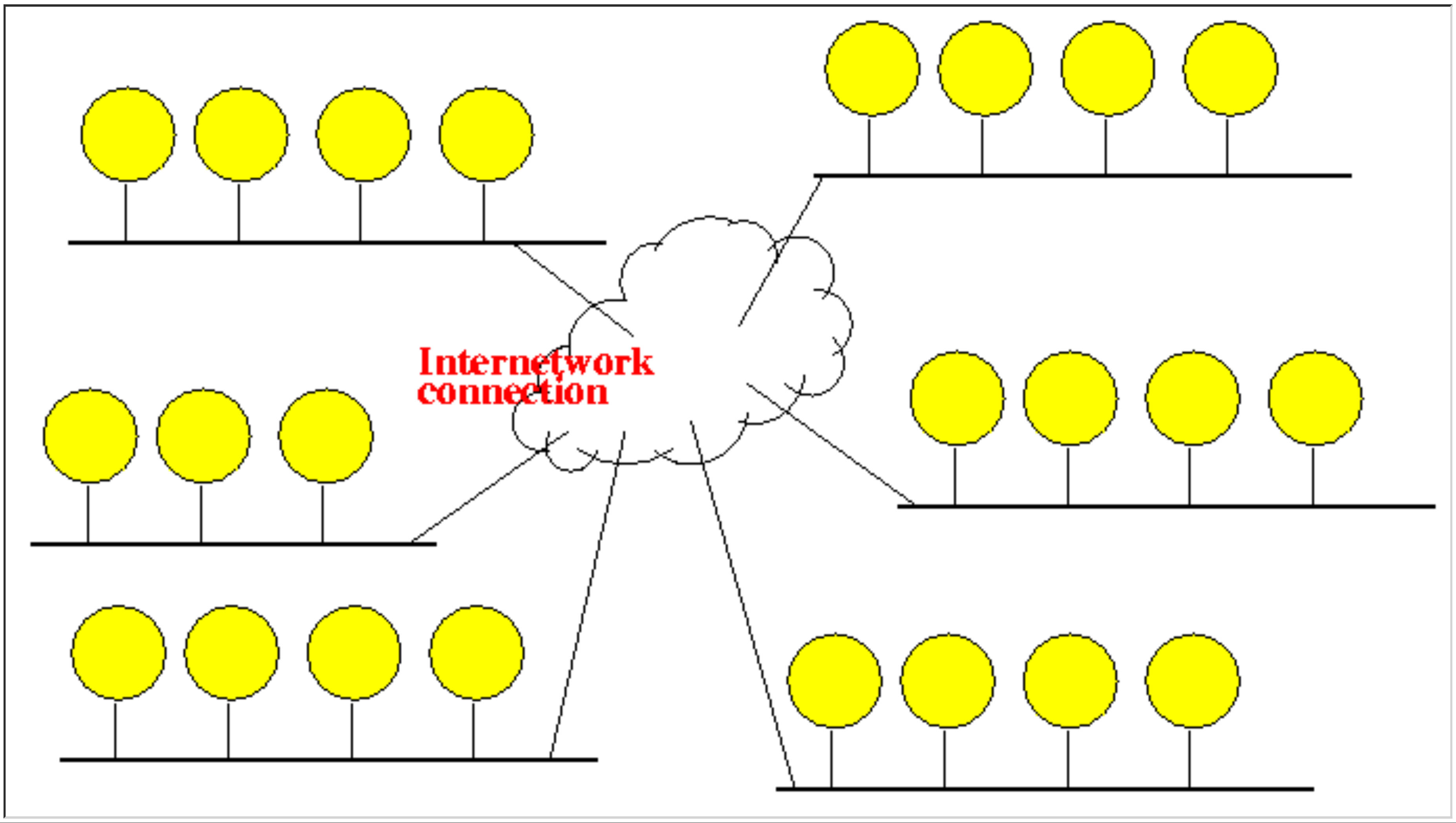
The challenges of making *larger* networks

- **Making a *bigger* network**
  - **So far:**
    - We discussed a ***single*** **802.x network**
  - ***Furthermore***:
    - The **802.x networks** span ***limited*** **geographical distances**

    Example:

    - **One Ethernet network** can support upto a few **100 computers**
    - **One Ethernet network** can span **distances upto 1000 ft**
  - We will **next** consider the **problem** of:
    - ***Interconnection*** multiple **networks**

    It is ***not*** as **easy** as ***simple*** connecting the **networks** together:



- **Order of discussion**

  - The **next 2** "sections" of the **webpages** will discuss:

    - Interconnecting ***homogeneous*** **networks**

      E.g.:

      - interconnect **multiple Ethernets**

    - Interconnecting ***heterogeneous*** **networks**

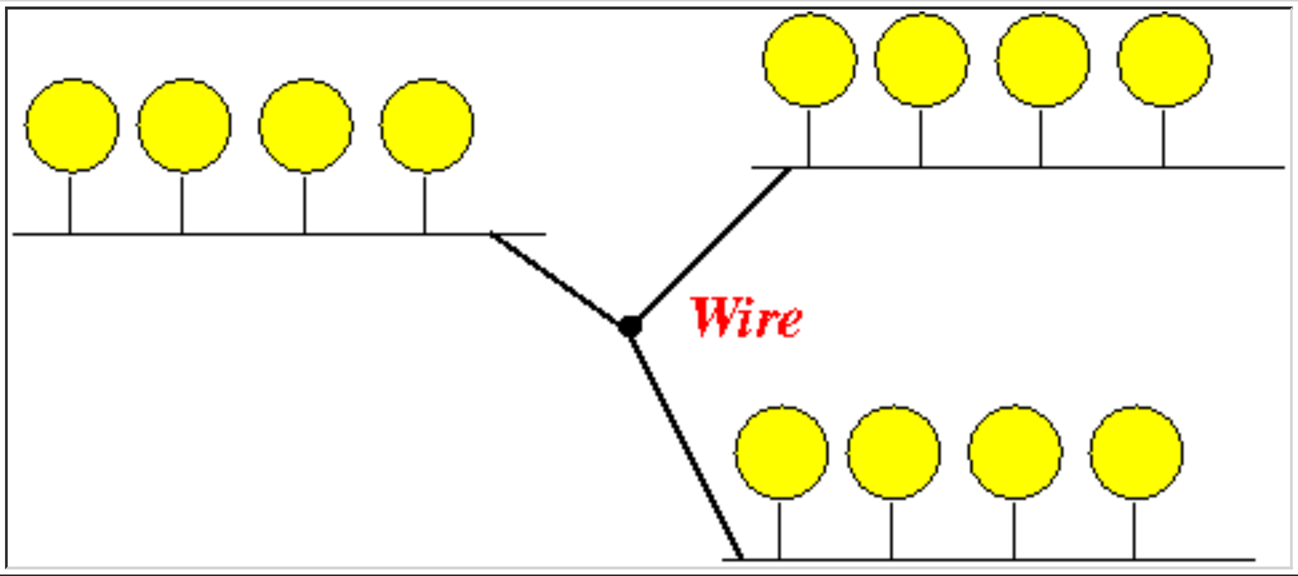
      E.g.:

      - interconnect **an Ethernet** and a **token ring**

Naive attempts to make *larger* networks....

- **Attempt #1: direct connection**
  - The **most *obvious* way** to **connect multiple networks**:
    - **Connect** the **networks *directly*** (e.g. with extension **wires**)

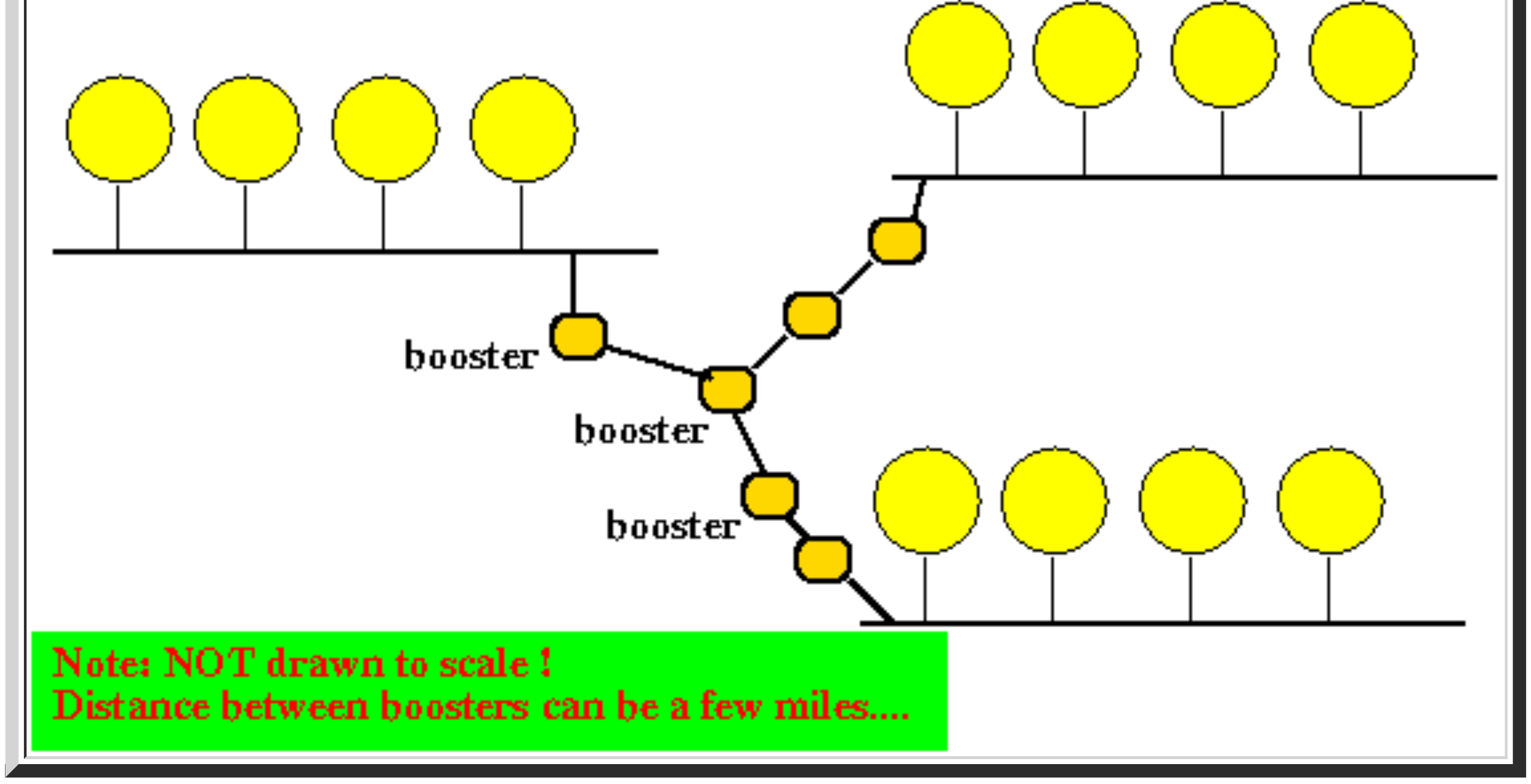
    Example:



  - **Why** this will **not work**:
    - **Signal strength *attentuates* (weakens)** over a **long distance**

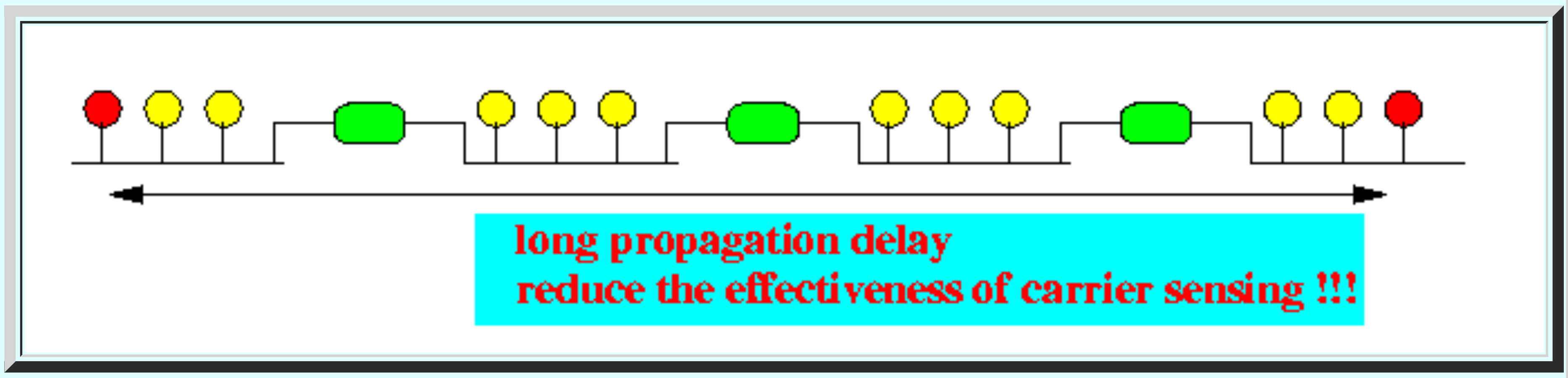
      Result:
      - **Many bit errors** due to **noise**

- **Attempt #2: use *amplifiers* (boosters)**
  - **Amplifier:**
    - **Amplifier (booster)** = a **circuit** that can **boost** the **signal strength**
  - **Attempt #2**:



- ***Problems*** with this **technique**:

  - The **resulting network** has a ***very*** **large end-to-end propagation delay**:

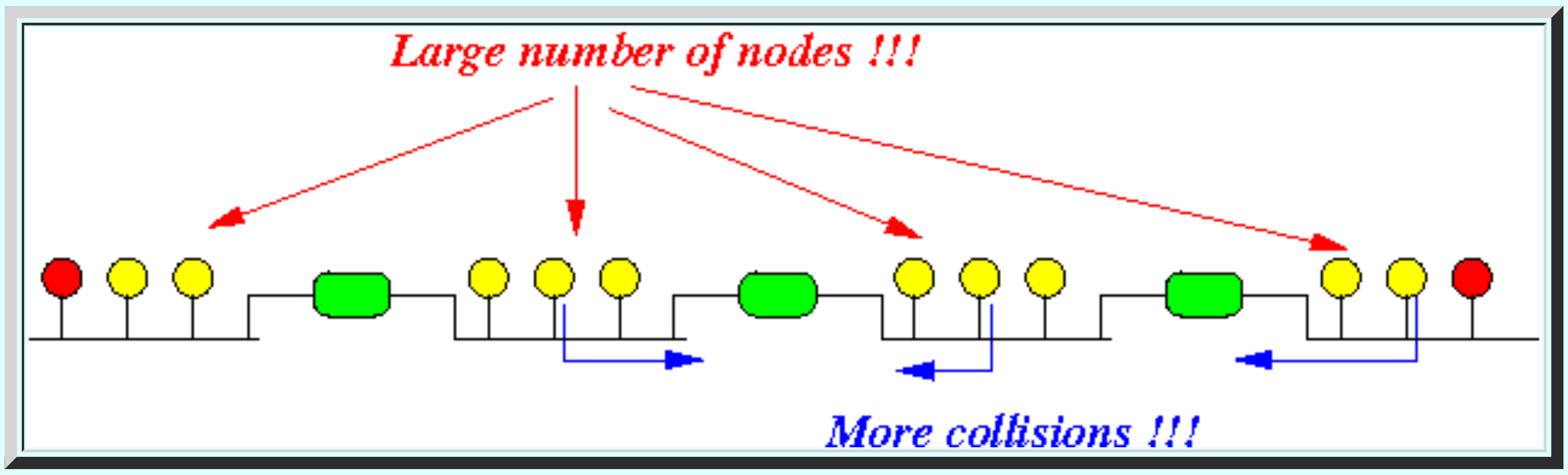
    

    Result:

    - ***Large*** **delays** will ***reduces*** the **effectiveness** of **carrier *sensing***
    - Remember **Aloha** ?
      - The **nodes** in the **Aloha network** were ***too far*** **apart** for **carrier sensing** to be ***useful***

        See: **click here**

  - There are ***many*** **nodes** in the **network**:

    Result:

    - **Increased number** of **transmissions**

- This will **increase** the number of **collisions**

- **Back off** will **more frequent** and will **take *longer*** to **resolve**

The *scalability* problem

- **The *Scalability* problem**
  - The **Scalability *problem*:**
    - A **larger network** have a ***larger* number** of nodes
    - **More nodes** means:
      - **greater chance** of **simultaneous transmissions**
  - **Fact:**
    - **Networking techniques** that works well with ***small* number** of **transmitters** will ***not* work** with a ***large* number** of **transmitters**
  - **CSMA/CD networks**:
    - **CSMA/CD protocol** does ***not* scale**

    Reason:

    - **More nodes** will cause ***more* collisions.**
    - **More *collisions*** results in **more *frequent* backoffs**
    - **More *frequent* back offs** results in ***longer* delays**
  - **Token based networks**:
    - **Token based networks** do ***not* scale**

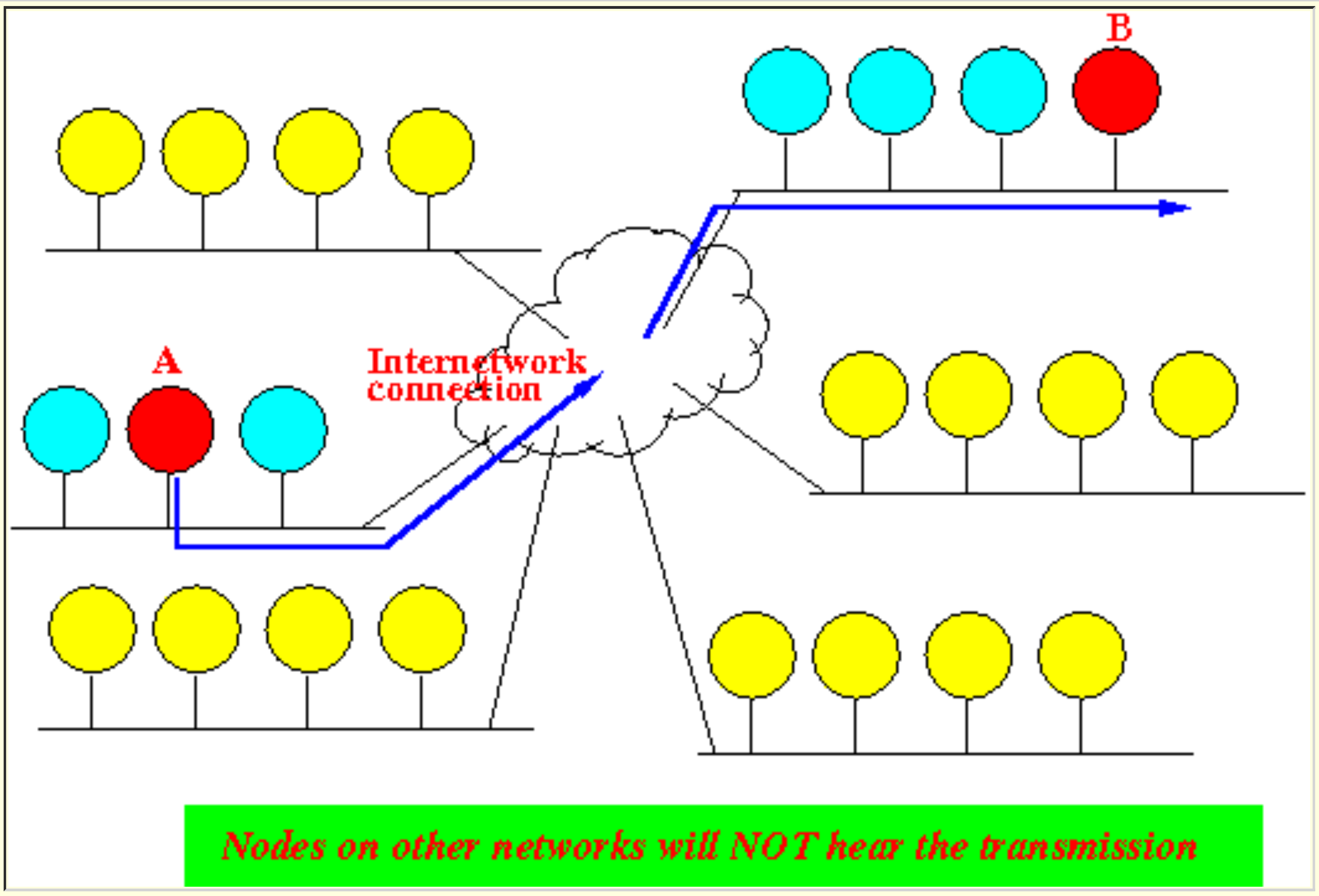
    Reason:

    - ***More* nodes** will **transmit *more* frames**

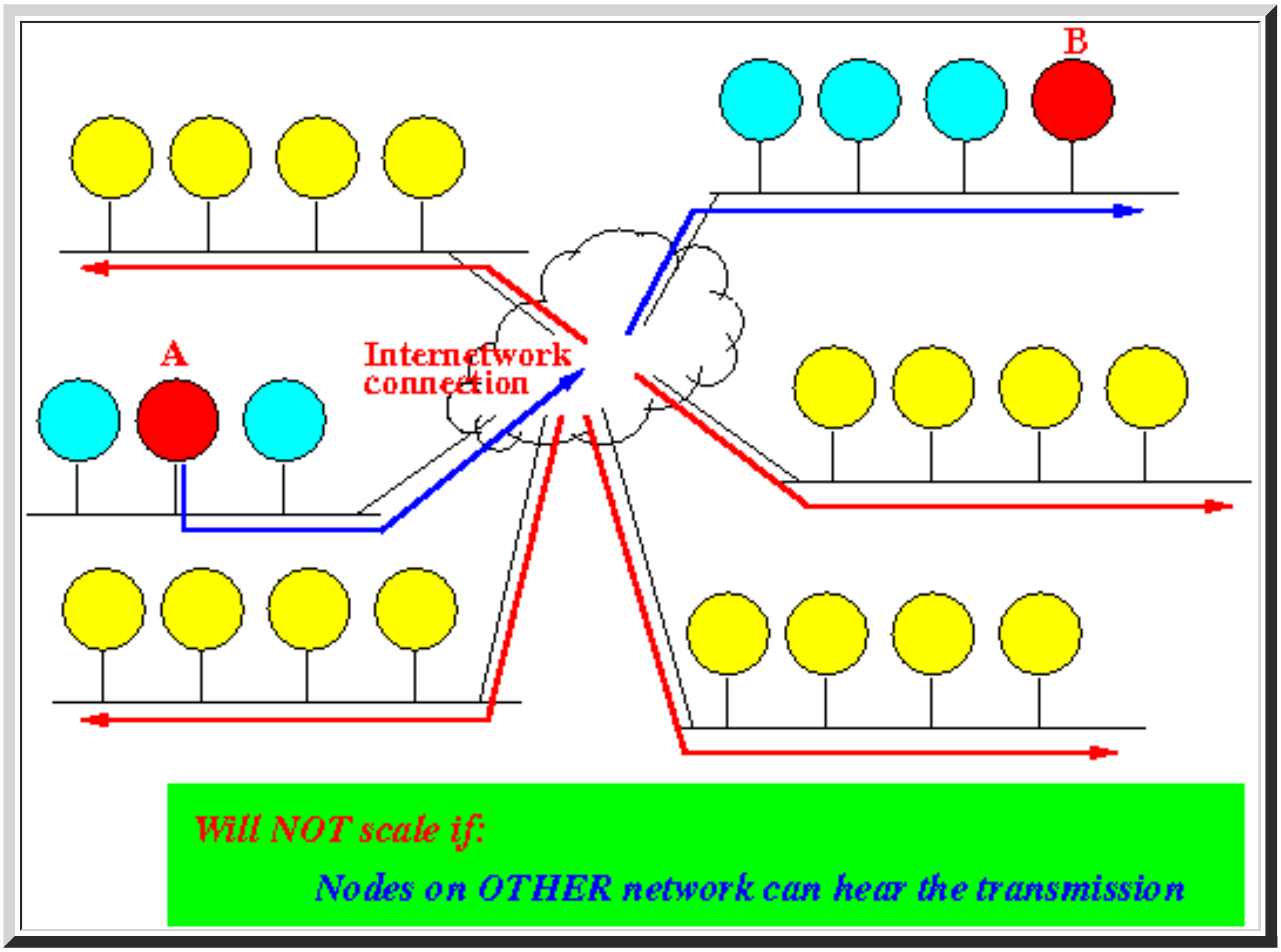
- ***More* frames** means ***longer* waiting time** for **your *next* turn**
- This **means:** ***longer* delays**

- ***Key* to achieve Scalability**
  - The **principle** to **achieve** **scalability**:
    - **Transmissions** between a **sender** and a **receiver** will
      - **"Intrude" (= use)** the ***home* networks** of the **sender** and the **receiver**
    - The **transmission** mist ***not* intrude (= use)** the ***home* networks** of ***non-participating* nodes**
  - **Graphically illustrated:**
    - ***If*** a **sender *A*** on **network 1** transmits a **message (frame)** to a **receiver *B*** on **network 2**, **then**:
      - ***only* nodes** on the **network 1** and **network 2** ***should*** be ***"disturbed"* (= hear)** the **transmission**

      Grpahically:



- **Example** of a ***non*-scalable transmission method**:



Because:

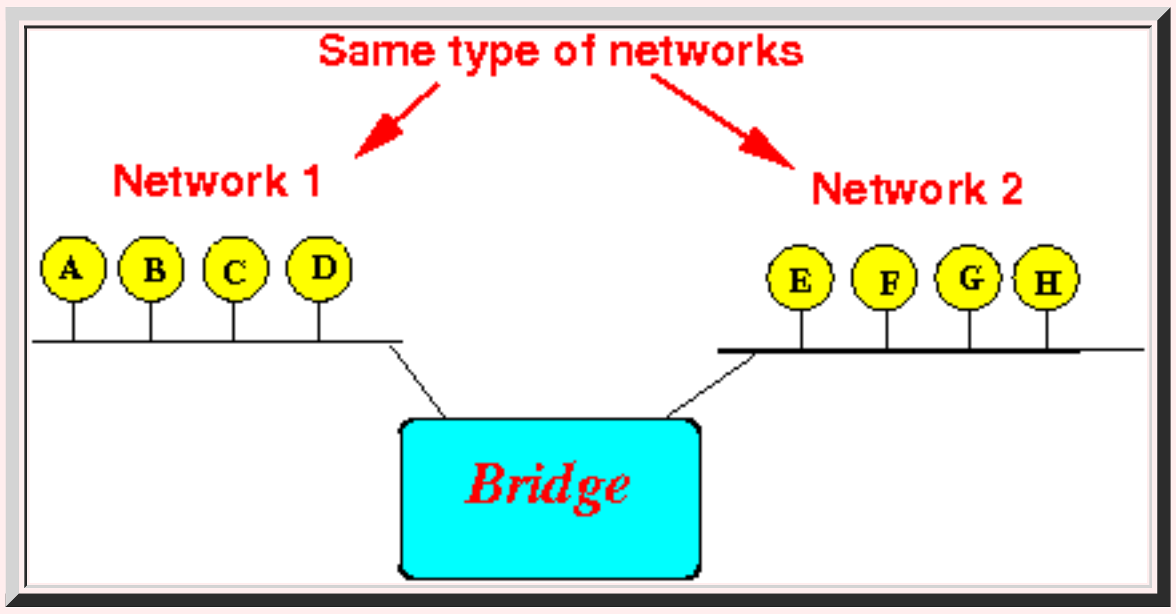
- ***Too many* nodes** can **hear** the ***each other's*** *transmissions*

  Result:

  - **More collisions** !!!

Using Bridges to Interconnect Homogeneous Networks

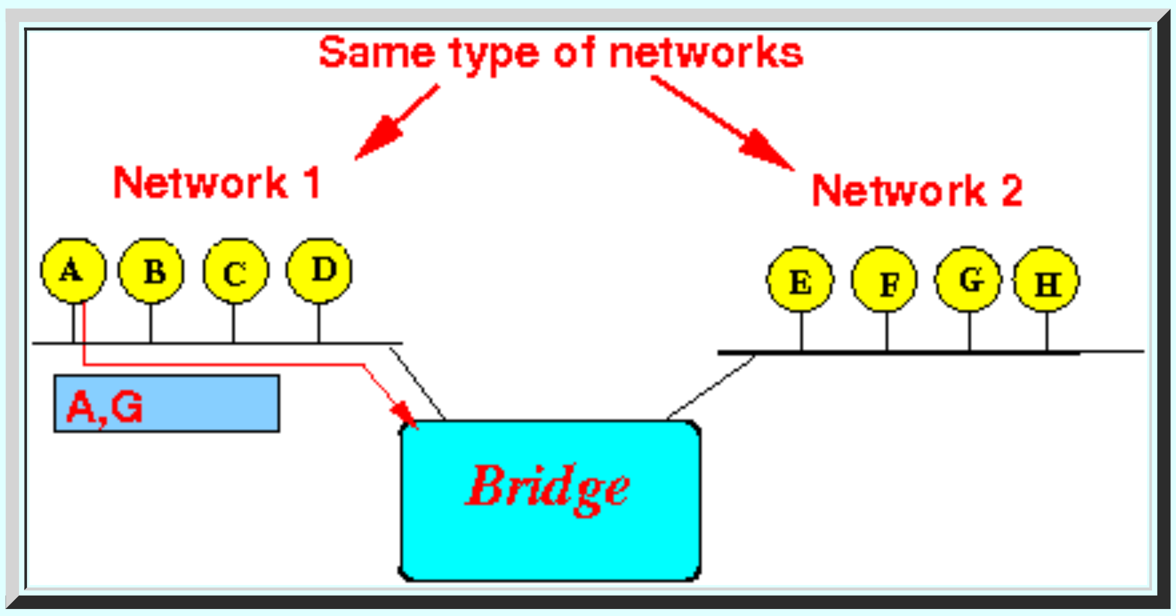
- **What is a Bridge**
  - **Bridge:**
    - **Bridge** (in Networking) = a **device** that **connects *two* networks** that uses the ***same* network protocol**



I found this **correct definition online**:

> DEFINITION
>
> **bridge**
>
> Share
>
> In telecommunication networks, a bridge is a product that connects a local area network (LAN) to another local area network that uses the same protocol (for example, Ethernet or token ring). You can envision a bridge as being a device that decides whether a message from you to someone else is going to the local area network in your building or to someone on the local area network in the building across the street. A bridge examines each message on a LAN, "passing" those known to be within the same LAN, and forwarding those known to be on the other interconnected LAN (or LANs).

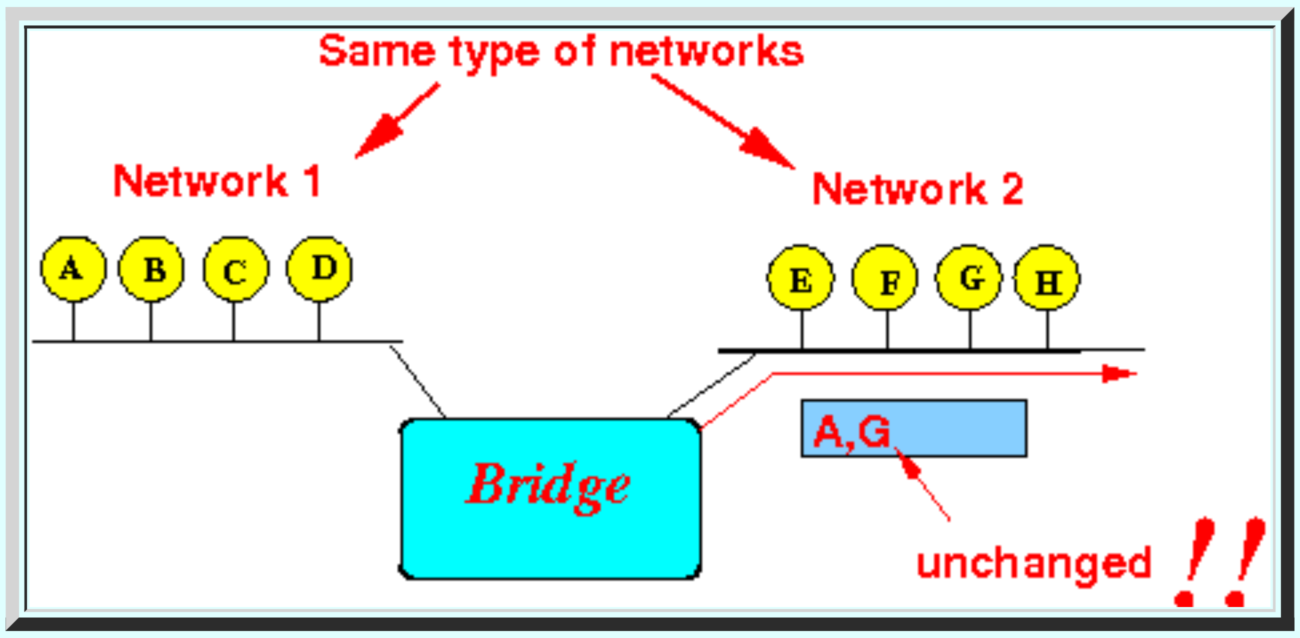
  - **Capabilities** of a **bridge**:
    - **Receive** a **frame** in the **network *protocol***



The **bridge** can **read (= interpret)** the **protocol fields** in thde **received frame**:

- The **bridge** can **read** the **source address** in the **frame**
- The **bridge** can **read** the **destination address** in the **frame**

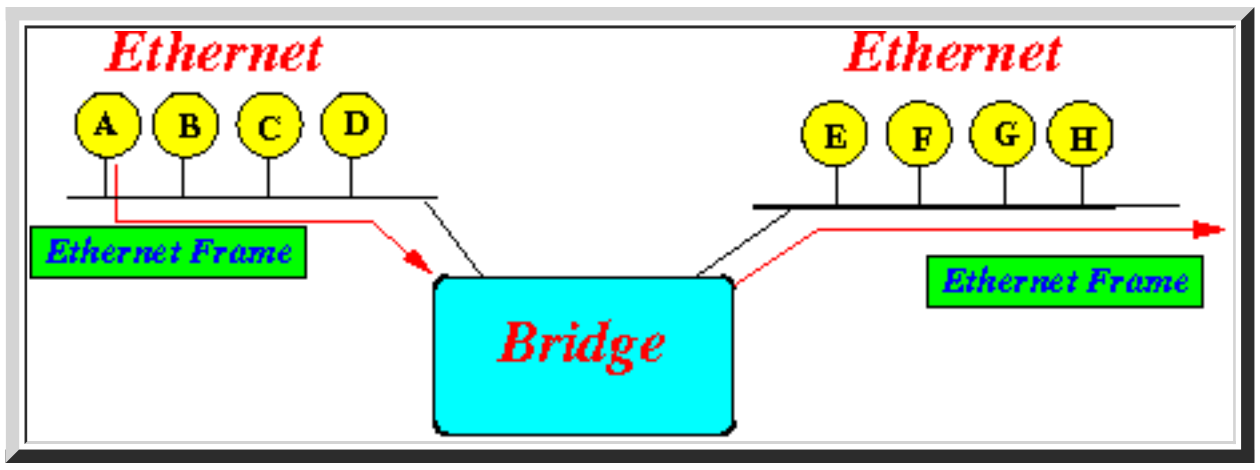
- ***Re-transmit*** a ***received*** **frame** ***unchanged***



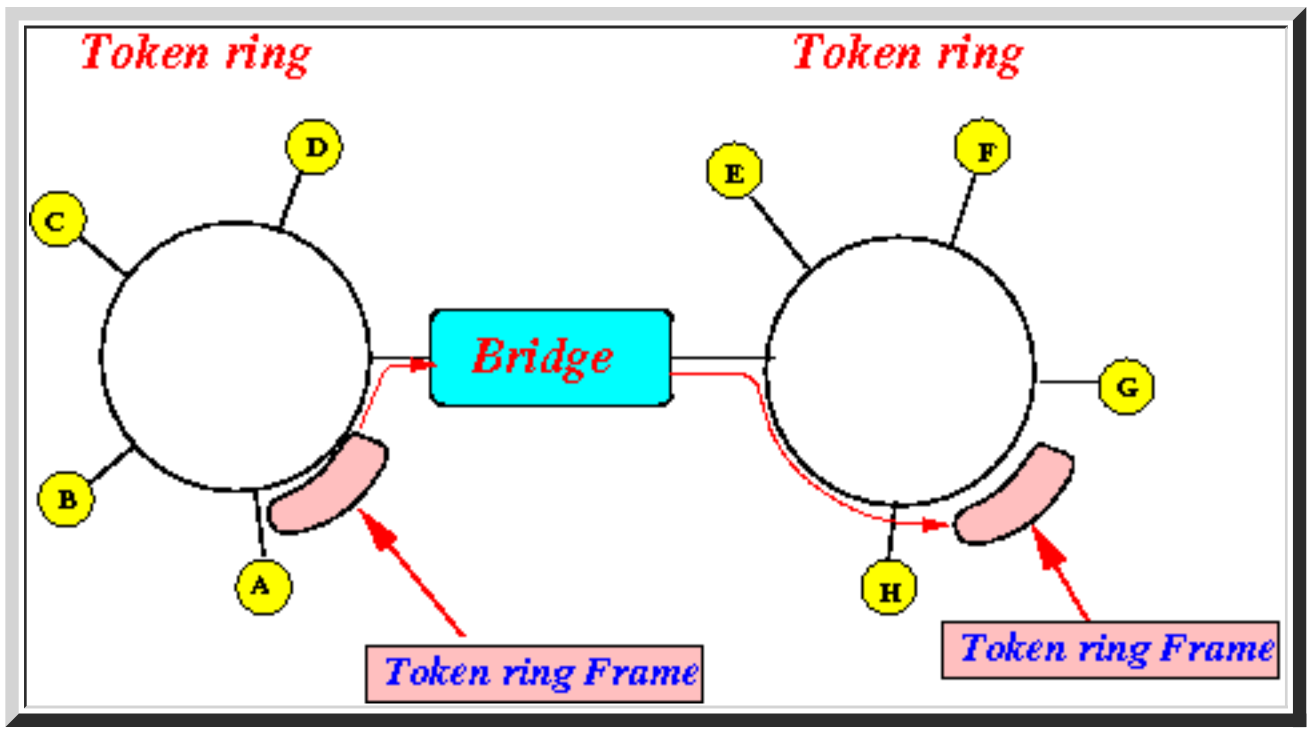
In particular:

- A **bridge** can ***not*** **translate** a **frame** from ***one*** **network format** into a **frame** in ***another*** **network format** !!!

- **Example: *Ethernet* bridge**

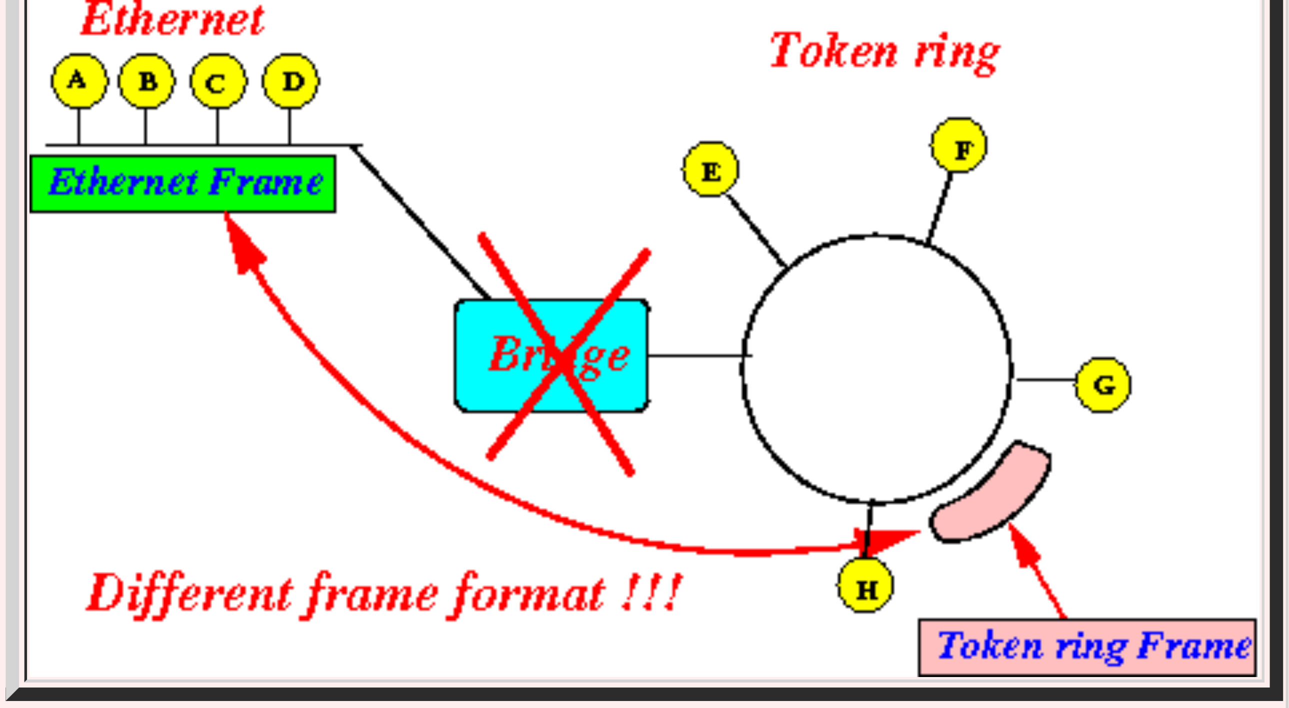


- **Example: *Token ring* bridge**



- **Some devices that are not bridges**
  - **Devices** that are ***not*** **(ordinary) bridges**:

1. A **device** that **connects** 2 ***different*** **types** of **networks**:



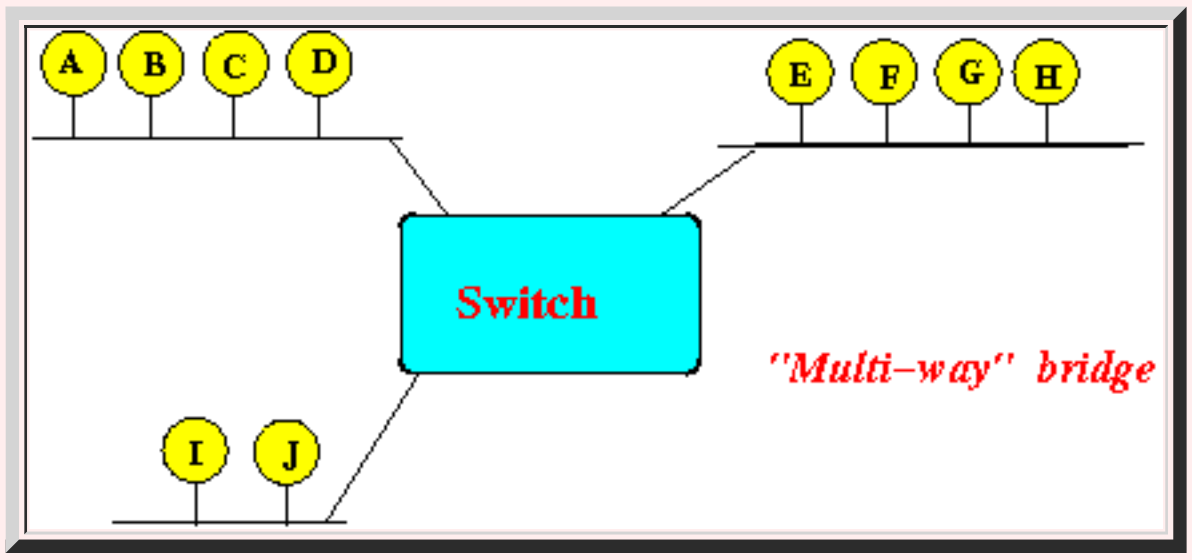
Comment:

- Such a **device** is called:
    - a **(protocol) *translating* bridge**

  See: click here or click here

---

2. A **device** that **connects *more* than 2 networks** (of the ***same* type**):



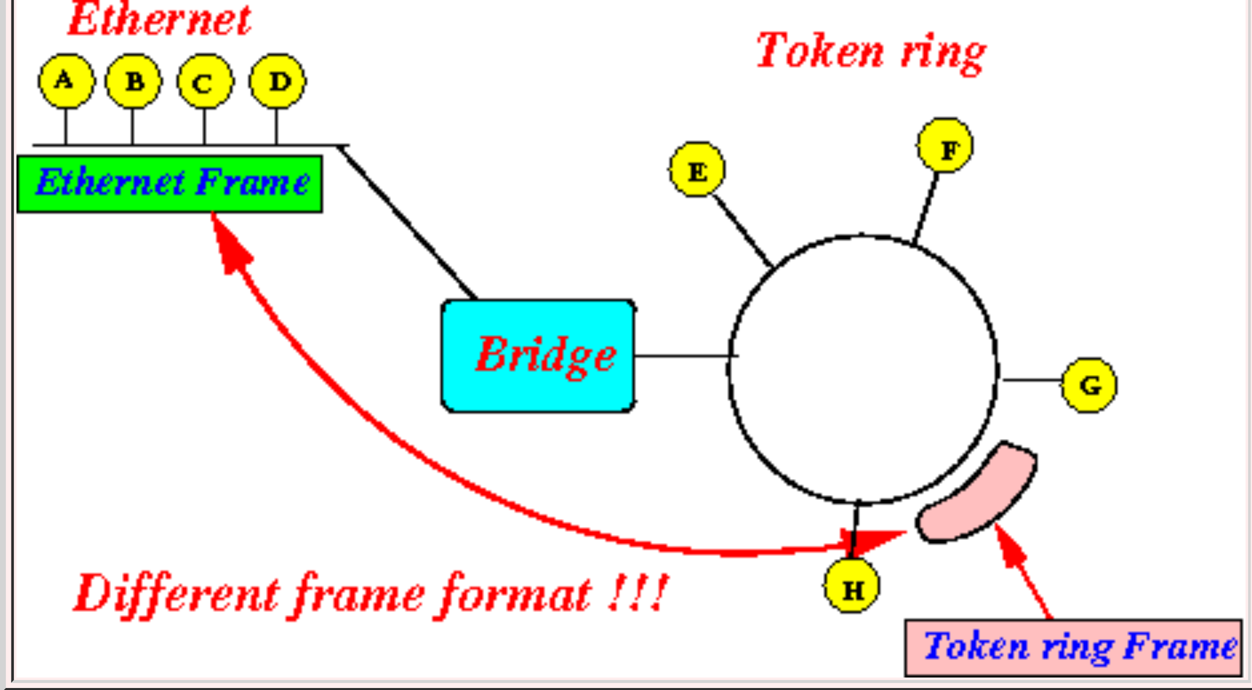
Comment:

- Such a **device** is called:
    - A ***multi-way* bridge**, **or**
    - A ***switch***

- ***Other* definitions of a *Bridge*....**
    - **Fact:**
        - ***Some* book/webpages** defines a **bridge** as:
            - A **network device** that connects ***any* 2 types** of **networks**

        - **Example:**
            - **Wikipedia**: A **network bridge** is a **network device** that connects **multiple network segments**.

