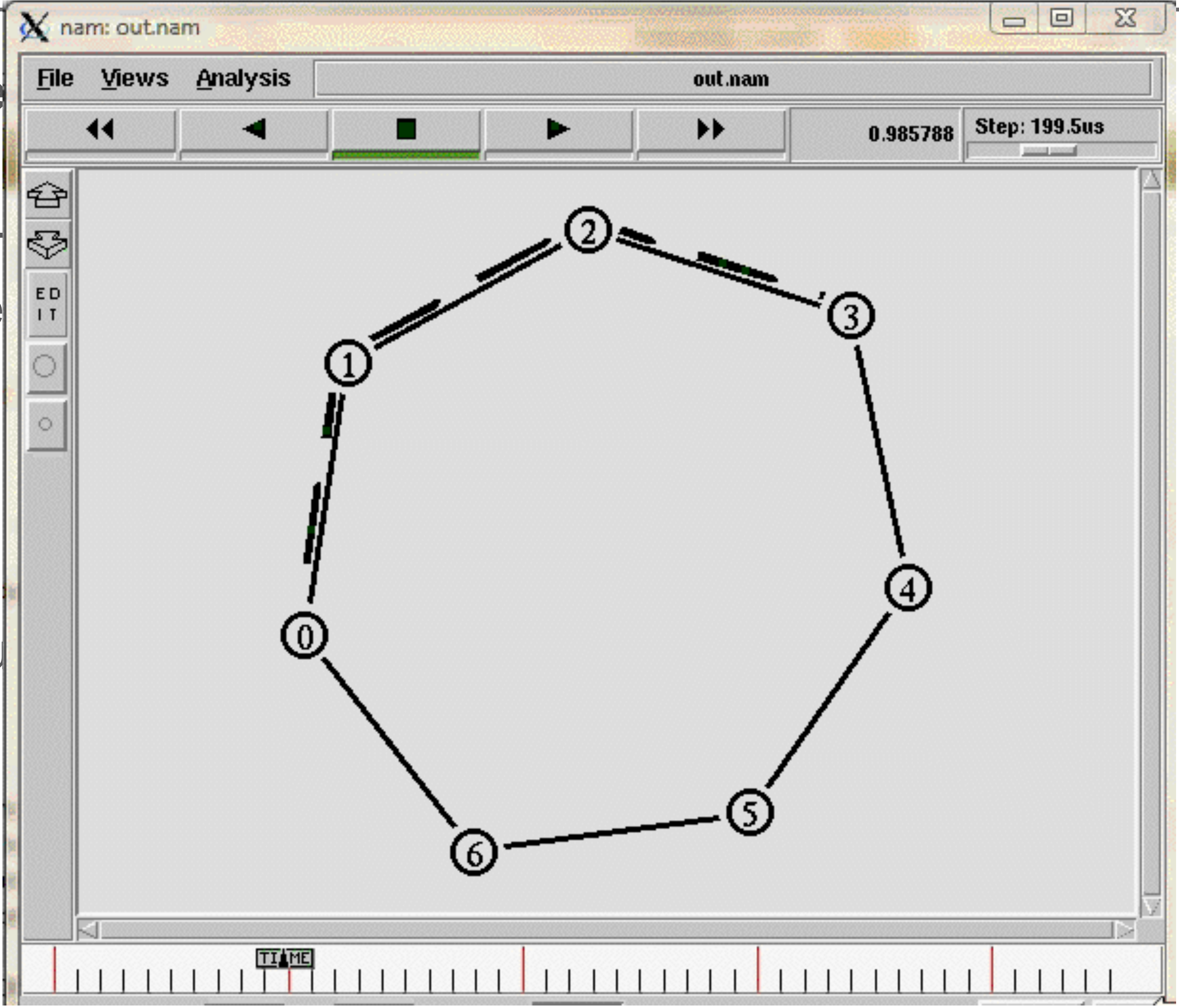
Basic idea



- Have failure information propagate the area quickly: on the fast path, without control plane involvement
- Distant nodes can switch to pre-calculated alternative next-hops (if needed)
- Preparing for remote failures also investigated in “Remote LFAPs”
  - draft-hokelek-rlfap-01
  - Needs a proper notification procedure

Network Simulator ns-2 did Fast Notification 13 years ago!

- Re
  - –
- ns-
- Se
  - –
  - –
  - –
- Ou
  - –
  - –



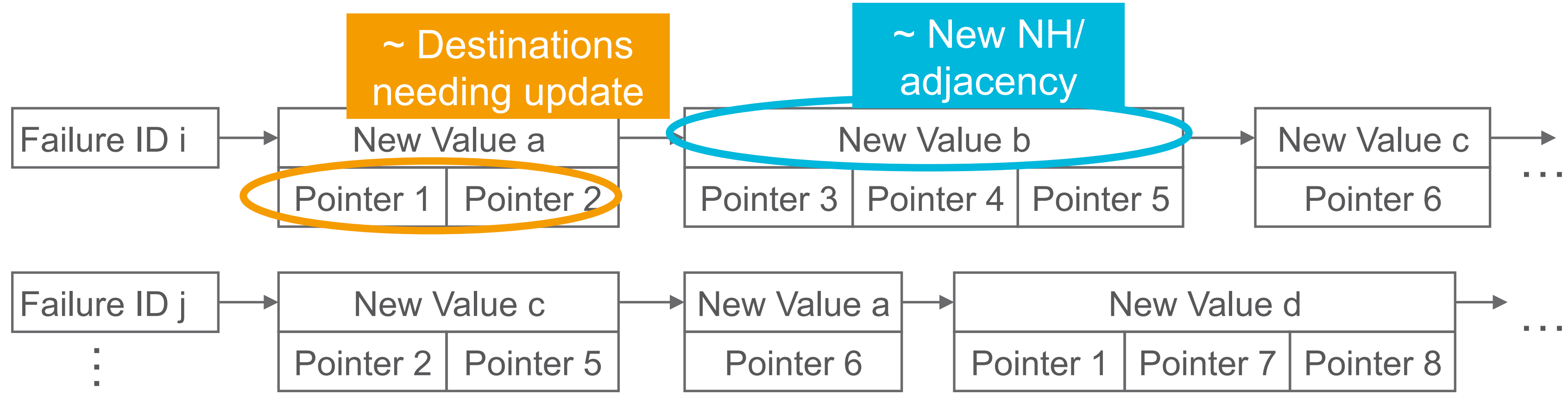
IPFRR based on Fast Notification: Main components

Preparation for the failure	**Fail-over mechanism**	
	› Quick local failure detection in the linecard	BFD, L2 trigger, LoS, etc.
› Pre-calculate and pre-install routes to distribute notifications	› Originate notification from within the linecard immediately › Distribute notification	Fast Notification service (draft-lu-fn-transport)
› Pre-calculate and pre-install failure specific alternative routes	› Process notification in the linecard – perform fail-over without consulting the control plane	**This draft**

Preparation Phase

- Let the IGP pre-calculate its response to protected failures
  - If the failure of a protected resource resulted in FIB change, pre-install this change to the forwarding plane
- Area scope: prepare only for intra-area failures
- Backup calculation/storage limited by:
  - Only need to prepare for failures on the SPT from the current node
  - Only need to install a backup route for a failure if failure specific alternate next-hop is different from primary NH

The IPFRR detour is identical to the “final” path after re-convergence!