              According to ***this* definition**, a ***translating* bridge** is **also** a **bridge**:



- In ***this* course**:
  - **Bridge** does ***not*** perform **protocol translation** !!!

- In ***this* course**:
  - **Devices** that **connect *different* types** of **networks** are:
    - ***Translating* bridges**
      - **Translating bridges send/receive MAC layer frames**
      - **Translating bridges** can **convert** one **frame format** into ***another* frame format**
    - ***Routers***
      - **Routers send/receive *Network* layer packets**
      - **Router** uses the **MAC level service** to **transmit *packets*** carried inside **MAC level frame**

      From Wikipedia:

      - A **router** is connected to two or more data lines from ***different* networks**.

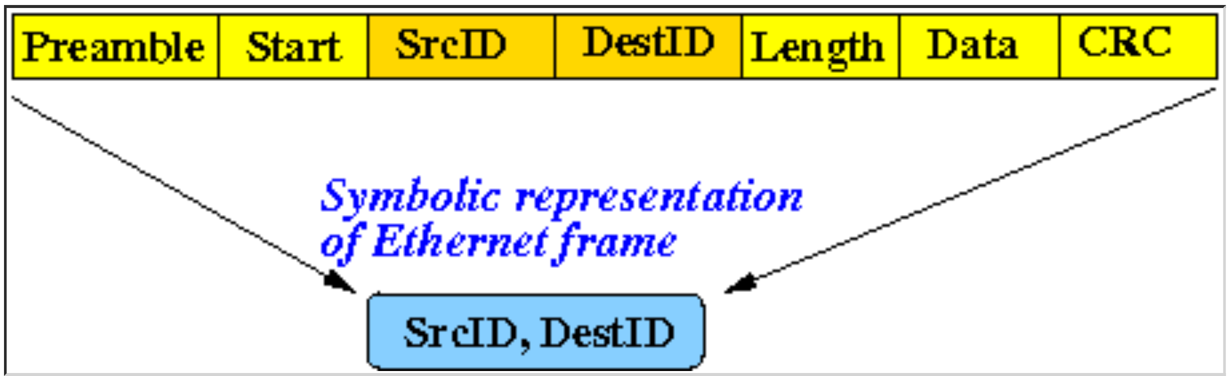
      See: click here

Ethernet Bridging

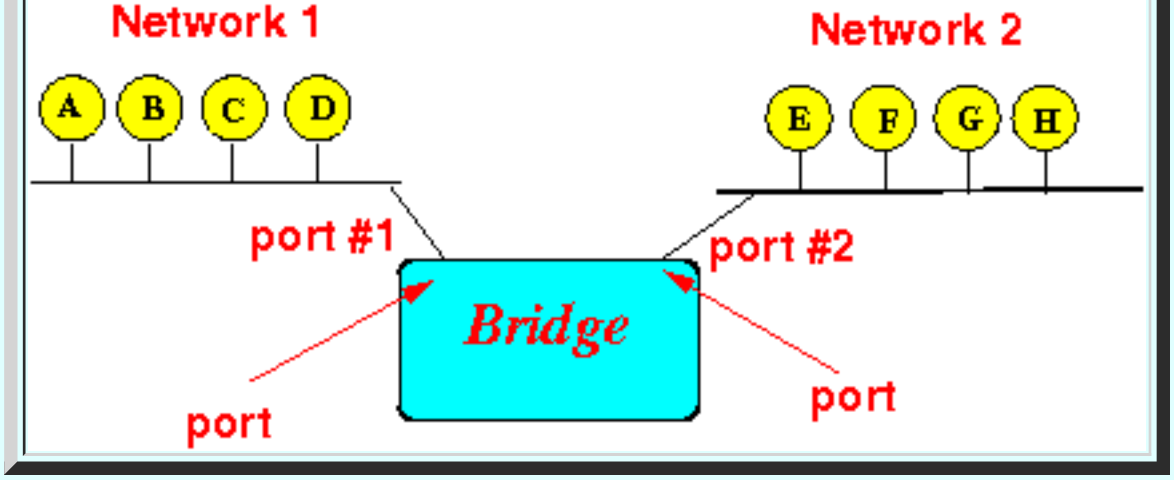
- **Introduction**
  - **Recall** the **Ethernet frame format**:

  - **Fact:**
    - A **bridge** ***only*** **uses**:
      - **SrcID** = the **Ethernet Address** (48 bits) of the **source**
      - **DestID** = the **Ethernet Address** (48 bits) of the **destination**

  - **Therefore**, I will **represent** an **Ethernet frame** as:

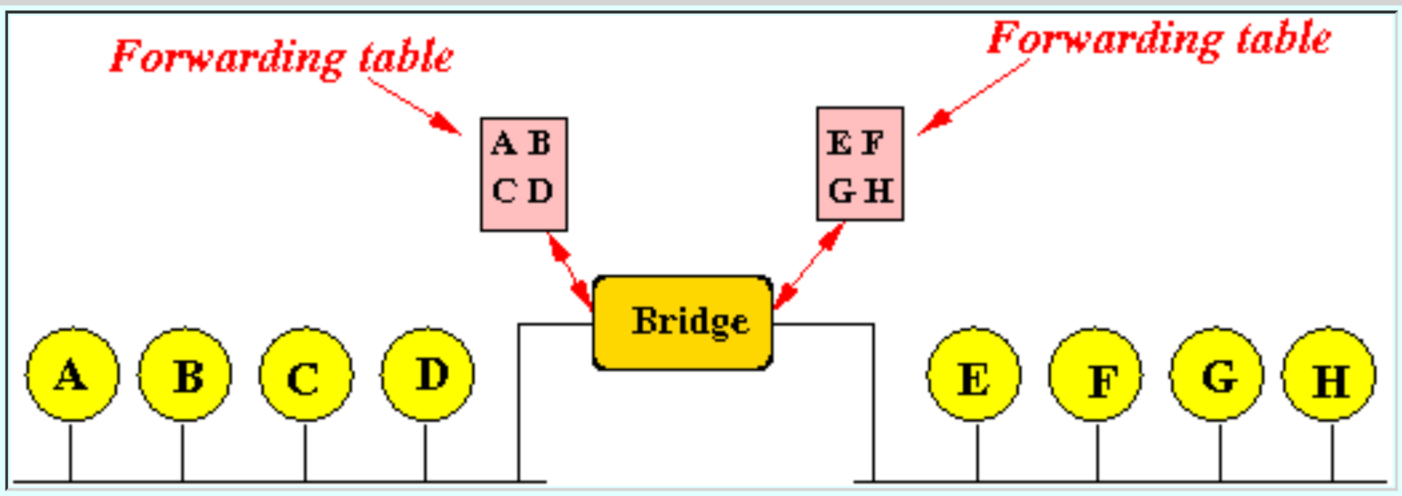


- **Structure of a Bridge**
  - **Structure of a bridge:**
    - A **bridge** has **2 ports**:



Each port is **connected** to a ***different* Ethernet segment**

- ***Each* port** of the **bridge** has an **associated *forwarding* table** :



Forwarding table:

- **Forwarding table** contains the **Ethernet Addresses** of **node** attached to the **Ethernet segment**

Note:

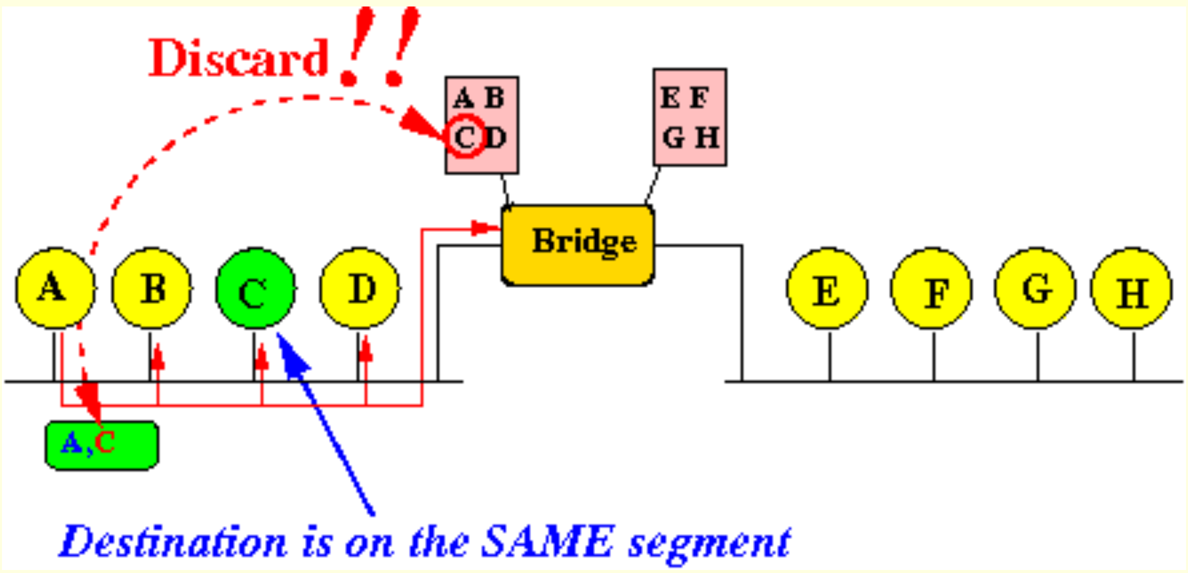
- **Recall** that:
  - **Ethernet** is a ***broadcast* network**
- **Therefere:**
  - A **bridge** can **hear (= receive) messages** transmitted on ***both* Ethernet segment**

- **Prelude: operation of an (ordinary) Ethernet node**
  - **Operation** of an **Ethernet node**:
    - When an **Ethernet node** receives an **Ethernet frame**:

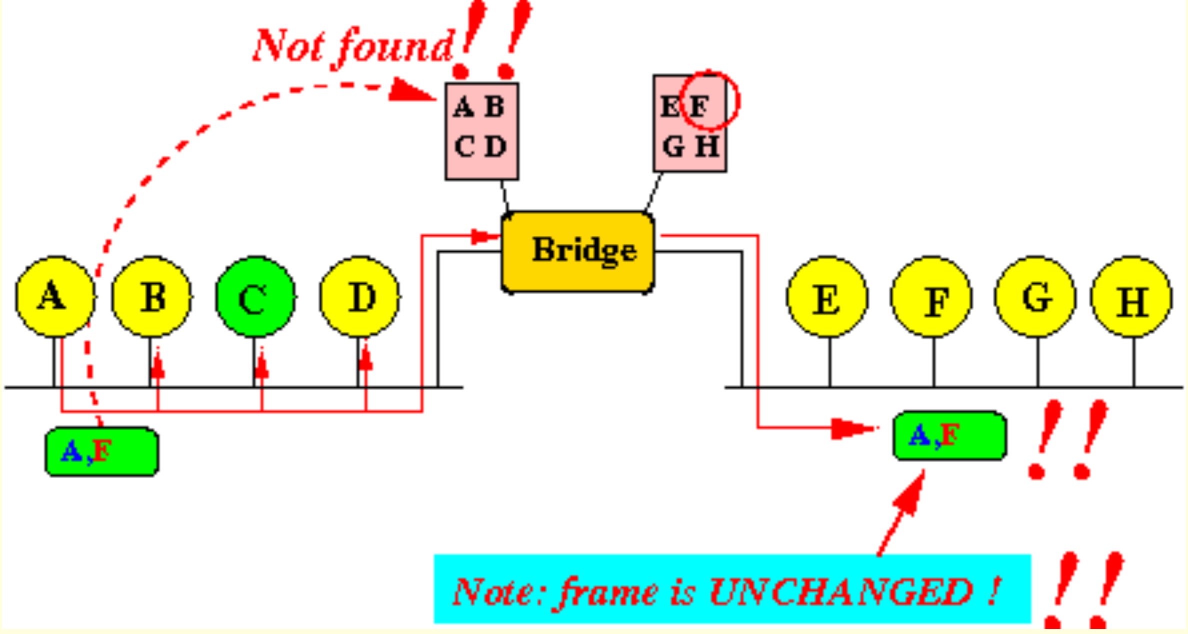
```
if ( destination address == my Ethernet address )
{
   Deliver the data portion in the Ethernet frame
}
else
{
   Discard frame
}
```

- **Operation of a bridge**
  - **Operation** of a **bridge**
    - When a **bridge** receives an **(Ethernet) frame** on a **port P**:

```
if ( Destination address ∈ Forwarding Table(P) )
{
   Discard frame
```




```
}
else
{
   Forward the frame on the other port
```



}

- **Note:**

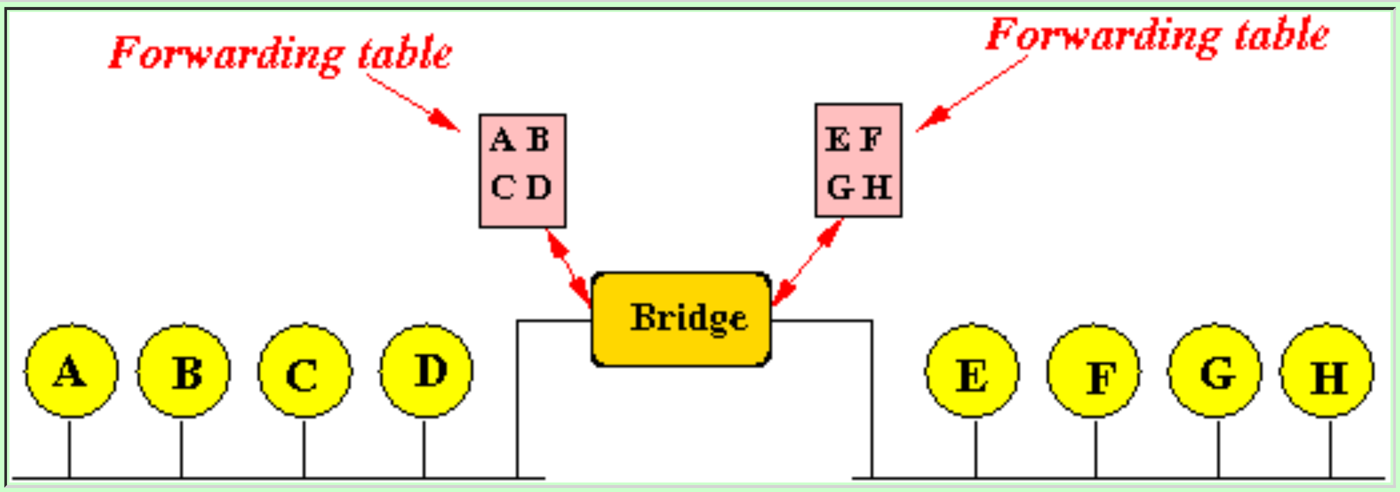
  - The **bridge** does **not** change ***any*** **part** of an **Ethernet frame** !!!

    - ***Later***, you will **learn** about **routers** do need to **change** ***some*** **part** of the **Ethernet frame**

- ***Bridged*** **Ethernet segments constitutes *one* (Ethernet) network**

  - **Fact:**

    - **Multiple** Ethernet segments **connected** by **bridges**:

      is considered as ***one*** **Ethernet network**

- **Adminstrative problem in the bridge operation**
  - ***Administrative* problem** in the **operation** of a **bridge**:
    - A **bridge** needs to know:
      - The **Ethernet Addresses** of the ***all* nodes** in **every Ethernet segment**

      (This **information needed** to ***correctly* forward** a **Ethernet frame**)

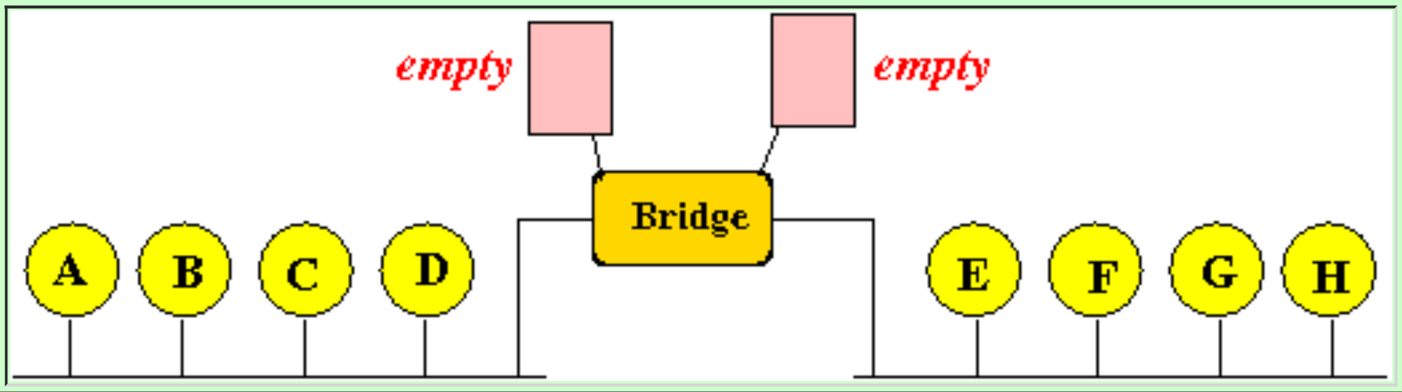
  - ***Early* days:**
    - **Once upon a time**:
      - **network administrators** had to **enter** the **forwarding tables** of a **bridge manually** !!!

  - **Transparent bridging:**
    - **Nowadays**, a **bridge** will ***learn*** the **Ethernet address**
      - This **technique** is known as ***transparent* bridging**

Transparent Bridging

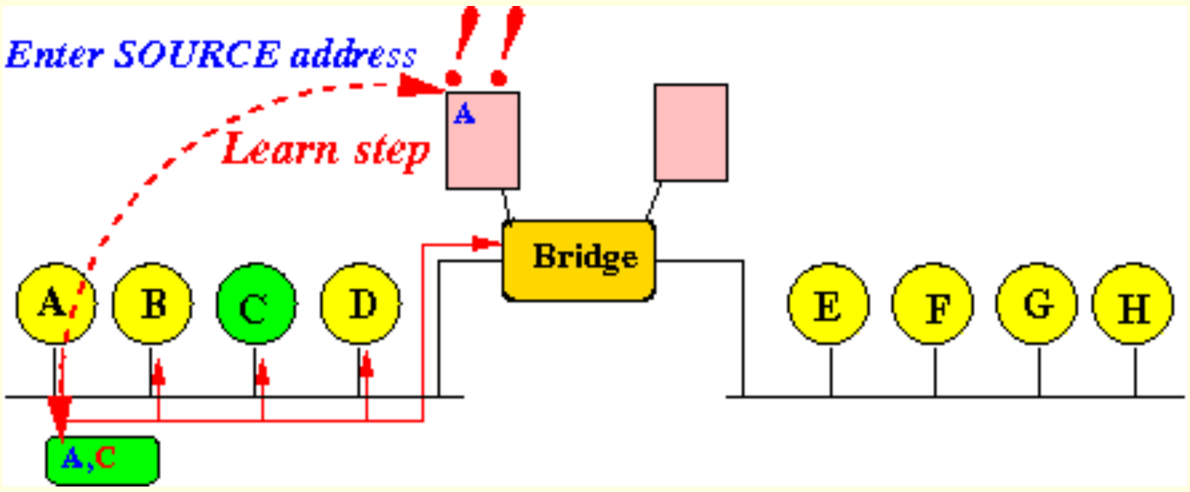
- **Transparent Bridging**
  - **Transparent bridging:**
    - **Bridges** can program the **forwarding tables** by **"learning"** the **Ethernet addresses** on an **Ethernet segment**.

  - **Bridge start up:**
    - When the **bridge starts up (reboot)**, the **forwarding tables** are ***empty***:

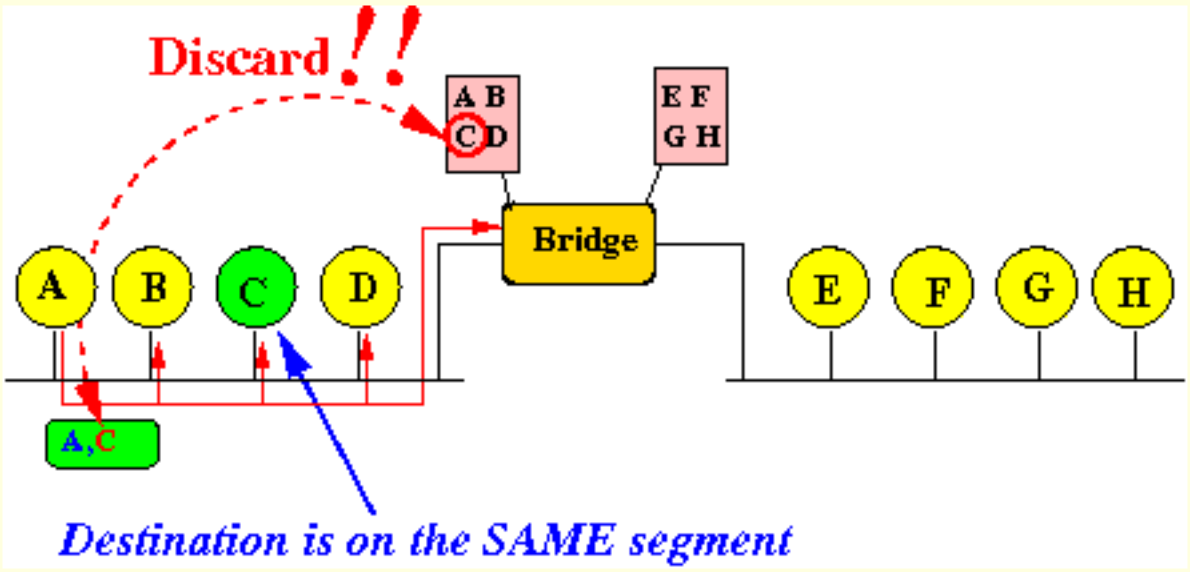


  - **Bridge operation:**
    - When the **bridge** ***receives*** an **Ethernet frame** on the **port** `P`:

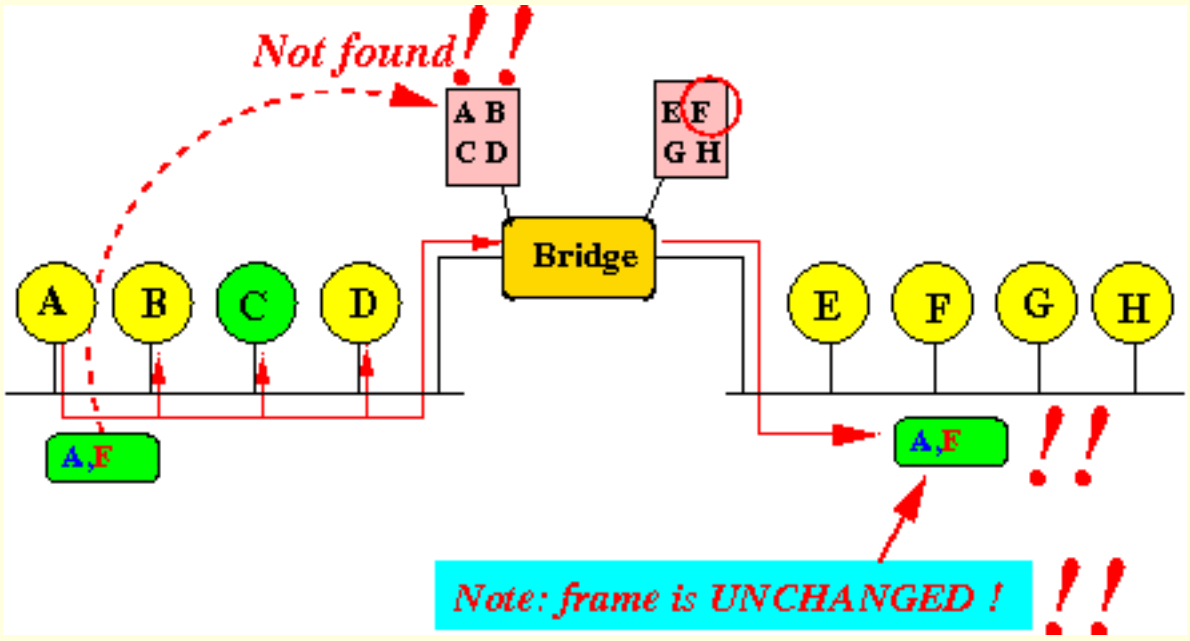
```
/* ==================================
   Learn step
   ================================== */
if ( Source Addr ∉ Forwarding Table(P) )
   Enter Source Addr in Forwarding Table(P);       // Learn step !!!
```




```
/* ==================================
   Forwarding step
   ================================== */
 if ( Destination address ∈ Forwarding Table(P) )
 {
   Discard frame
```



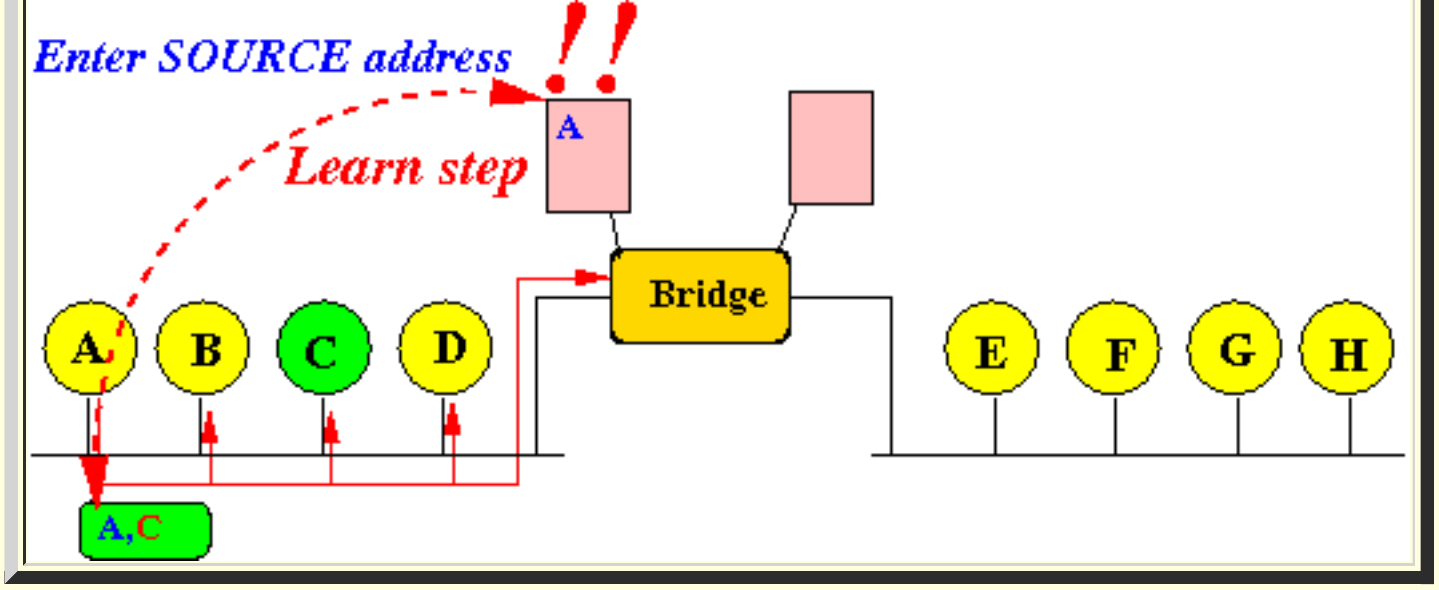

```
}
else
{
   Forward the frame on the other port
```



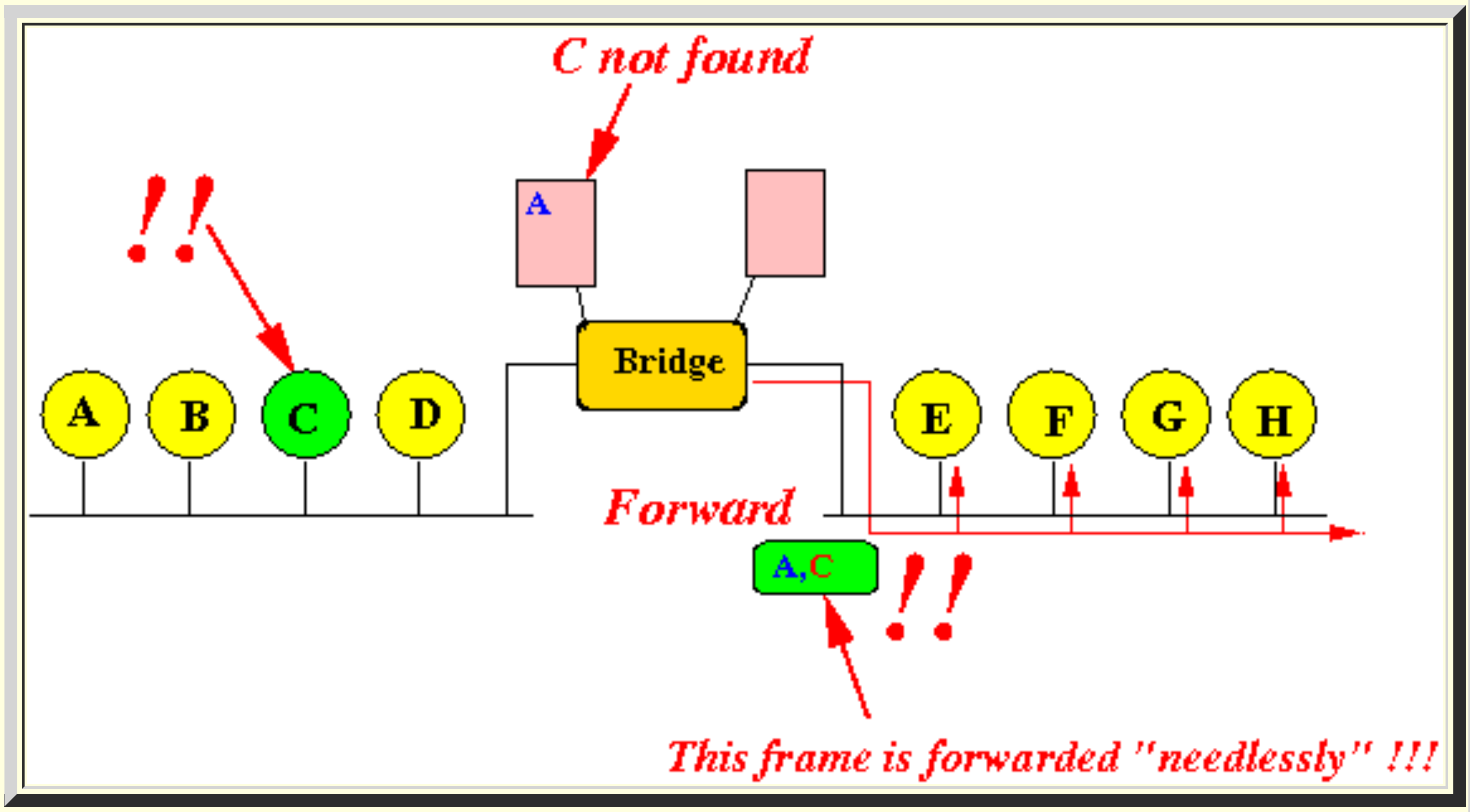

```
}
```

- **Transparent bridging has initial inefficiency**
  - **Fact:**
    - **During** the ***initial* learning phase**:
      - **Transparent bridging** may **forward** some **frames *unnecessarily***

  - **Example:**
    - **Node *A*** trnasmits a **frame** to **node *C*** on the ***same* Ethernet segment**:

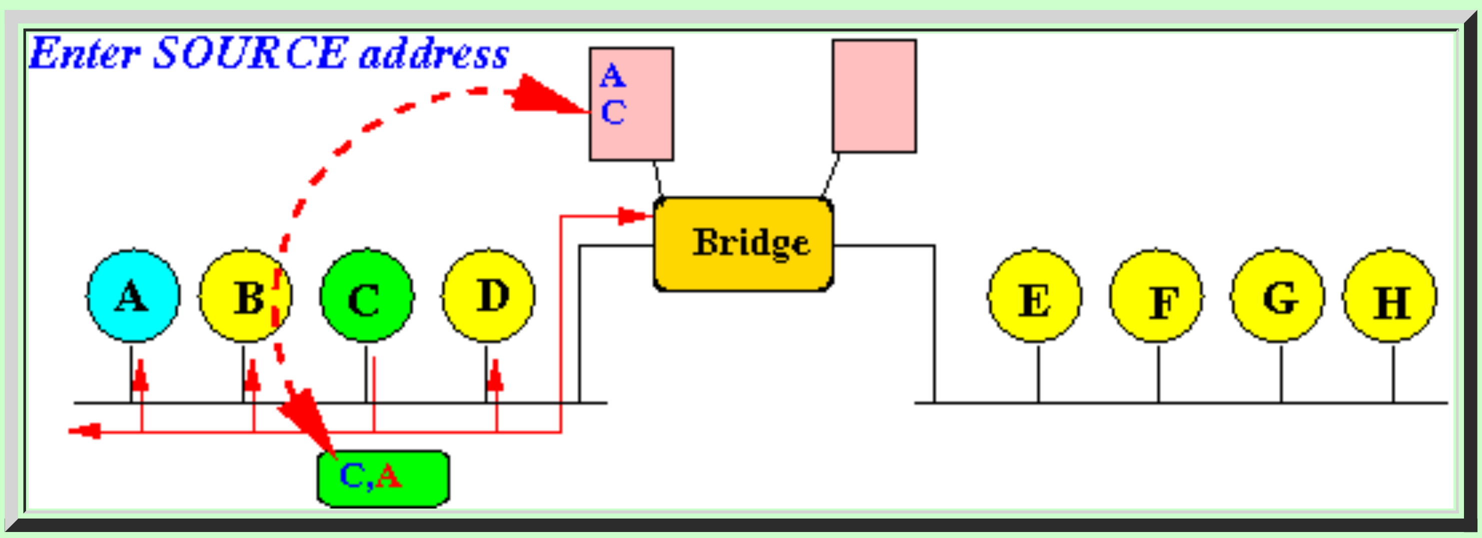


- **Because** C is ***not* found** in the **Forwarding table**, the **bridge** will **forward** the **frame** (needlessly):



- The ***inefficiency*** will **eventually cease**:

  - Suppose **node *C* replies** to **node *A*'s** message:

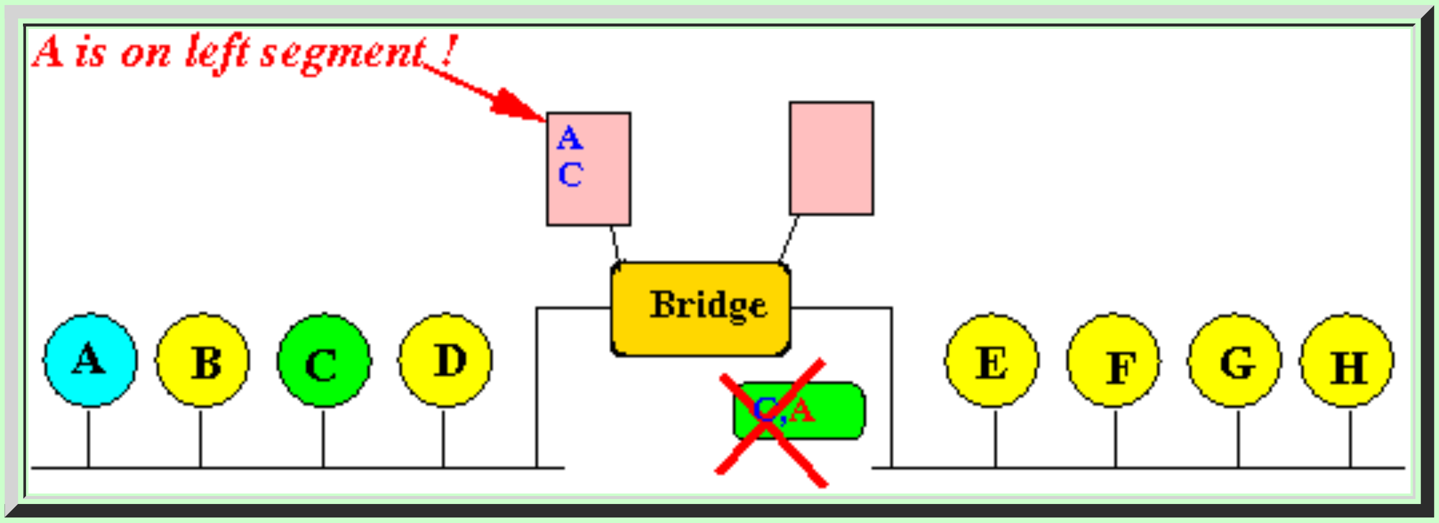


The **bridge** will ***learn*** that **node *C*** is the the ***left* segment** !!

- **From now on:**

  - the **bridge** will ***not* forward** the **frame** destined for **nodes *A* and *C*** to the ***other* Ethernet segment** !!!

Graphically:

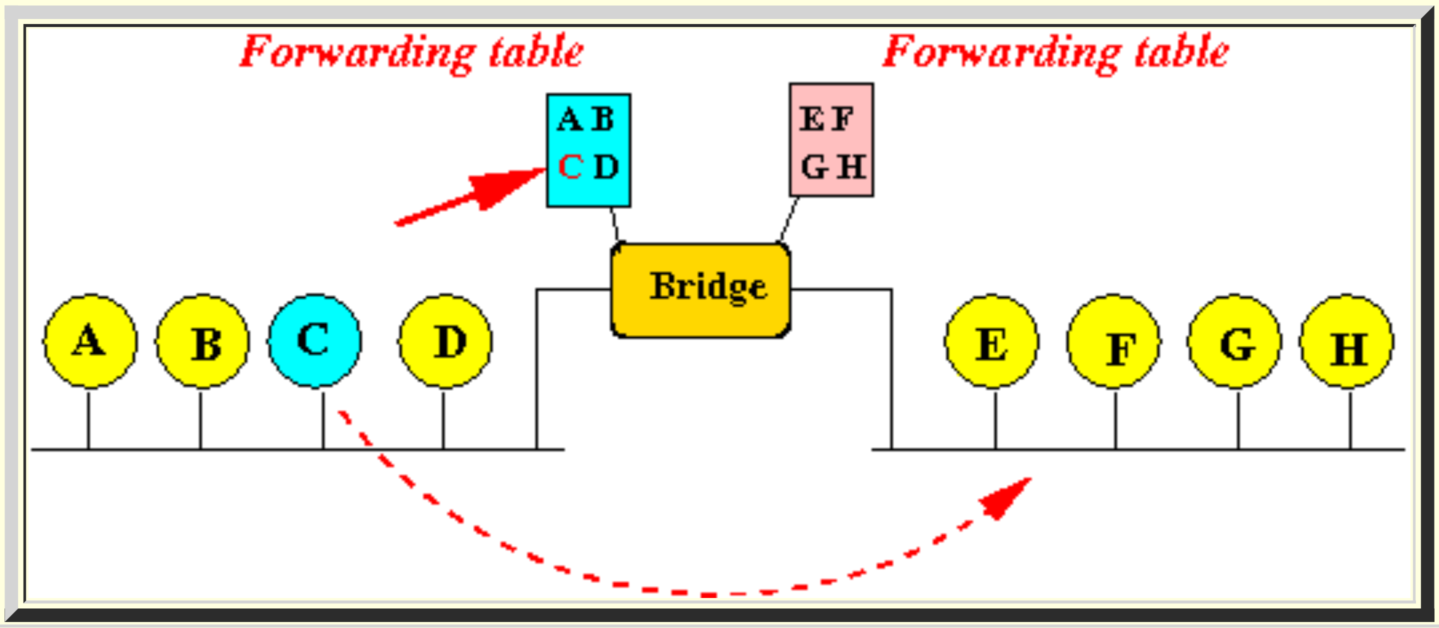


Because ***now*** the **bridge** knows ***where*** the **nodes *A* and *C*** are **located** !!!

The node relocation problem and its solution

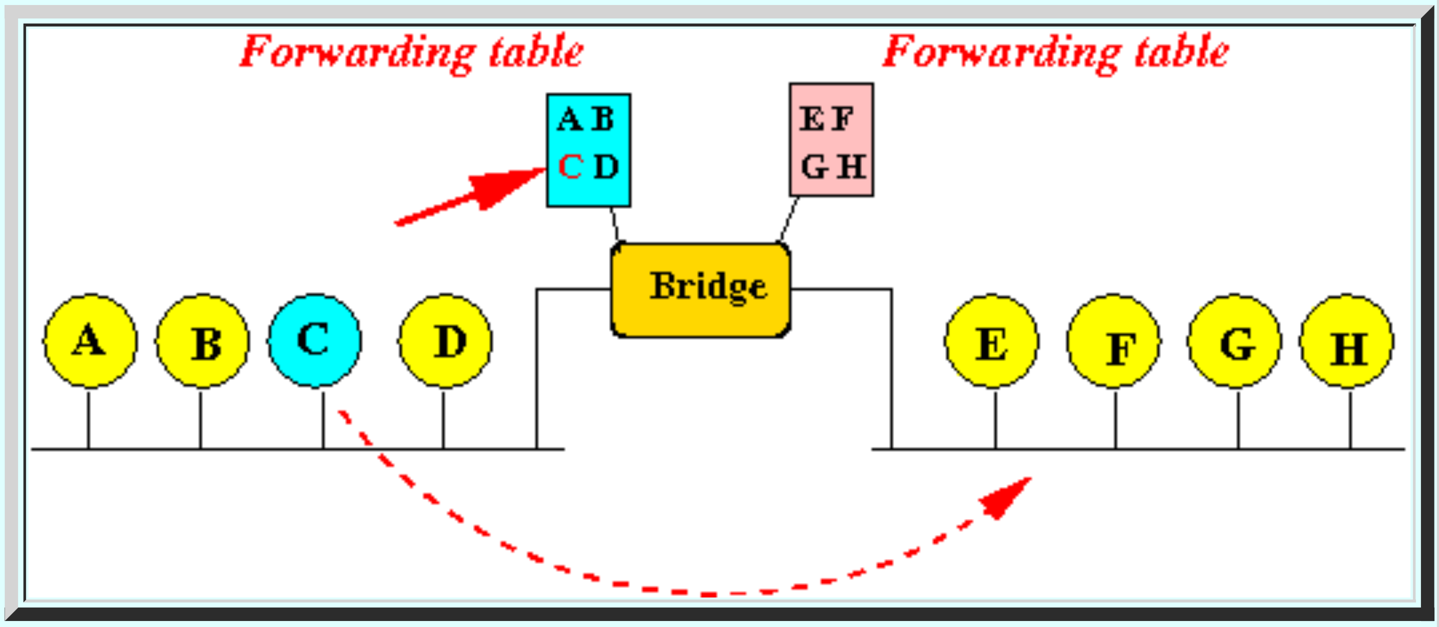
- **Problem: *Relocating* nodes in Transparent Bridging**
  - **Scenario:**
    - We want to ***re-locate* (move)** a **node (e.g.: *C*)** from **one Ethernet segment** to ***another* Ethernet segment**:

      Example:

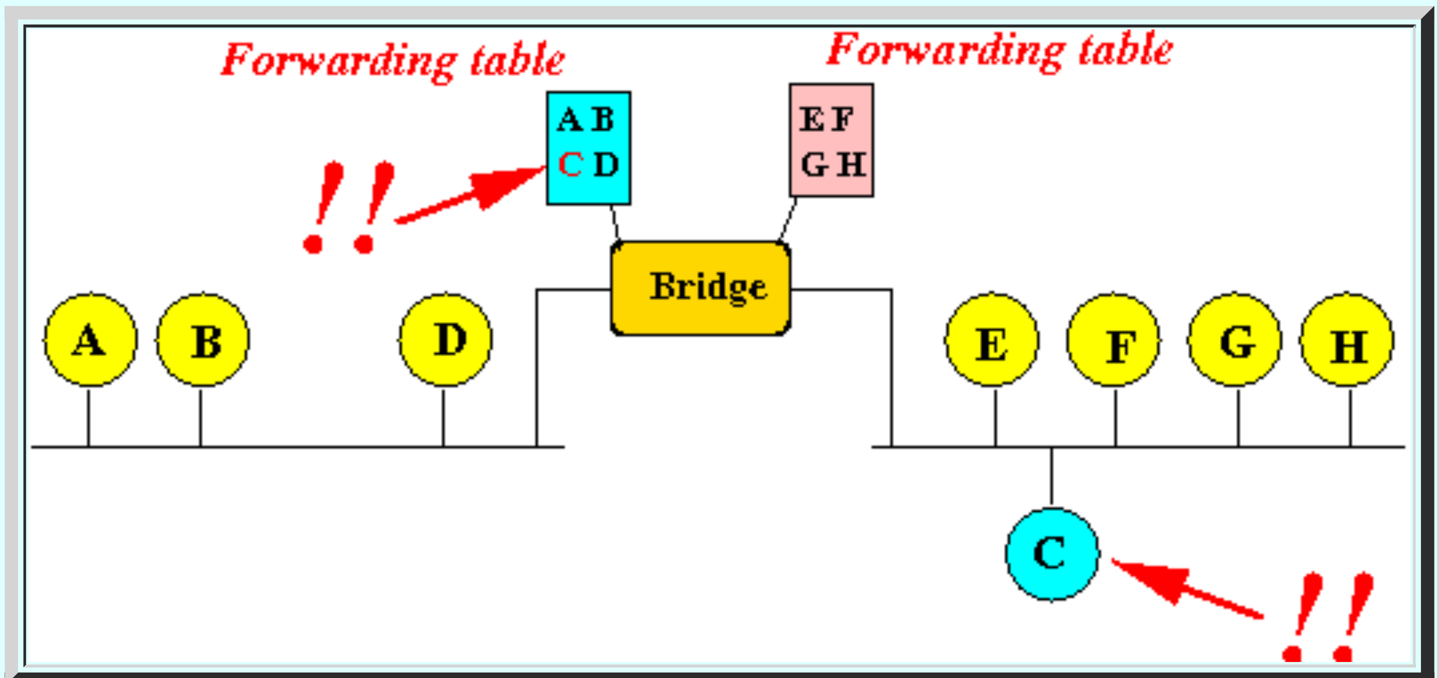
      

    Result:

    - **Before** moving **node *C***:

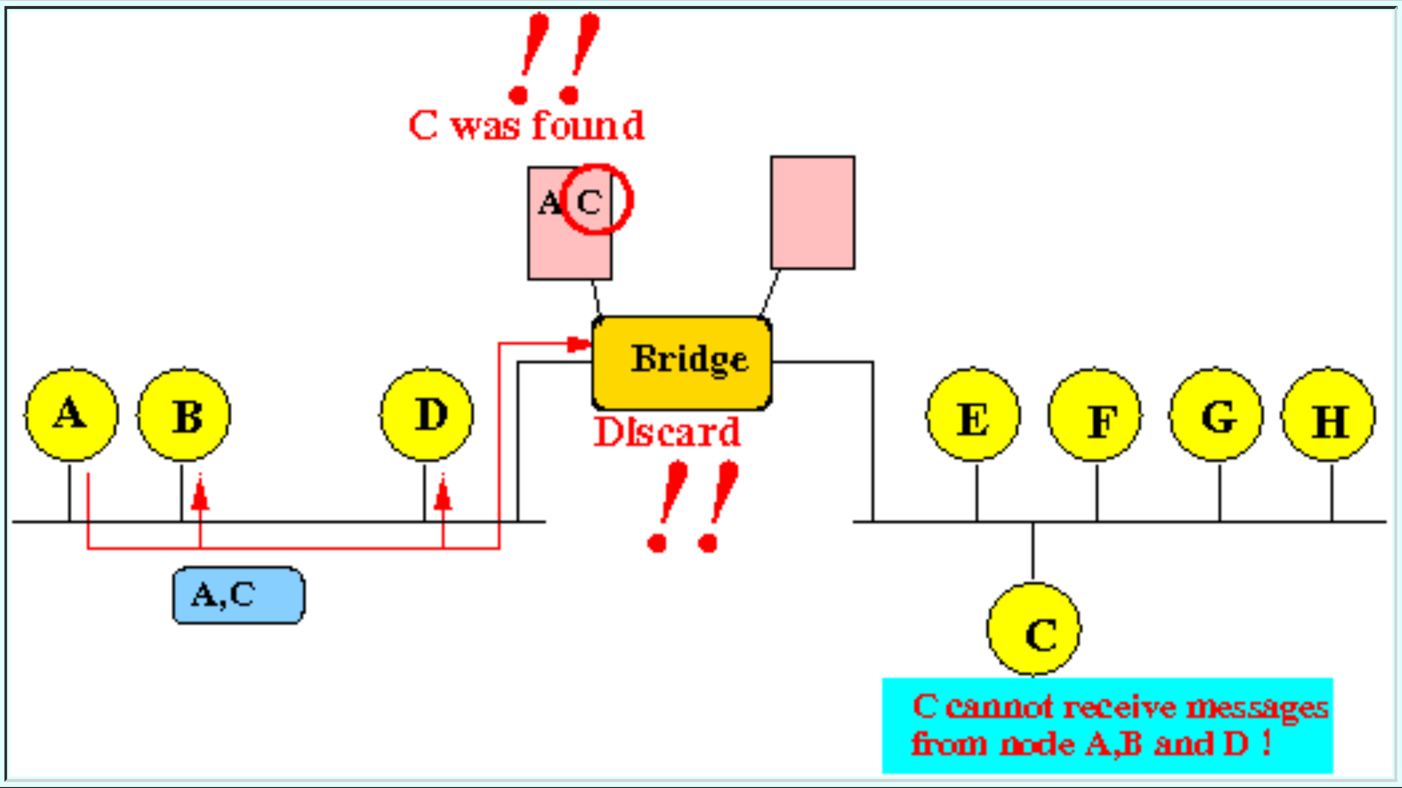
    - **After** moving **node *C***:

- **Consequence:**

  - **Node *C*** can **not** receive **messages** from **some nodes**

  Example: if **node *A*** transmits a **frame** to **node *C***

- The **cause** of the **relocation problem**:

  - The **entry** for **node *C*** is:

    - **still associated** with the ***old* port**

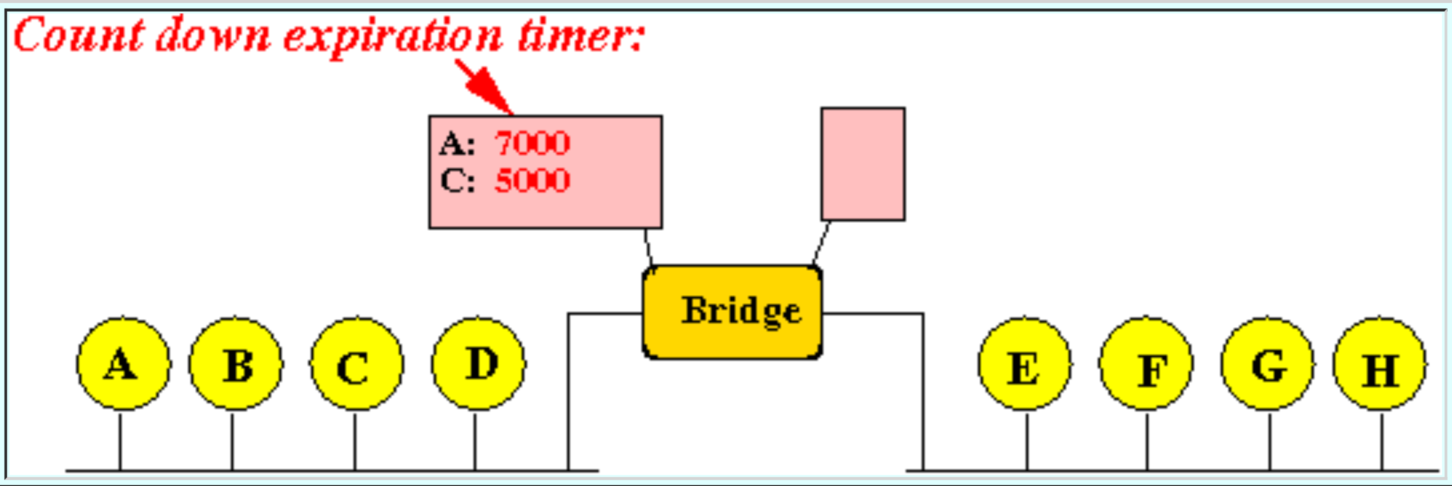
  I.e.:

    - The **entry** for **node *C*** is in the ***wrong* forwarding table** !!!

- **Solving the node relocation problem**

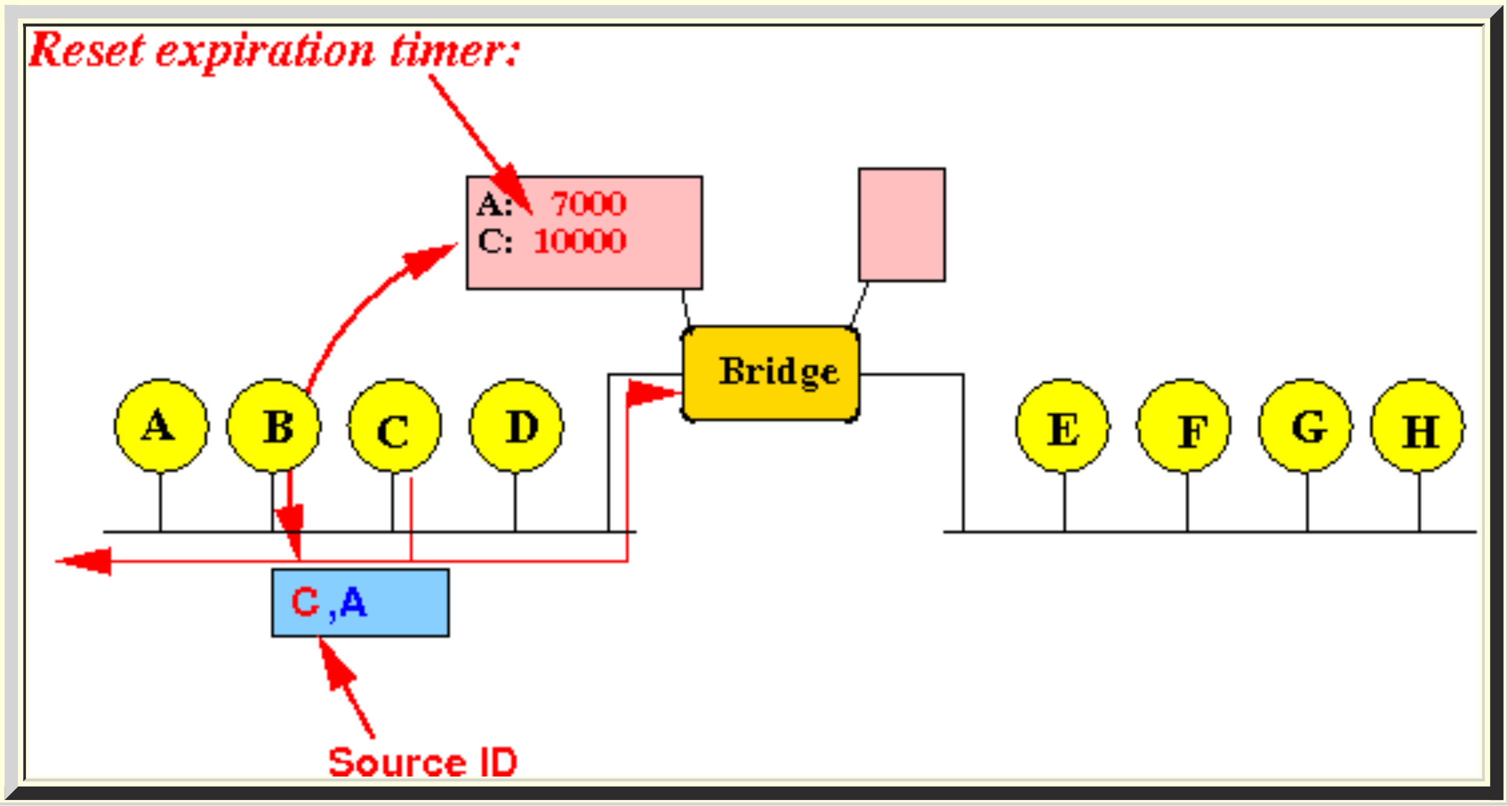
  - **Solution:** use ***time out*** to **remove *inactive* entries** !!!

    - **Entries** in the **forwarding table** in **Transparent bridging** has a ***expiration* time**:

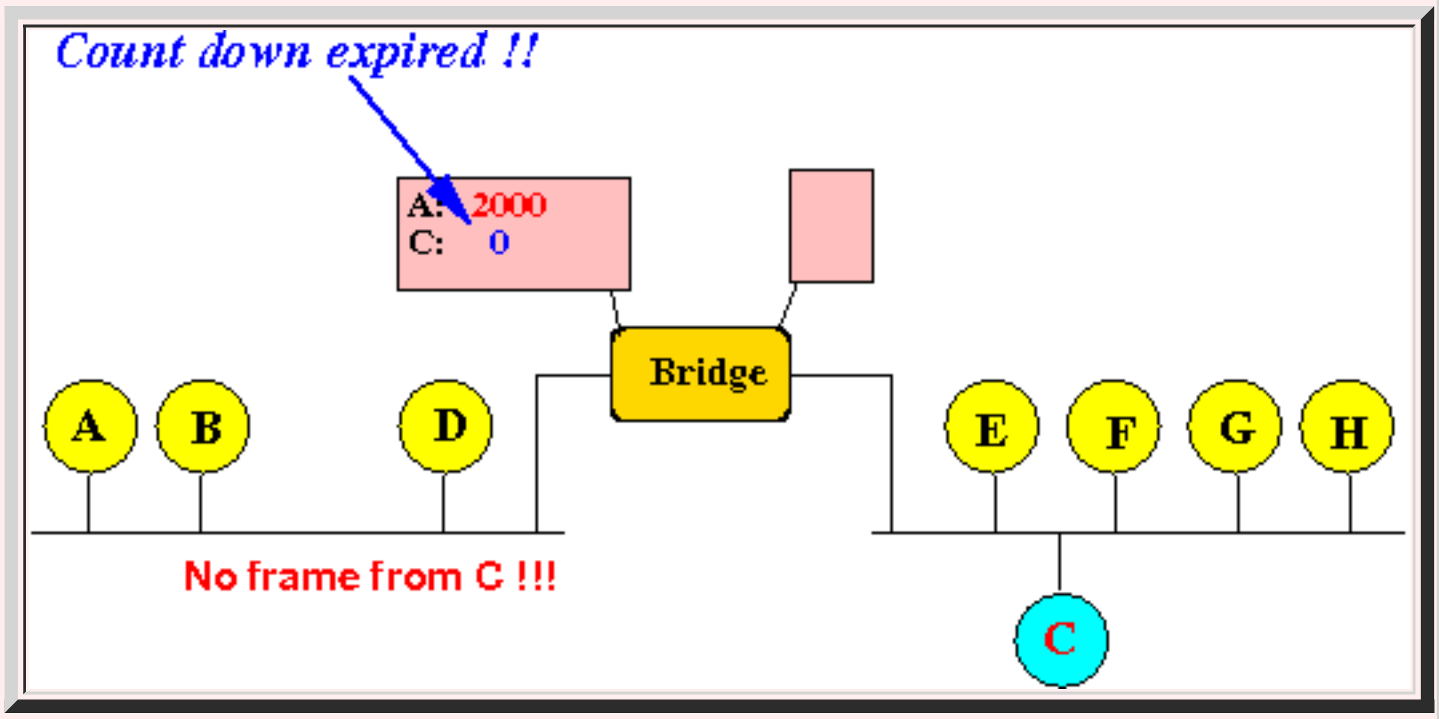
- Timer ***refresh***:

  - When the **bridge** **receives** a **transmission**, from **source ID**, the **bridge** must ***reset*** the **expiration timer** (= **"renewal"**)



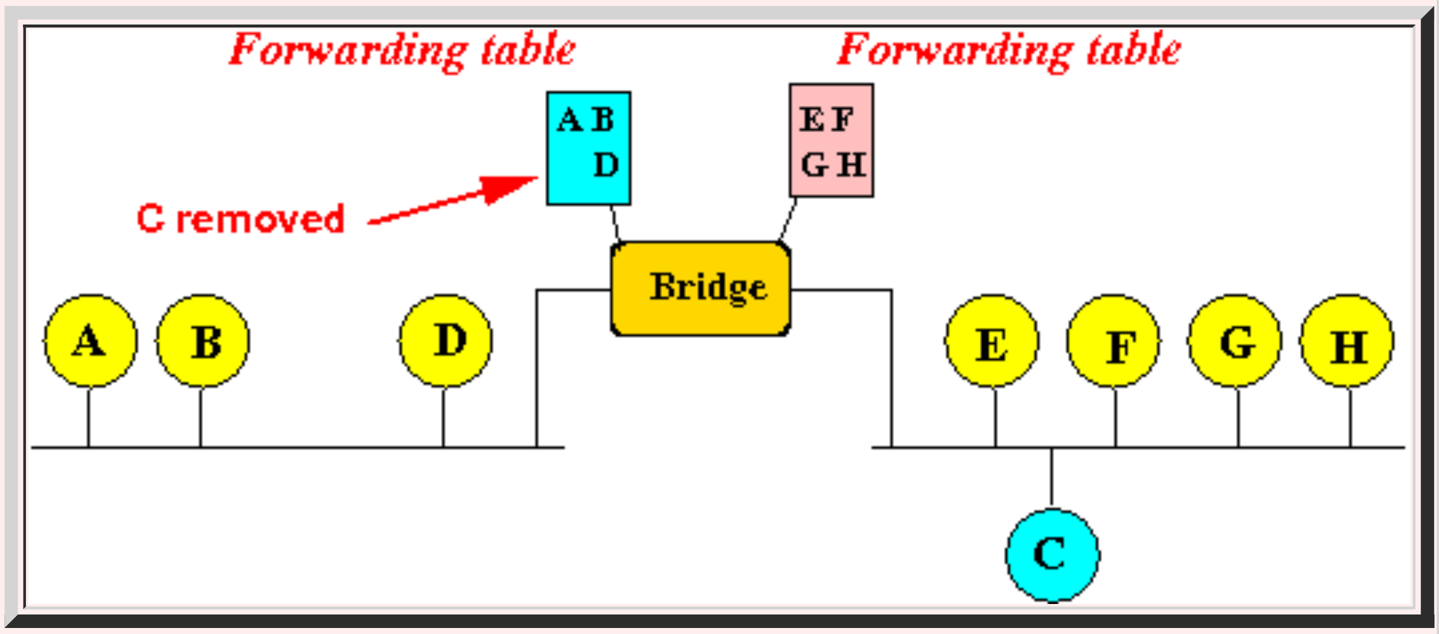
- ***Expiration***:

  - If a **node** has ***not*** transmitted for a **long time** on a **segment**:



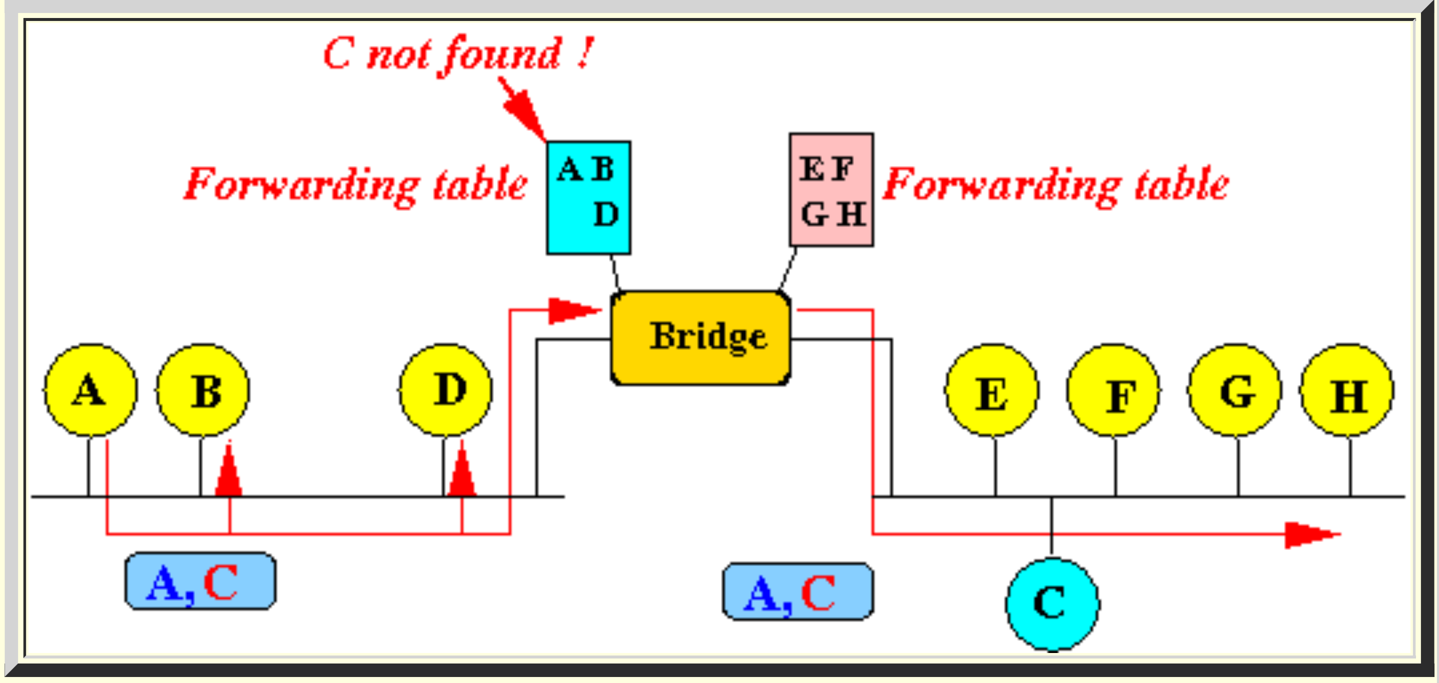
    The **timer** will **expire** !!!

  - The ***expired*** **entry** will be **removed**:

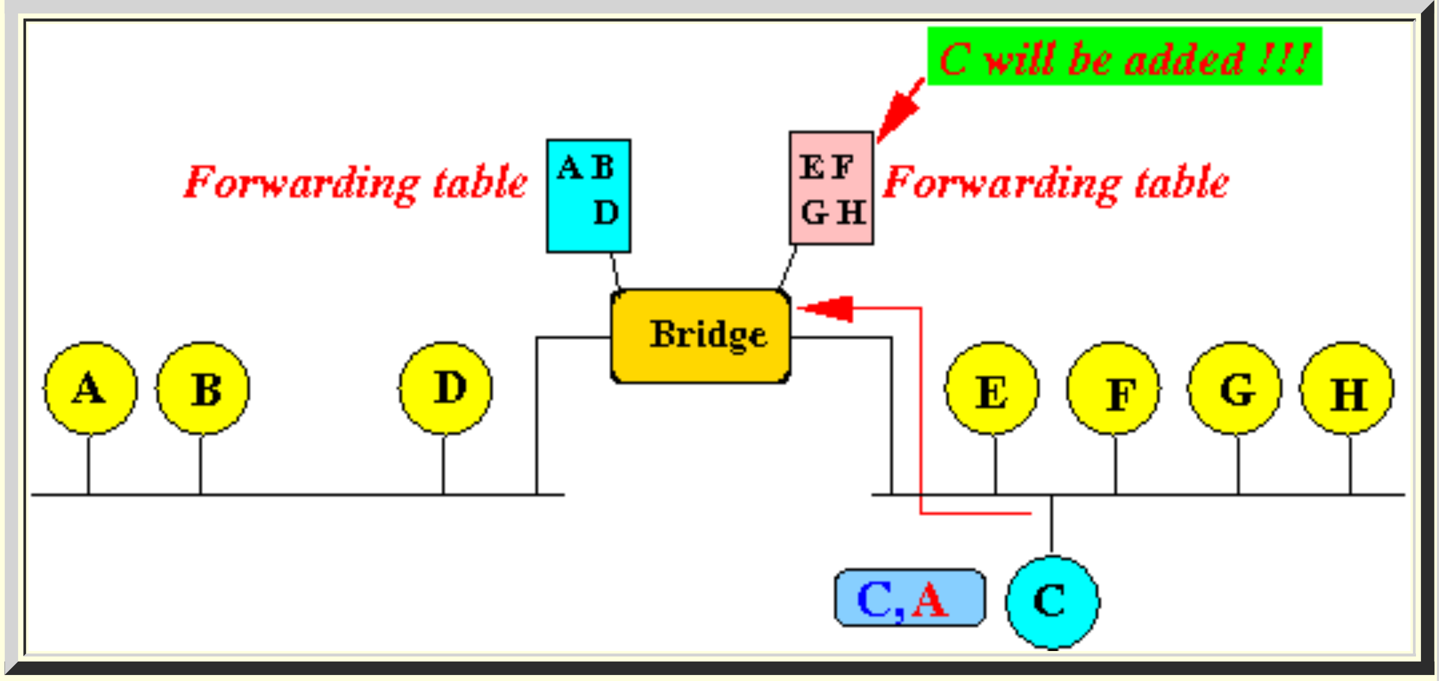


- **Result:**

  - The **bridge** will **now forward frame** destined for **node *C***:

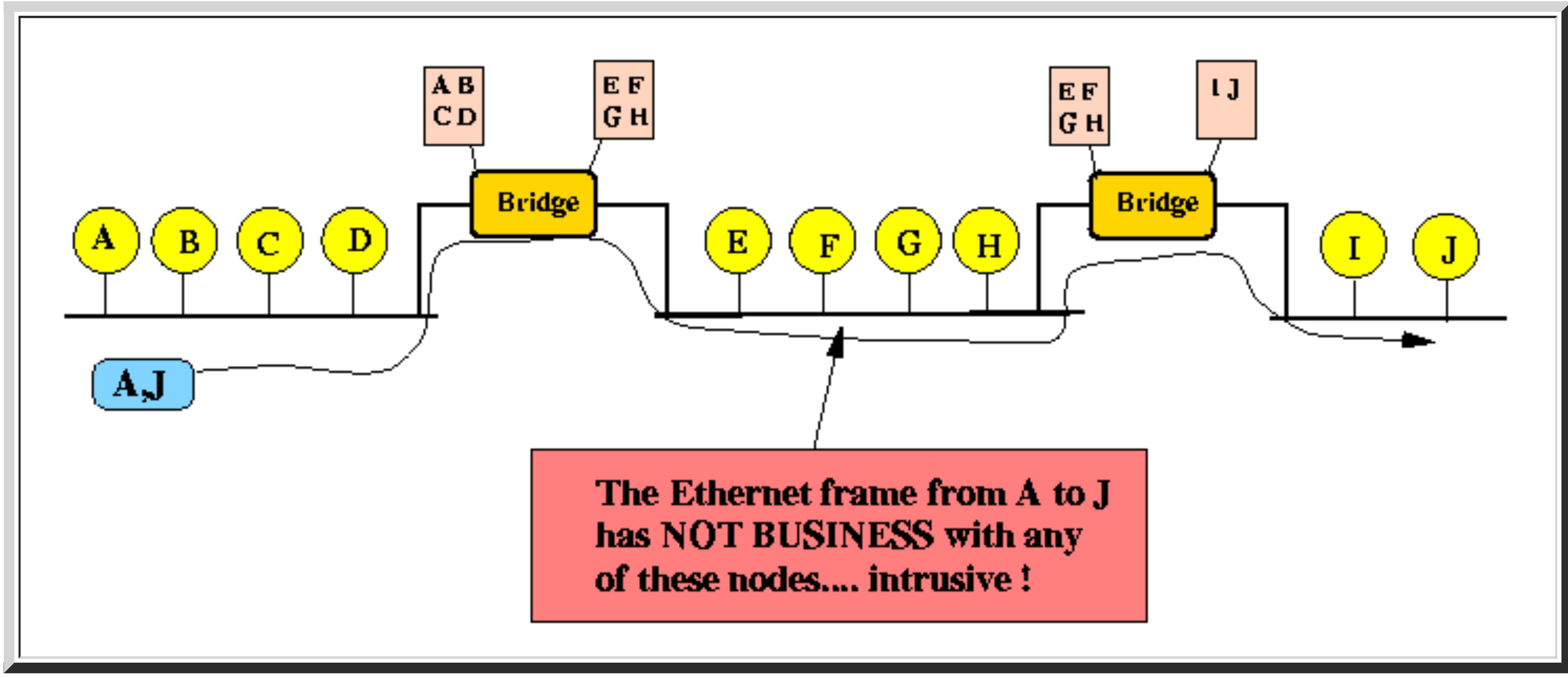


  - When **node *C* replies**, the **bridge** will **learn** the **location** of ***C***:



Connecting more Ethernet segments

- **Connecting more than 2 Ethernet segments:**
  - **Bridges** are **ineffective** to **connect more than 2** Ethernet segments:



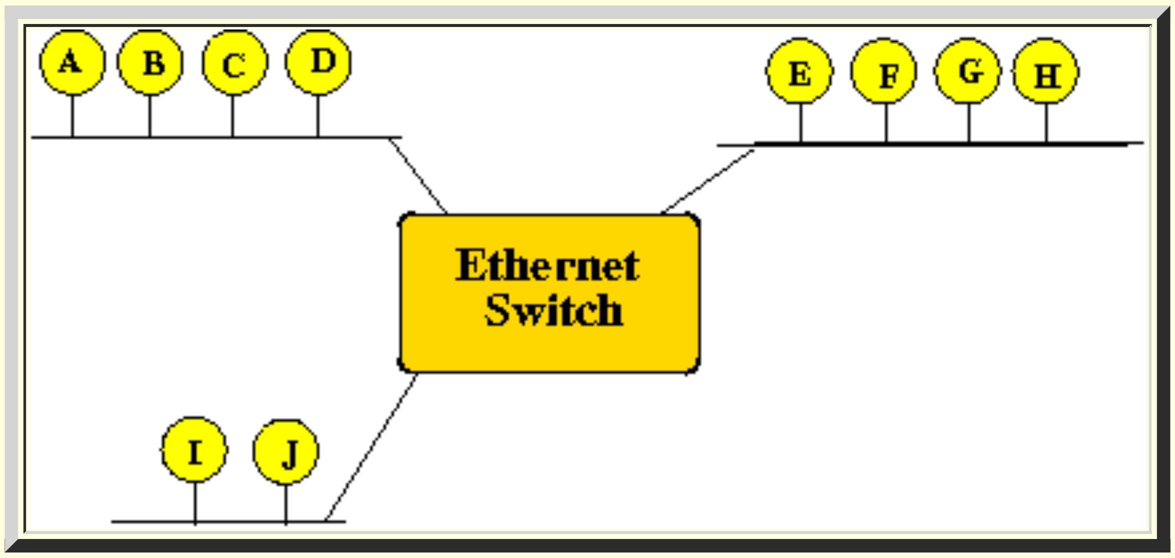
Because:

- **Many nodes** will be **inconvenient (= hear)** the **transmissions**
  - We will have ***scalability* problems**

- **Solution:**
  - Use ***multiway* bridges** or:
    - **Ethernet *Switches***

Ethernet Switches

- **Switch**
  - **Switch**
    - A **switch** is an **interconnection device** that connects ***multiple* networks** that uses the ***same* network protocol**:

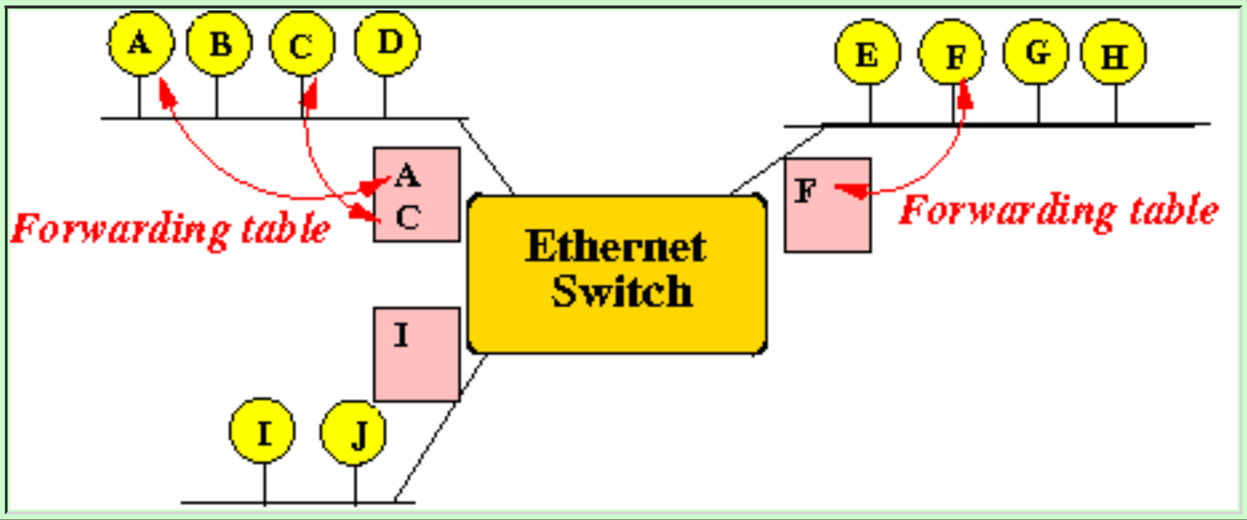


Another name for a **switch**:

- A **switch** is **also known as**: **"*Multi-way* Bridge"**

- **Heads up: router**
  - **Networking *Babel*:**
    - ***Switch*** = a **device** that connect (multiple) **networks** of the ***same* type**
      - A ***switch*** **operates** at the **Data Link layer level**
    - ***Router*** = a **device** that connect **networks** of the ***different* type**
      - A ***router*** **operates** (= **processes**) at the ***Network layer* level**
      - A **router** (See: [click here](#))
  - **Note:**
    - ***some* textbooks** will call **device** that connect **networks** of the ***different* type**:
      - A ***switch*****.....** (instead of a ***router***)

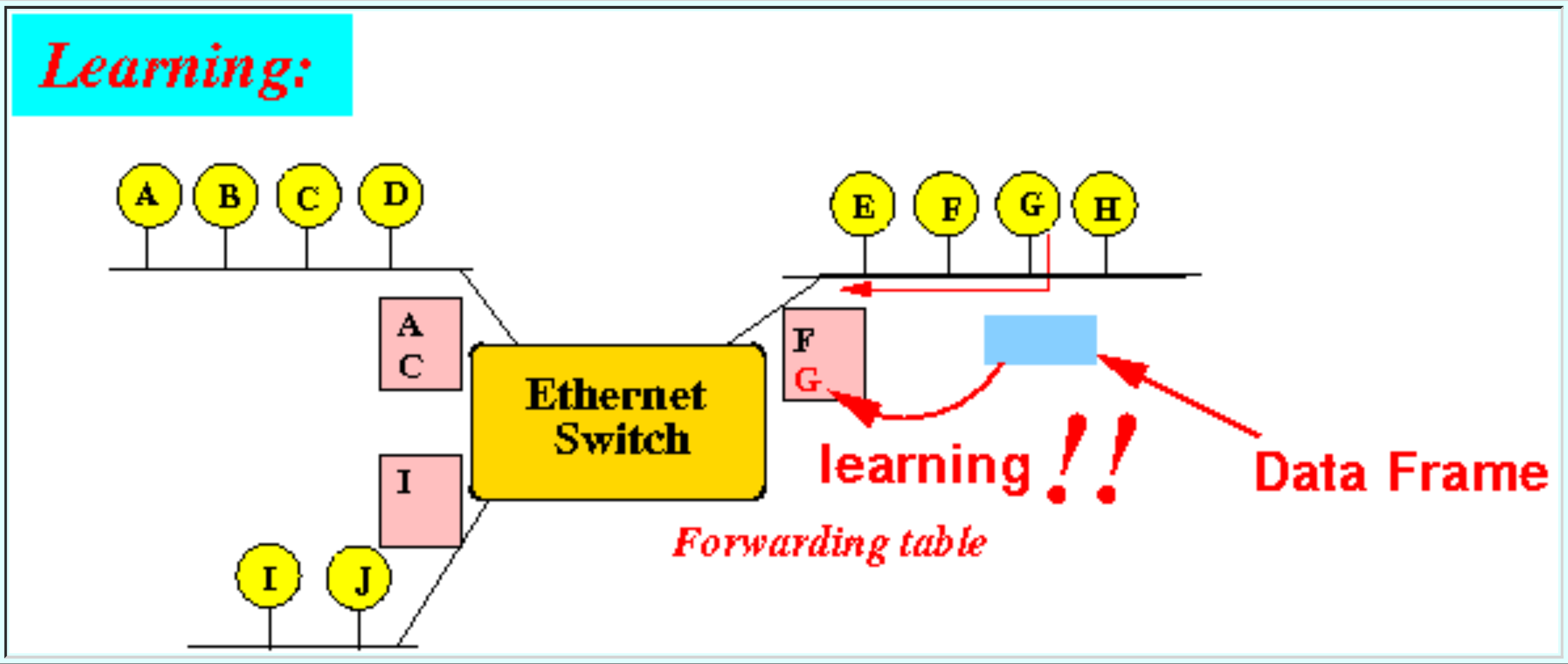
- **Operation of an (Ethernet) Switch**
  - **Forwarding table:**
    - **Each port** of the **switch**, has an associated ***forwarding* table** :



Forwarding table contains:

- **Node addresses** of **nodes** that reside on the **network segment**

  - **Learning:** (**adding** entries to the **forwarding table**)
    - When the **switch** receives a **frame** from a **node** on **port** `P`:



The **switch** will:

```
if ( source address ∉ Forwarding Table(P) )
{
   Insert source address into Forwarding Table(P);
}

Reset expiration timer for source address;
```

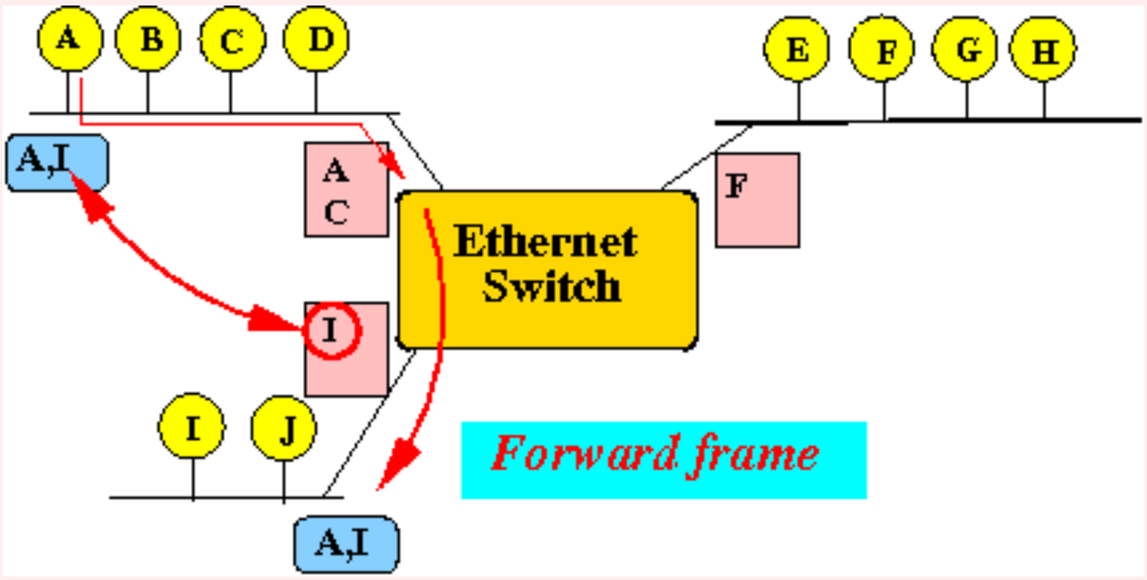
Note:

- The ***learned*** **entries** can ***expire***
- ***Expired*** **entries** are **deleted**

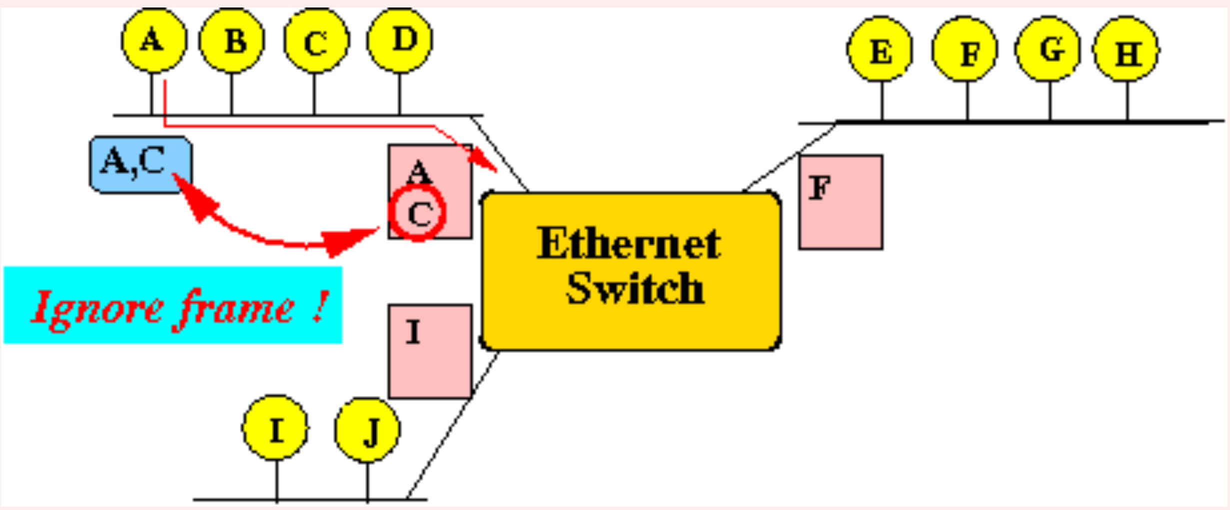
- ***Forwarding*** **Operation** of a **switch**:

  - Suppose an **Ethernet switch** receives an **Ethernet frame** on some **port *P***:

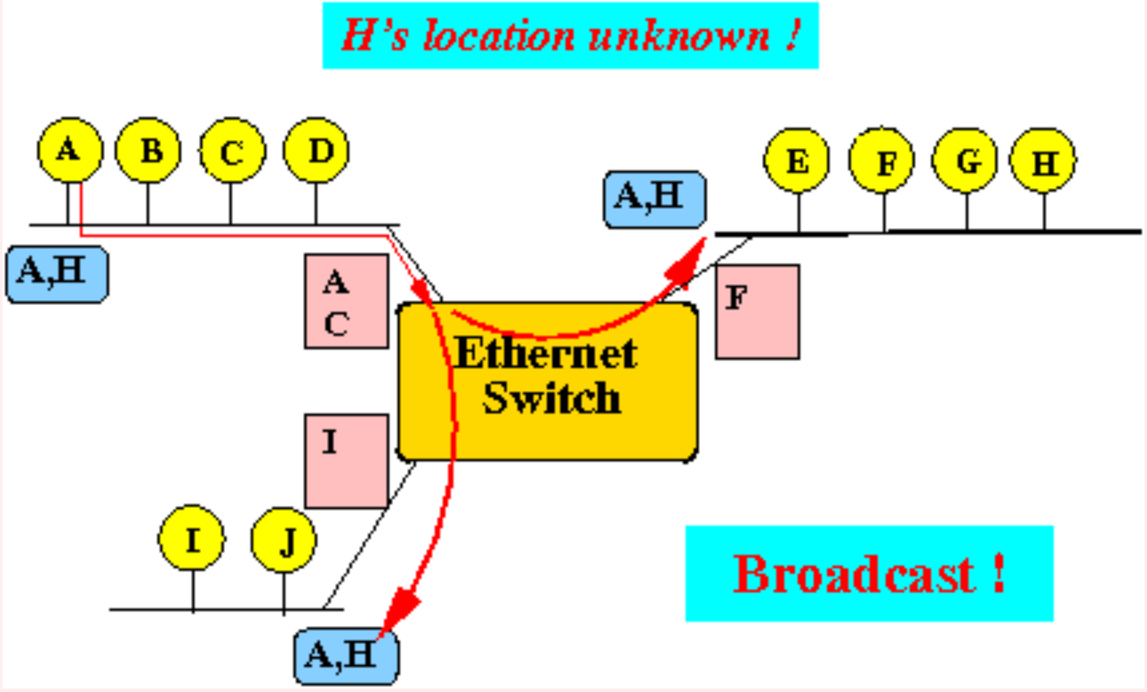
```
if ( Destination Addr ∈ Forwarding Table(Q (≠ P)) )
{
   Forward frame on port Q;
```




```
}
else if ( Destination Addr ∈ Forwarding Table(P) )
{
   Discard frame;
```




```
}
else
{
   Forward frame on all other port except port P;
```

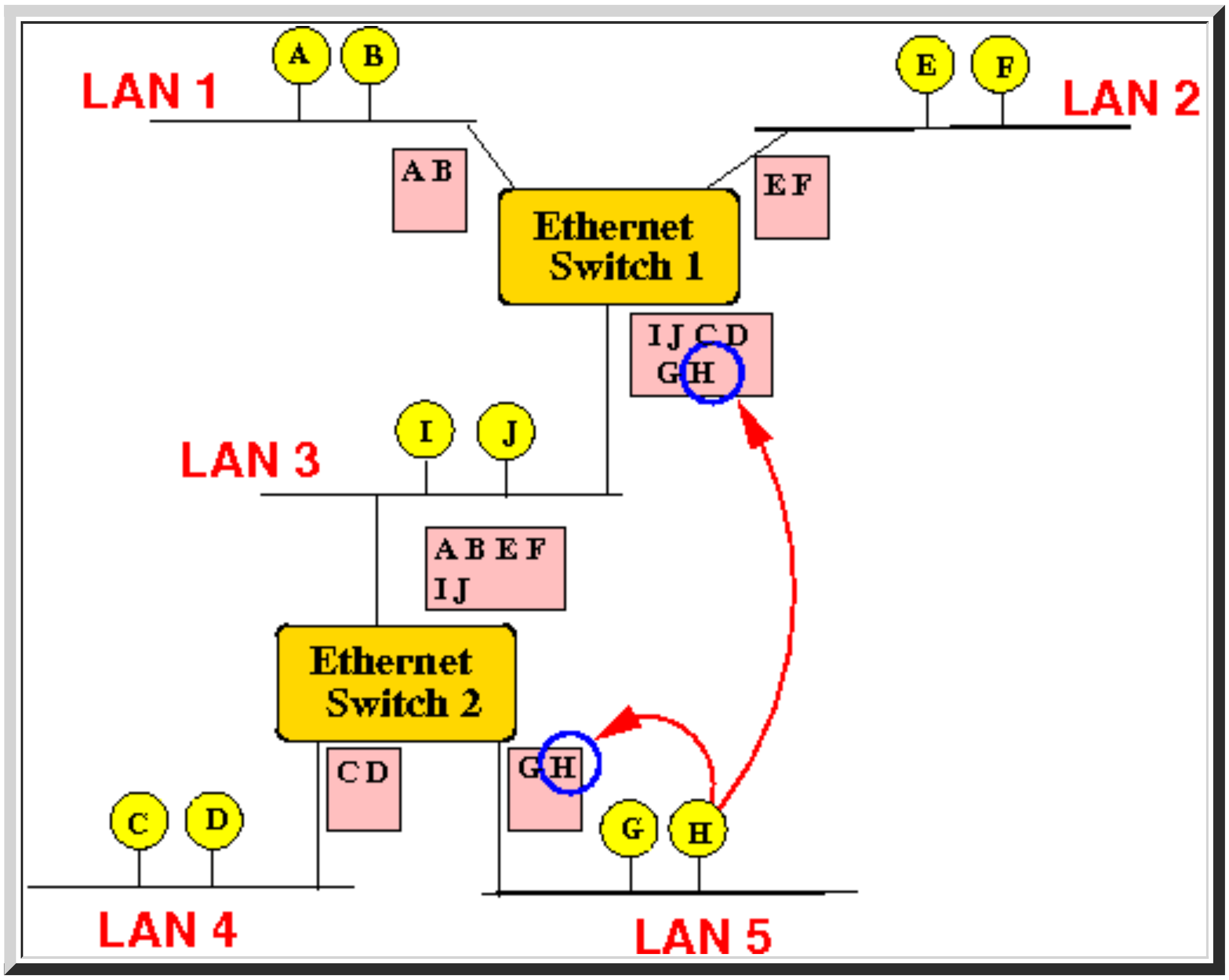


(This operation is called: broadcasting)

}

Loop-free Ethernet switched networks

- ***Interconnecting* (Ethernet) networks with Switches**
  - **Example network 1:**