~ Destinations needing update

~ New NH/ adjacency

Failure ID i → New Value a | Pointer 1 | Pointer 2 → New Value b | Pointer 3 | Pointer 4 | Pointer 5 → New Value c | Pointer 6 → …

Failure ID j → New Value c | Pointer 2 | Pointer 5 → New Value a | Pointer 6 → New Value d | Pointer 1 | Pointer 7 | Pointer 8 → …

⋮

Fail-Over: Step by Step
Initiate FN

- Detect failure
- Create (or just load) payload of FN packet
  - OrigID: Identifier of the originator
  - NbrID: Identifier of the node to which the connectivity was lost
  - LinkID: Identifier of the link/SRLG through which the connectivity was lost
  - Sequence number: LSDB digest
    - To protect against replay attacks
- Disseminate using FN
  - Redundant tree distribution mode is preferred
  - Punt and forward in each hop
  - FN packet loss should be minimised (e.g. priority)
    - Redundant trees ➔ likely receiving multiple replicas
      - At least one should get through to every node

} Data pre-loaded by IGP to forwarding plane

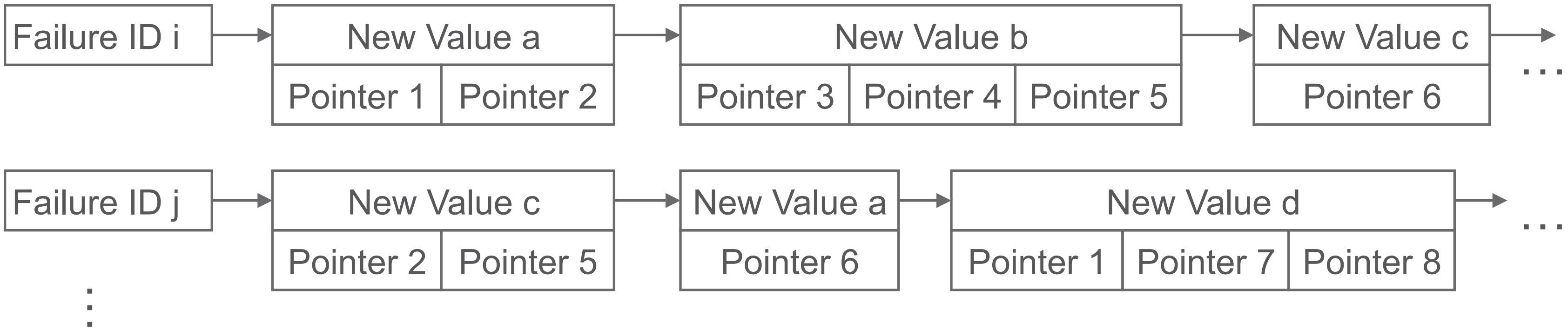
Fail-Over: Step by Step
Activate Backups after Receiving FN

› Implementation dependent

› In general:

– Find routes which have alternates installed for `FailureID`

› E.g. if Failure ID in pkt is “j”, make updates described by second row:



Failure ID i → New Value a (Pointer 1 | Pointer 2) → New Value b (Pointer 3 | Pointer 4 | Pointer 5) → New Value c (Pointer 6) → …

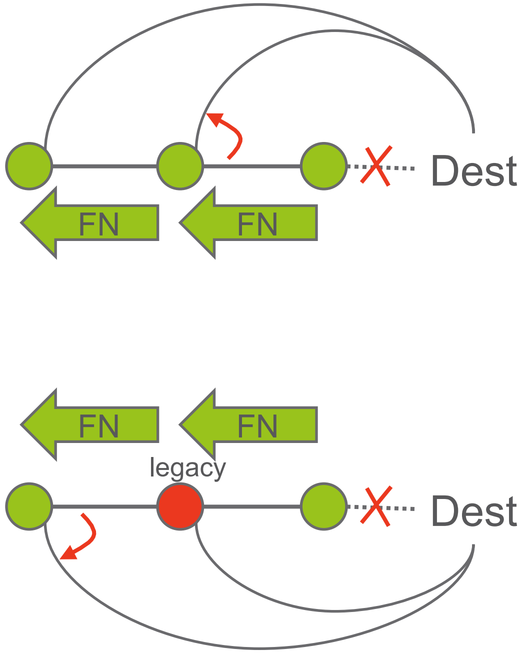
Failure ID j → New Value c (Pointer 2 | Pointer 5) → New Value a (Pointer 6) → New Value d (Pointer 1 | Pointer 7 | Pointer 8) → …

⋮

IPFRR with Fast Notification

Legacy Node Bypass

- Legacy
  - It can at least forward the multicast packets of FN
  - FN packets are not recognised/processed → routes are not changed!
- FN-capable nodes
  - When pre-calculating backups, have to consider that legacy nodes won't change routes
- Needs:
  - Advertisement of FN capability
  - Router Capability TLVs
    - OSPF [RFC4970]
    - IS-IS [RFC4971]



Summary

- FN enables preparing for remote failures
- IGP runs pre-calculation in the background preparing for topology changes
  - IPFRR detour identical to “final” path (after IGP re-convergence)
    ➔ eliminates IGP micro-loop

Questions, comments?

- FN Framework
- FN Transport
- IPFRR FN

BACKUP SLIDES

IETF’s Existing IPFRR

- Assumes very quick failure detection – not part of IPFRR
  - e.g. BFD, lower layer upcall