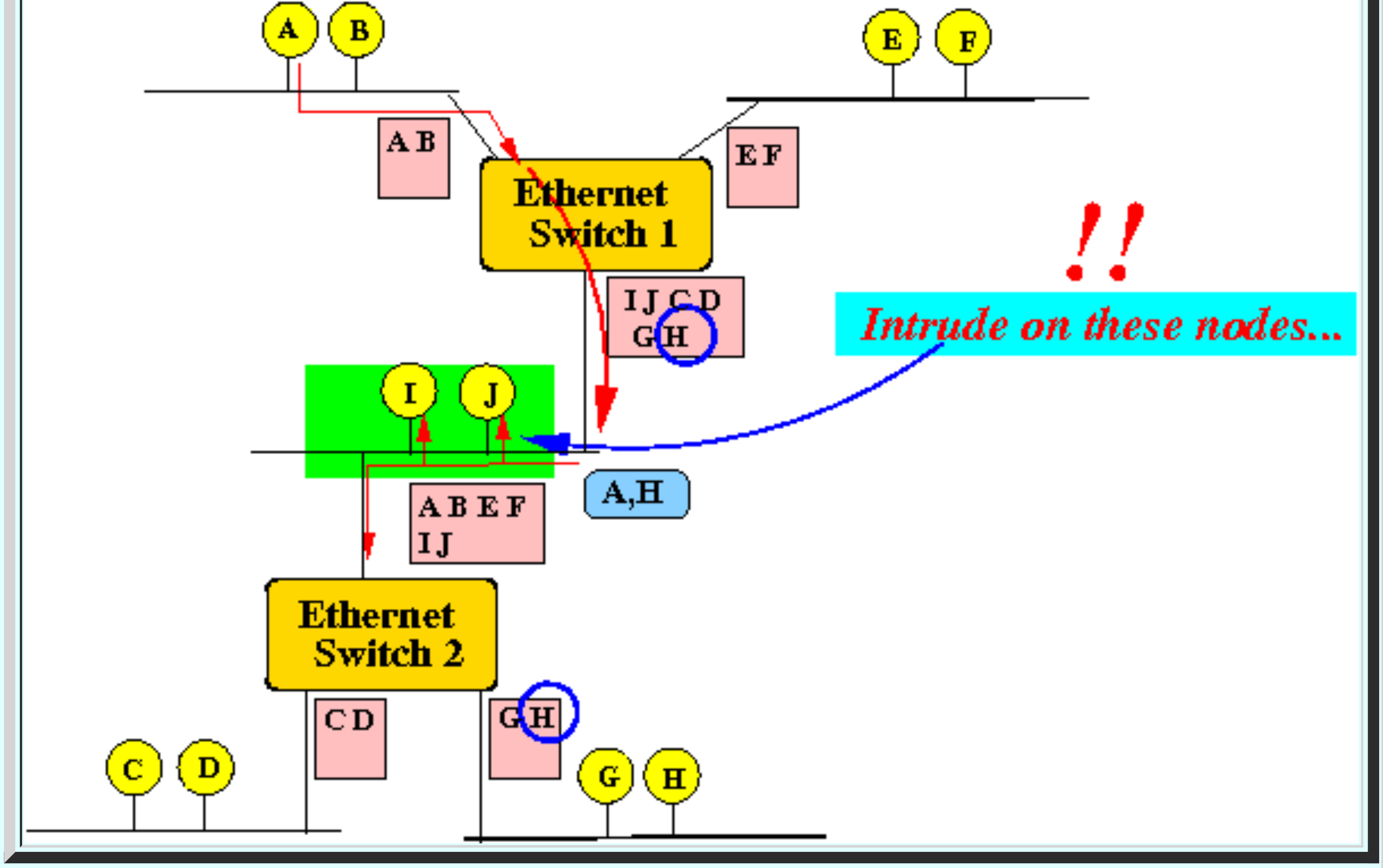


Notice that:

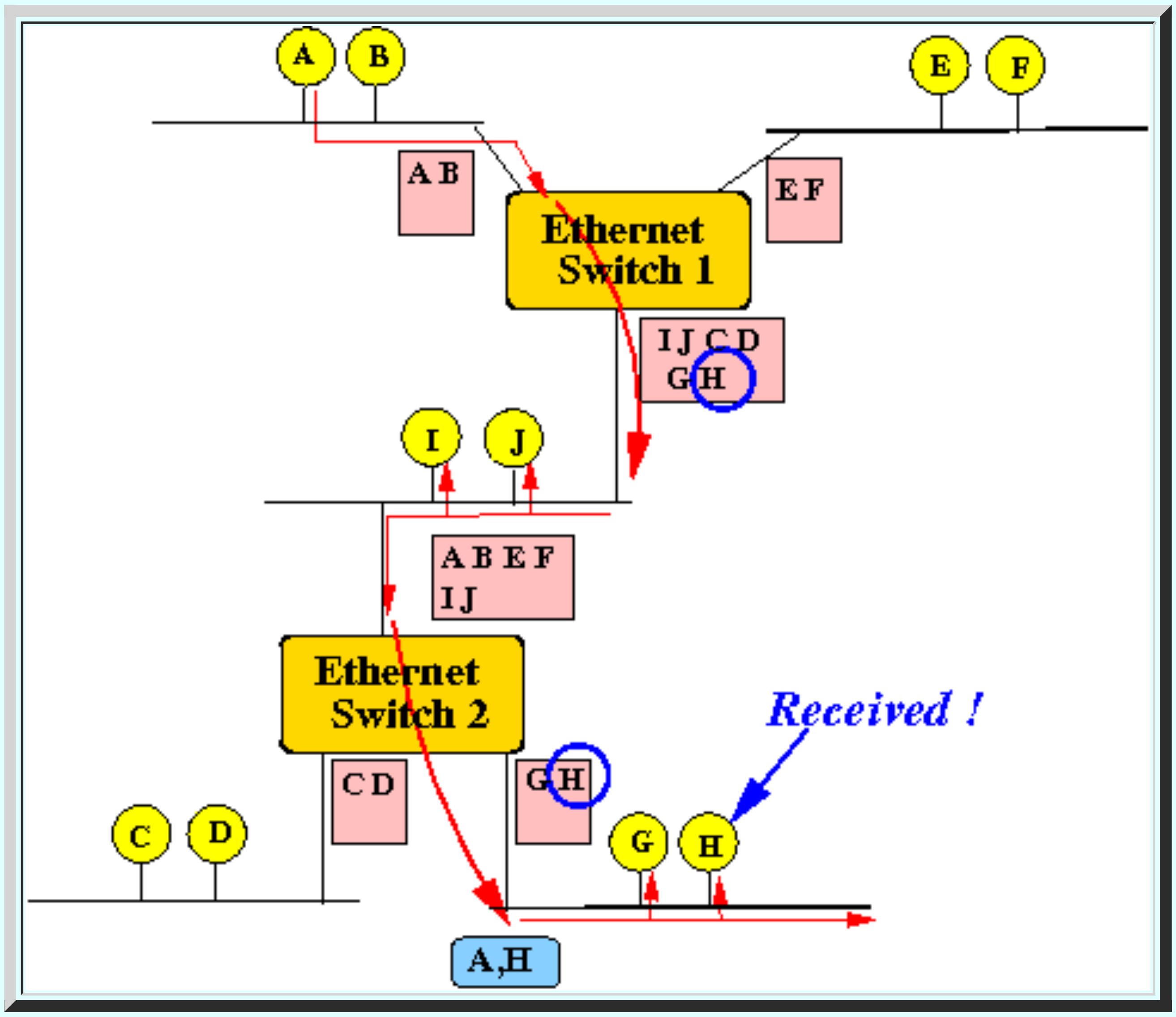
- The **node *H*** is ***not*** on a **network** that is ***directly* attached** to **switch 1**

- But ***H*** can ***still*** be stored in the **forwarding table** of **switch 1** !!!
  - **Node *H* resides** on ***that* network "segment"**

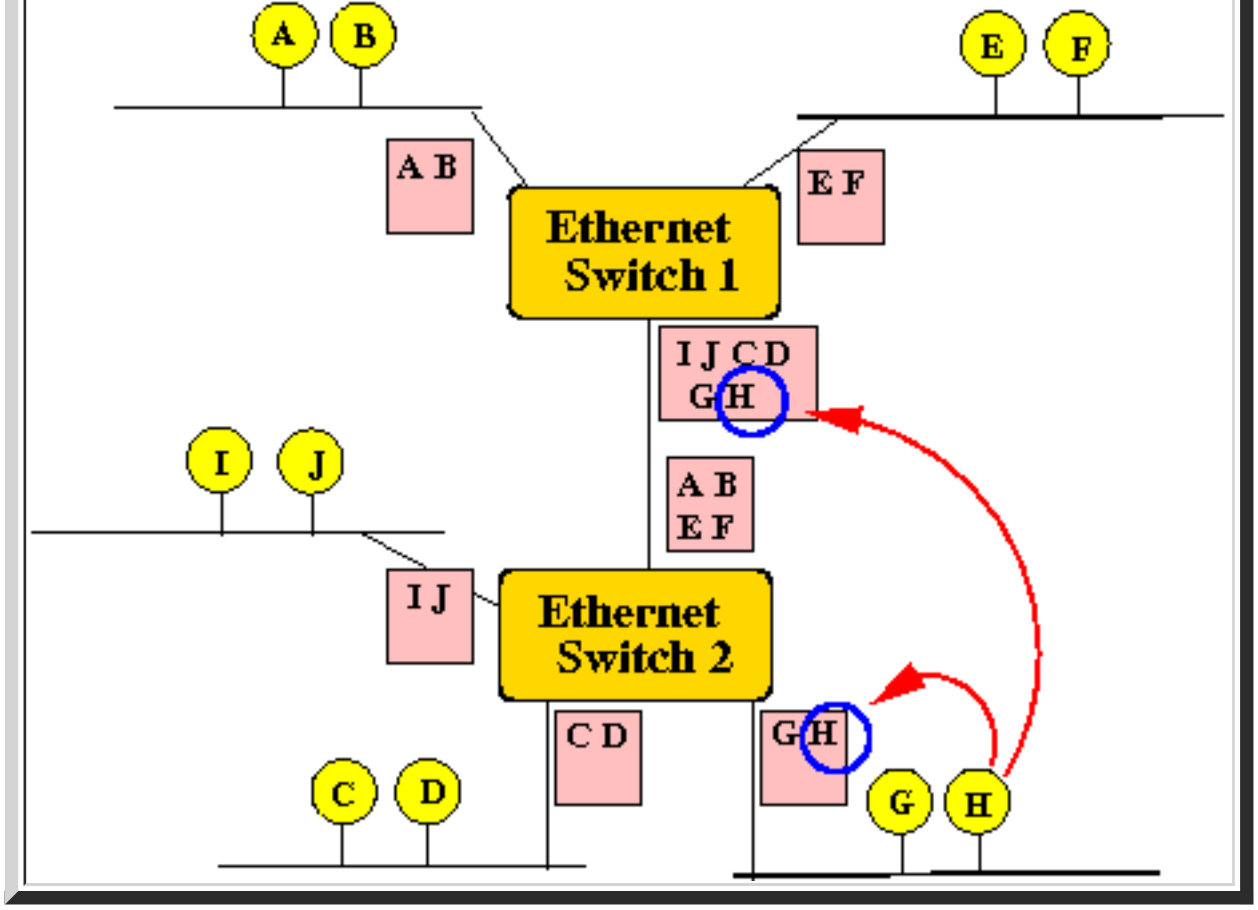
- **How** the **frame** from ***A*** ⇒ ***H*** is ***forwarded***:
  - **Switch 1** finds ***H*** in its ***forwarding* table** and **forwards** the **frame**:



- **Switch 2** finds ***H*** in its ***forwarding* table** and **forwards** the **frame**:



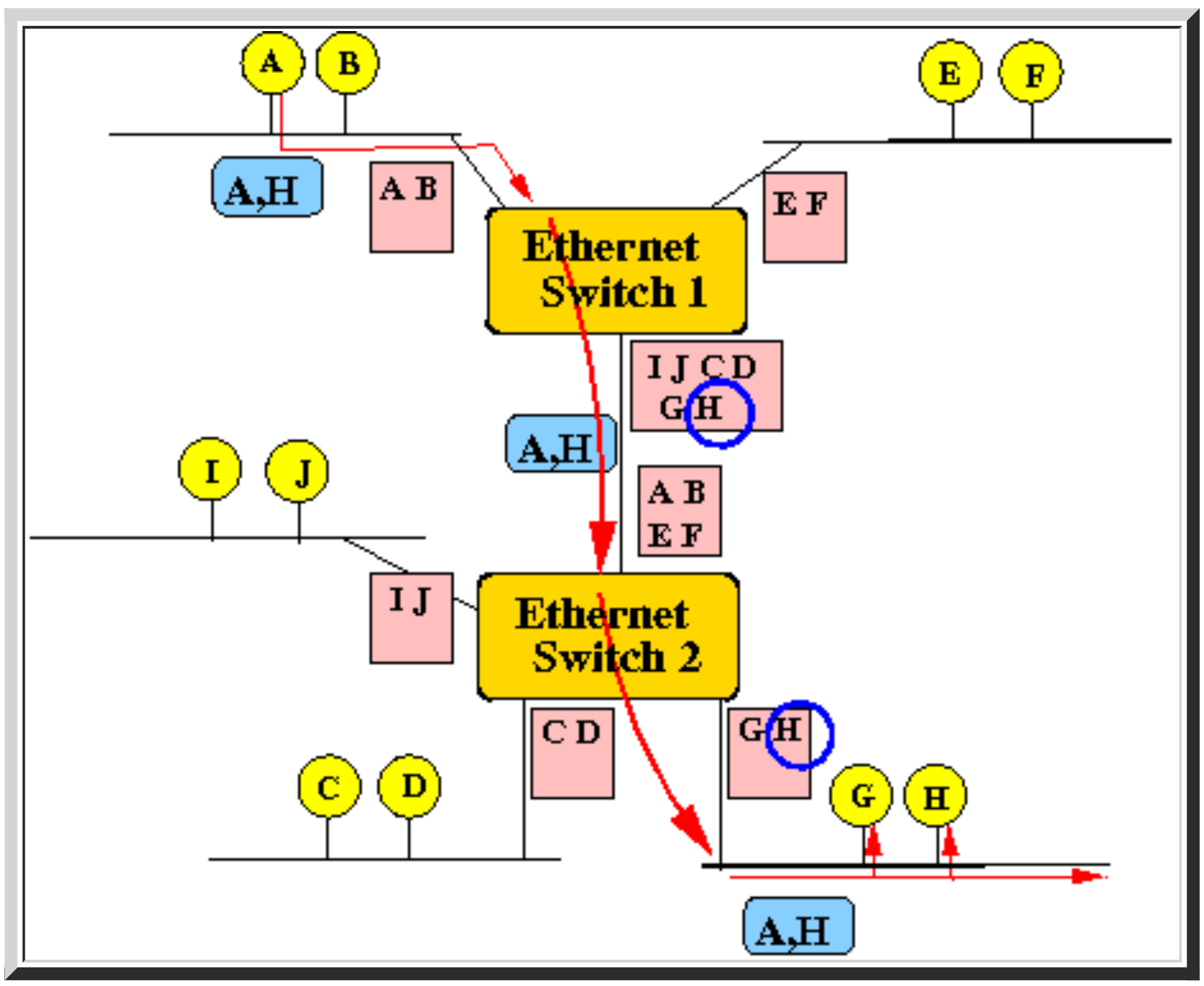
- **Improved network topology**
  - **Example newtork 2:**



Changes made to the **topology**:

- The **switches** are **connected *directly*** to each other:
  - There are ***no* networks** in **between** the **switches** !!!

- **How** the **frame** from ***A*** ⇒ ***H*** is ***forwarded***:



Result:

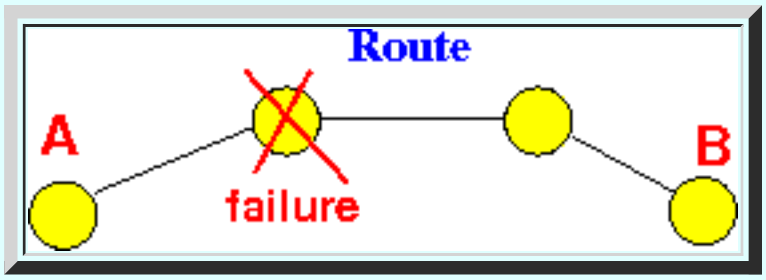
- ***Only*** nodes on the ***source* network** and the ***destination* network** will **hear** the **frame transmission** !!!

- **Switched networks** are ***highly* scalable** !!!

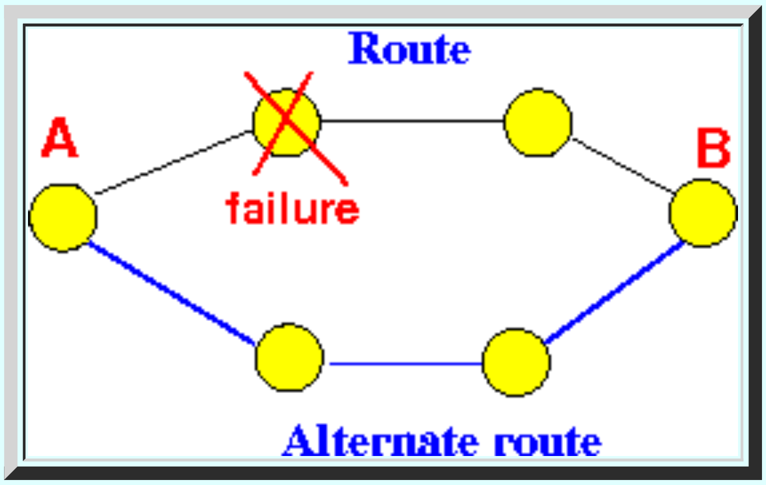
Fault-tolerant computer networks

- ***Fault-tolerant* Network Topologies**
  - **Network failures:**
    - ***Links* can fail (accidentally cut)**
    - ***Nodes* (switches) can fail**
  - **Fault *tolerant* computer networks:**
    - **Fault *tolerant* computer networks** = the ***remaining* operational portion** of the **computer network** will **operate *normally*** (= can **(send/receive)**) when ***some* component(s)** have **failed**

- **How to construct fault-tolerant networks**
  - **Design** of **fault tolerant** computer networks:
    - A **fault** will cause a **disruption** of a **route**:



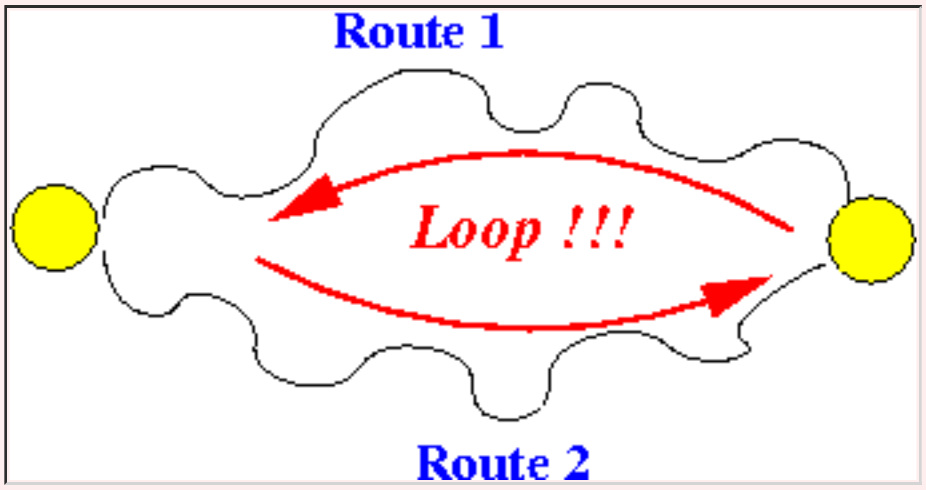
    - **Fault tolerance** is **achieved** by providing ***alternate* routes** between **nodes**:



Result:

- When ***one* of the route** is **disrupted**:
  - the **frames** can **still** reach the **destination** through ***another* route**

- **A very unfortunate consequence of fault tolerant networking**
  - ***Very unfortunate* consequence** of **fault tolerant networking:**
    - A ***fault tolerant* network** will ***always*** contain **loops**:



  - **Warning**:
    - ***Loops*** in a **computer network** has ***very* unpleasant effects**...

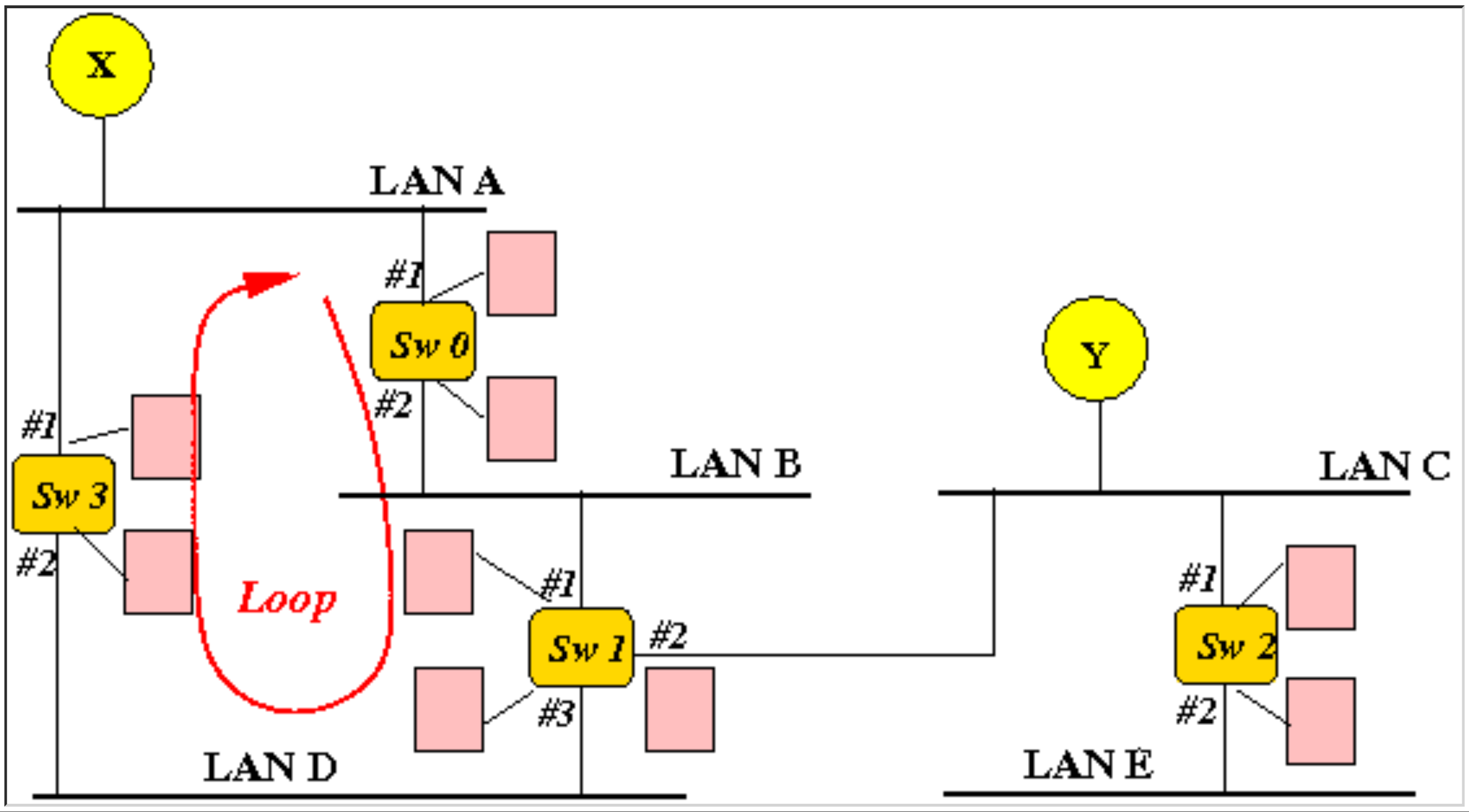
Effect of loops in computer networks

- **Effect of Loops in Networks**
  - **Effect** of **loops** in a **computer network**
    - **Frames (messages)** will be **repeated** ***indefinitely*** (if a **network** contains ***loop***) !!!

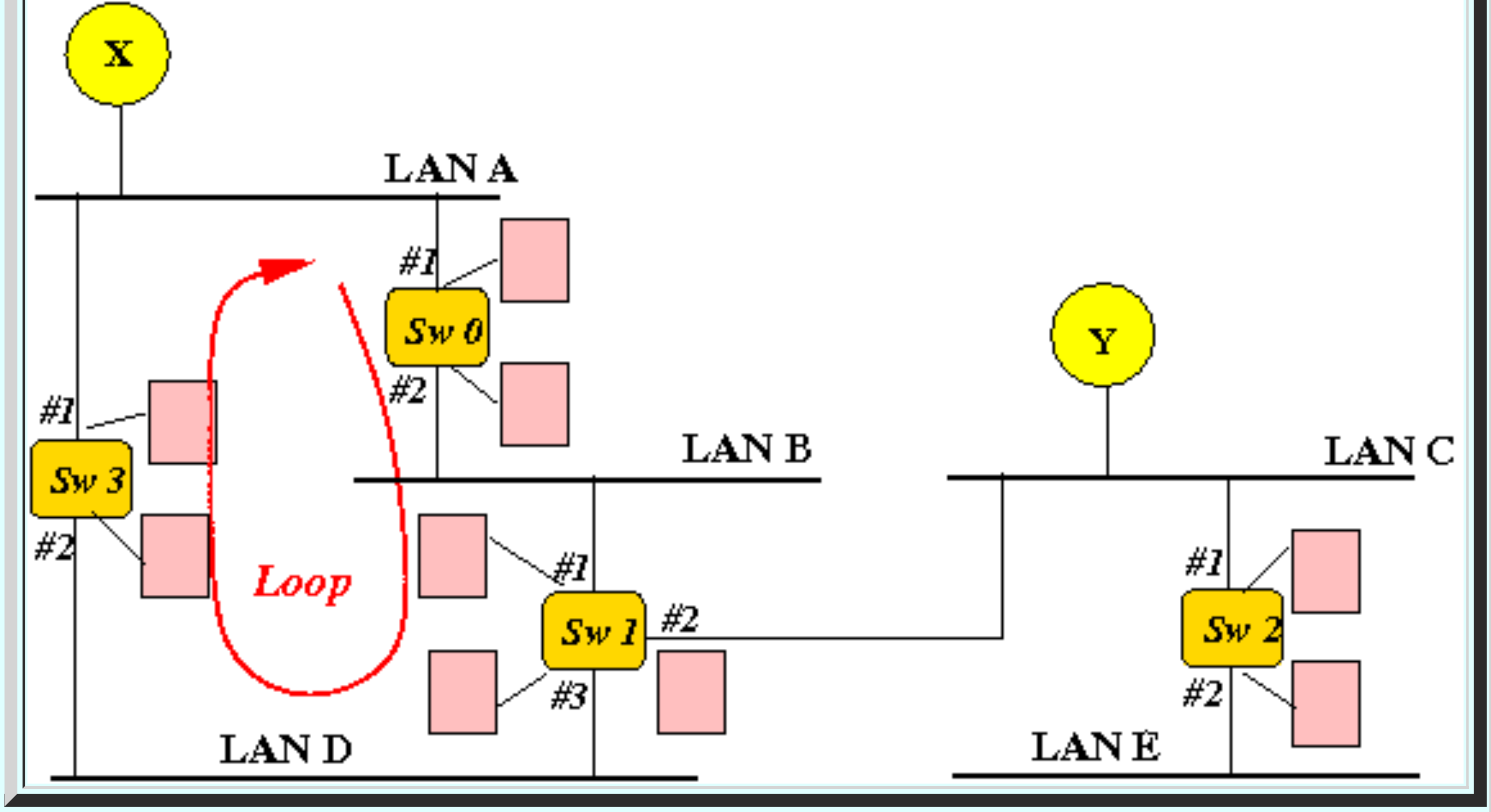
    - When **messages** are **repeated** ***indefinitely***, the **network** will become ***extremely*** **congested**

    - **Network performance** will ***drop*** **dramatically** and the **network** becomes ***useless*** !!!

  - **Example:** a **network** with a **loop**



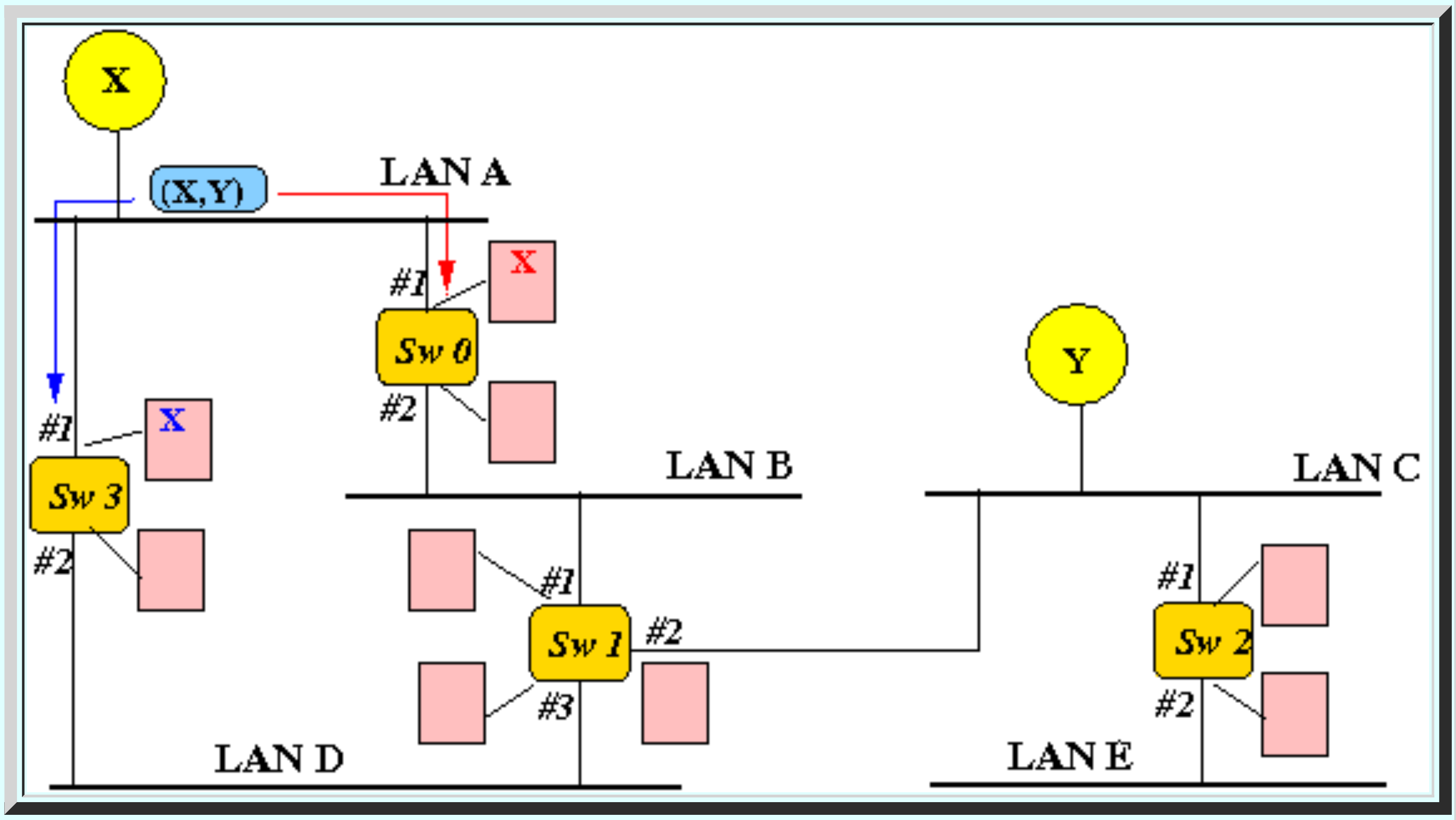
  - The ***loop*** **problem** in a **network** illustrated:
    - We **begin** with ***empty*** **forwarding tables:**



Suppose:

- **Node *X*** transmits a **frame** to **node *Y***

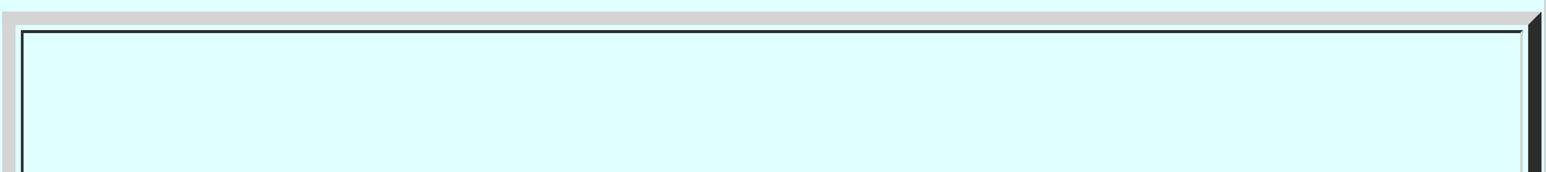
- The **frame** will **first** be **received** by the **switches Sw0 and Sw3**:

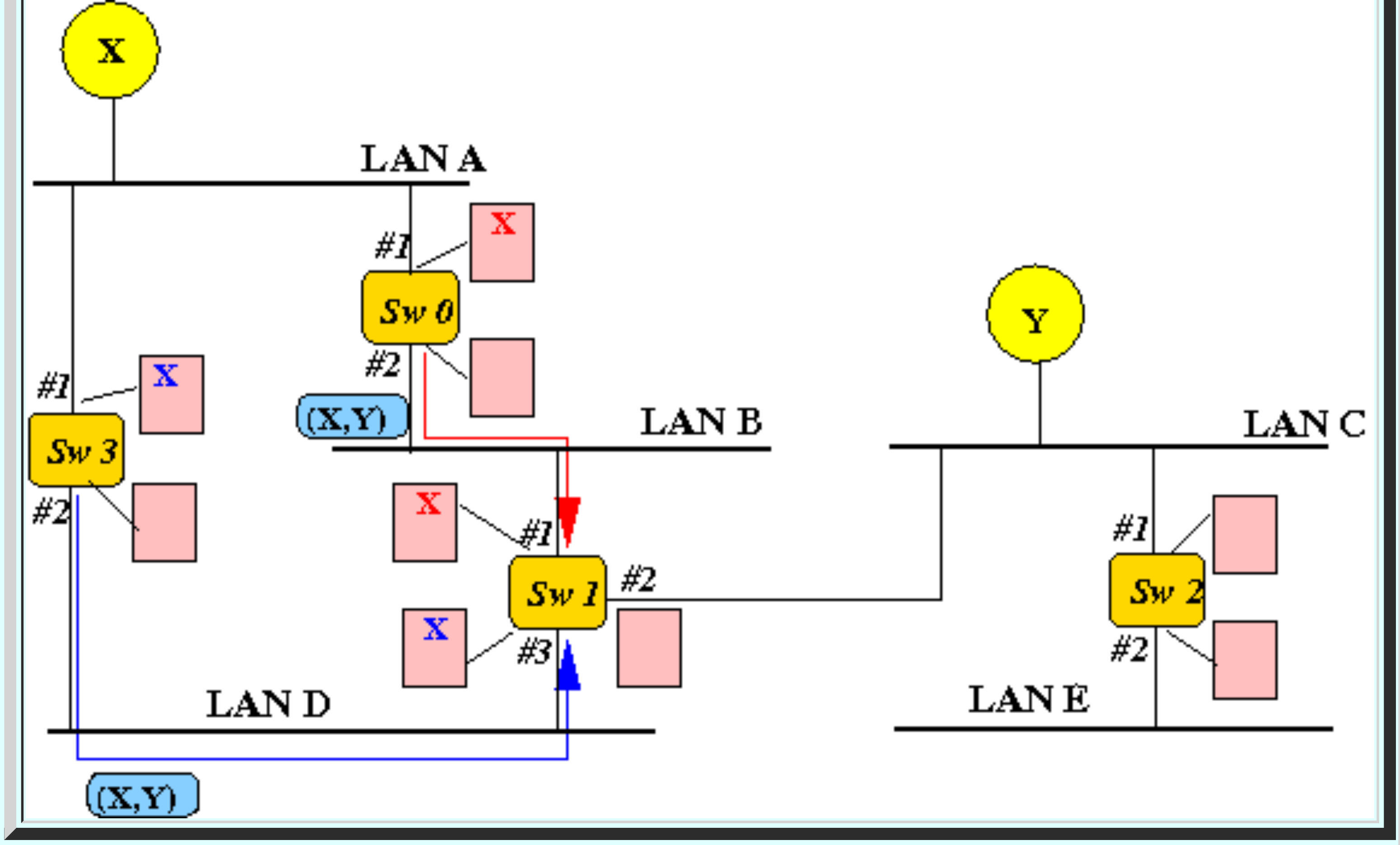


Result:

1. **Sw0** and **Sw 3** will **learn (insert)** the **source node *X***
2. **Sw0** and **Sw 3** will **then forward** the **frame** on ***all* other ports**

- After **one more round** of **forwarding**:





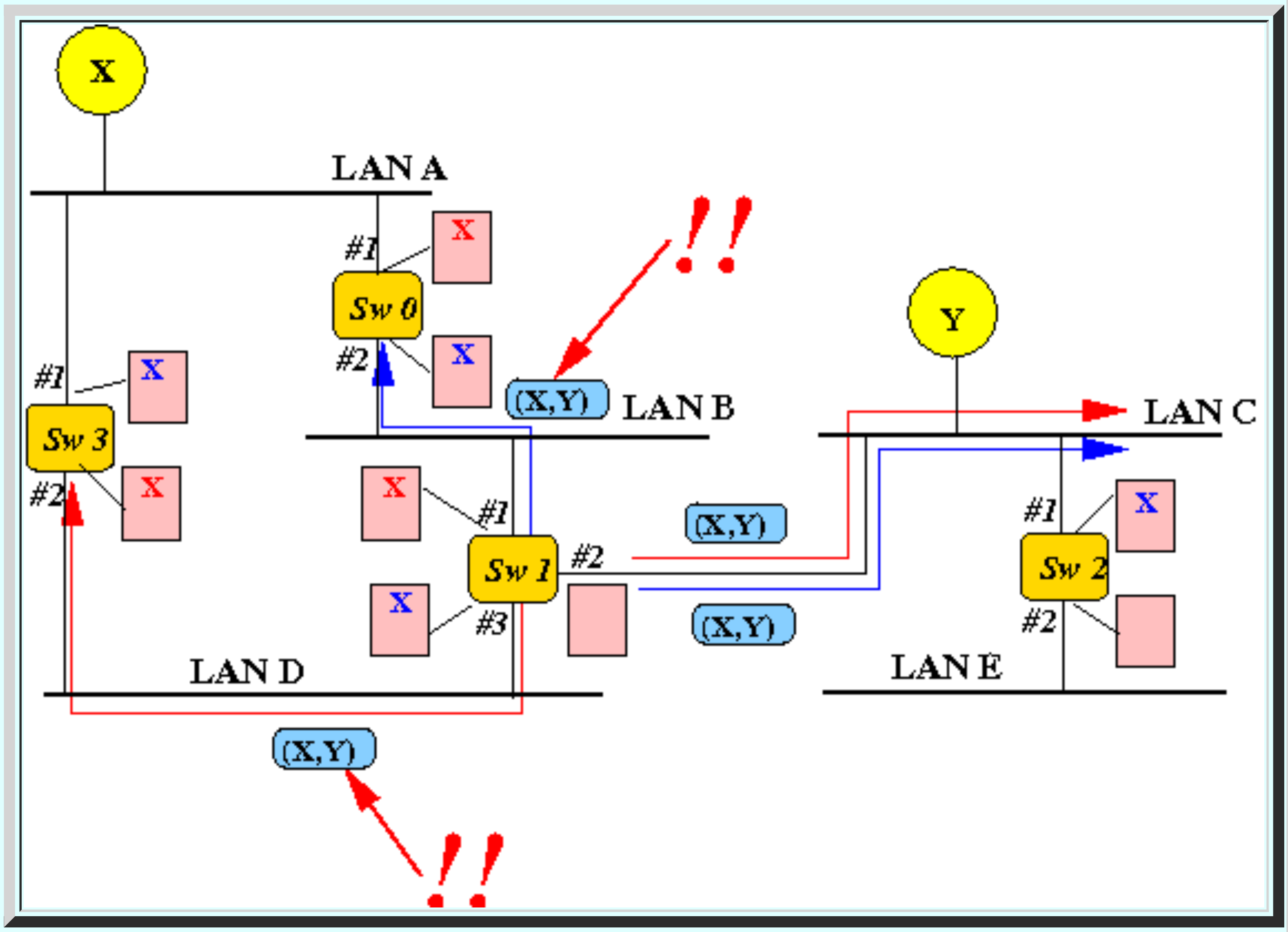
Result:

- **Sw 1** will **learn (insert)** the **source node *X***

Notice that:

- The **entry** for ***X*** in the ***two* forwarding tables** of **switch *Sw1*** !!!!!

- After **one more round** of **forwarding**:



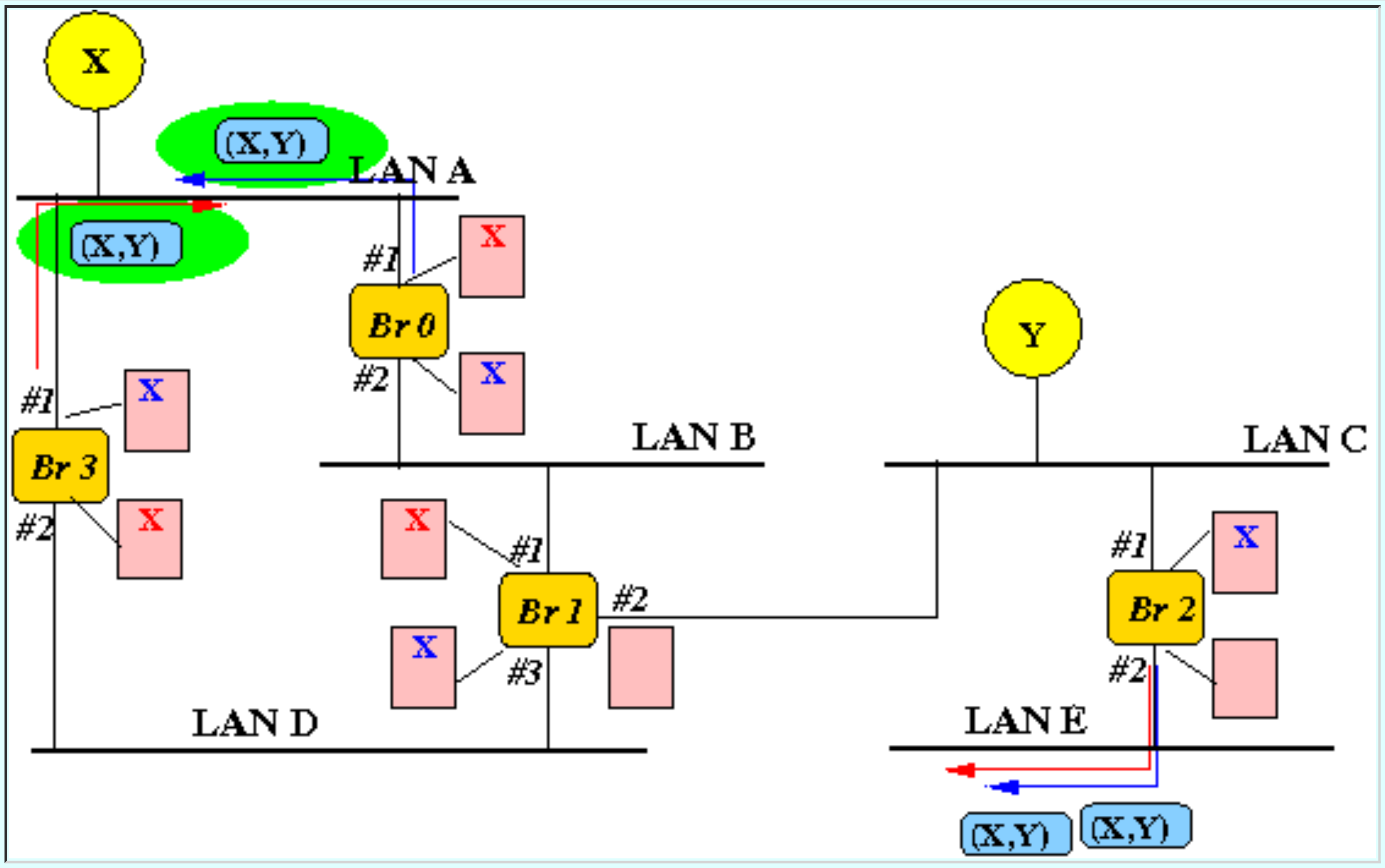
Notice that:

- The **frame** is **forwarded** on

  - **LAN *A*** and **LAN *D***

  becase the **destination address** was **found** in ***another* forwarding table** !!!

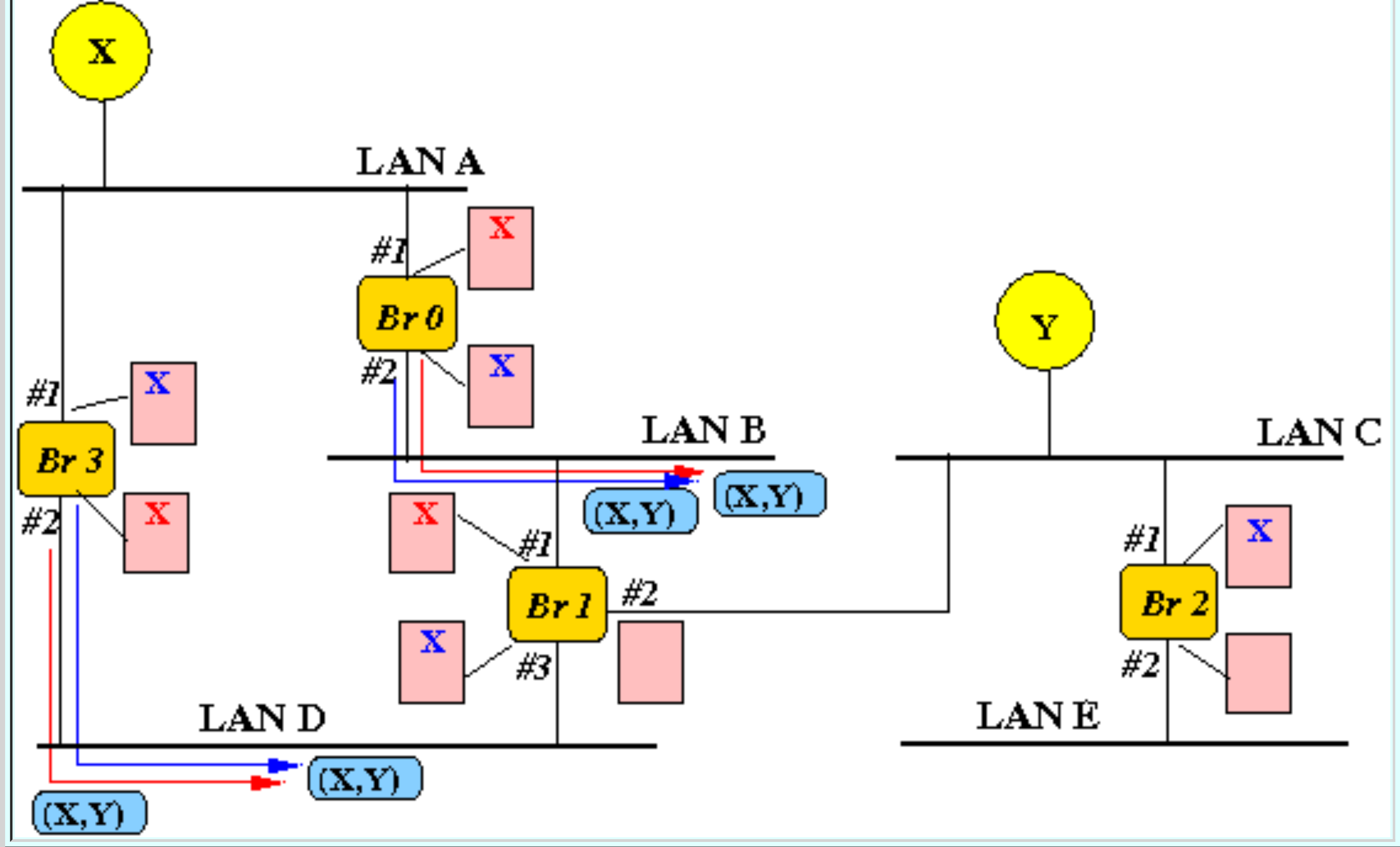
- **Now** the **network** is in ***trouble***:



There are **now**:

- **2 copies** of the **frame** on the **source LAN *A*** !!!

- ***Each* copy** will be **forwarded** through the ***loop* again**:



***Each* copy** will **generate *another* 2 copies** of the **frame** !!!

- **Conclusion:**

  - A **loop** will cause **frames (messages)** to ***multiply* indefinitely** in the **network**

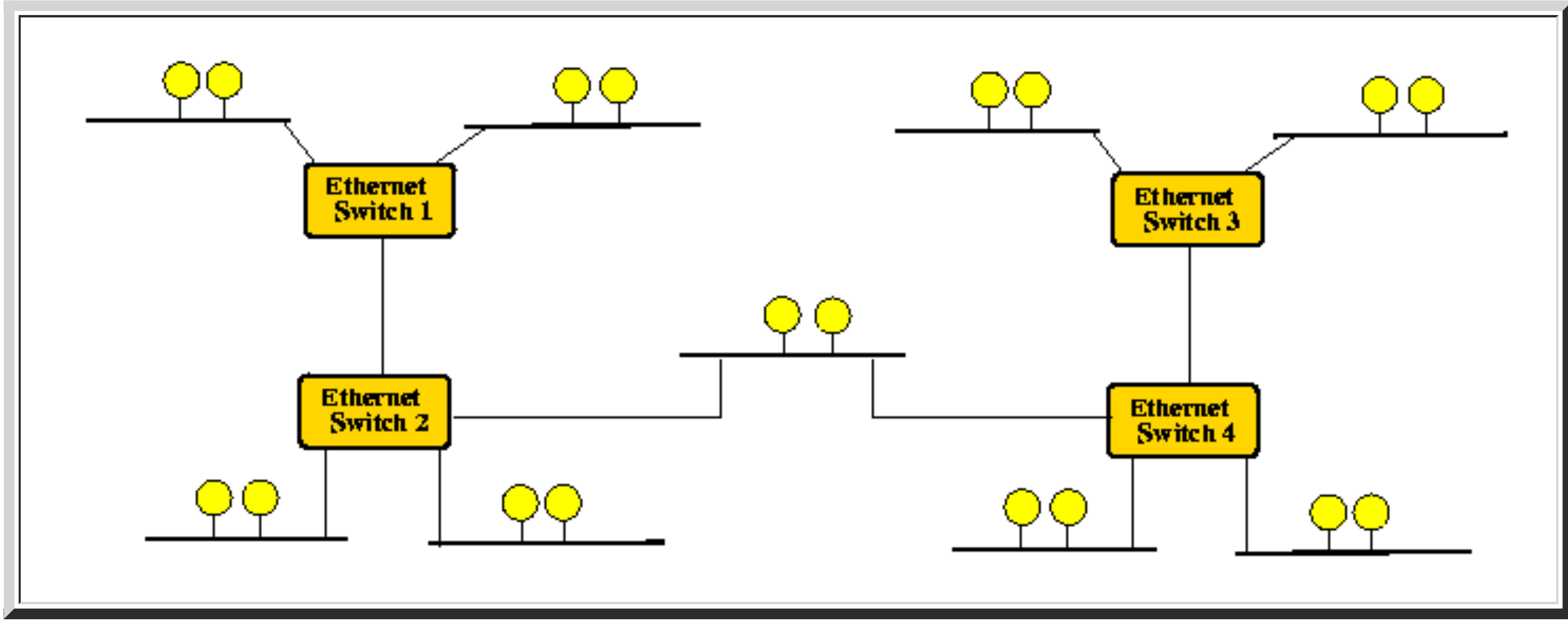
  - **Furthermore**:

    - The **copies** of a **frame (message)** will **cycle *indefinitely*** !!!

Dealing with loops --- physical network and logical network

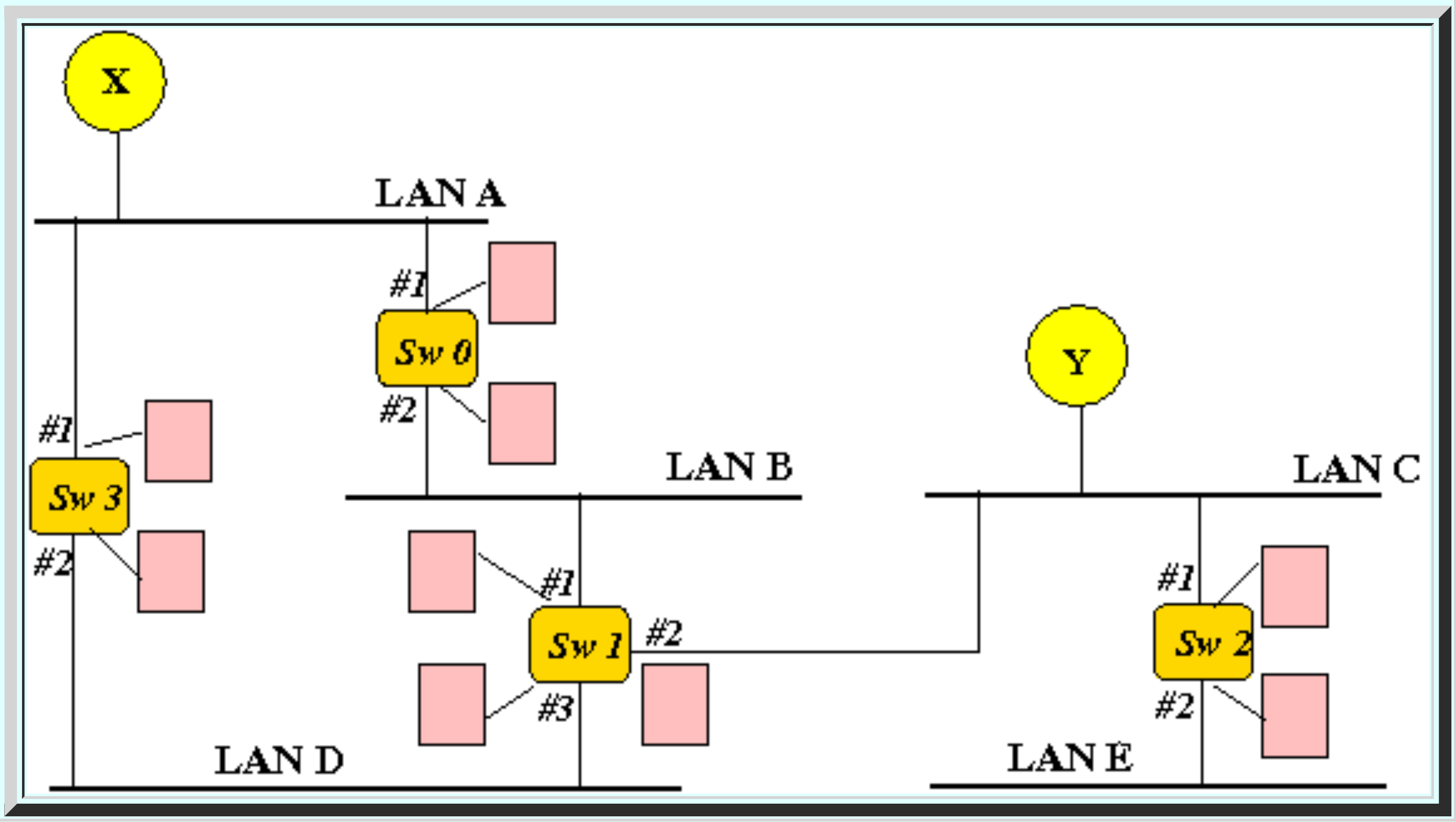
- **Loop-free networks**
  - A ***loop-free* network** is:
    - a **tree**

    Example:



- **Logical and Physical networks**
  - ***Physical* network**:
    - **Physical network** = the **network** that consists of the ***actual* nodes and links**

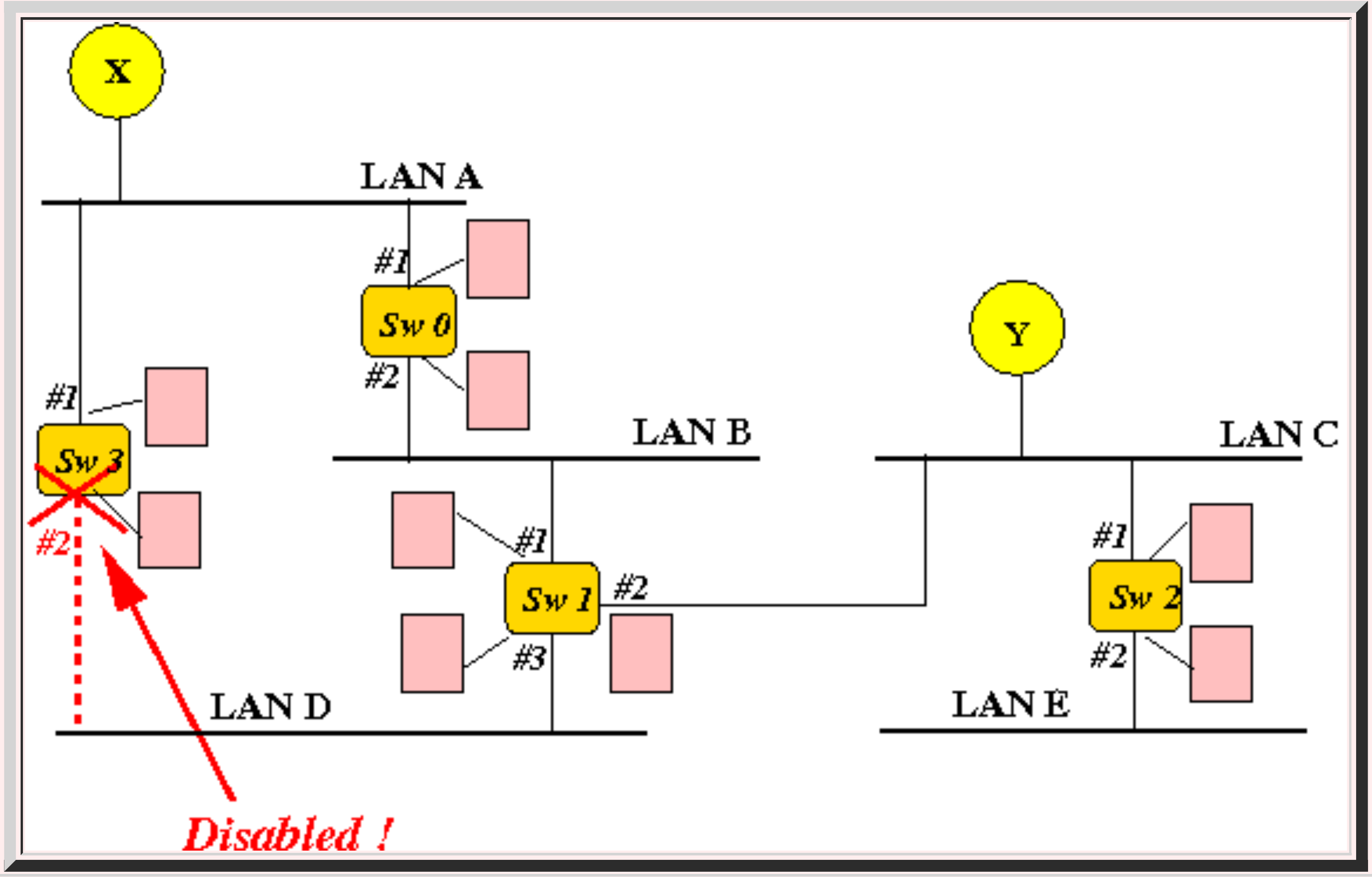
      Example:



- *Logical* **network:**

  - **Logical network** = the **(sub)network** that consists of the **nodes and links** that are ***used*** to **forward frames**

    Example:

  Note:

  - We **must** make sure that the **logical network** is a ***tree*** !!!

- **How to provide fault tolerance**

  - **Fault-tolerant networking**:

    - The ***physical*** **network** contains **loops**

      - This is **necessary** to provide **alternative routes**

    - We **create** a ***loop-free*** **logical network (= tree) on top** of the **physical network** by:

      - **Disabling** one or more **ports** on **switches** that is **part of a loop**

    - **Frame forwarding** will ***only*** use the **logical network**

- **The IEEE 802.1D "Spanning Tree" algorithm**

- **Next:**
  - We will study a ***distributed*** **algorithm** that the **Ethernet switches** execute to:
    - **Form** a ***tree*** (= the ***logical*** **network**)

  - The algorithm is called the **IEEE 8021D Spanning Tree Algorithm**

Intro to the IEEE 802.1D *Spanning Tree* Protocol

- **IEEE 802.1D documentation**
  - Here is the complete 802.1D standard document: click here
    - The **tree-construction algorithm** is on **pages 77-124** in the **PDF file) (or pages 59 - 106 in the document page numbering)**
  - I will only cover a **subset of the standard** (described in **text books**) which contains the **gist of the protocol**:
    - how to **set up** a **loop-free (tree) structure** and
    - how to **recover** from **failures**.

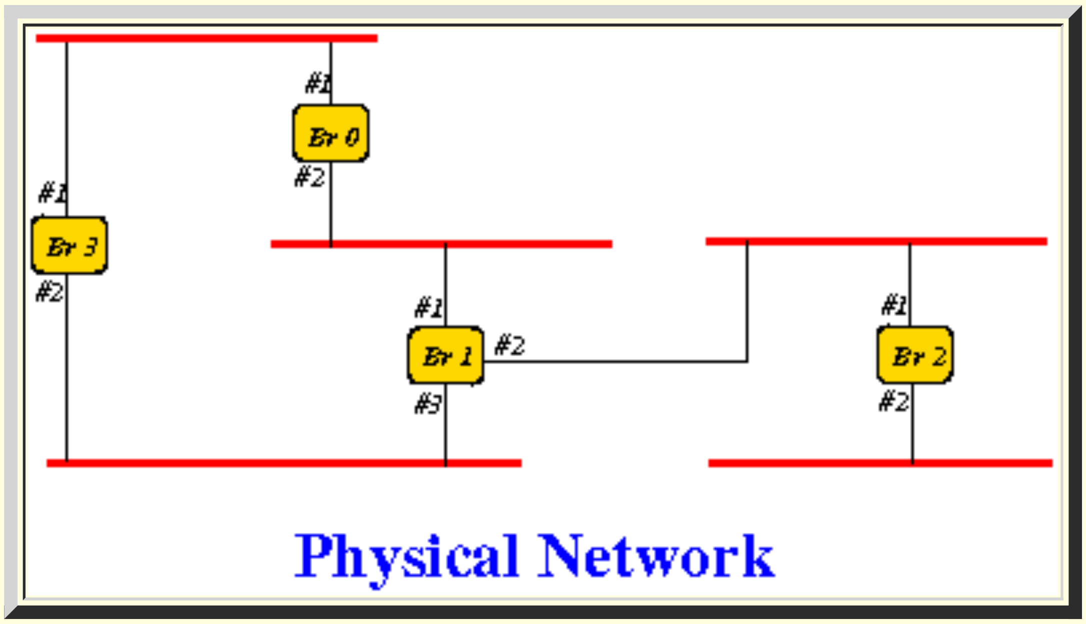
- **Nomenclature comment**
  - Since the **IEEE Standard** calls the **Ethernet *switch*** a:
    - ***Multiway* Bridge**

  - **Note:**
    - I will use the term ***bridge*** in my **IEEE 802.1D Spanning Tree Protocol lecture webpages**
      - The **proper term** should be: ***switch***

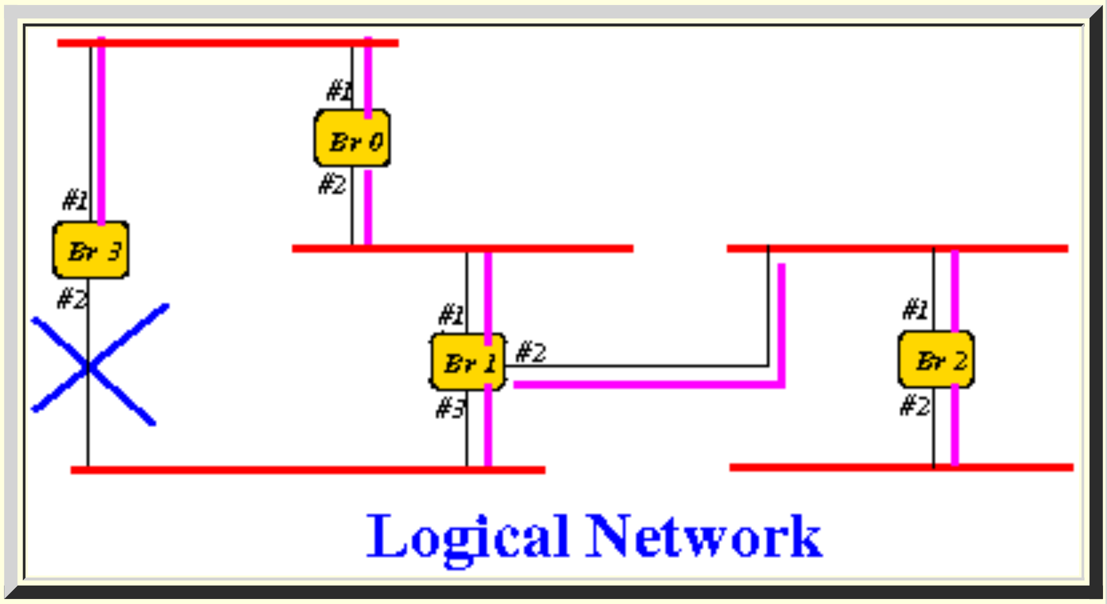
- **Goal of the Spanning Tree Algorithm**
  - **Purpose** of the **Spanning Tree Algorithm**:
    - The **Impose** a ***logical* tree structure** on top of the **physical network** (that may contain ***loops***)

    Example:

- The ***physical* network**:



- The ***logical* network**:



(A ***tree*** !!!)

- The **logical network** will be **used** in **message forwarding operation**

- **How to create a *logical* network on top of the physical network**

  - **State information:**

    - Each **port** has a **state** associated with the **port**

    - **Blocked port**:

- If the **state** is ***block***, the **port** will ***not* forward frames** !!!

  (And the **loop** is ***broken***)

- **Intro to the 802.1D Spanning Tree Algorithm**
  - The **802.1D Spanning Tree Algorithm** is a ***distributed* algorithm**:
    - ***Every* node** will **perform** the **computation *on its own***

    - **Each node** maintains some ***local* information**
      - ***Local* information** is ***only* accessible** by the **owner node**

    - **Nodes** will ***exchange* information** to **help each other** perform the **computation**
      - **After** receiving some **information**, a **node** may **update** its ***local* information**

      - If the **node's *local* information** is **changed**, the **node** will transmit its ***updated* local information** to ***other nodes*** *!!!*

  - **Distributed Algorithms**:
    - ***Distributed*** means:
      - There is **not** a **place (= node)** in the **entire system** that has ***all* (= complete) information**

    - **Distributed Algorithms** always performs their **computation** by:
      - ***Exchanging*** their ***local* state informations** (by sending **your information** to your ***neighboring* nodes**)

- **Overview of the Spanning Tree Algorithm**

  - ***Overview*** of the **Spanning Tree Algorithm:**

    1. Each **bridge** will ***first*** **determine** (by itself) the ***root*** **bridge**

       - The ***root*** **bridge** is the **bridge** with the **smallest node ID (= Ethernet Address)**

    2. Each **bridge** **then** uses the **root bridge ID** to **determine** (for itself) its **root port**:

       - **Root port** of a **bridge** = the **port** of the **bridge** that ***leads*** to the **root bridge**

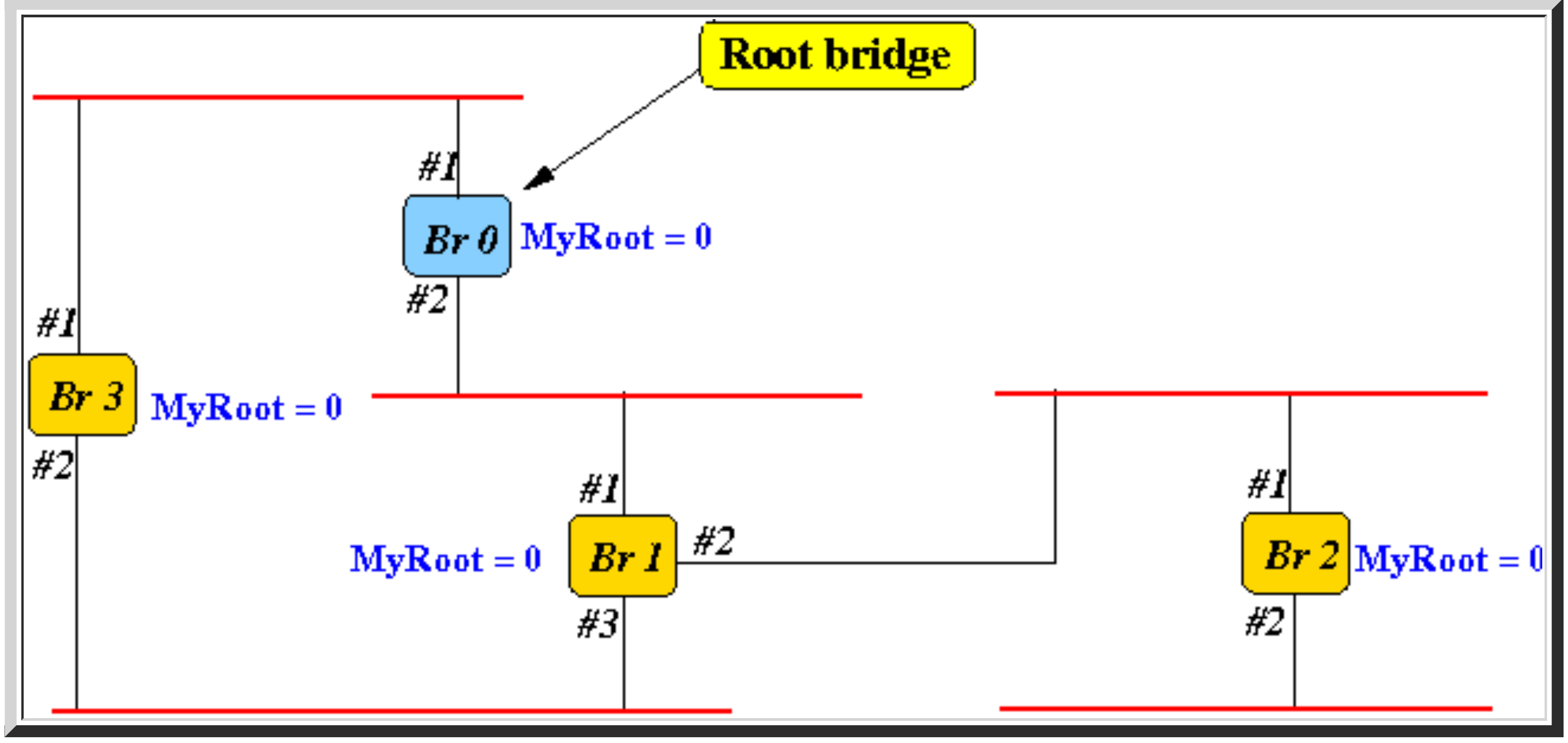
    3. For each of the **other ports** (the ***non-root*** **ports**) a **bridge** will then **determine** (for itself) if the **port** is:

       - A ***block*** **port** (status = ***blocked***) **or**
       - A ***unblock*** **port** (status = ***designated***)

The *root* bridge and root ports

- **Prelude...**
  - **Assumptions:**
    - In these **webpages**, I will **use**:
      - **bridge** to mean a ***switch*** (because the **text book** and **other IEEE 802.11D doucments** do so)
    - **Each bridge** has:
      - A ***unique* node ID**
      - ***Multiple* ports**

- **The Root Bridge**
  - **Root bridge:**
    - **Root bridge** in the **network** = the **bridge** with the ***smallest* ID**
  - **Example:**



  - In the **figure above**:
    - **Bridge 0** is the **root bridge**

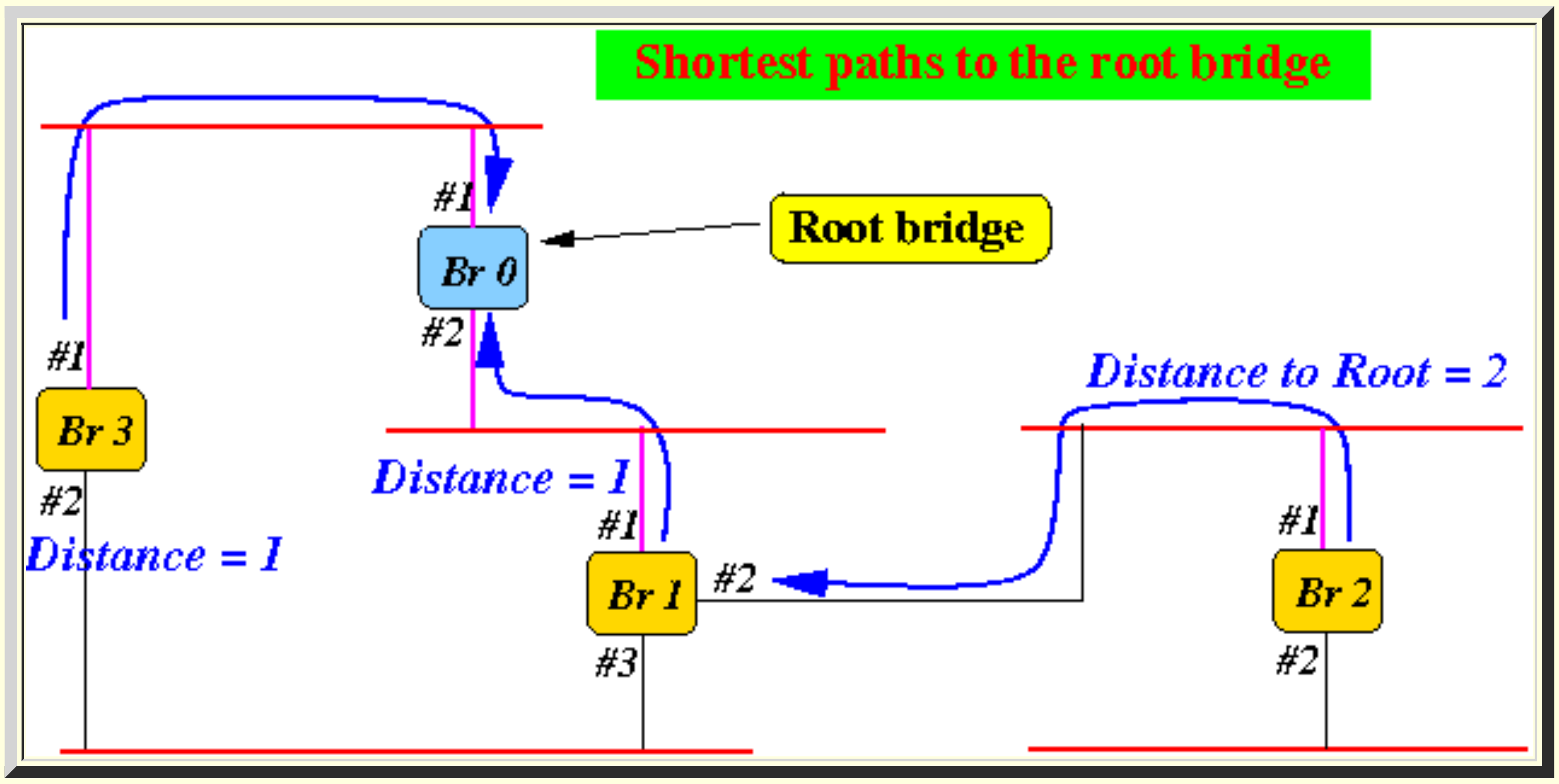
- **A comment before we continue.....**
  - **Comment:**
    - Right now, I ***only*** want to **define** the **terminologies/concepts**

- We will **study** the ***algorithm*** **later**....

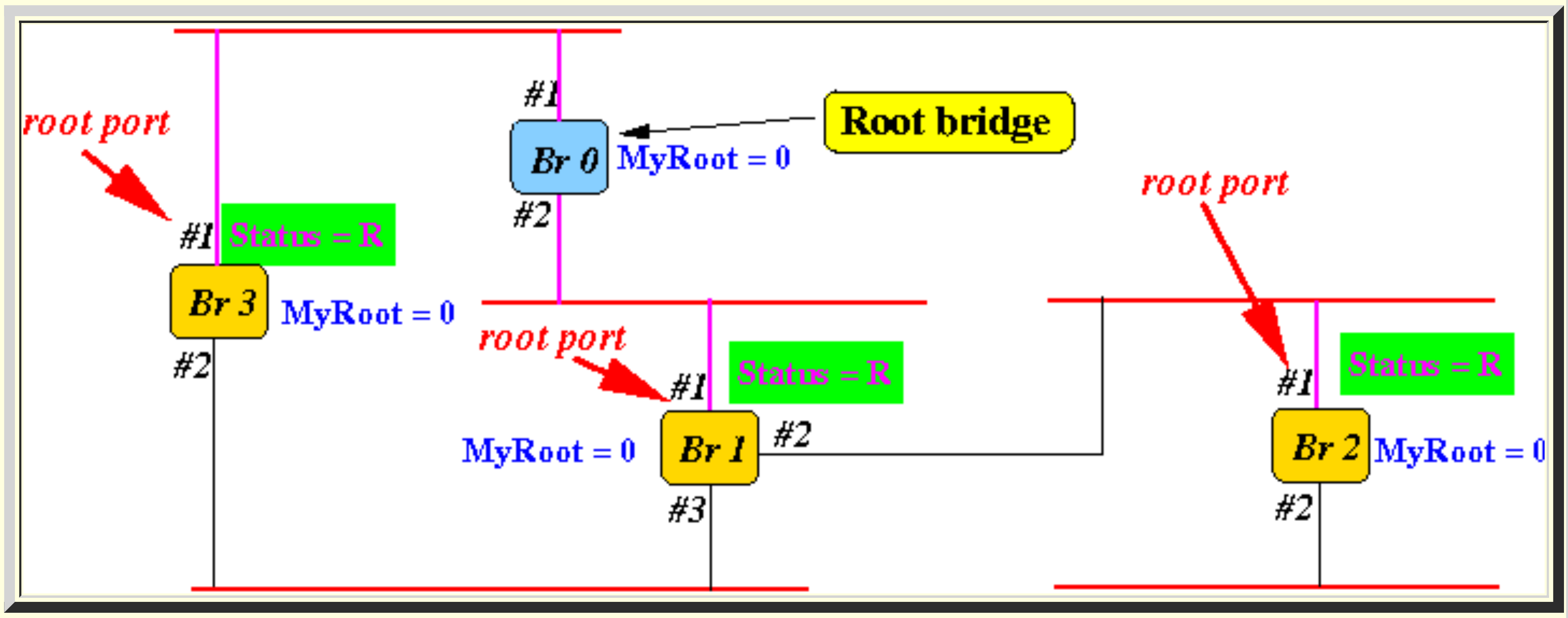
- **Role of the root bridge**
    - **Role** of the ***root*** **bridge**:
        - **Root bridge** = the **root** of the **tree** that is **constructed** by the **Spanning *Tree* algorithm**

- **The Root Port (of a *non*-root bridge)**
    - **Root port:**
        - The **root port** of a **(non-root) bridge** = the **port** that a **bridge** uses to **reach** the **root bridge** in the ***smallest*** **number of hops**.

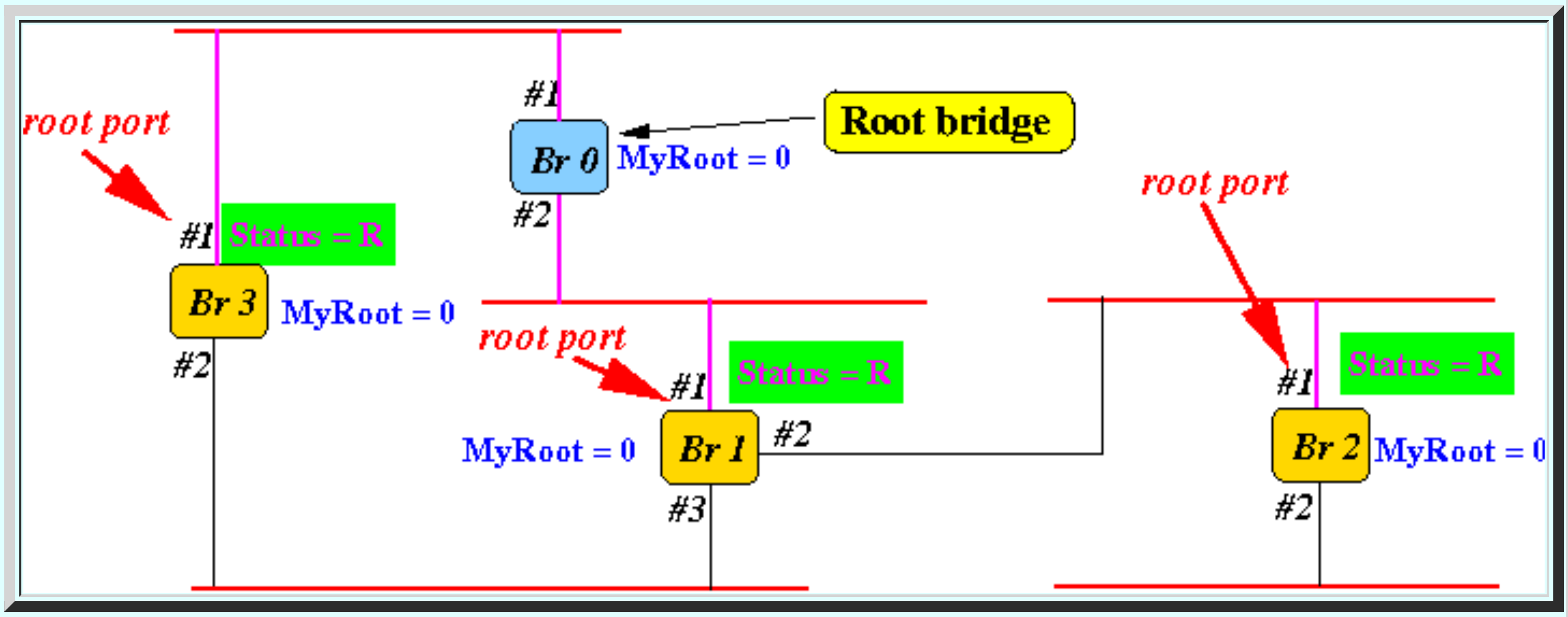
    - **Example:**
        - The **blue lines** in the **following figure** show the ***shortest*** **routes** from a **bridge** to the **root bridge**



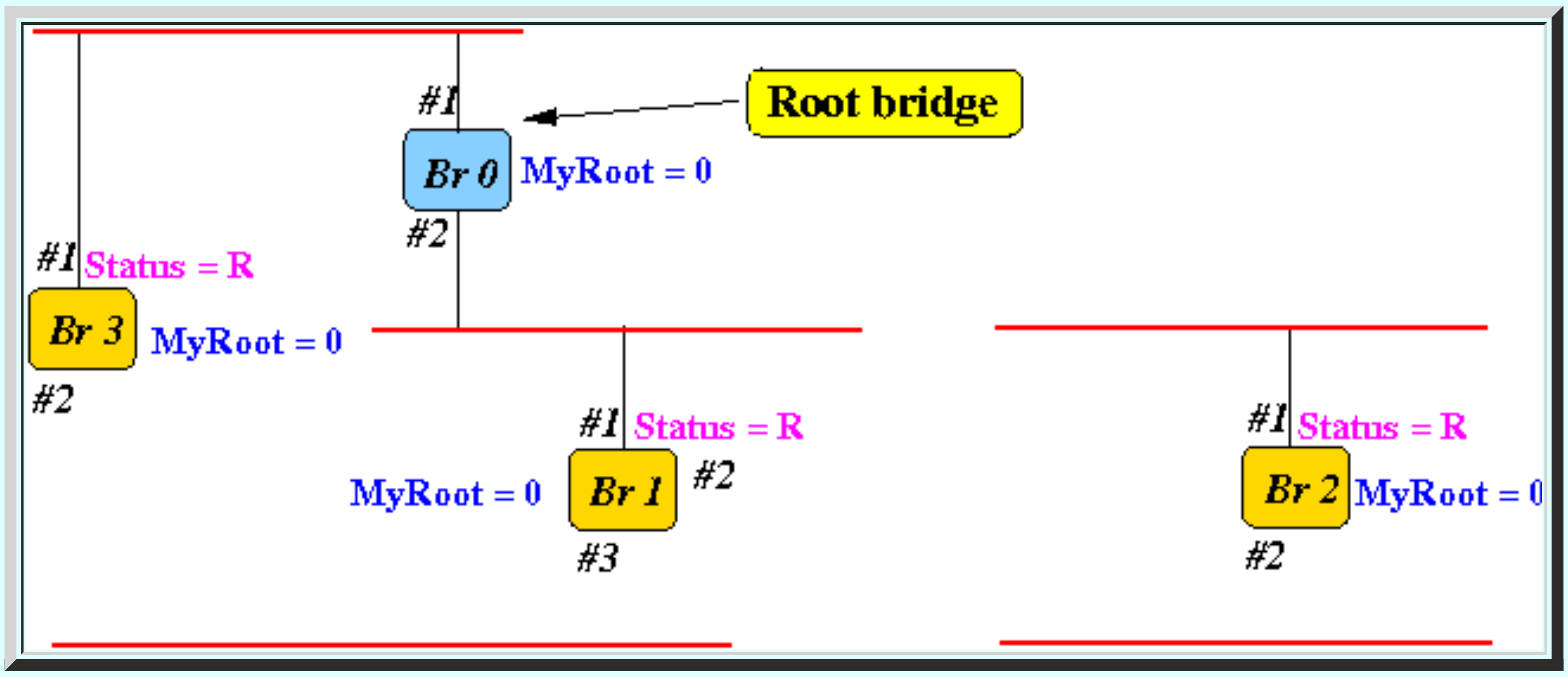
- The **root ports** of **each bridge** are marked with **"status = R"** in the **following figure:**



- **Taking stock....**
  - Consider the ***logical* network** spanned by **root bridge** with the **root ports** of the ***non-root* bridges**:
    - **Original network:**



    - ***Logical* network** with **only** the ***root* bridge** and the **root *ports***:



  - **Observations:**
    1. The ***logical* network** is ***loop-free*** (which is what we want)...
    2. **However**, the **network** so far **does *not* connect** to ***all* LAN segments**
       - We still have to **fix** this !

  - **Subsequent discussion**:
    - We need to **add** more **ports** to connect ***all* the LANs** in the network

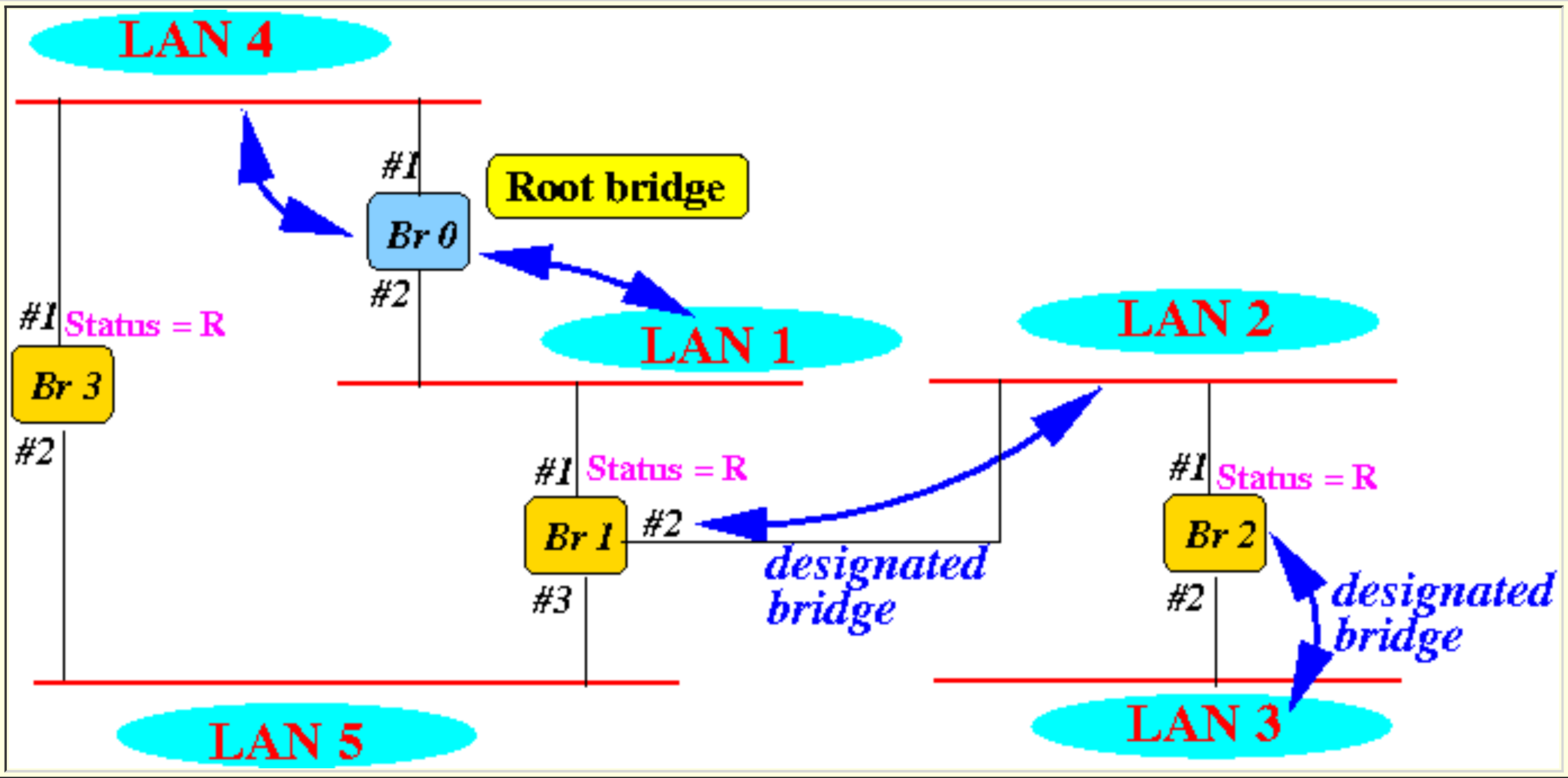
- These **additinal** ports are called ***designated* ports**

- We must be **careful** ***not*** to **create** a **cycle** while **adding ports** !!!