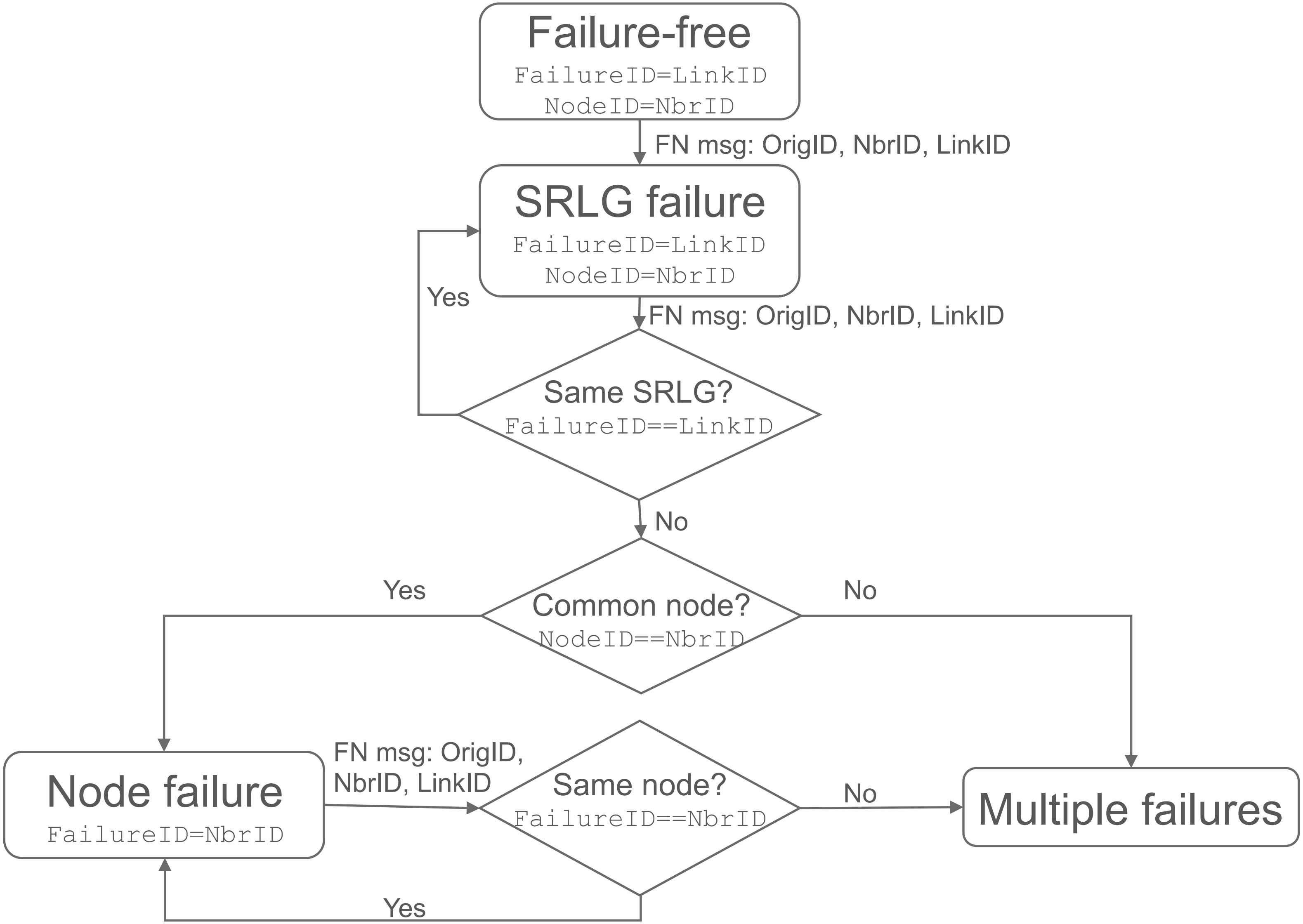
- “…to compute backup routes that allow the failure to be repaired locally by the router(s) detecting the failure without the immediate need to inform other routers of the failure…”

- Options
  - Loop free alternates (LFA)
  - “Multi-hop repair paths”

Existing Solutions

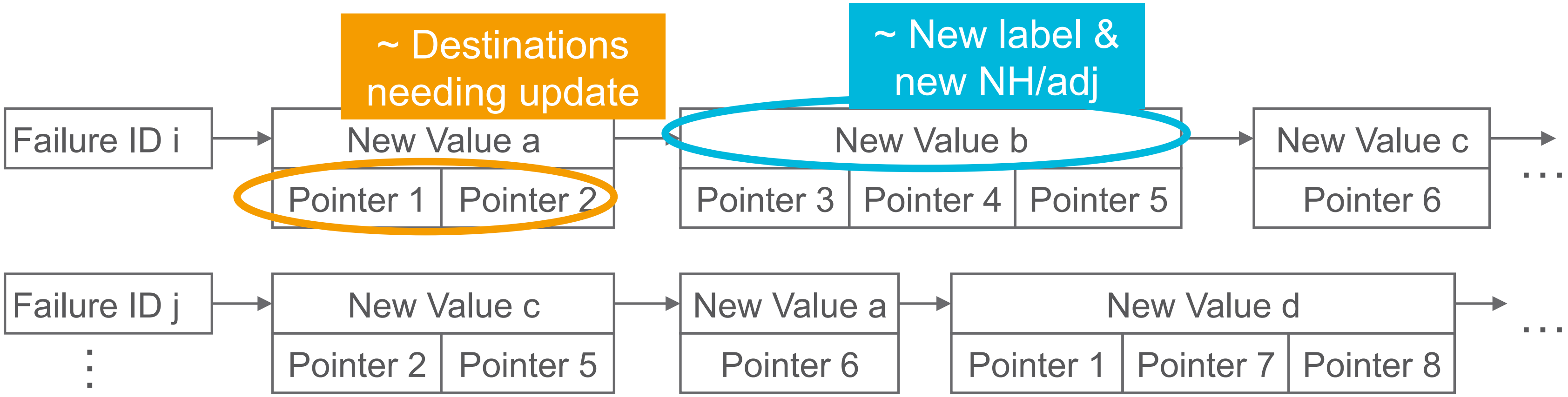
- Rely on safe, loop-free neighbours (LFA)
  - Loop free alternative next-hops not always exist: not full coverage (ca. 80% in practice)
    - Especially, if failure handling is needed bi-directionally
    - LFA can be good enough if we are in control of topology
- Multi-hop repair paths (full coverage)
  - Rely on tunnelling/encapsulation (e.g. Not-Via Addresses)
    - Encapsulation not preferred due to fragmentation at MTU (SAR decreases forwarding performance)
    - Special tunnel endpoint addresses represent extra management burden
  - Rely on interface-specific forwarding (e.g. FIFR, U-Turn LFA)
    - Existing router design often have the same replica of the forwarding table at each linecard (serving multiple interfaces/adjacencies) – an assumption deep in HW/SW → hard to change
  - Assume packet marking to encode routing configuration ID (e.g. MRC)
    - No free bits in IP header for this purpose
    - Alternative would be encapsulation (see above)

Fail-Over: Step by Step
Processing FN: Identifying the Failure



Application to LDP

- LDP unsolicited / liberal retention mode
- FN initiates MPLS FIB update



~ Destinations needing update

~ New label & new NH/adj

Failure ID i → New Value a (Pointer 1 | Pointer 2) → New Value b (Pointer 3 | Pointer 4 | Pointer 5) → New Value c (Pointer 6) → …

Failure ID j → New Value c (Pointer 2 | Pointer 5) → New Value a (Pointer 6) → New Value d (Pointer 1 | Pointer 7 | Pointer 8) → …