  - The ***remaining* ports** (= ***not* added**) are called ***blocked* ports**

The *designated* bridge and *designated* ports

- **Designated Bridges**
  - **Designated *bridge*:**
    - **Designated bridge** = a **bridge** that provide the ***shortest* connection (path)** to the ***root* bridge** for **some LAN segment**
  - **Example:**
    - The **designated bridges** of **LANs 1**, **2**, **3** and **4**:

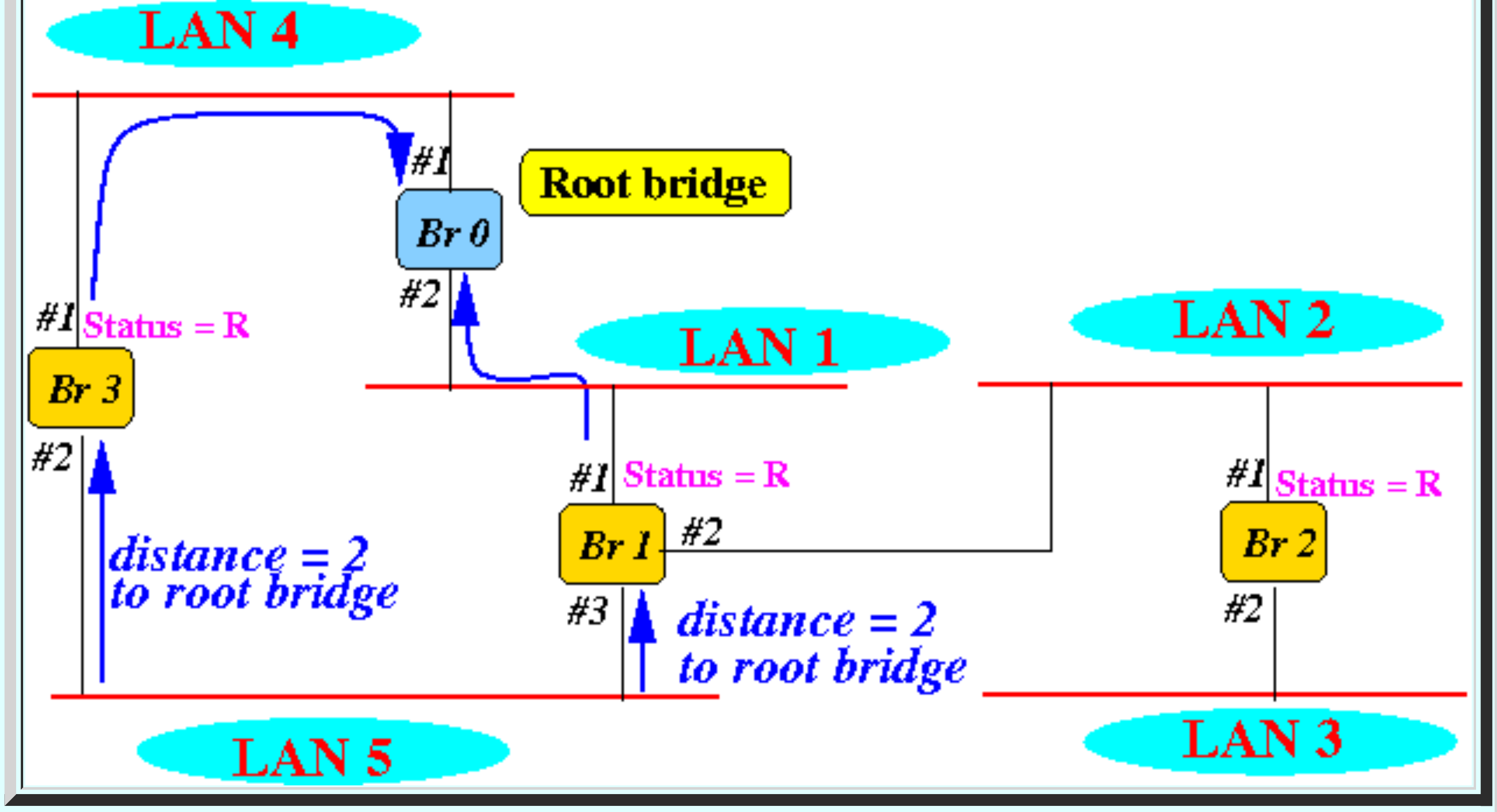


Note:

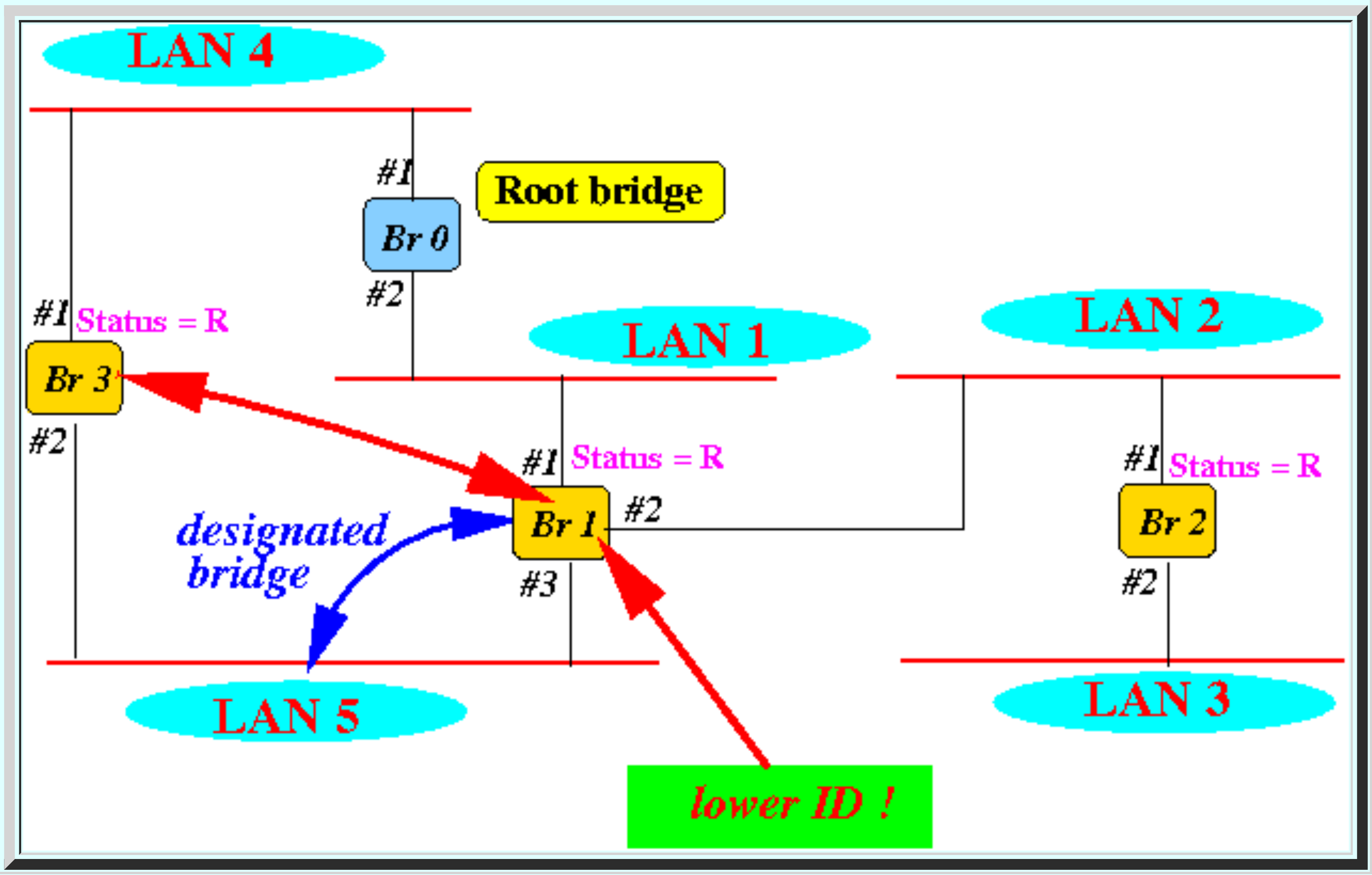
- The ***root* bridge** is ***also***:
  - the ***designated* bridge**

  for **LANS 1** and **4**

- ***Multiple* shortest paths**:
  - ***If*** there are ***multiple* shortest paths** from a **LAN** to the **root bridge**, **then**
    - The **designated bridge** for a **LAN** = the **bridge** with the ***smallest* ID**
- **Example:**
  - **LAN 5** has ***multiple* shortest paths** to the **root bridge**:



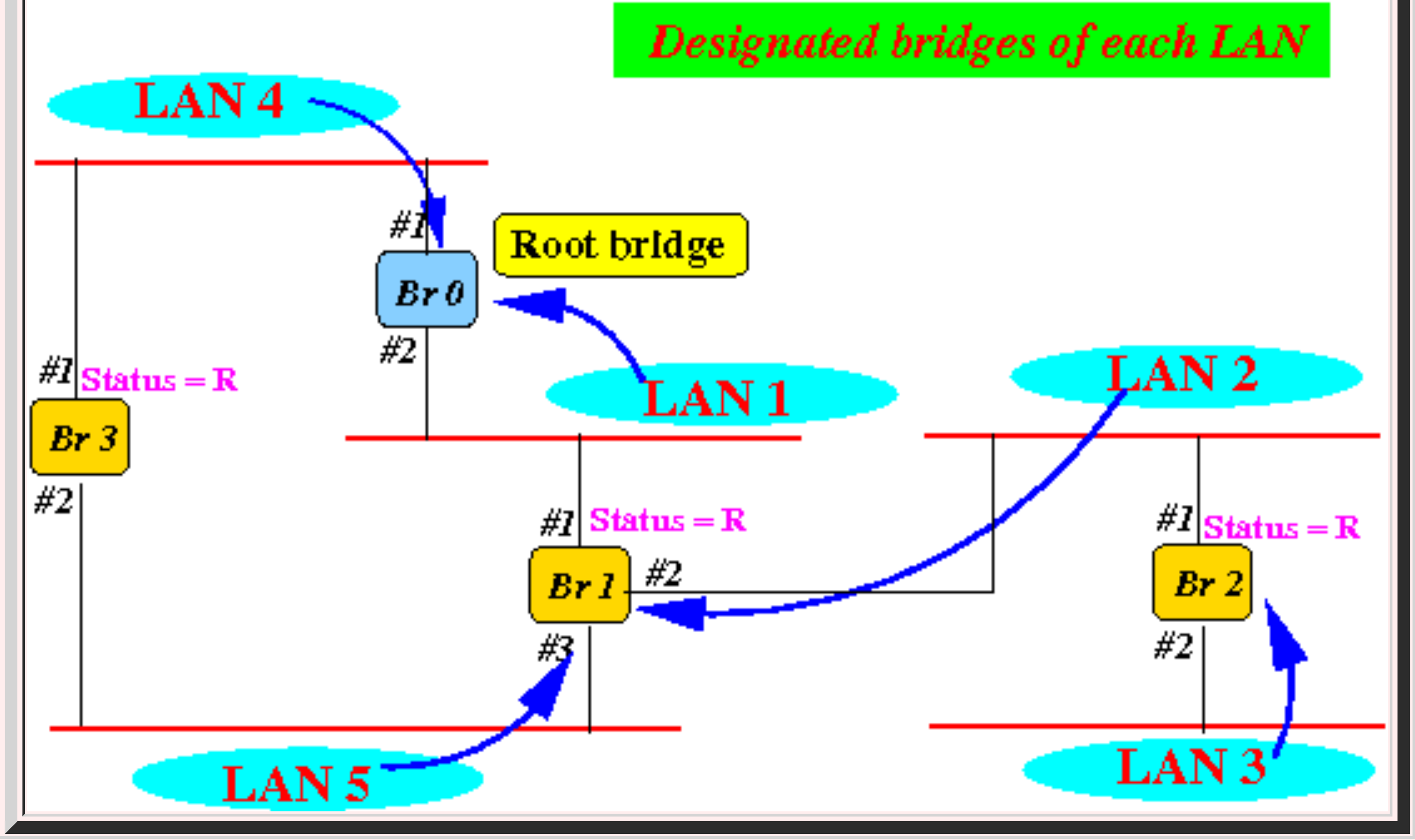
The **designated bridge** for **LAN 5** is **bridge 1** which has the ***smallest* ID**:



- **Notice that:**

  - ***Every* LAN (segment)** has a ***unique* designated bridge**

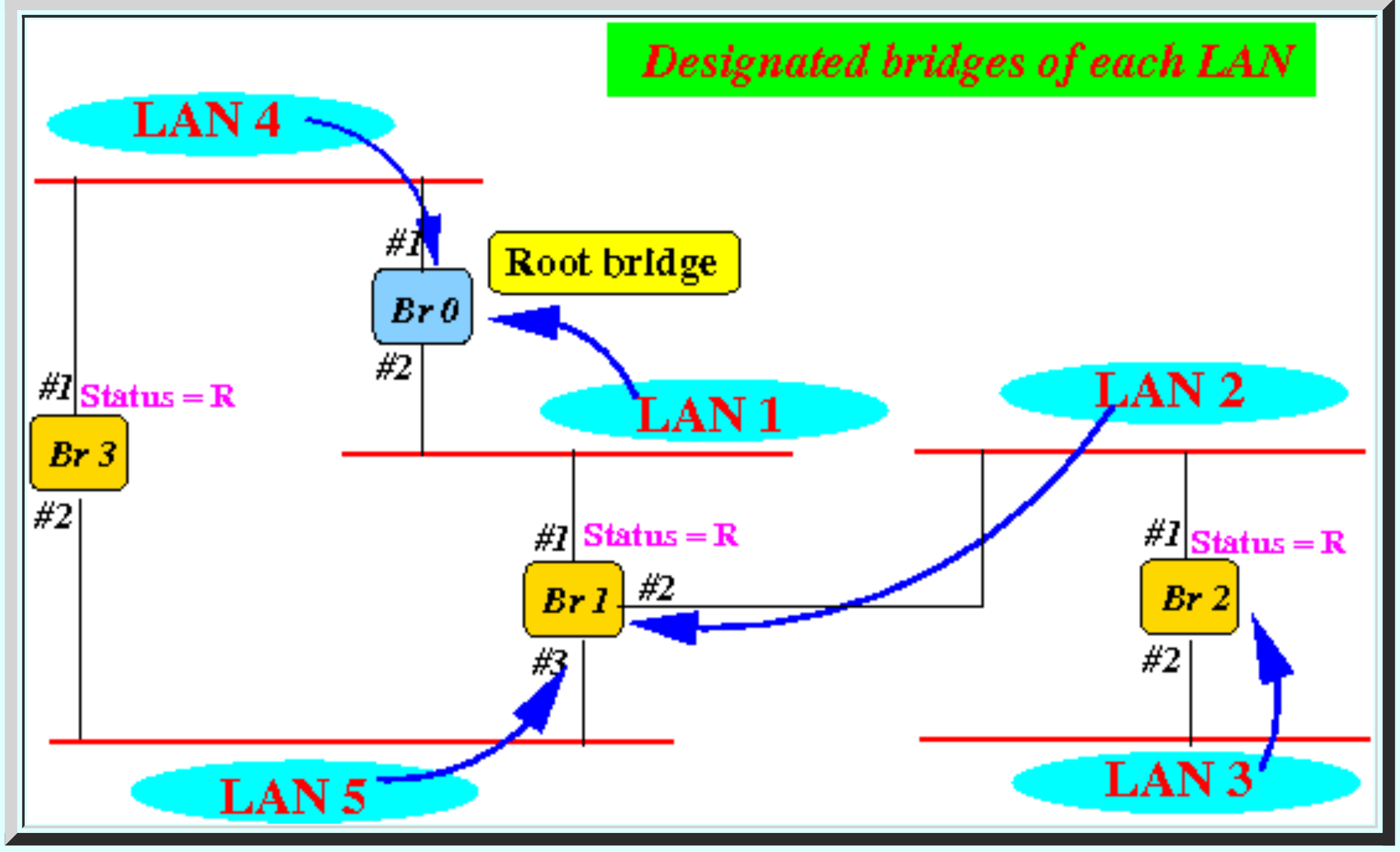
    Example:



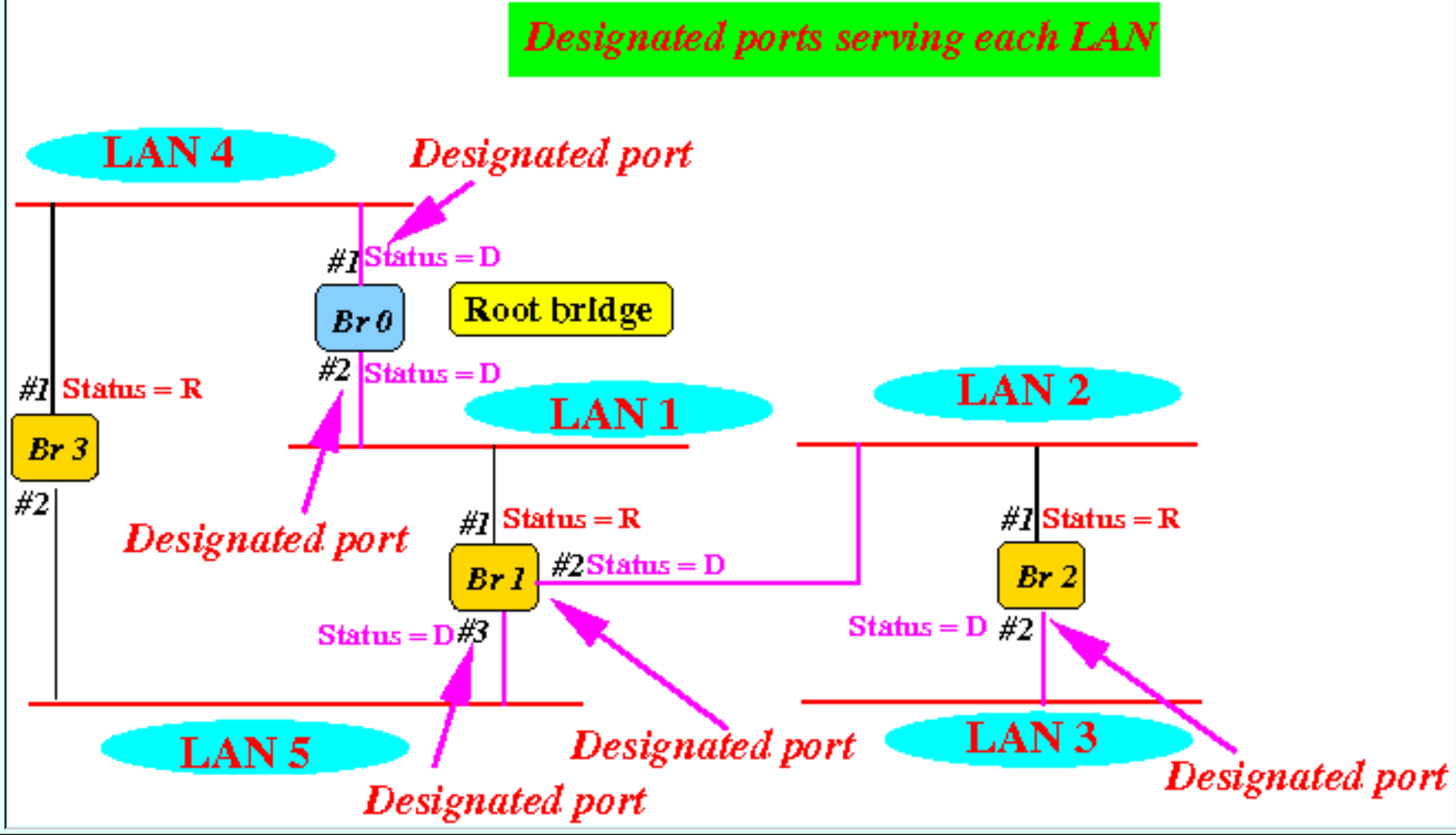
* **Designated ports**
  * **Designated *port*:**
    * **Designated port** = the **port** of a ***designated* bridge** that **connects** to the **LAN** (on the ***shortest* path** to the ***root* bridge**)

Example:

* **Previously**, we have **identifid** the **designated *bridges*** for each **LAN**:



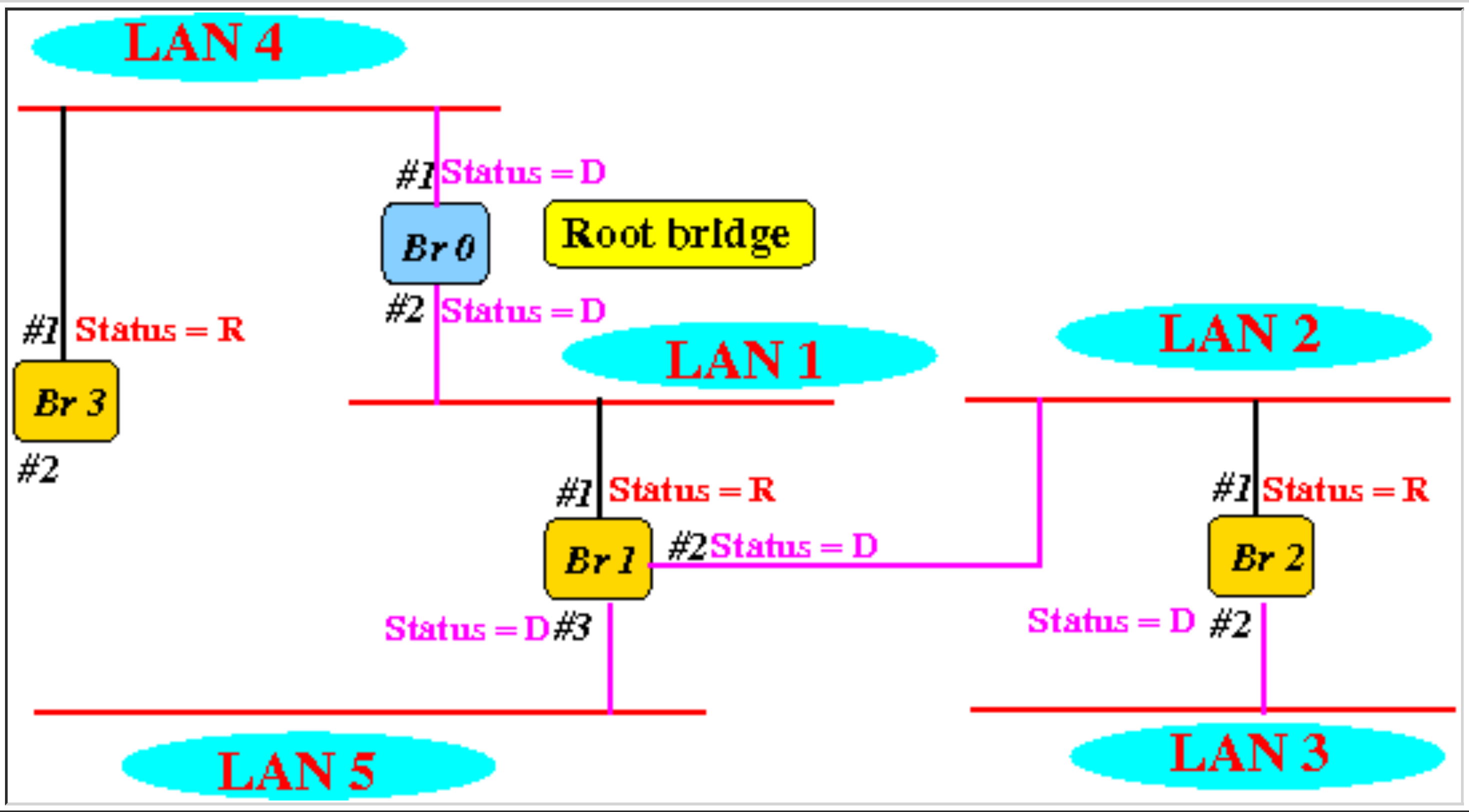
* The **designated *port*** are **marked** by **Status = D** in the **following figure**:



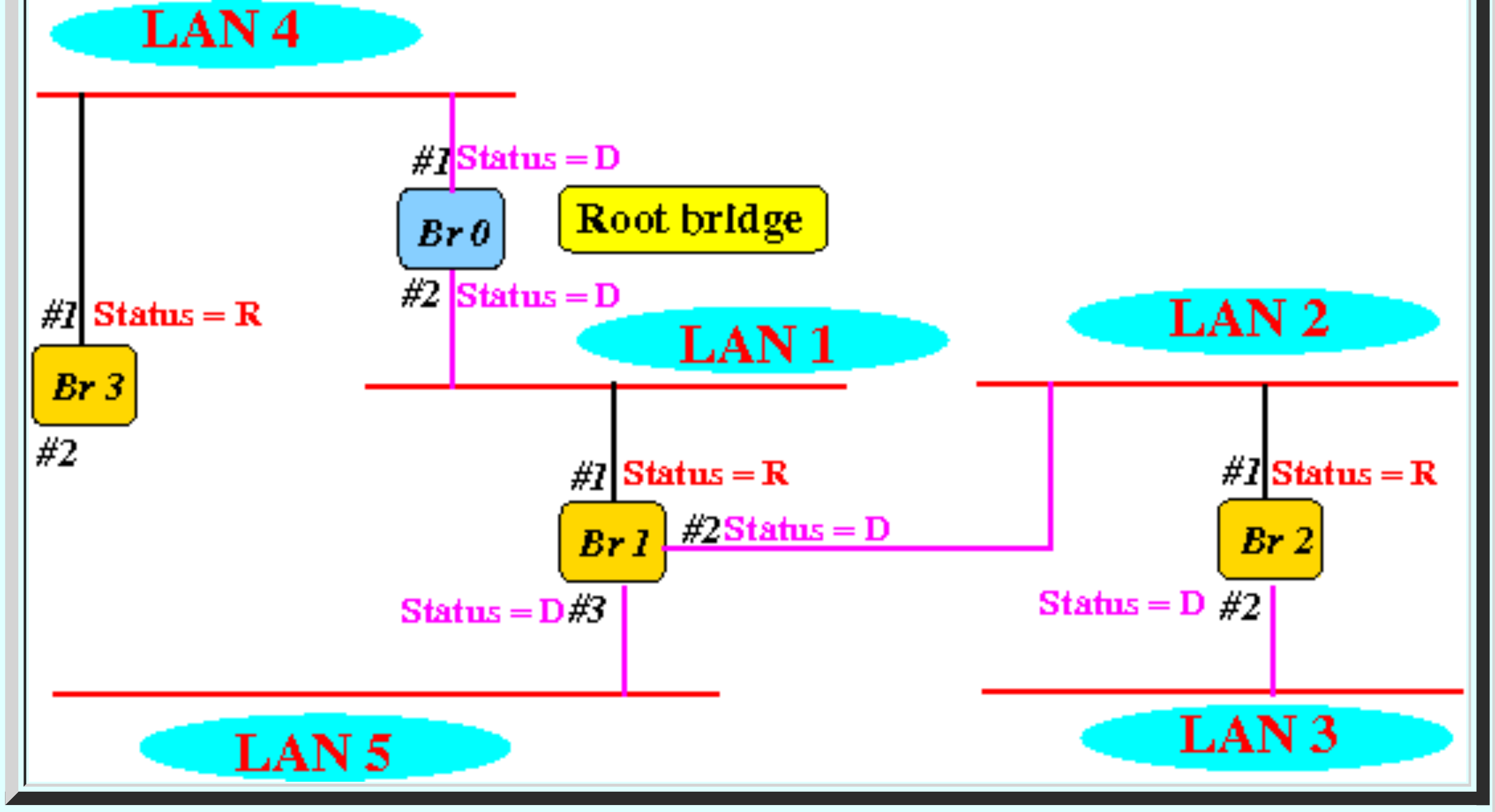
* **The spanning *tree* constructed by the IEEE 802.1D algorithm**
    * **Consider** now the ***logical* network** consisting of:

        1. ***All*** the **LAN (segments)**,
        2. The **root *bridge*** and all **designated *bridges***
        3. The **root *ports*** and all **designated *ports***

        This is that ***logical* network**:



    * **Notice that:**



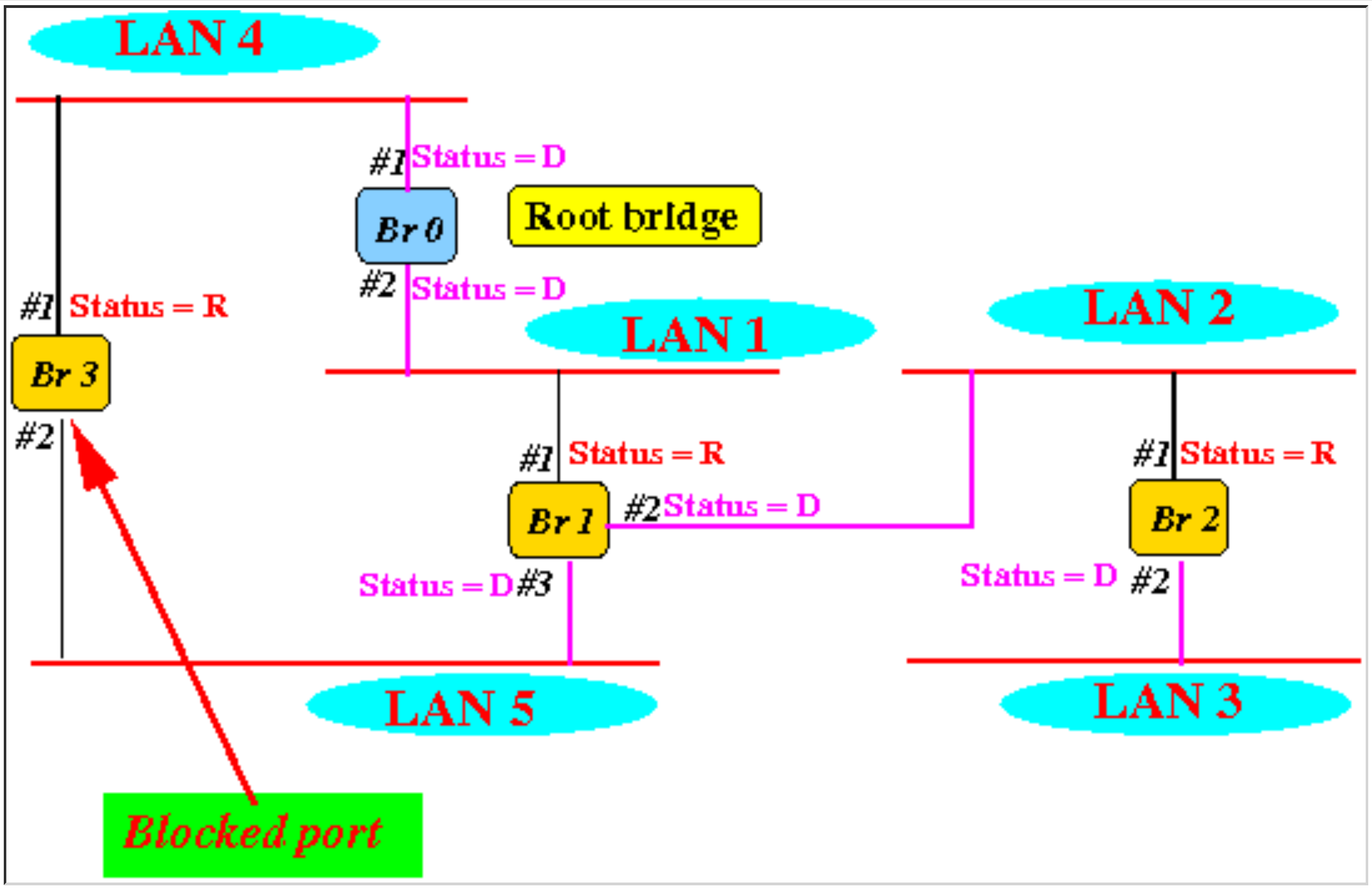
- **Contain *all* LANS**
- The ***logical* network** is a ***tree*** !!!

Blocked ports

- **Blocked ports**
  - **Blocked port:**

    - **Blocked port** = the ***remaining* ports** of **bridges** that are **neither**:
        - a **root port** **nor**
        - a **designated port**

    Example:



- **Operation of a blocked port**

    Note:

    - **Blocked port** is ***not*** a **"dead" port** !!!

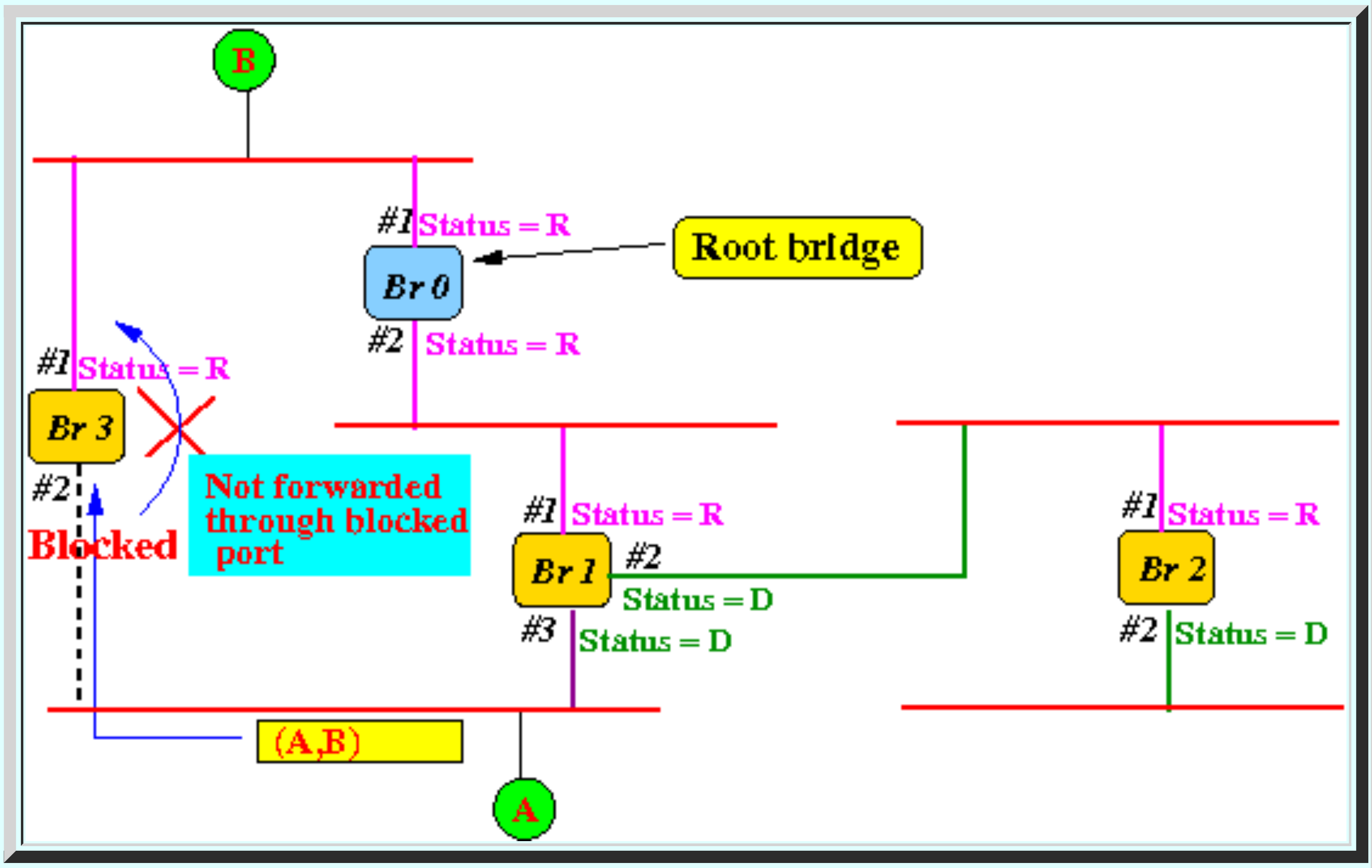
    - **Block port** = a ***stand-by* port** !!!

  - **Operation** of a **Blocked port**:

- A **frame** received on a **blocked port**:

  - will ***not* be forwarded** on ***any* other ports** of the **bridge**

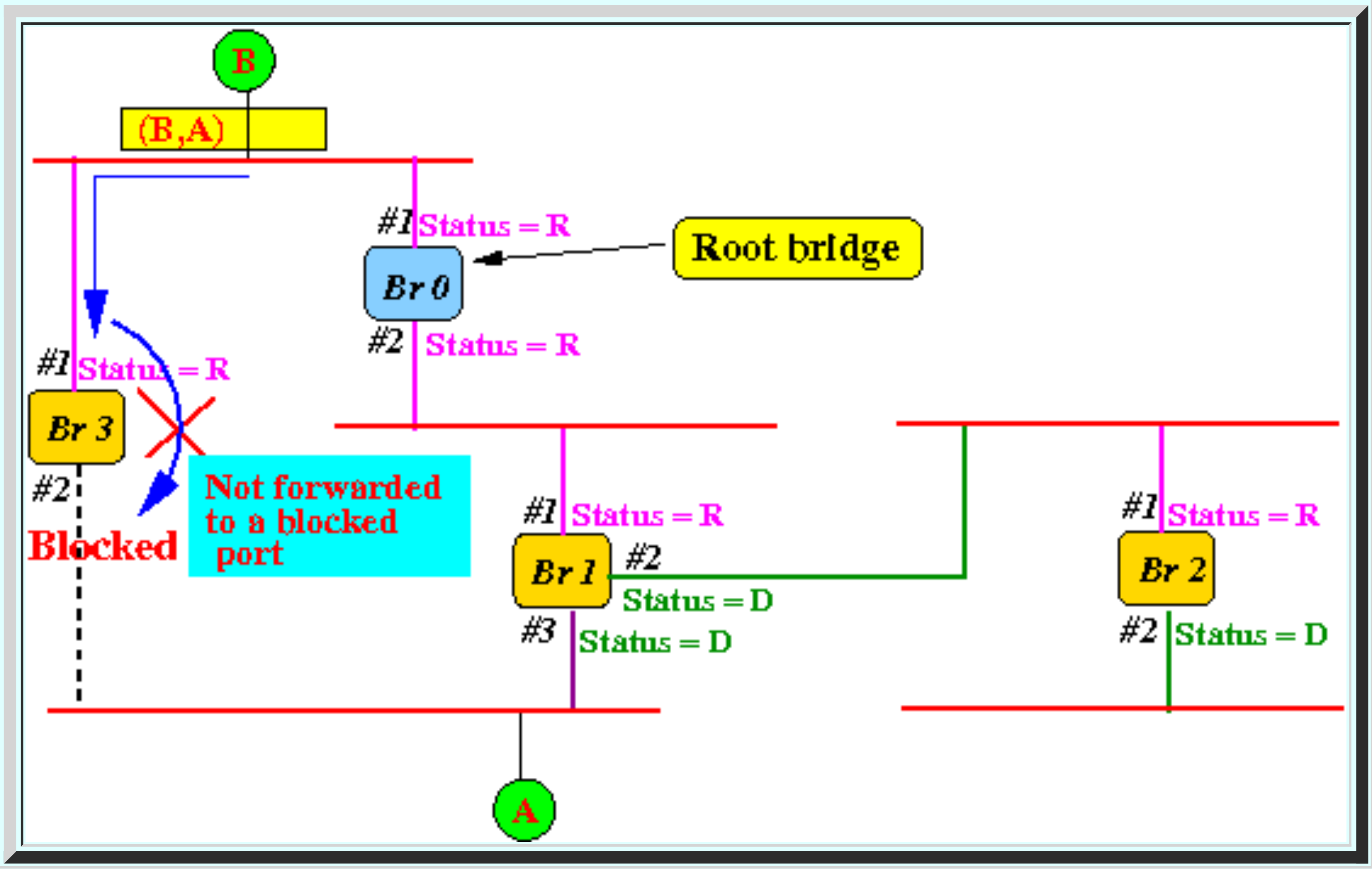
Example:



- A **frame** received on a ***non*-blocked port** will:

  - ***not*** be **forwarded** onto a ***blocked* port**

Example:

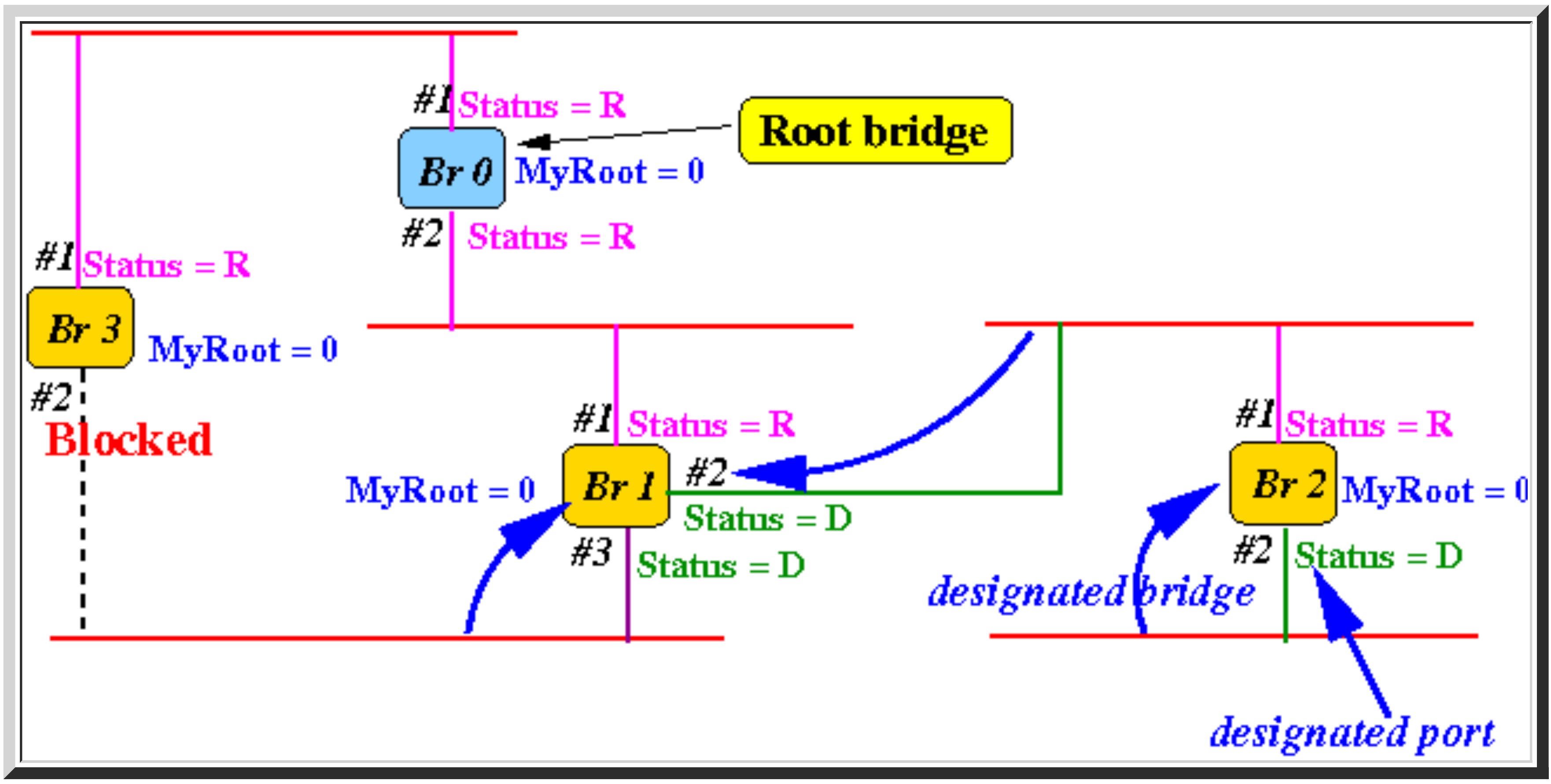


- So ***logically speaking***:

  - A ***blocked* port** is ***detached* (= disconnected)** from the **network** !!!

- **Summary**

  - **Summary of the above concepts:**



State information and Configuration control messages

- ***State information*** **(data structure) maintained by a bridge in IEEE 802.1D Spanning Algorithm**
  - ***Each*** **bridge** maintains the following **state variables**:
    - **MyRoot** = the **ID** of the ***root*** **bridge** that the **bridge** has **discovered** so far

      ---
    - **Root Path Cost** = the **cost (= distance)**, (in number of ***hops***) to the **root bridge**

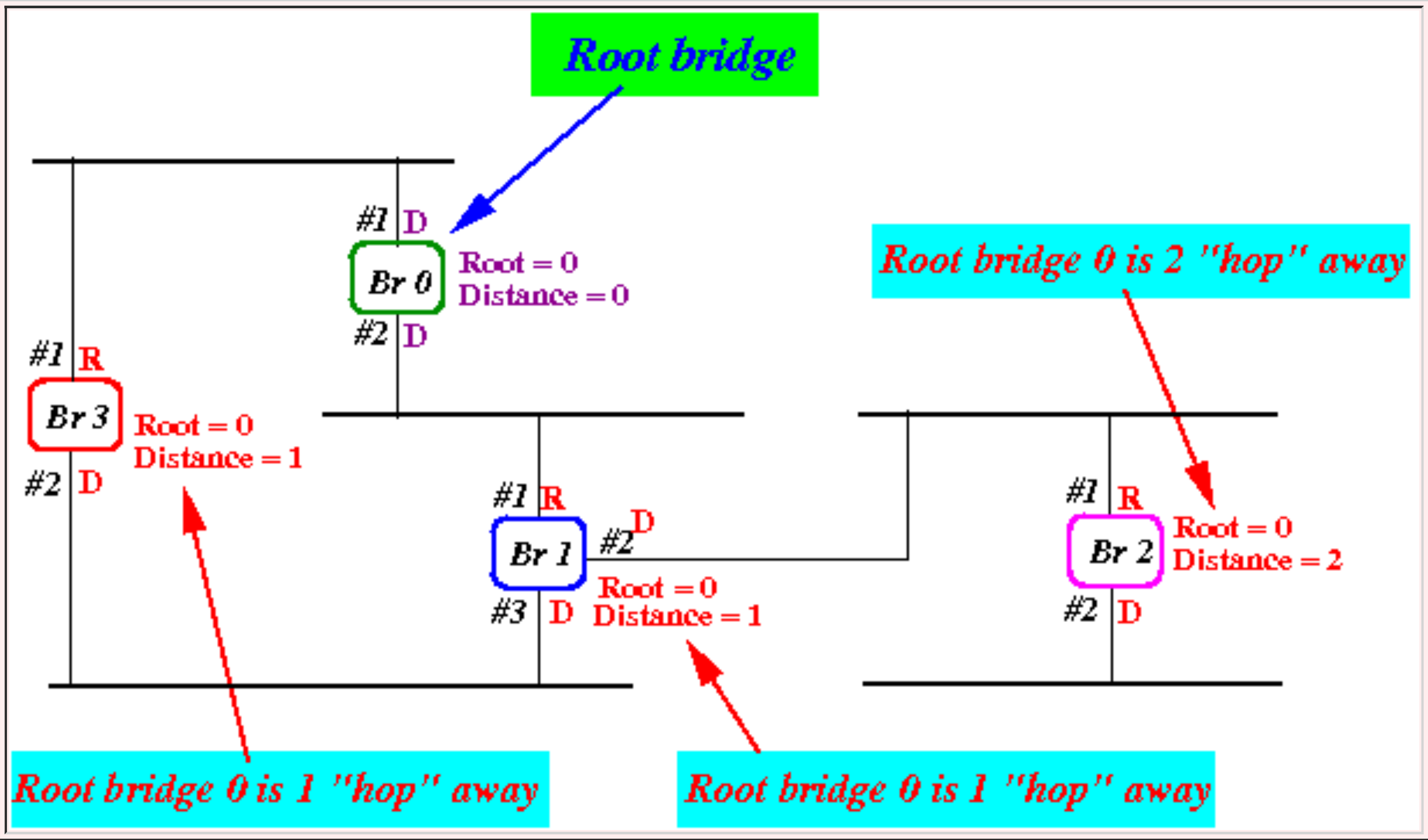
    - **Port Status** = the **status** of a **port**

      Possible states:

      - **R** = the **port** is a **root port**
      - **D** = the **port** is a **designated port**
      - **B** = the **port** is a **blocked port**

    Example:

    - **Sample state information** of **bridges**:

      Note:

      - The ***root*** **bridge** is **bridge 0**

      - The `Distance` variable is **equal** to the ***number*** **of hops** to the ***root*** **bridge**

      - The ***root*** **ports** of **each bridge** is the **direction** of the ***shortest*** **path** to get to the ***root*** **bridge**

- **Initialization**

- System initialization:

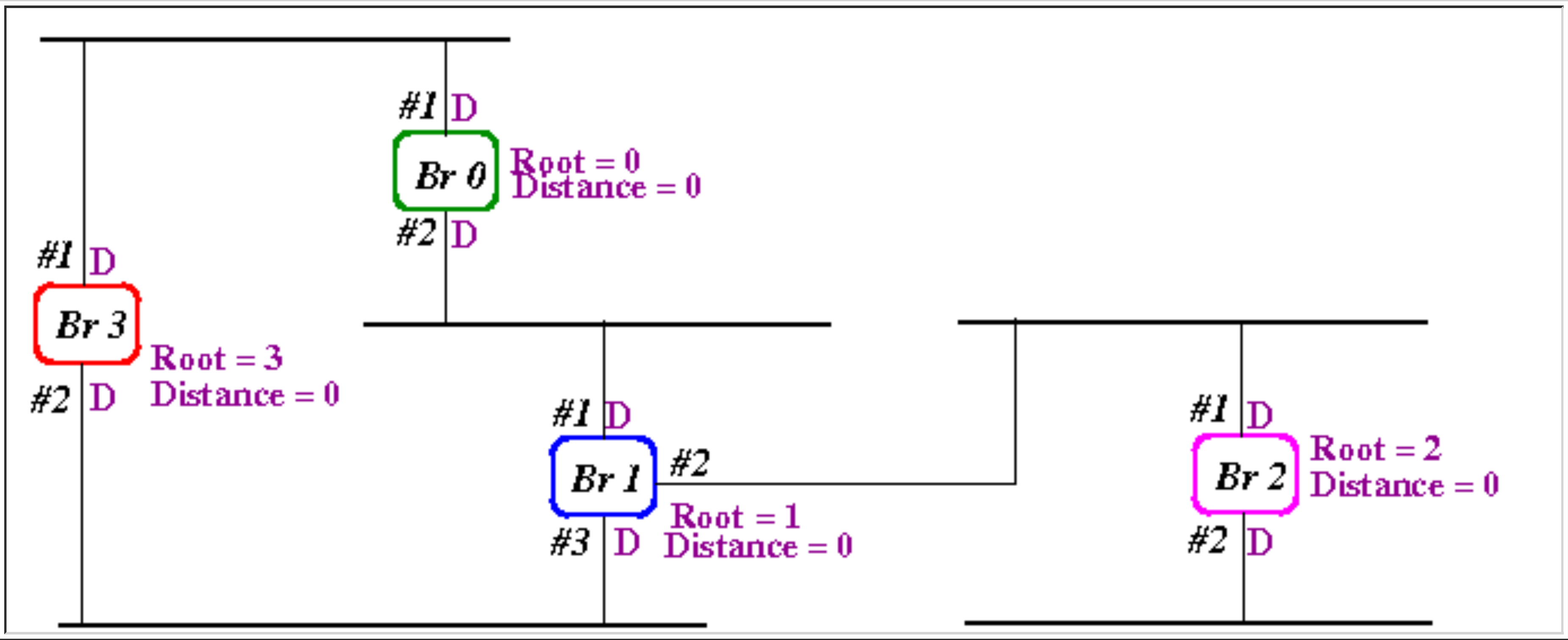
  - When a **bridge starts up**, the **bridge** will **initializes** its **state variables** to:

```
RootID   = myID;        // Bridge assumes it is the root bridge

Distance = 0;           // Distance to the root bridge = 0

for ( each port p )
{
    status[p] = D;      // Each port is "designated"
}
```

    I.e.:

    - The **bridge** assumes that ***it*** is the ***root* bridge** !!!

Example:



- **Overview of the Spanning tree algorithm**

  - **Overview** of the **Spanning Tree** algorithm:

    - **Bridges** in the **network** transmits ***configuration control* messages** (**only**) to its ***neighbor* nodes**

    - When a **bridge** receives a ***configuration control* messages**

      - The **bridge** will **process** the **configuration data** and **may update** some of its **state variables**

    - If the **node** has ***updated*** some of its **state variables**:

      - The **bridge** will **transmit** its ***new* configuration** in a **configuration message** to ***all* its neighbors**

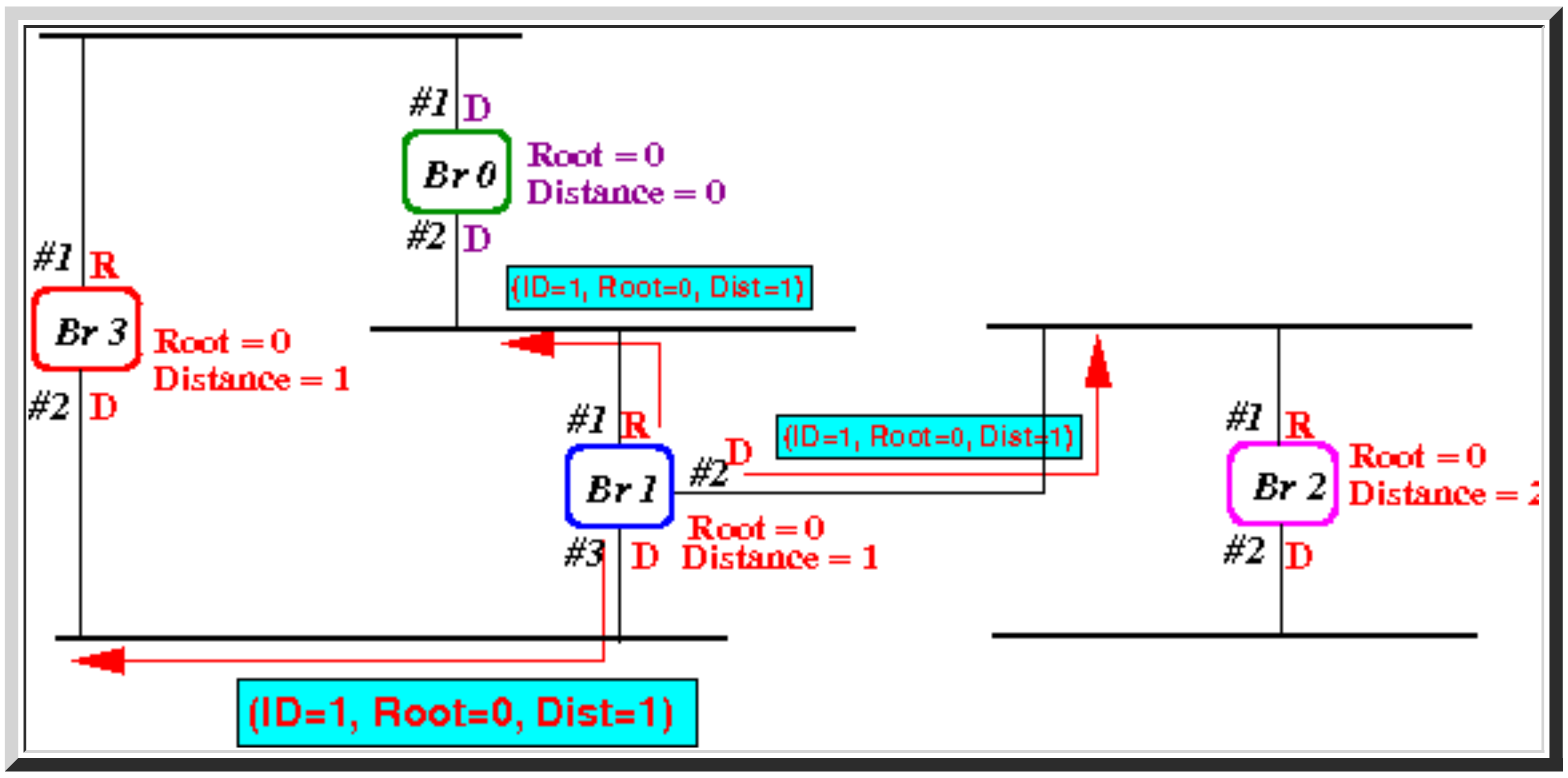
        (**Except** to the **neighbor** that **sent** the **configuration message**)

- **Structure of *Configuration Control Messages*** **(exchanged in the IEEE 802.1D algorithm)**

- The **structure** of the **Configuration Control Message** is as follows:

My ID	My Root bridge	My Distance to root

  Example:



  Explanation:

  - **MyID** = the **ID** of the **bridge** that sends the **configuration message**
  - **MyRootBridge** = the **ID** of the **root bridge** that the **bridge** has **discovered so far**
  - **MyDistanceToRoot** = the **distance (# hops)** of the **bridge** to the **root bridge**

- **Purpose** of **Configuration Control Message**:

  - **Enables** a ***neighbor*** **bridge** to **identify** the ***root bridge***
  - **Enables** a ***neighbor*** **bridge** to **compute** the ***shortest distance*** from itself to the **root bridge**
  - **Enables** a ***neighbor*** **bridge** to **determine** its ***root*** **port**
  - **Enables** a ***neighbor*** **bridge** to **determine** all its ***designated*** **ports**
  - **Enables** a ***neighbor*** **bridge** to **determine** all its ***blocked*** **ports**

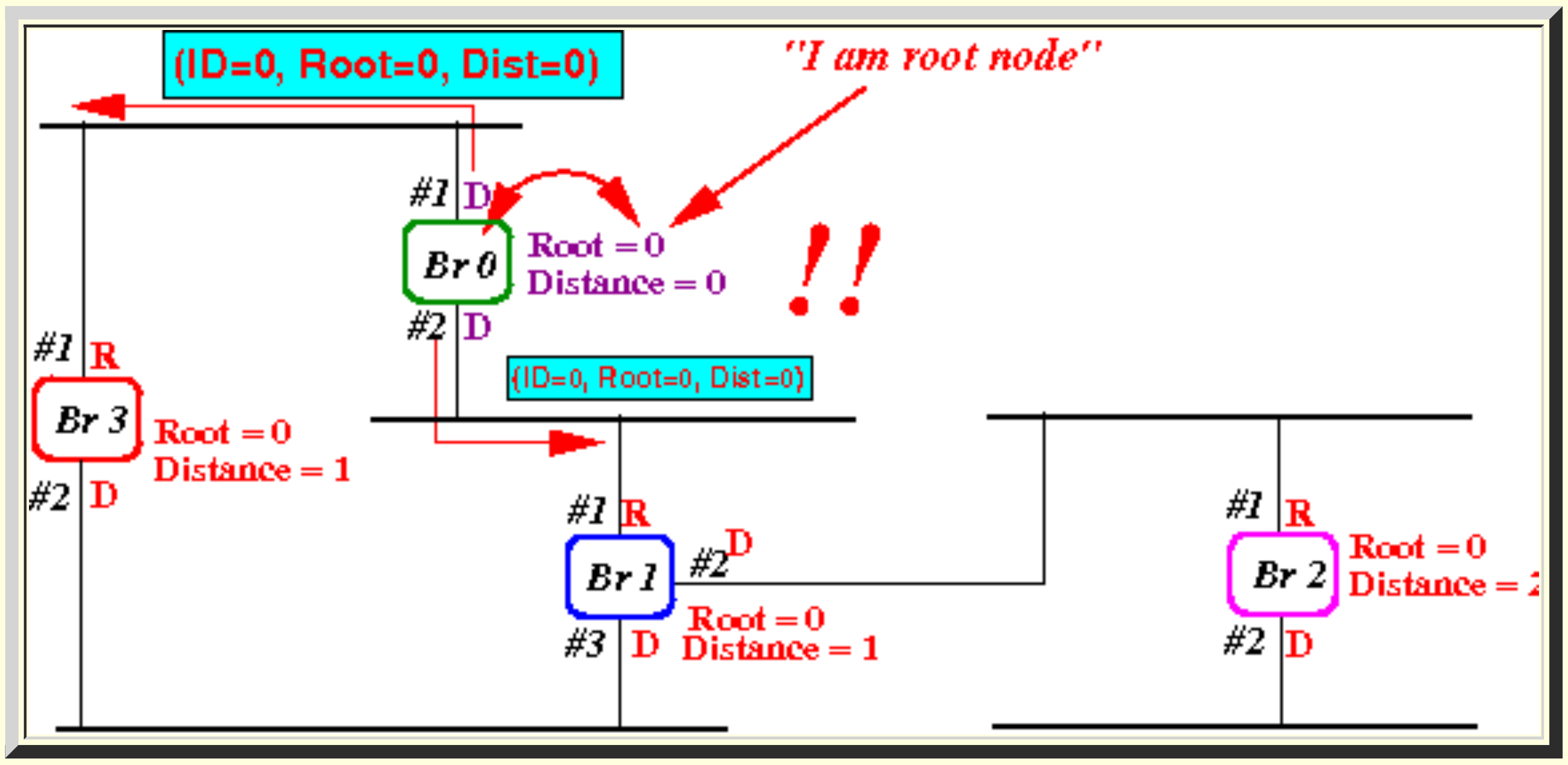
- **When to *send* a configuration control message**

  - **When** does a **node** send a **configuration control message**:

    1. A ***root*** **node** (i.e.: a **node** with `RootID == myID`) will:

       - ***periodically*** transmit a **configuration control message**

Example:



---

2. A ***non-root* node** (i.e.: a **node** with **`RootID ≠ myID`**) will **transmit** a **configuration message**:

    - When ***some*** of **its *state* variables**:

        - **`RootID`**
        - **`DistanceToRoot`**
        - **`status[port]`**)

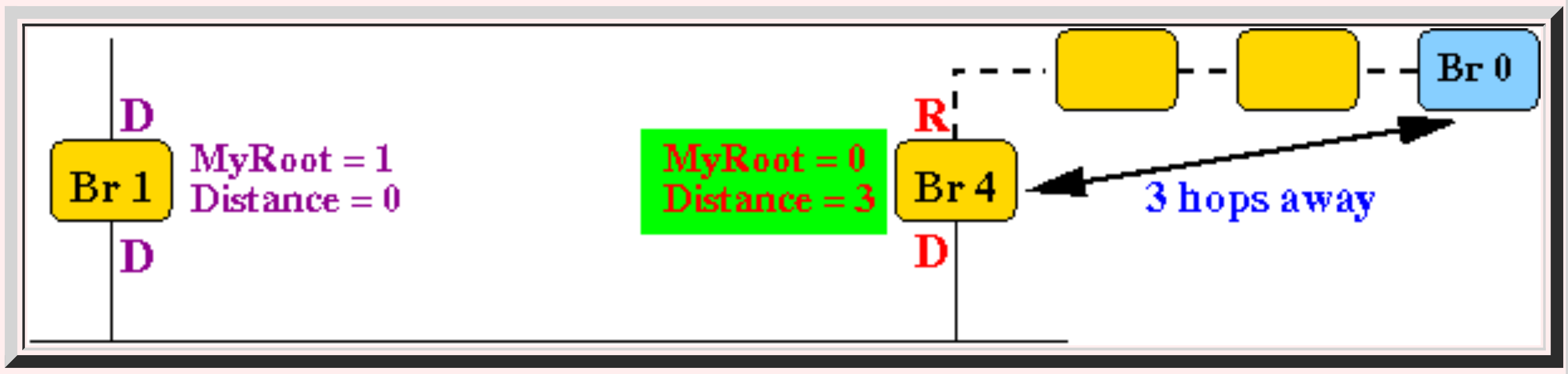
      has **changed** its **value**

Example:

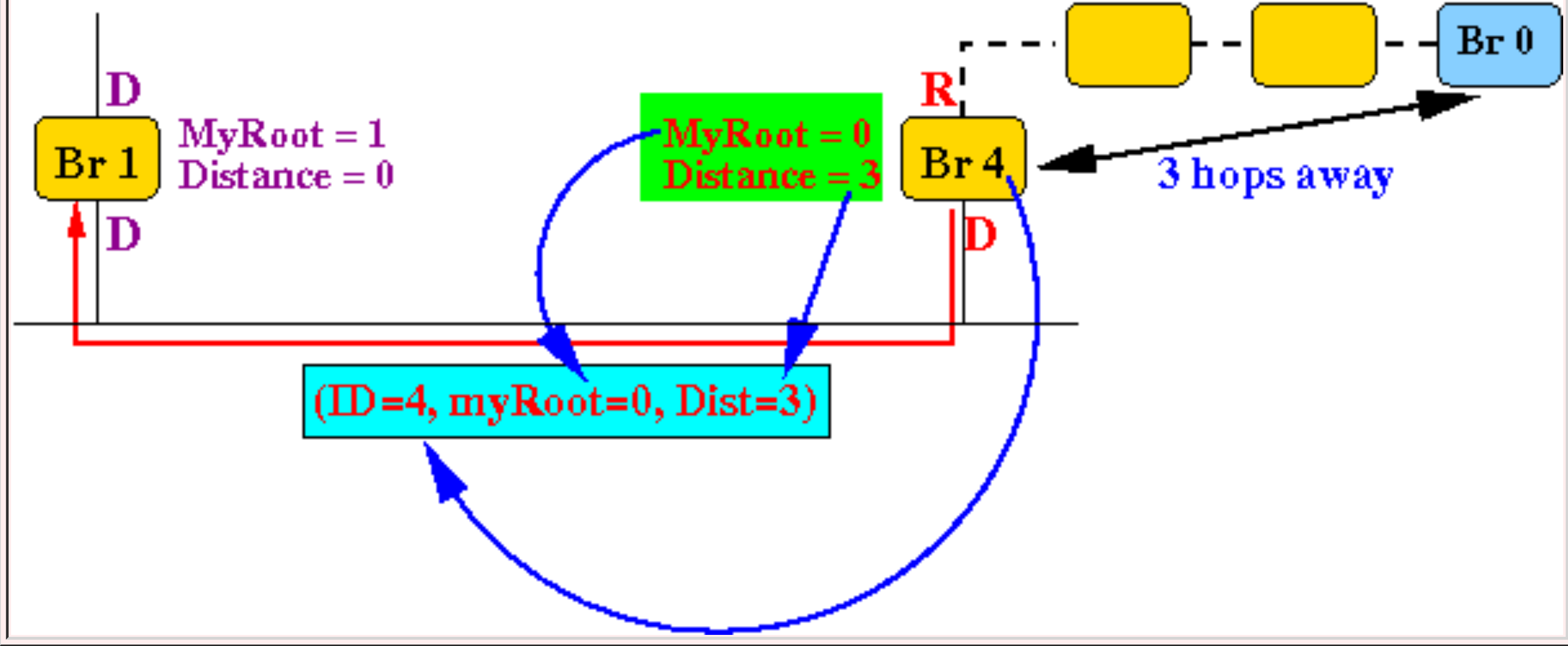
- **Suppose** the **bridge 4** just ***updated*** (**some** of) its **state variables** to:

```
MyRoot    = 0
Distance  = 3
```

  Its **new state** is:



- Then **bridge 4** will transmit the following **configuration message** to **all its neighbors** --- **except** to the **neighbor node** who **sent** the **configuration message**:

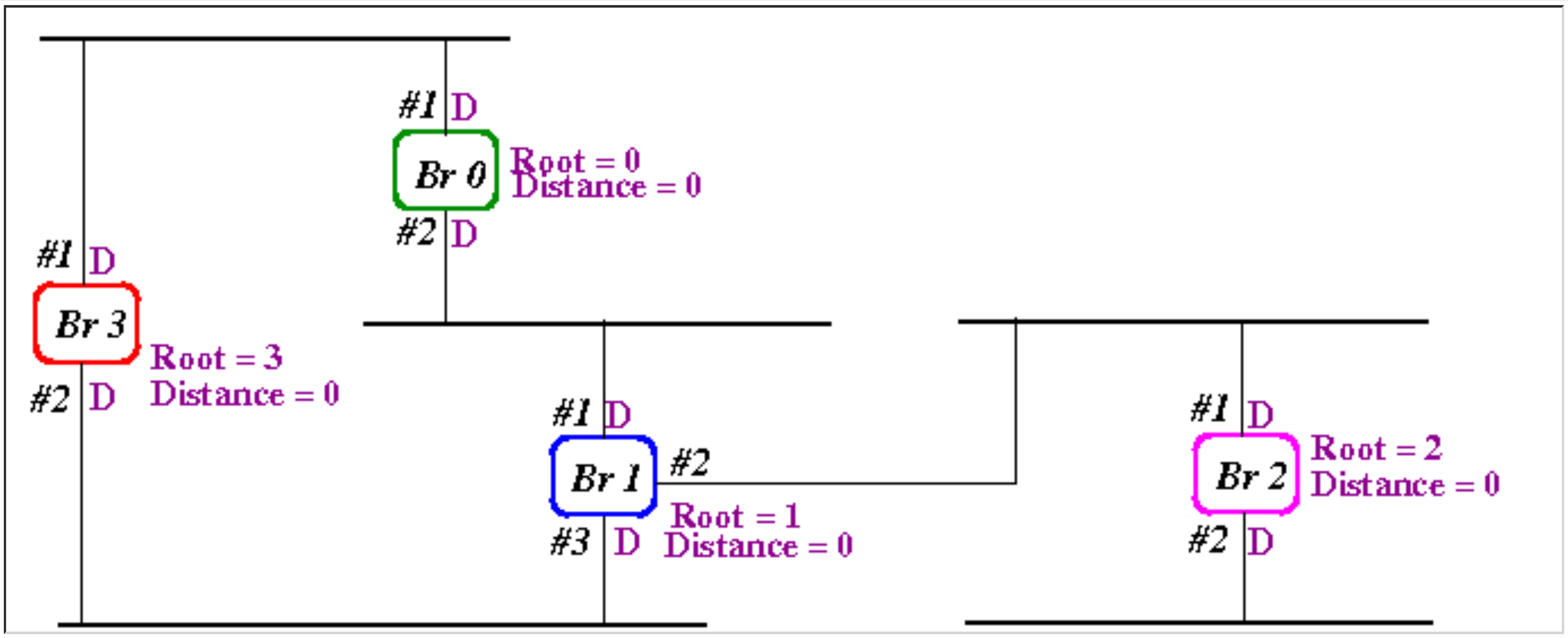


D
Br 1
MyRoot = 1
Distance = 0
D
MyRoot = 0
Distance = 3
R
Br 4
D
Br 0
3 hops away
(ID=4, myRoot=0, Dist=3)

Processing a configuration control message

- **Recall: purpose of the configuration messages**
  - **Recall**:
    - **Bridges exchange configuration messages** in order to:
      - **Find** the **tree** that is **rooted** at the **bridge** with the ***smallest* ID**

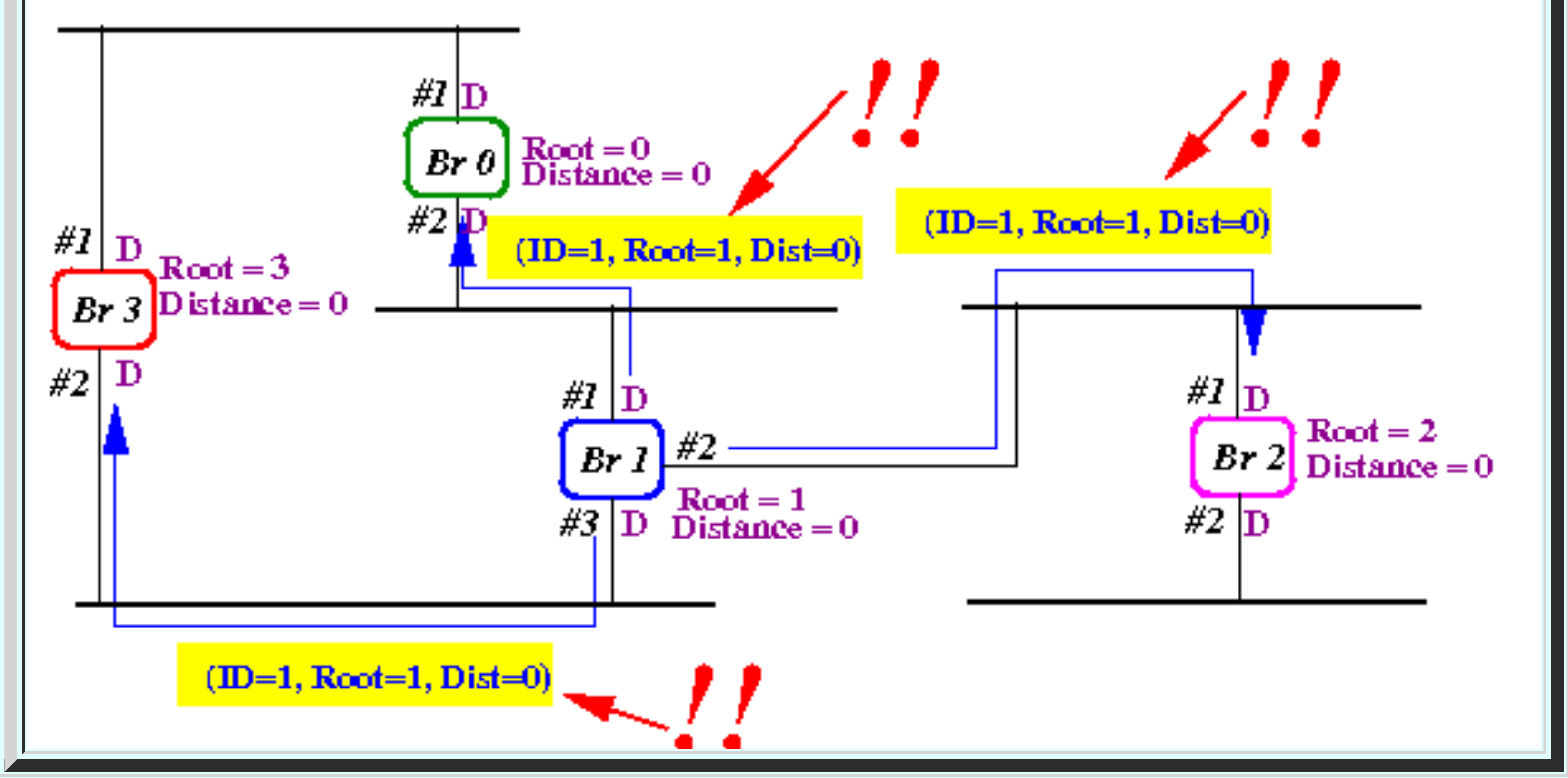
- **Processing a configuration message**
  - I will ***first* illustrate** the **processing** with a ***concrete* example**
  - I will **start** with the **initial state:**



  - **Suppose** that:
    - **bridge 1** sends the **configuration message**:

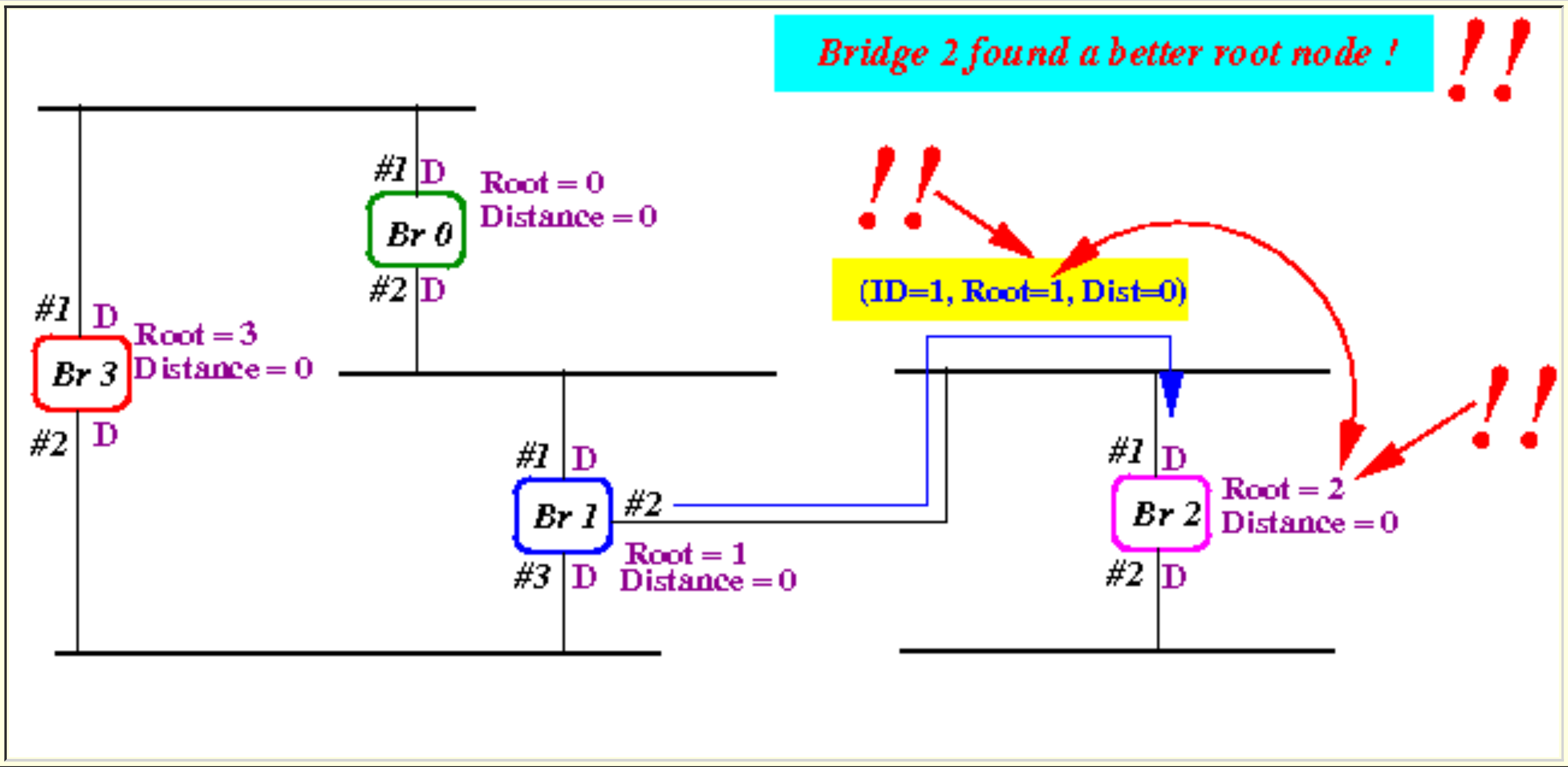
      ```
      (ID=1 , Root=1 , Dist=0)
      ```

      to ***all*** its **neighboring nodes**:



- *How* the **bridge 2** processes the **configuration message** from **bridge 1**:

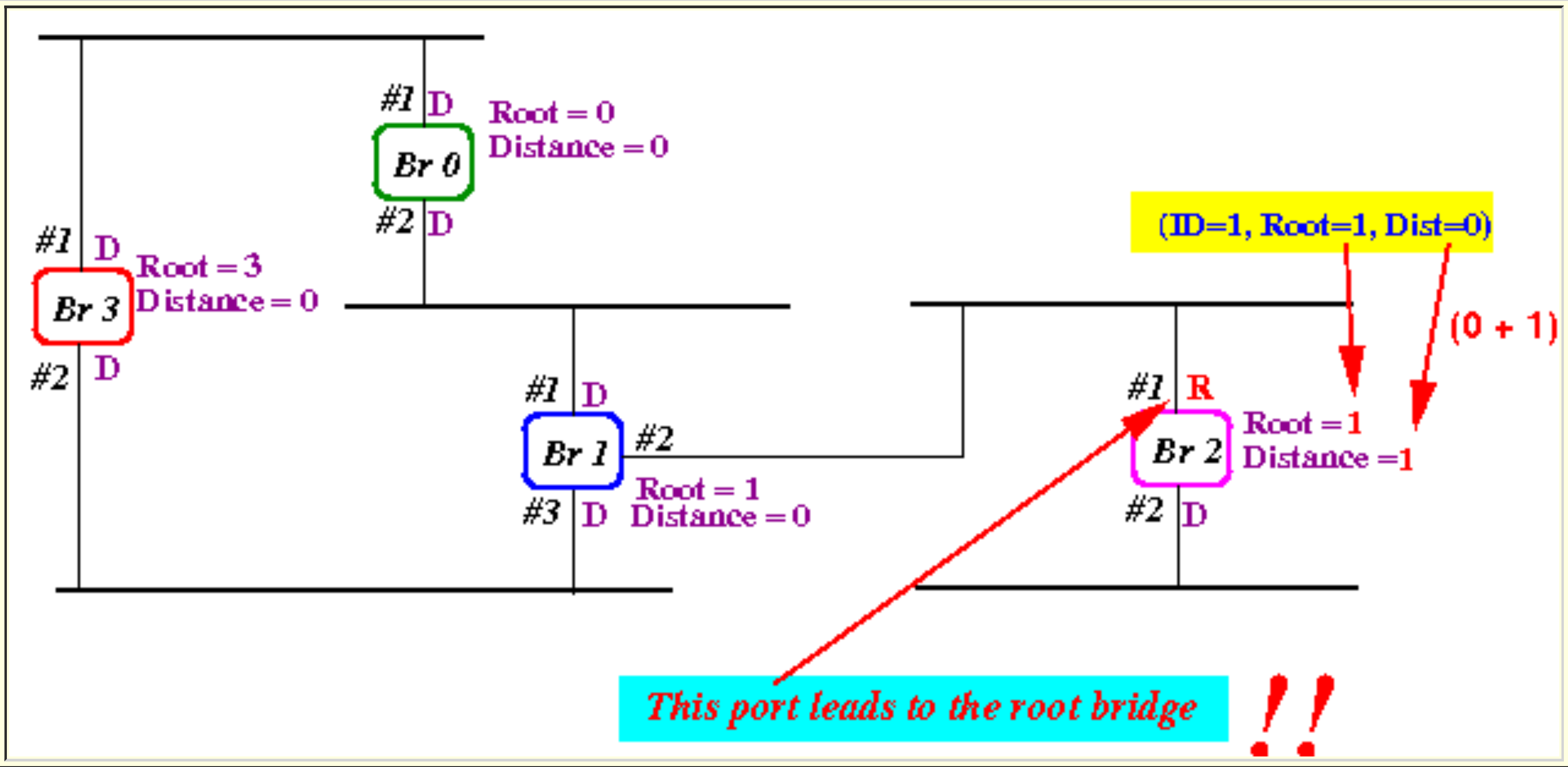
  - **Bridge 2** detects that the **information** in the **configuration message** can **improve** its ***current* state (= assumption)**:



Explanation:

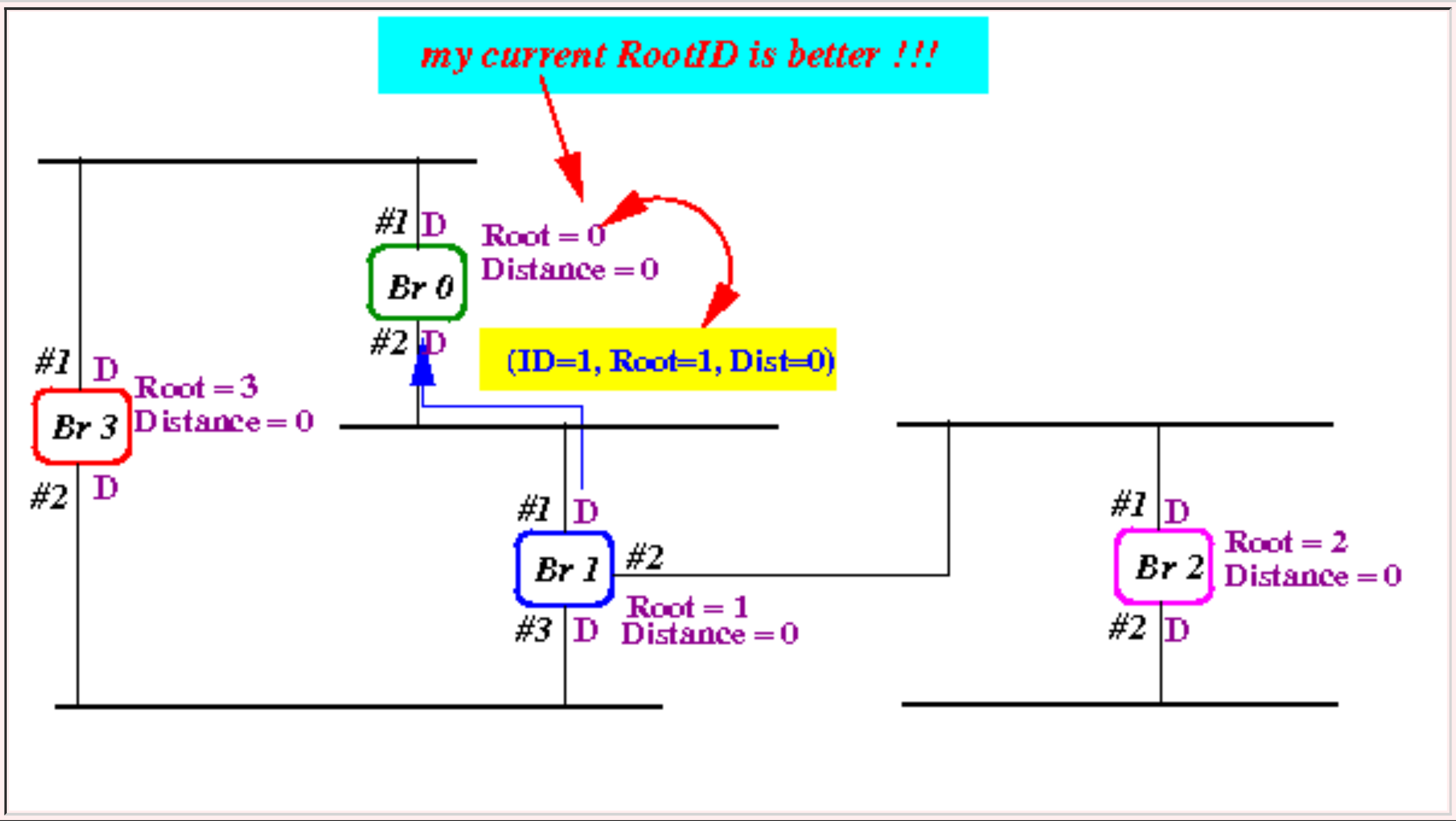
```
Bridge 2:
                         Root ID  = 2
                         Distance = 0

Configuration messages:
                         Root ID  = 1   <---- Smaller !!
                         Distance = 0

Bridge 2 has discovered a better candidate for the root bridge !!!

Bridge 2 will update its state variables to:

                          Root ID  = 1
                          Distance = 1    (= 0 + 1)
```

Result:



- *How* the **bridge 0** processes the **configuration message** from **bridge 1**:

  - **Bridge 0** detects that the **information** in the **configuration message** can ***not*** **improve** its ***current*** **state (= assumption)**:



Explanation:

```
Bridge 0:
                               Root ID  = 0
                               Distance = 0

Configuration messages:
                               Root ID  = 1   <---- Smaller !!
                               Distance = 0

Bridge 0 itself has the better candidate for the root bridge !!!

Bridge 2 will NOT update its state variables
```

Detecting and handling a *better* configuration

- ***When*** **is a new configuration *better* better than the current configuration**
  - There are **2 cases** that can **improve** the **current configuration**:

1. The **root bridge ID** in the ***new*** **configuration** is ***smaller***:

   - We have found a ***better*** **root bridge** !!!

   (Since the **root bridge** is the **bridge** with the ***smallest*** **ID**, the ***current*** **root brigde** is **actually** ***wrong*** !!!)

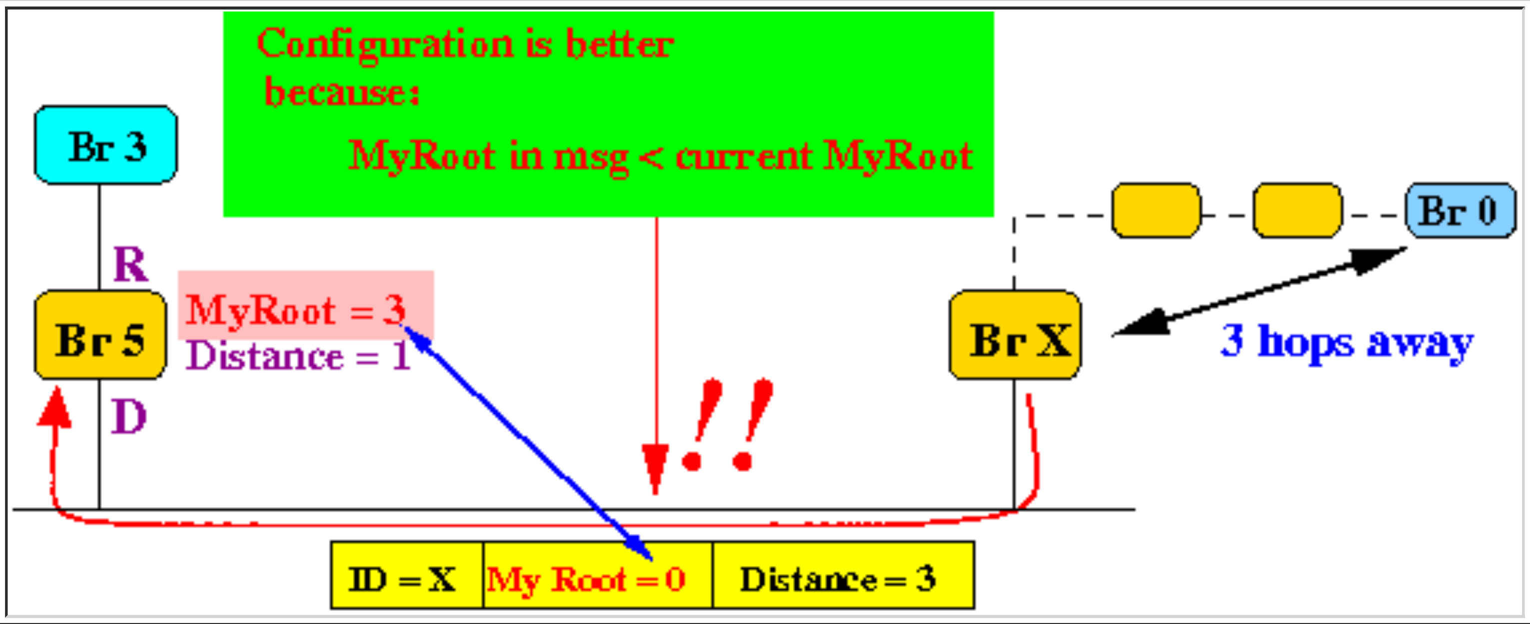
2. **A:** The **root bridge ID** is ***equal*** to the ***current*** **root bridge ID** ***and***
   B: The ***distance*** to the **root bridge** is ***smaller*** than the ***current*** **distance**:

   - We have found a ***shorter*** **route** to the **(same) root bridge** !!!