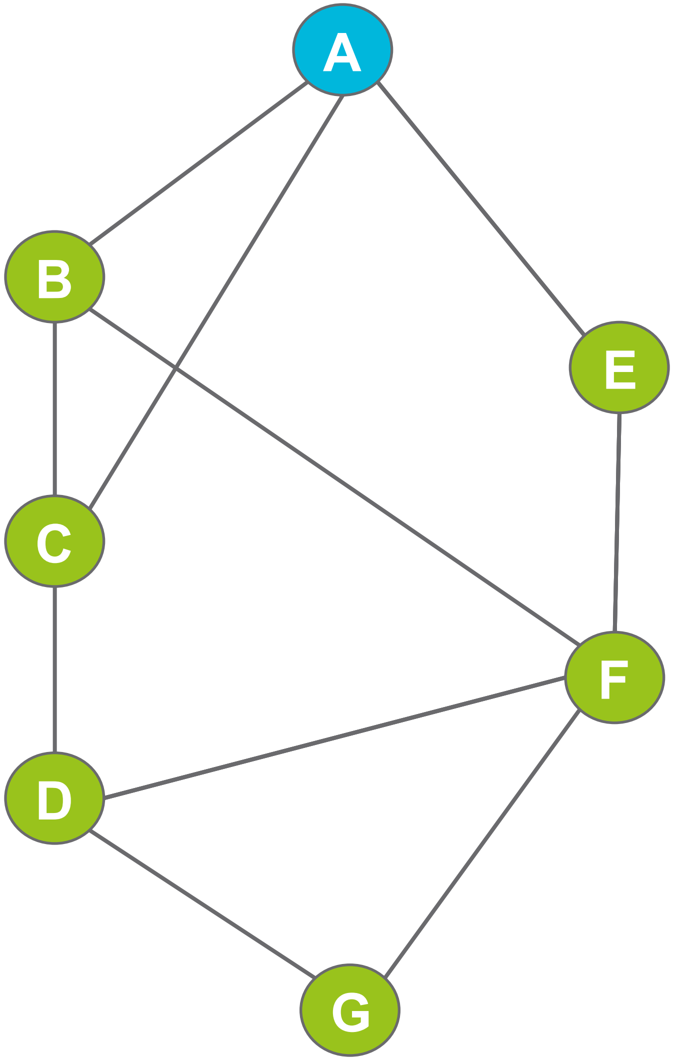
⋮

Building a Redundant Tree – An Example

1. Select a Root

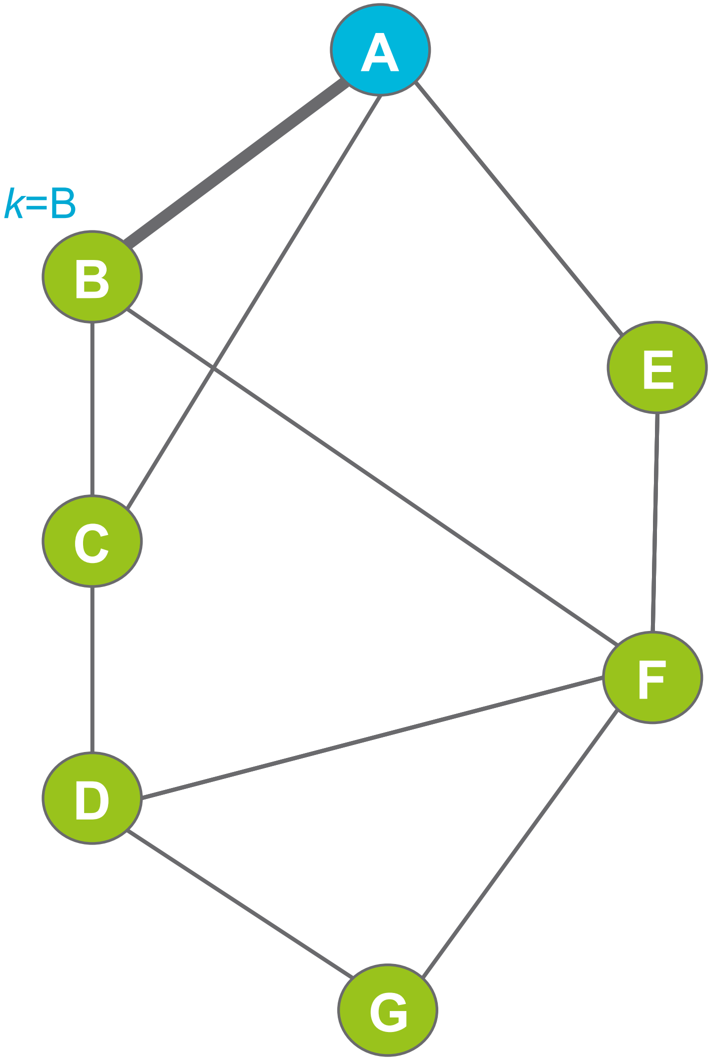
Irrelevant which router,
but it has to be consistent

2. Let *k* be one of Root's neighbours
3. Find the shortest path from *k* to the Root without the direct k-Root link
4. On the resulting cycle, start from Root and add all links until the last node to red tree
5. Do the same in reverse direction, adding links to the blue tree
6. Add all nodes incl. Root and *k* to the *Ready* set



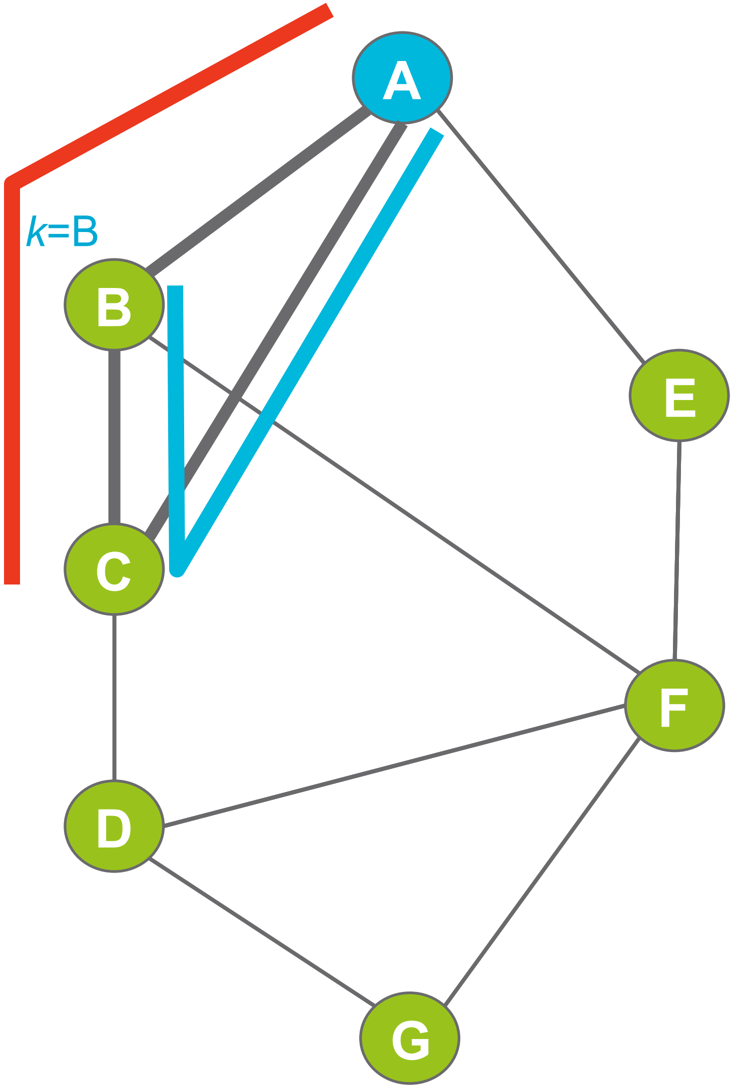
Building a Redundant Tree – An Example

1. Select a Root
2. Let *k* be one of Root' s neighbours
3. Find the shortest path from *k* to the Root without the direct k-Root link
4. On the resulting cycle, start from Root and add all links until the last node to red tree
5. Do the same in reverse direction, adding links to the blue tree
6. Add all nodes incl. Root and *k* to the *Ready* set



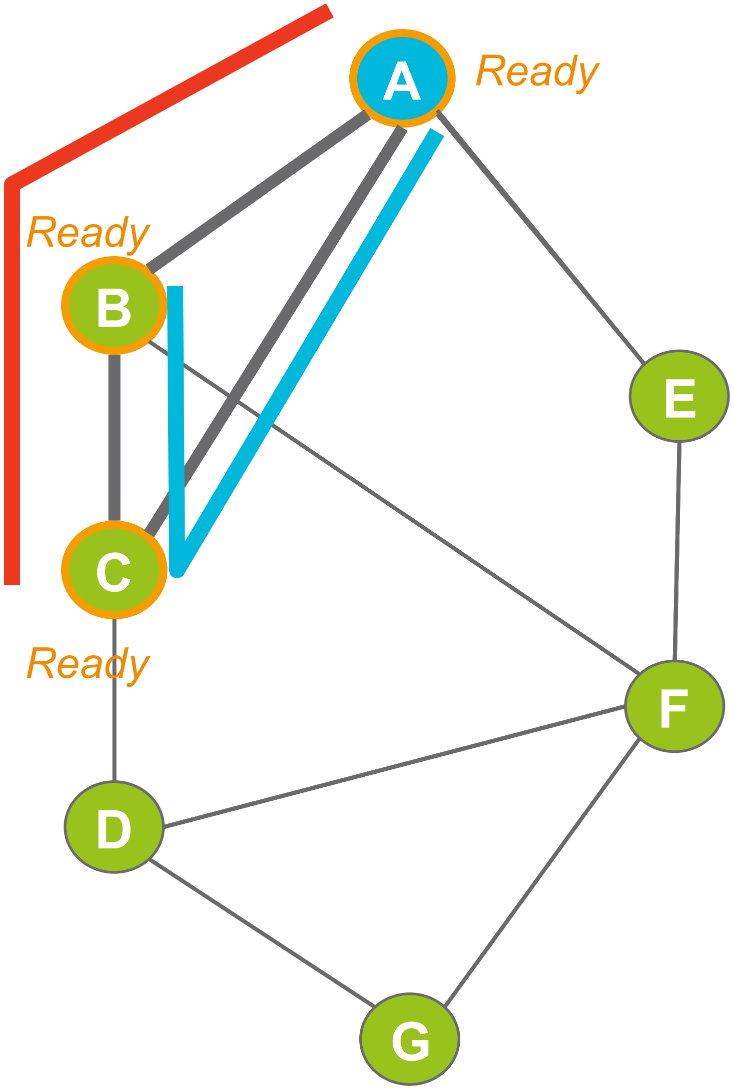
Building a Redundant Tree – An Example

1. Select a Root
2. Let *k* be one of Root’s neighbours
3. Find the shortest path from *k* to the Root without the direct k-Root link
4. On the resulting cycle, start from Root and add all links until the last node to red tree
5. Do the same in reverse direction, adding links to the blue tree
6. Add all nodes incl. Root and *k* to the *Ready* set



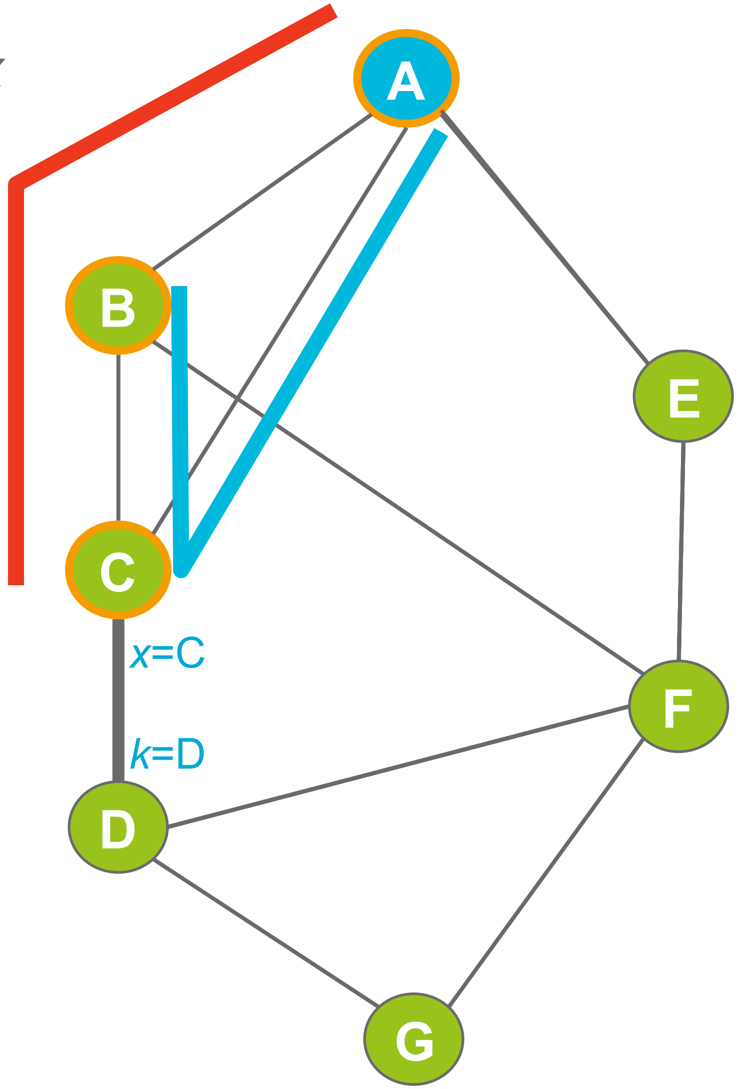
Building a Redundant Tree – An Example

1. Select a Root
2. Let *k* be one of Root's neighbours
3. Find the shortest path from *k* to the Root without the direct k-Root link
4. On the resulting cycle, start from Root and add all links until the last node to red tree
5. Do the same in reverse direction, adding links to the blue tree
6. Add all nodes incl. Root and *k* to the *Ready* set



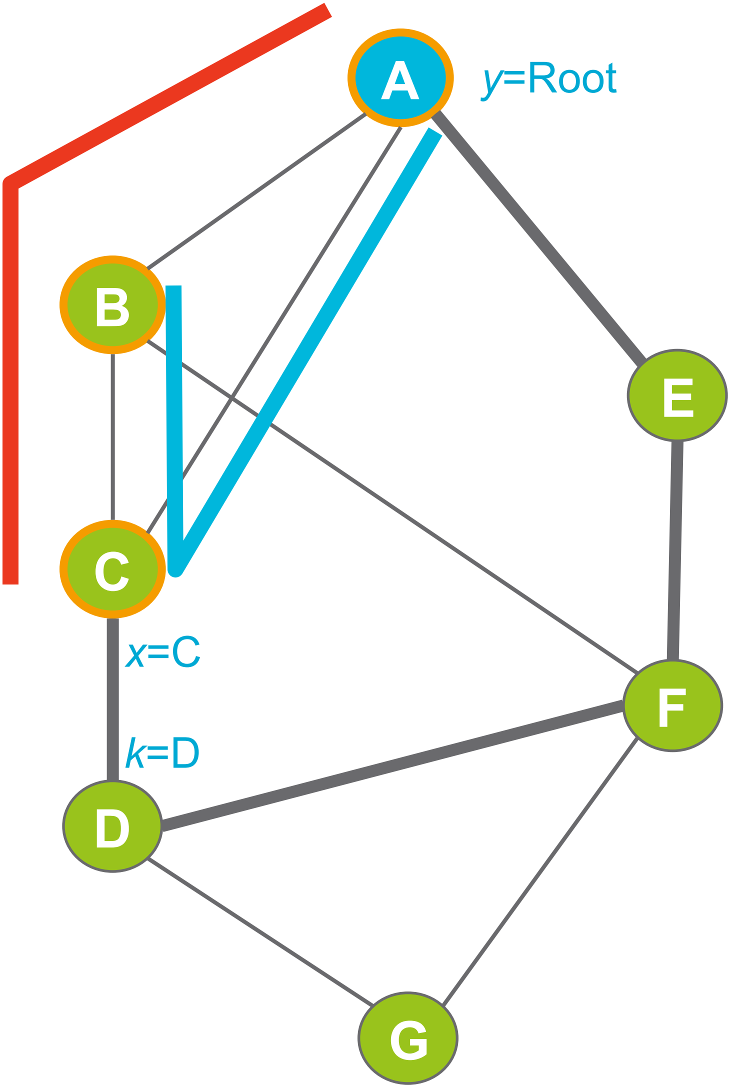
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



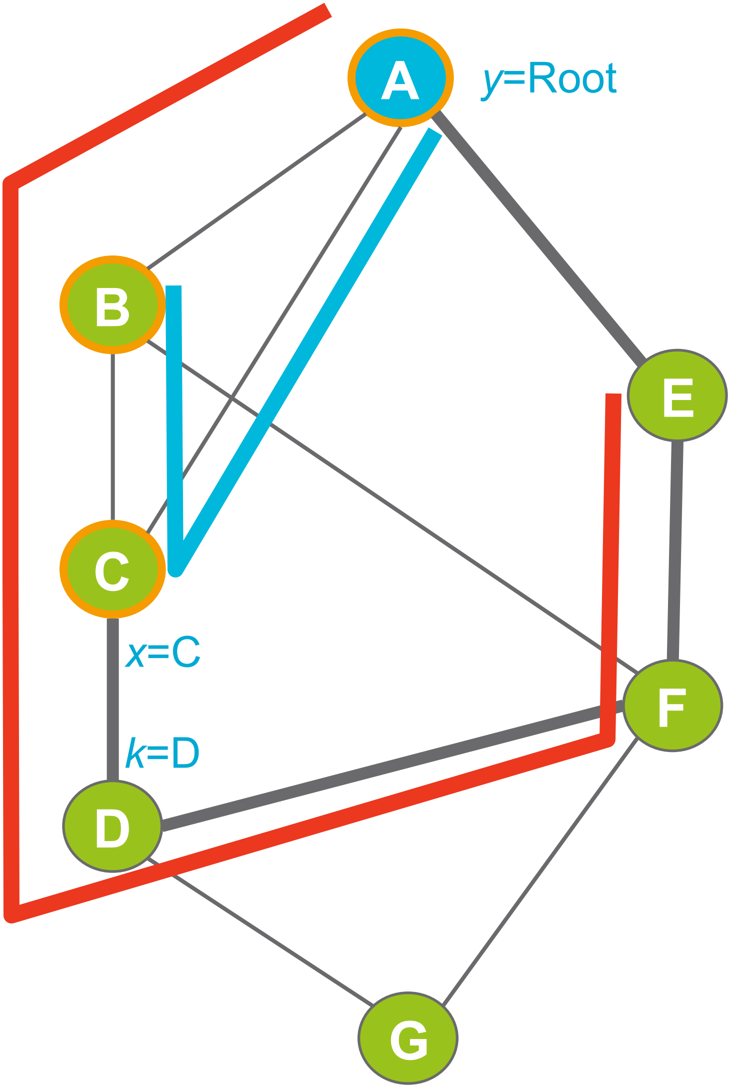
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



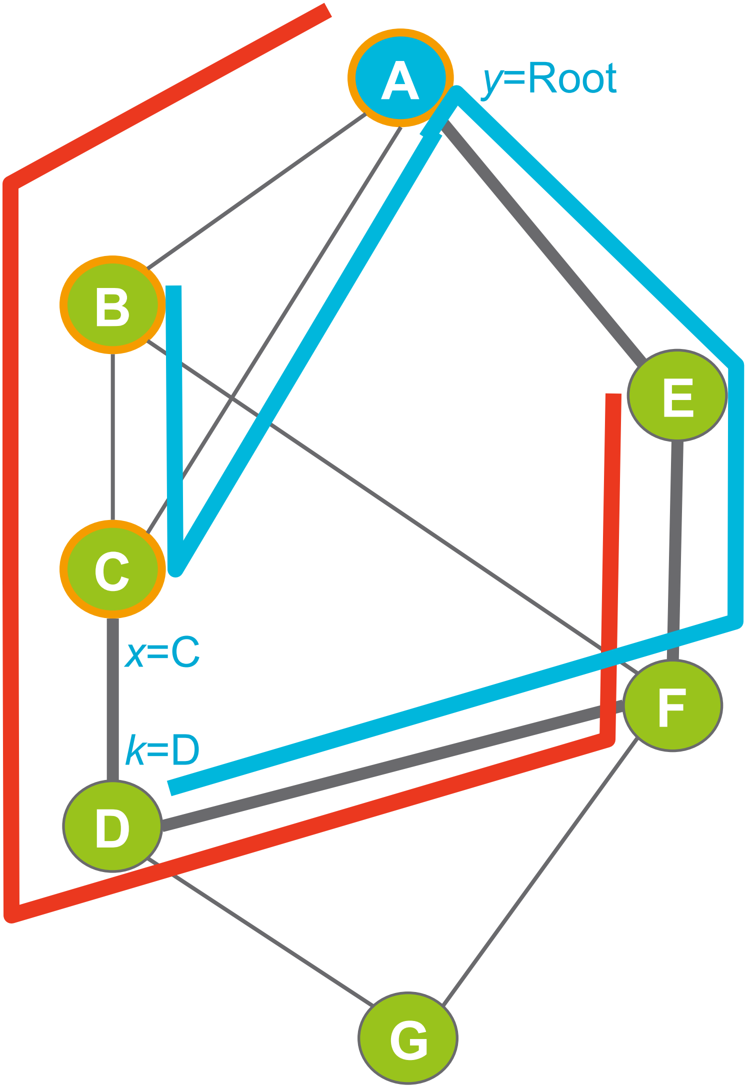
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



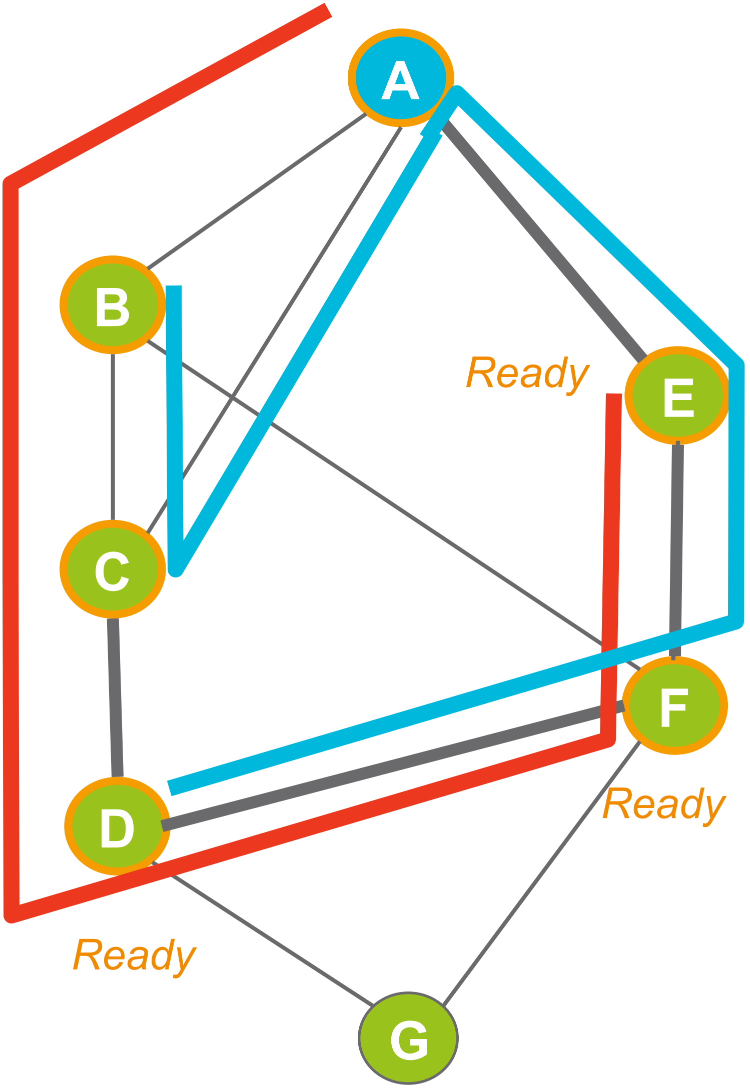
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k' s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



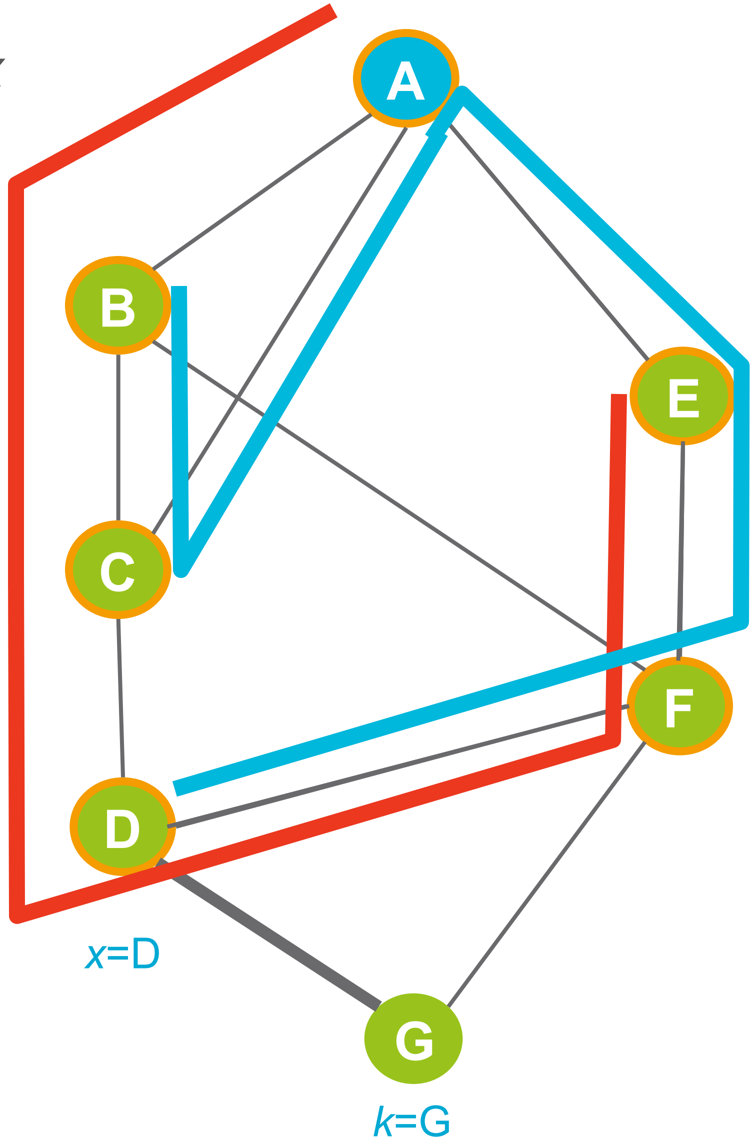
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k' s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



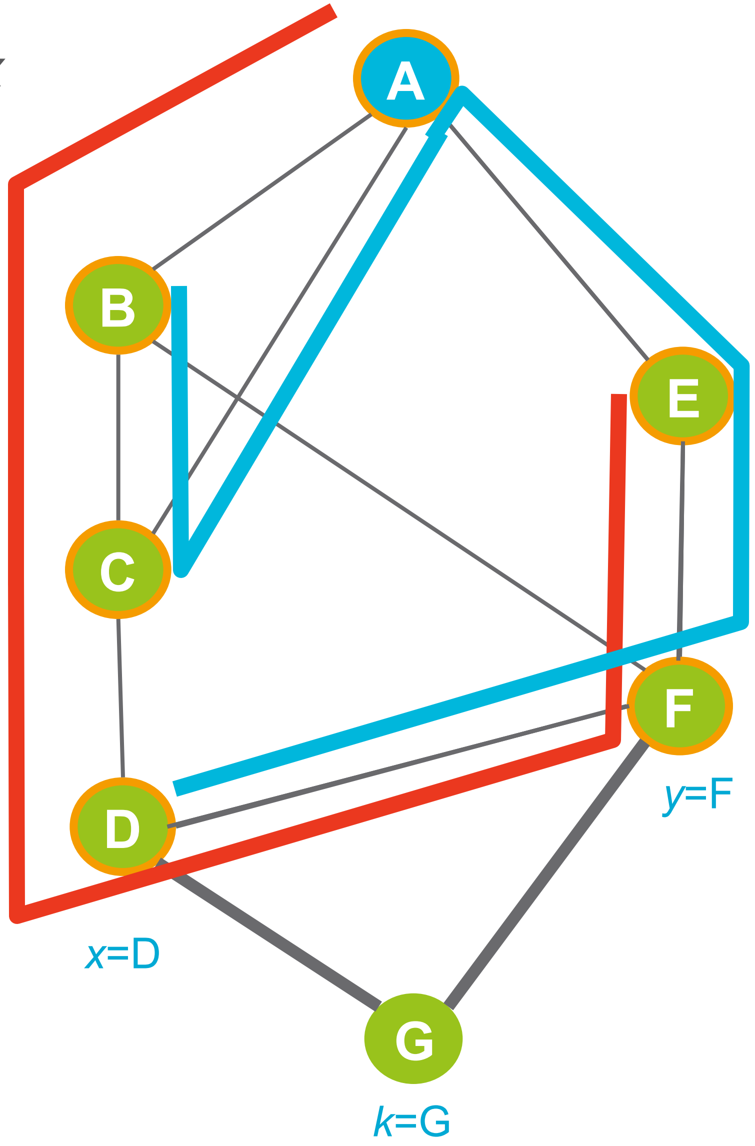
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



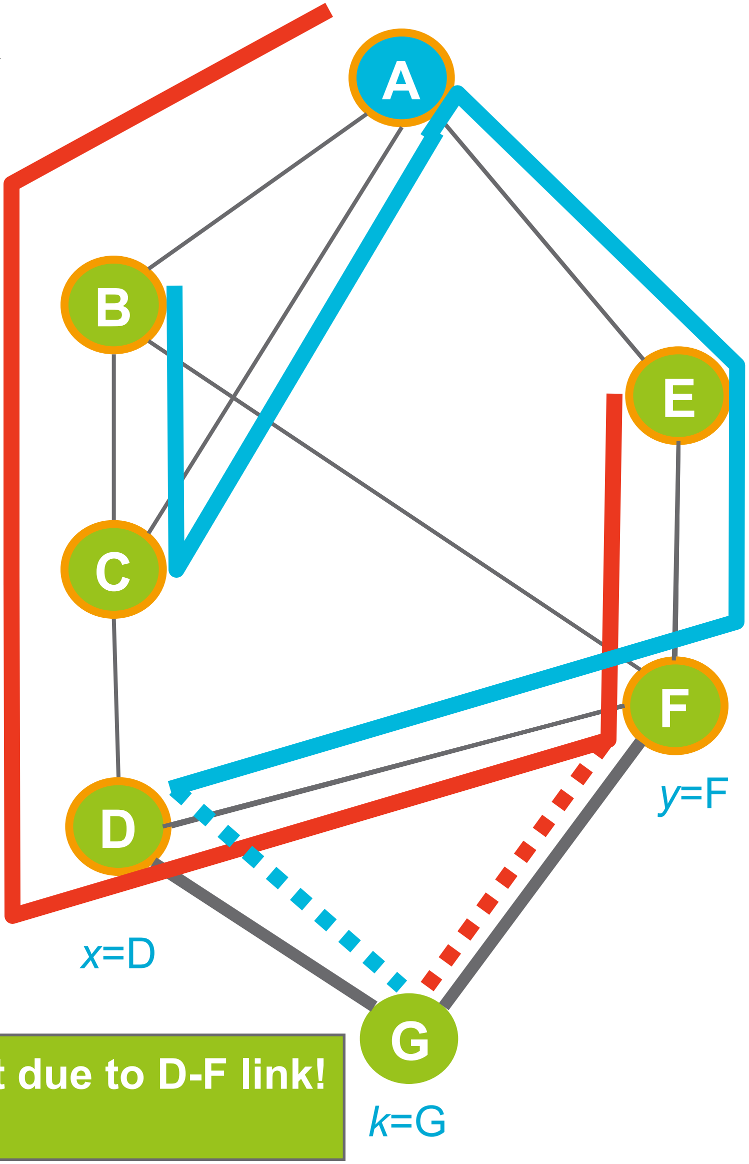
Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k's two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



Building a Redundant Tree – An Example

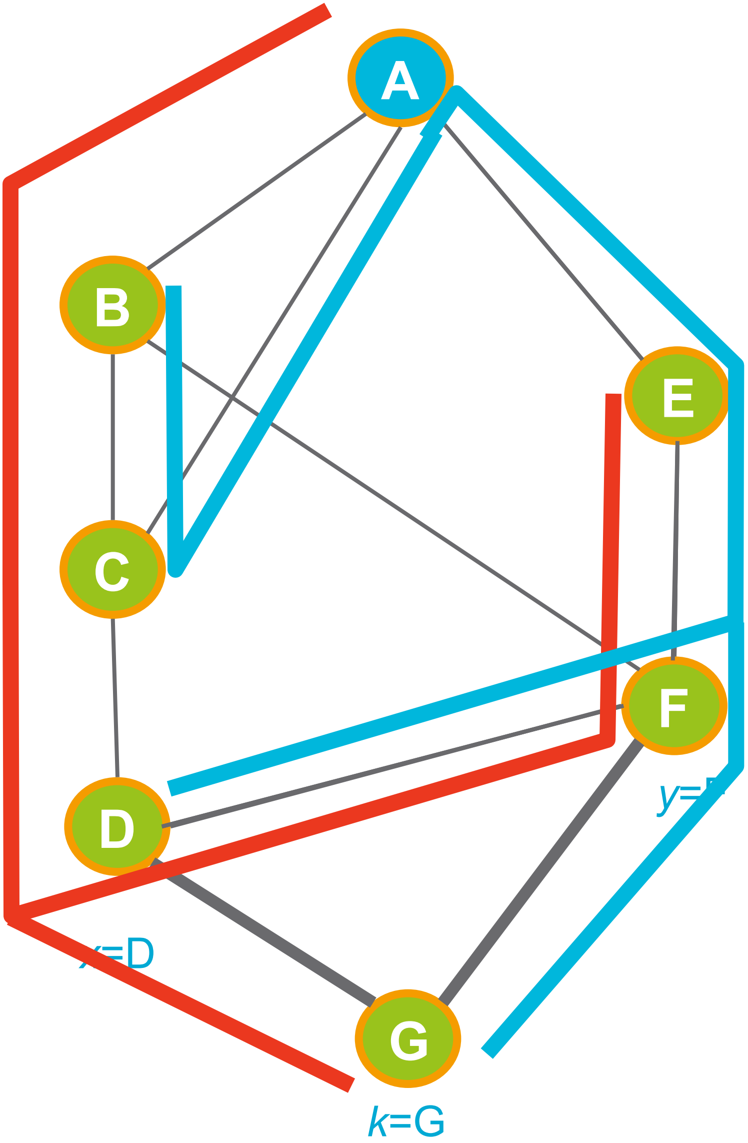
7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



Path from G to Root is not redundant due to D-F link!
REVERSE!

Building a Redundant Tree – An Example

7. Select a neighbour *k* of a *Ready* node *x*
8. Find the shortest path from *k* to Root without *x*
9. Let *y* be the first *Ready* node of the previously found path
10. Add the links/nodes between *x* and *y* to the to red tree in one direction, to blue tree in the other direction
11. If k’s two paths to the Root are not independent, do 10 in the reverse direction
12. Add all newly touched nodes to the *Ready* set
13. If there are non *Ready* nodes, Goto 7.



Tie Braking

- Irrelevant which node is selected for Root, it just has be consistent
- Irrelevant which neighbour is selected, just keep it consistent
- Irrelevant which shortest path is selected, just keep it consistent

- E.g. highest Router ID