- **How to determine a *better* root bridge**
  - **Test** to find a **better root bridge**

    - **Root ID** of the ***control message*** **<** **MyRoot**

    Example:



  - **Actions** taken by the **bridge** when a ***better*** **root bridge** is **discovered**:

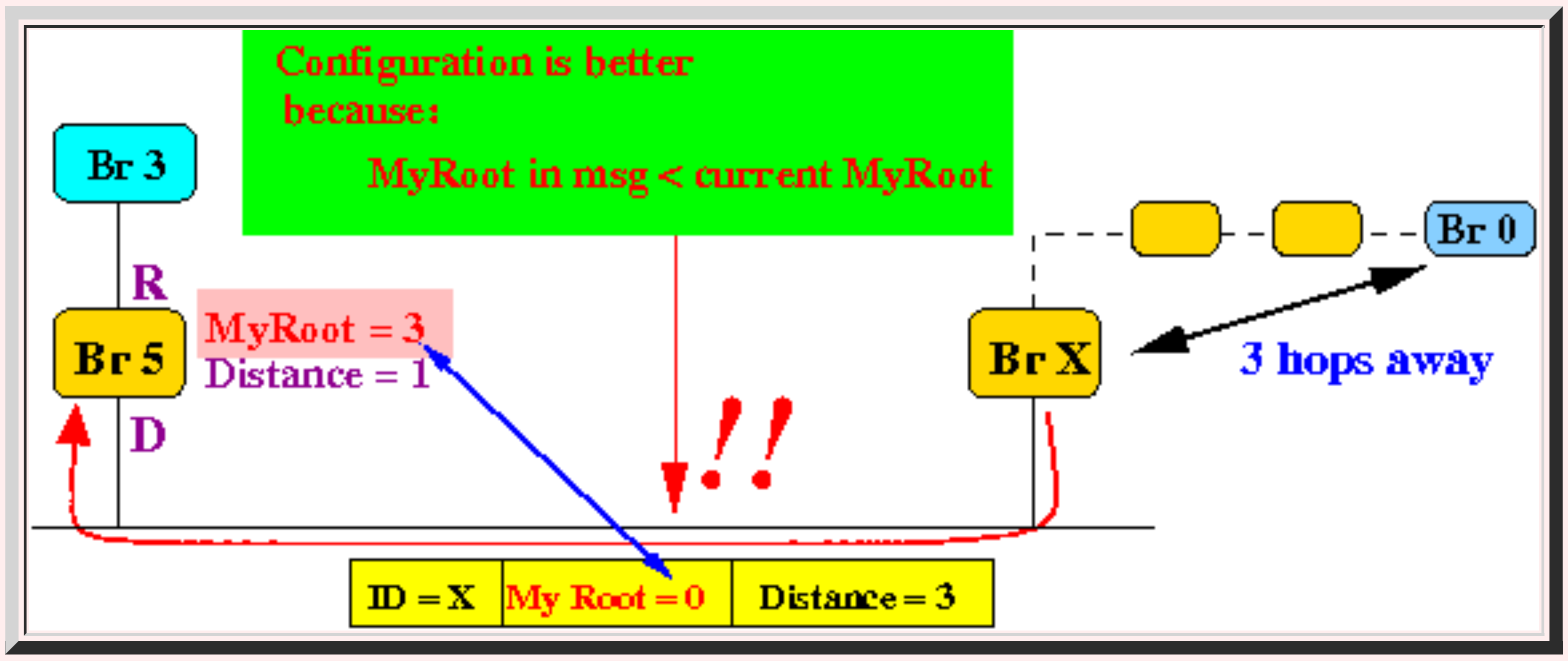
```
MyRoot = Root Bridge ID in configuration message;

Distance = Distance in configuration message + 1;       // One more hop !

State(Port on which configuration message was recv) = R (Root);

for ( all other ports P )
{
    State(P) = D (Designated);
    Transmit configuration msg "(myID, MyRoot, Distance)" on port P;
}
```

Example:

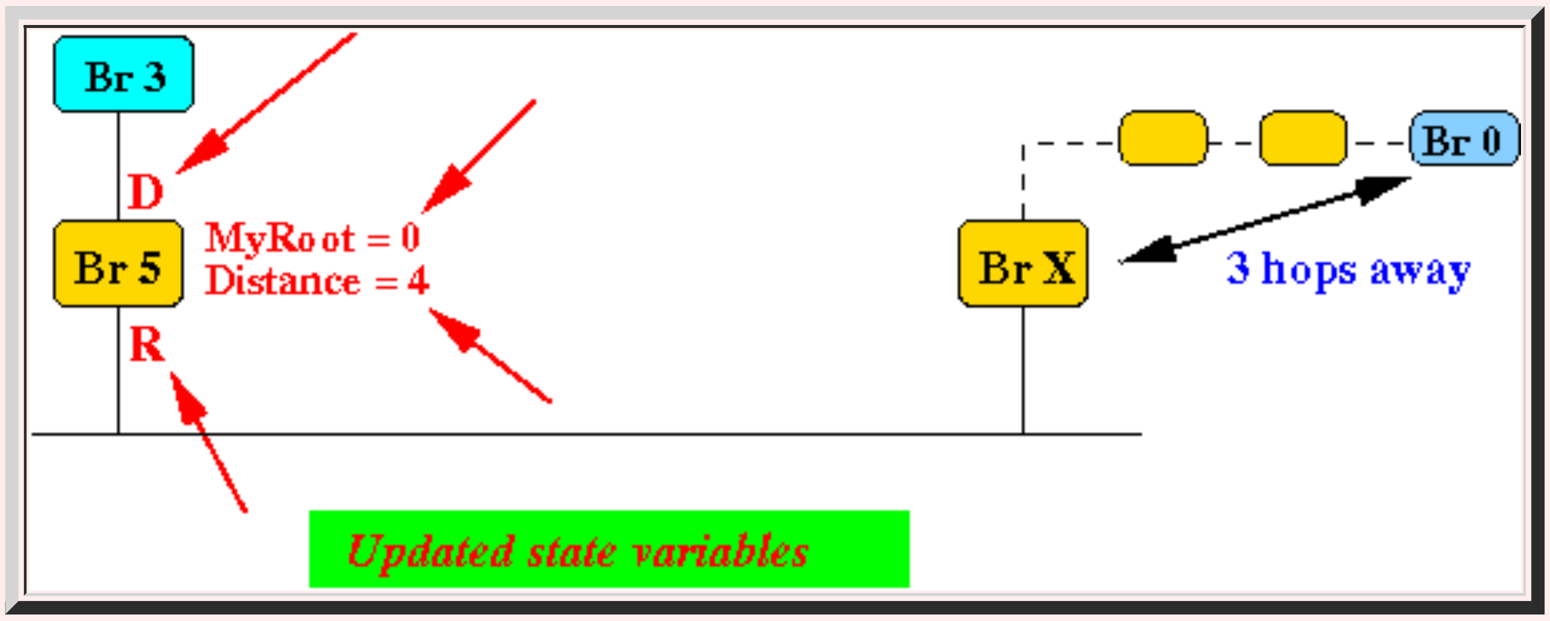
- **Bridge 5** has just **received** a **configuration message** with

```
MyRoot in configuration msg  <  Current Root
```

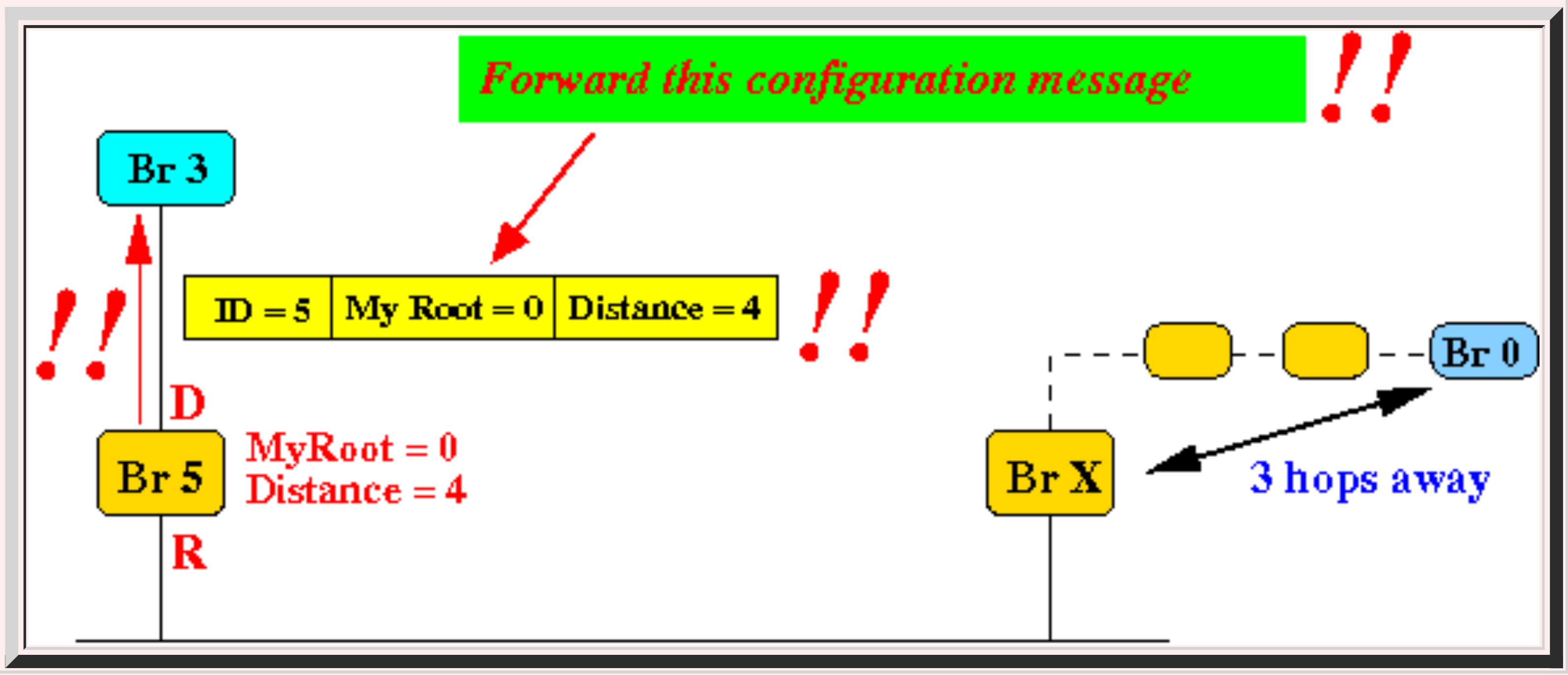
Scenario:



- **Bridge 5** will **update** its **state variables** to:



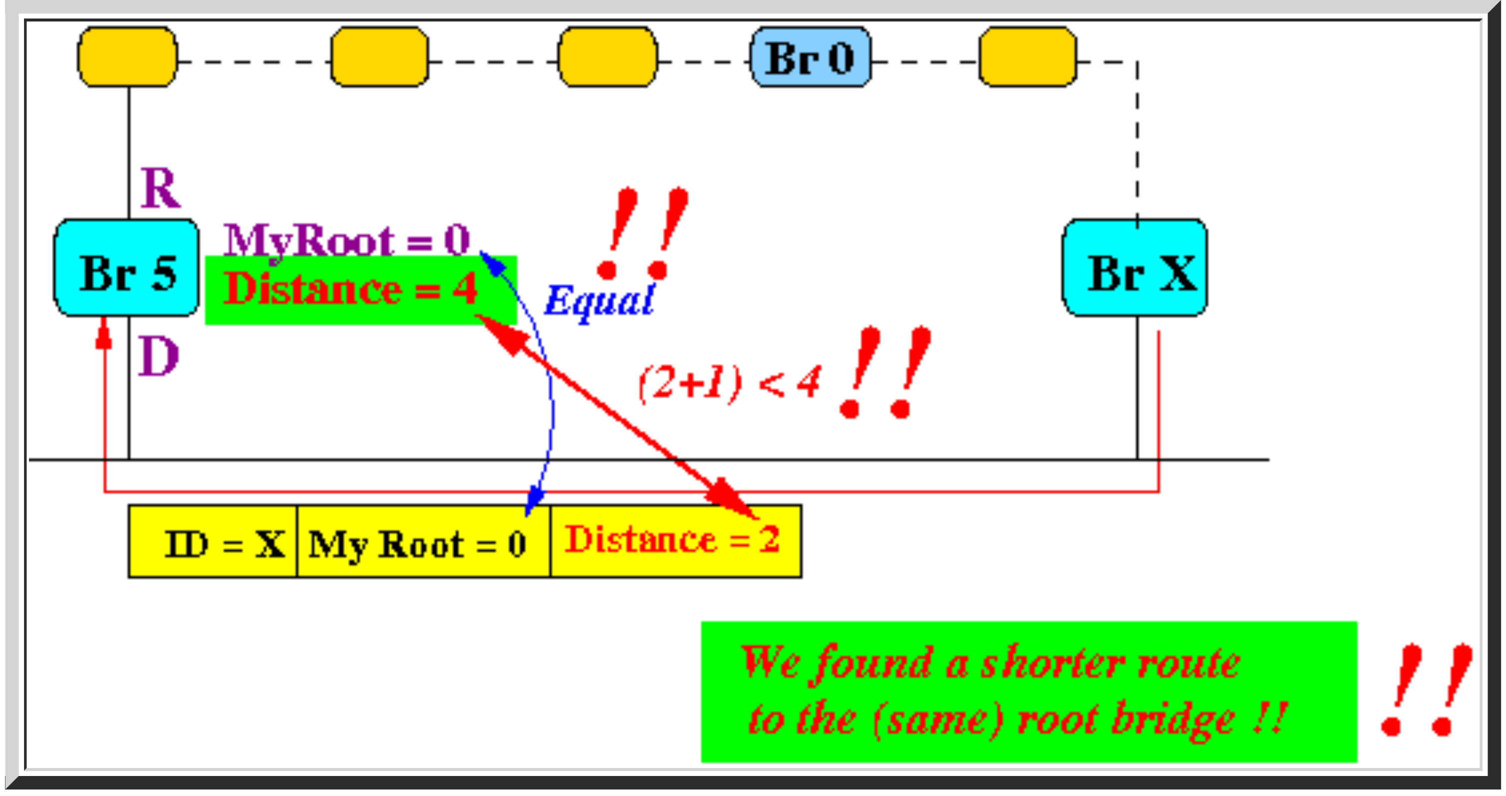
and **transmit** the following **configuration message** on **all** its ***other* ports**:



- **How to determine a shorter path to the *same* root bridge**
  - **Test** used to **find** a ***shorter* path** to the **(same) root bridge**:

  - **(Root ID in control message == my Root)** **and** **(*Distance* in control message + 1 < my *Distance*)**

  Example:



- **Actions** taken by the **bridge** when a ***better* root bridge** is **discovered**:

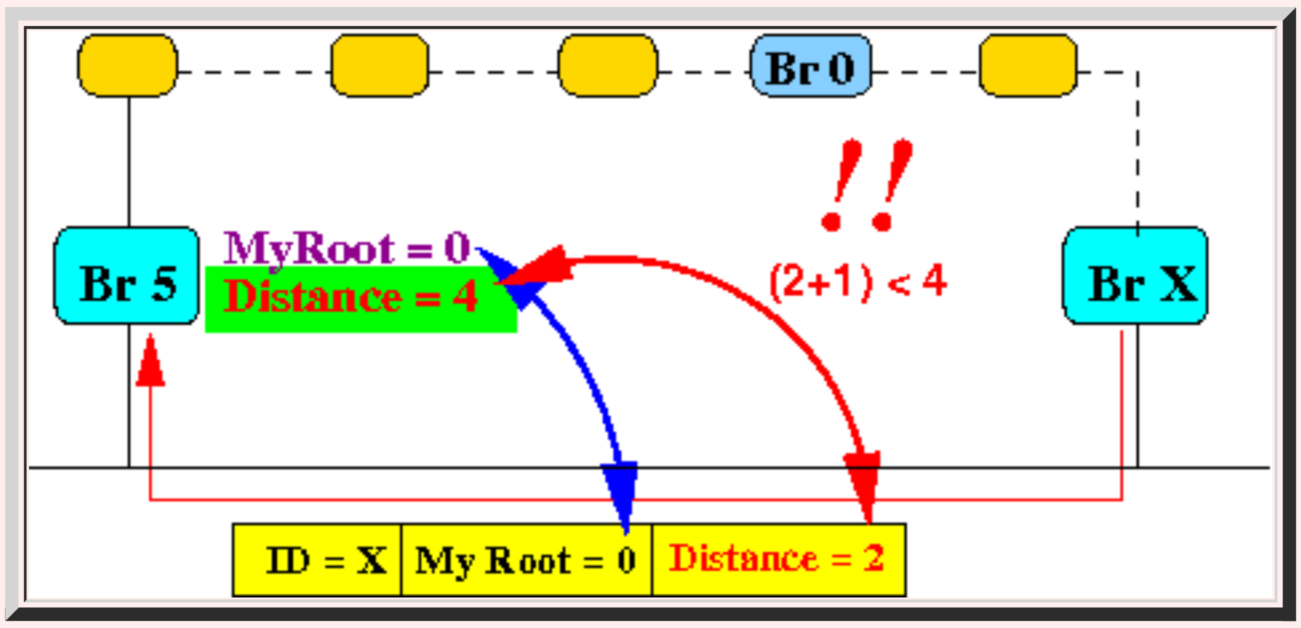
```
( MyRoot = Root Bridge ID in configuration message; )  // They are equal !!!

Distance = Distance in configuration message + 1;      // One more hop !

State(Port on which configuration message was recv) = R (Root);

for ( all other ports P )
{
    State(P) = D (Designated);
    Transmit configuration msg "(myID, MyRoot, Distance)" on port P;
}
```

Example:

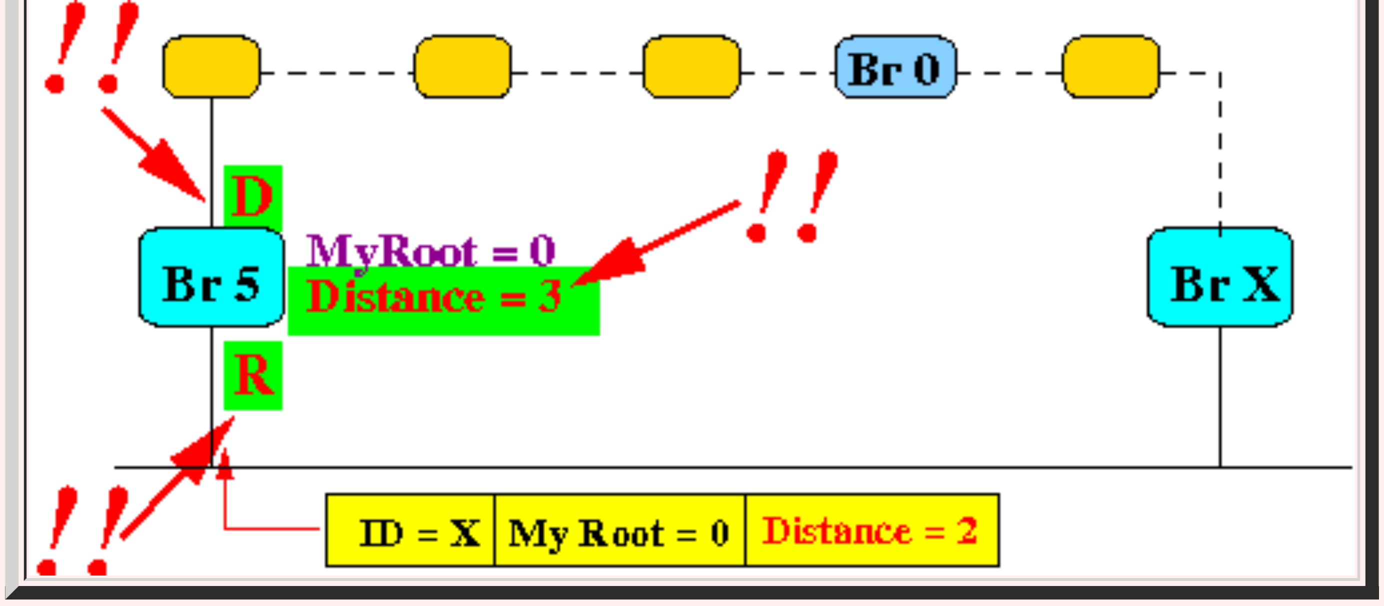
- **Bridge 5** has just **received** a **configuration message** with

```
     MyRoot   in configuration msg  =  Current Root
and  Distance in configuration msg  <  Current Distance
```

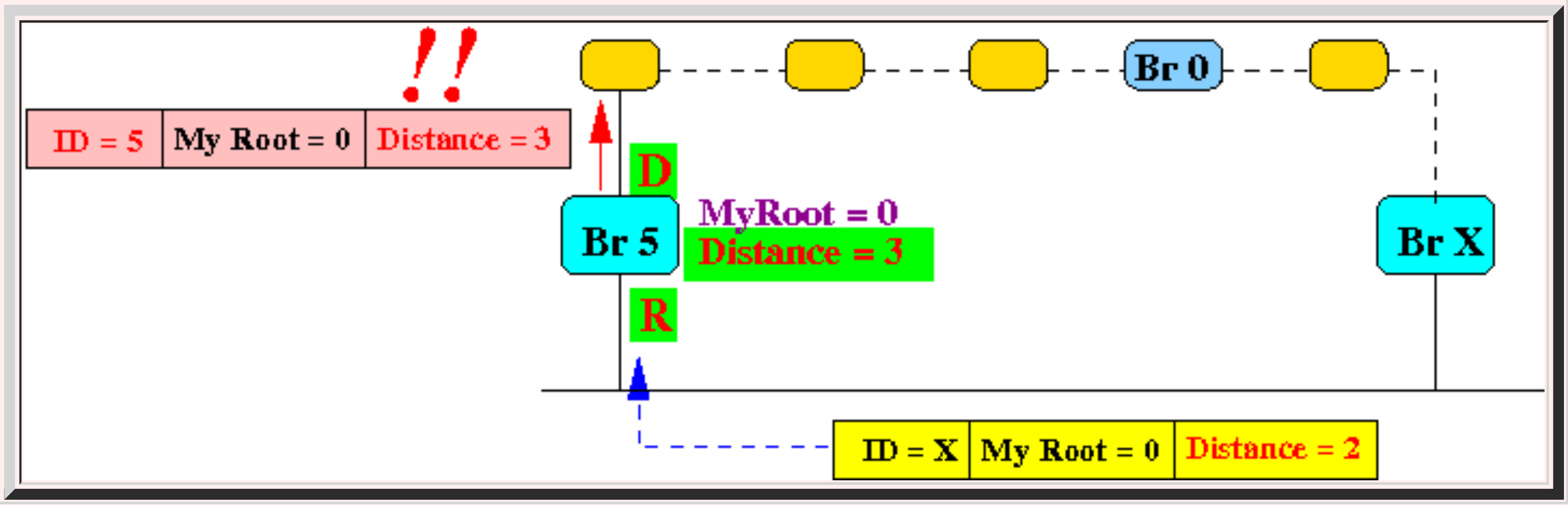
Scenario:



- **Bridge 5** will **update** its **state variables** to:



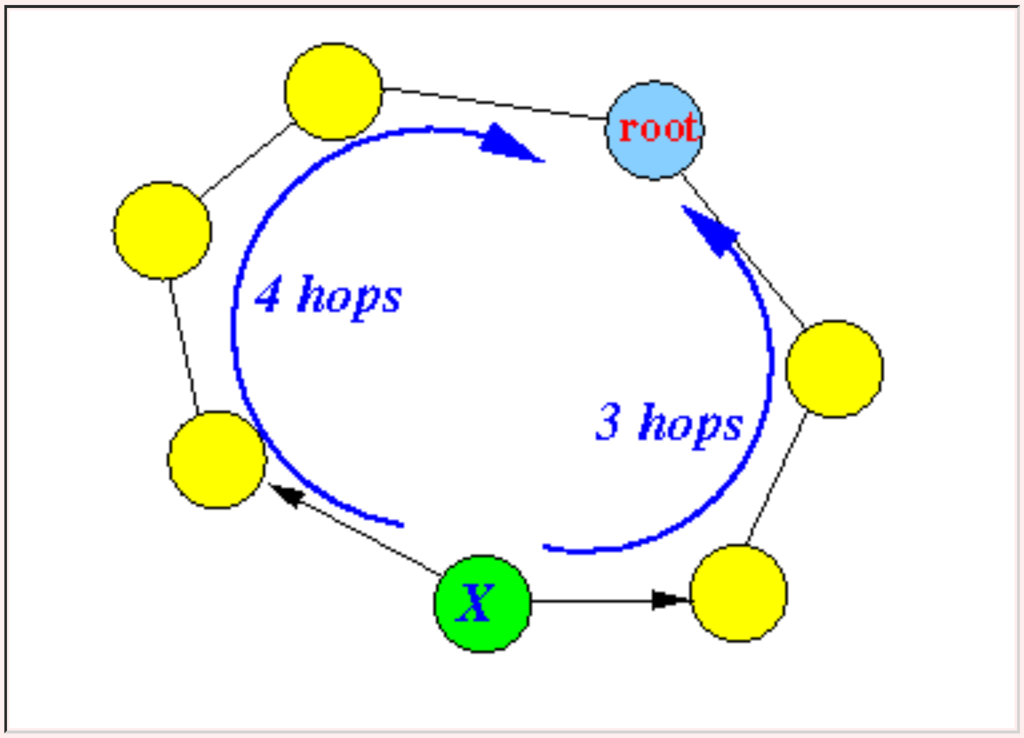
and **transmit** the following **configuration message** on **all** its ***other*** **ports**:



Nodes that are in a cycle

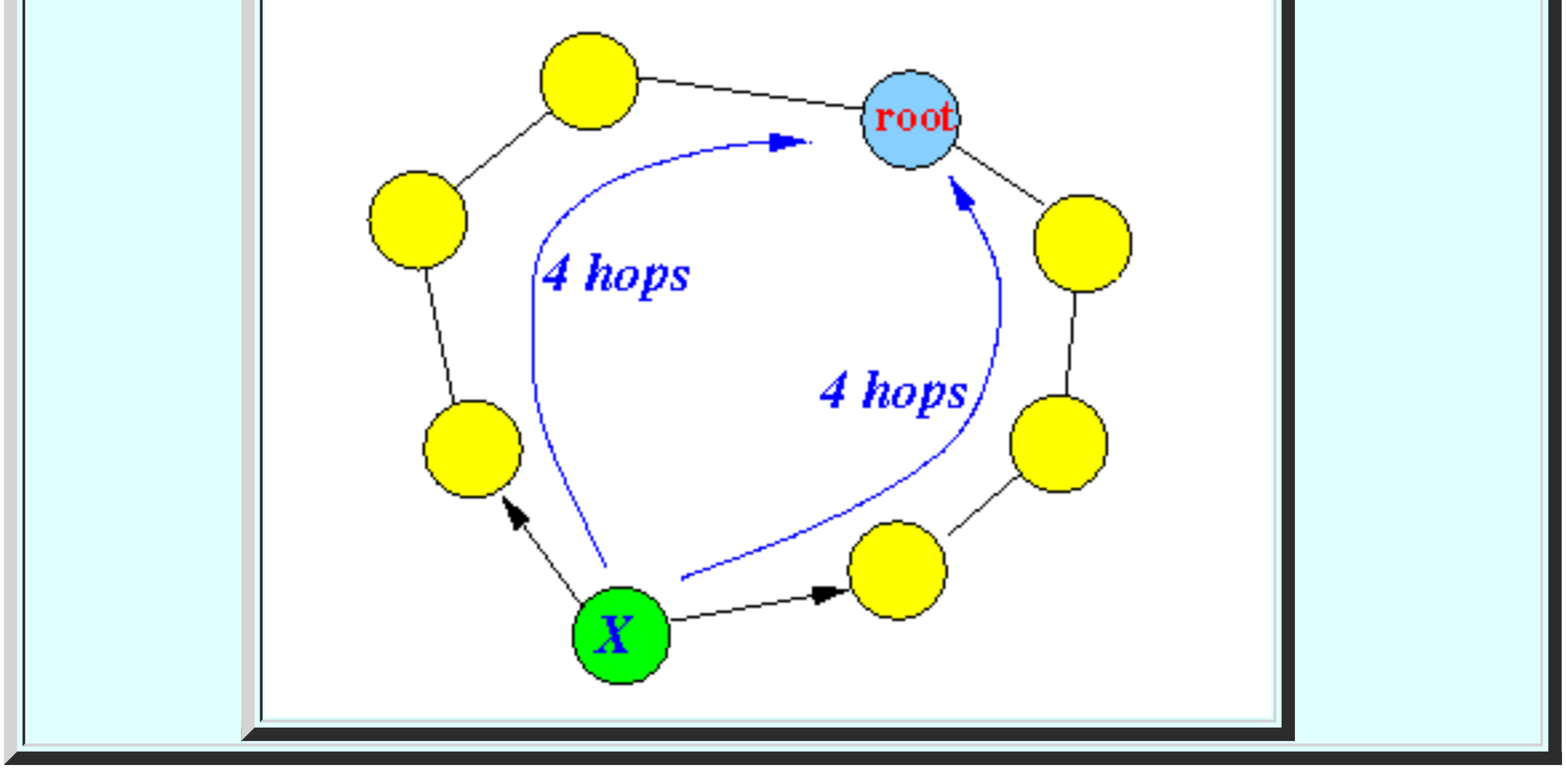
- **Different distances to the root bridge**
  - **Fact:**
    - A **bridge** can have ***different* path length** to the **root bridge** due to:
      - **Cycles (loops)**

      Example:



- **Multiple paths with the same distance to the root bridge**
  - **Fact:**
    - A **bridge** can have ***multiple* paths** to the **root bridge** that has the ***same* distance**

      Example:

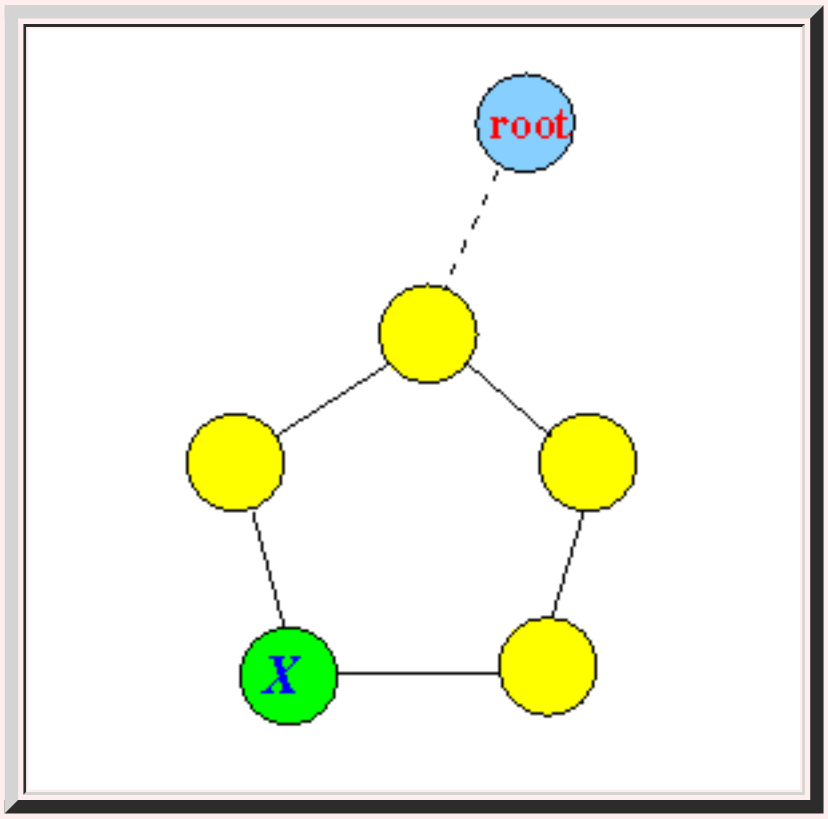


* We will discuss **how to** handle **cycles** next.....

Detecting and handling *cycles*

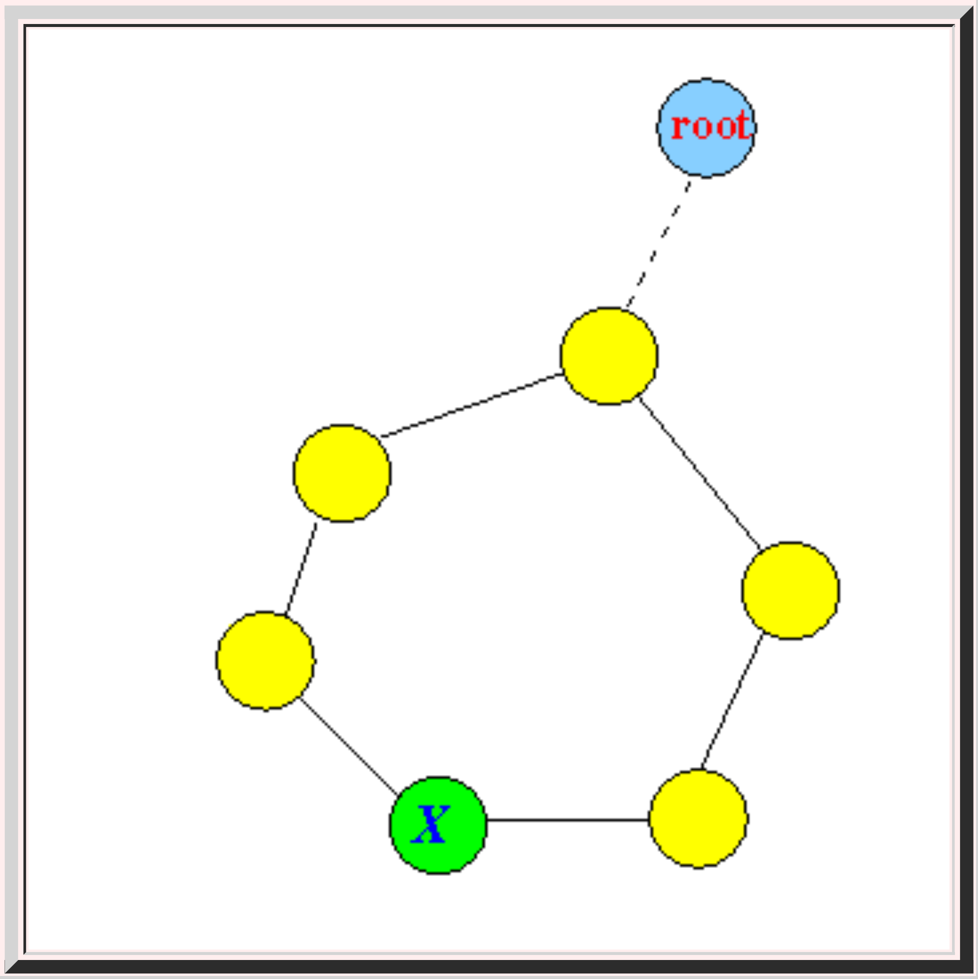
- **Cycles...**
  - There are **two types** of **cycles** in a network:
    - ***Odd* cycles**: the **number of nodes** in the **cycle** is **odd**

      Example: this **cycle** contains **5 nodes**

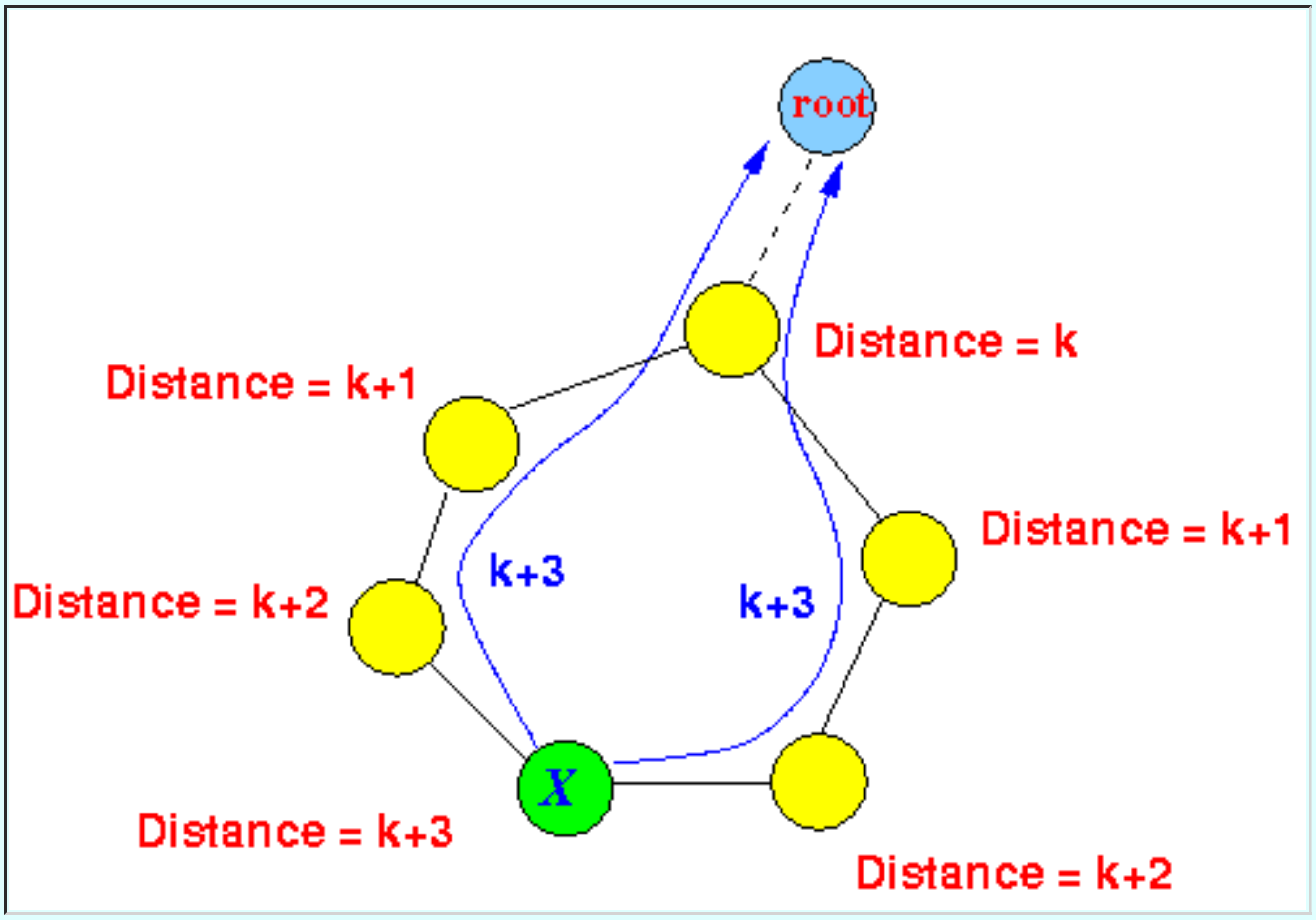
    - ***Even* cycles**: the **number of nodes** in the **cycle** is **even**

      Example: this **cycle** contains **6 nodes**

- **Detecting *even* cycles**
  - **Observation:**
    - In an **even cylce**:
      - There is **exactly *one* bridge** that have **two paths** with the ***same* distance** to the ***root* bridge**

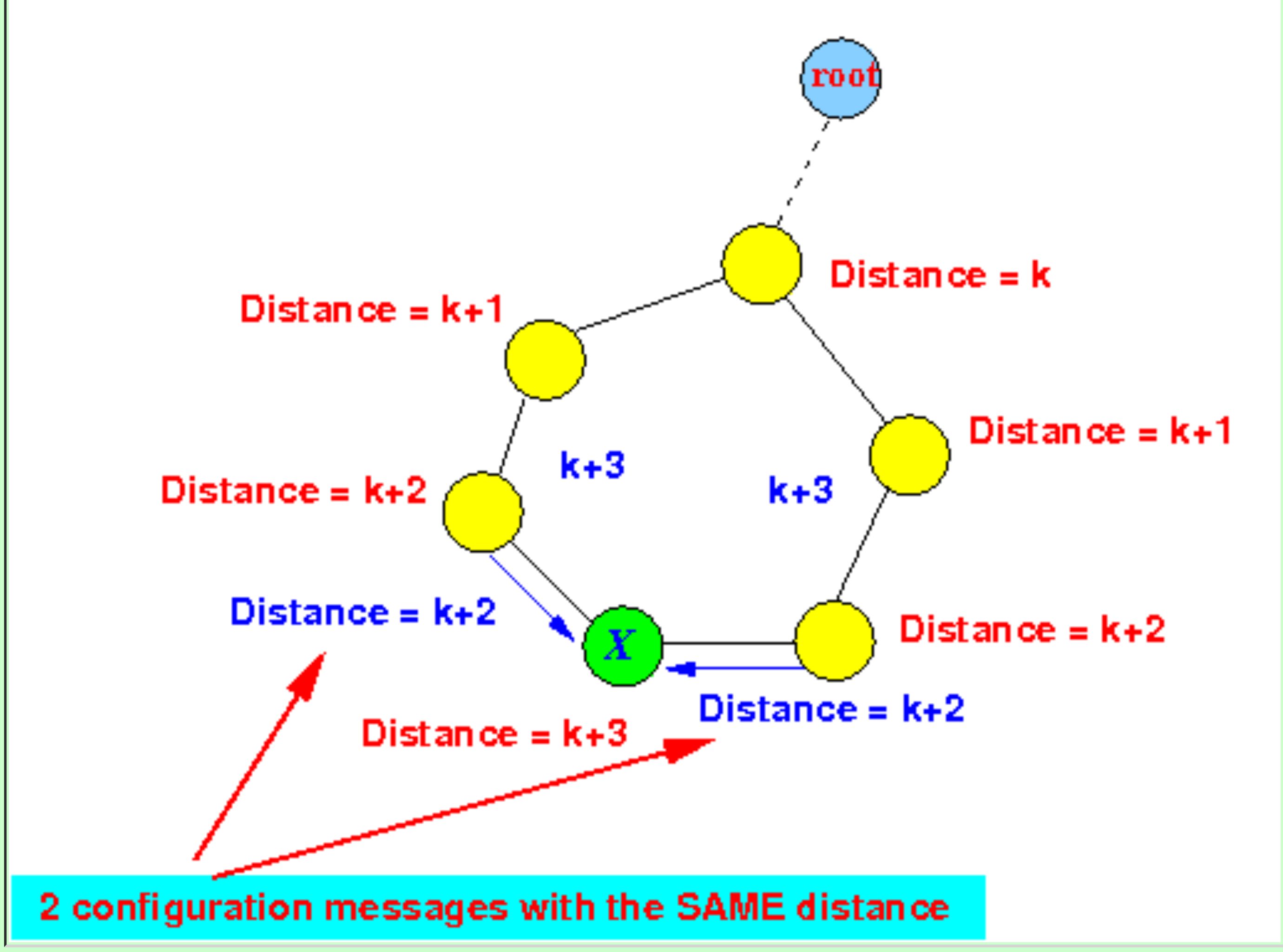
    Example:

  - ***My* nomemclature:**
    - For **simplicity**, **I** (not in any book) will **call** this **node**:
      - the ***last* node** in the **even cycle**

  - **Fact:**
    - The ***last* node** in the **even cycle** can **tell** that:
      - ***It*** is the ***last* node** in an ***even* cycle**

    How:

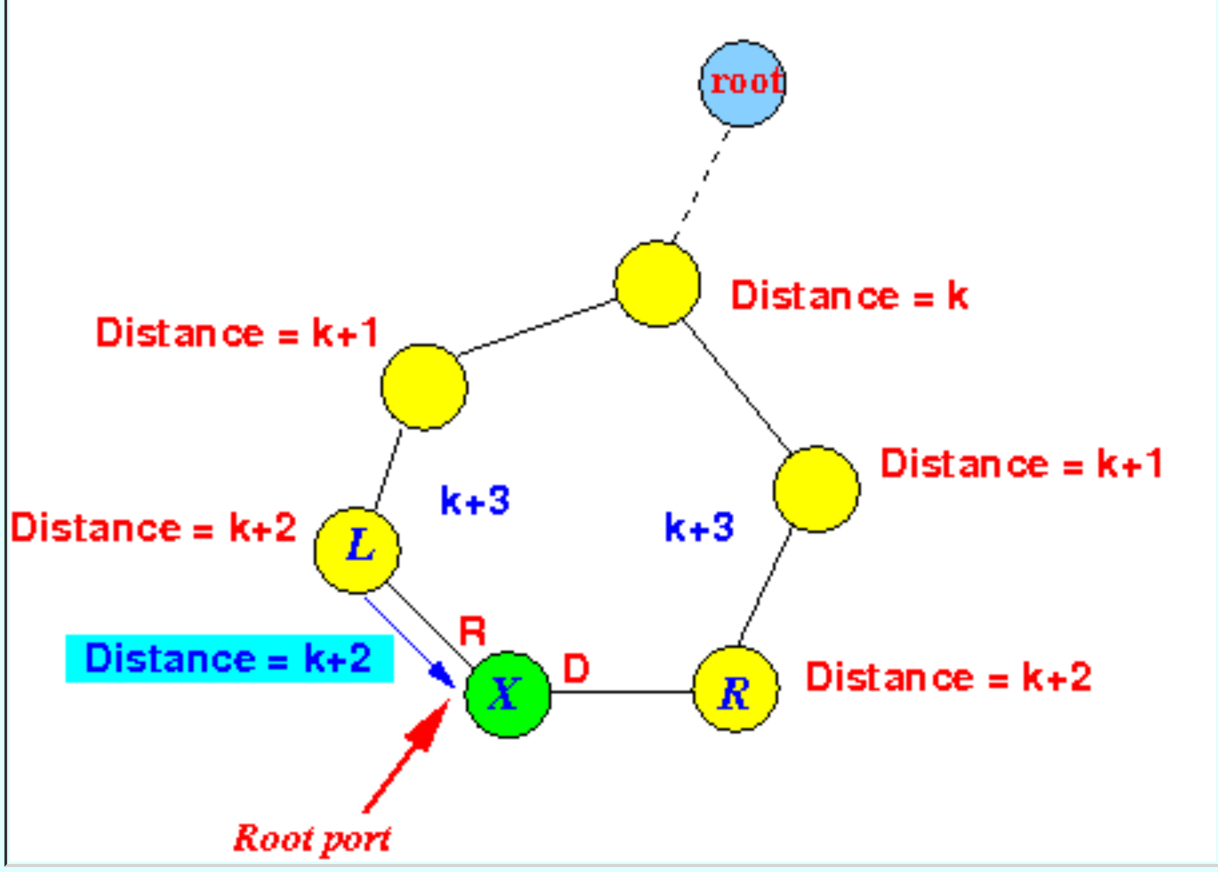


- **Handling even cycles**
  - **Goal:**
    - We want to **obtain** a ***tree* (= cycle-free)** ***logical* network** !!!

    Therefore:

    - When the ***last* node** in an ***even* cycle** detects that ***it*** is the ***last* node**:
      - The ***last* node** in an ***even* cycle** must ***block*** **one** of its **ports** (to **break** the **cycle**) !!!

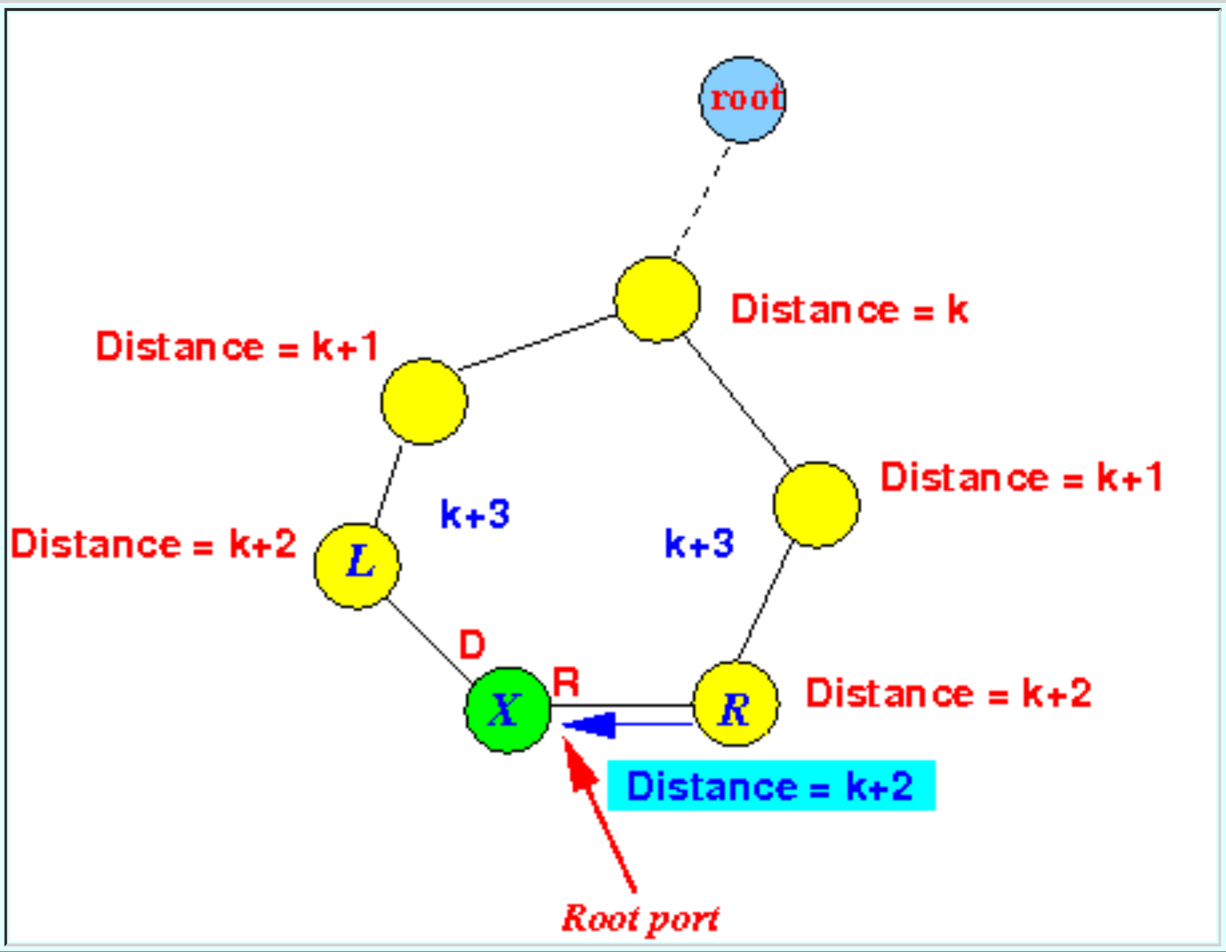
  - **Depending** on the ***order*** in which the **2 configuration messages** are **received**, you can have one of two **possible states**:
    - **If** the ***configuration message*** from the ***left* bridge (L)** arrives ***first***:



then:

- The ***left* port** will be the ***root* port**

- **If** the ***configuration message*** from the ***right* bridge (R)** arrives ***first***:



then:

- The ***right* port** will be the ***root* port**

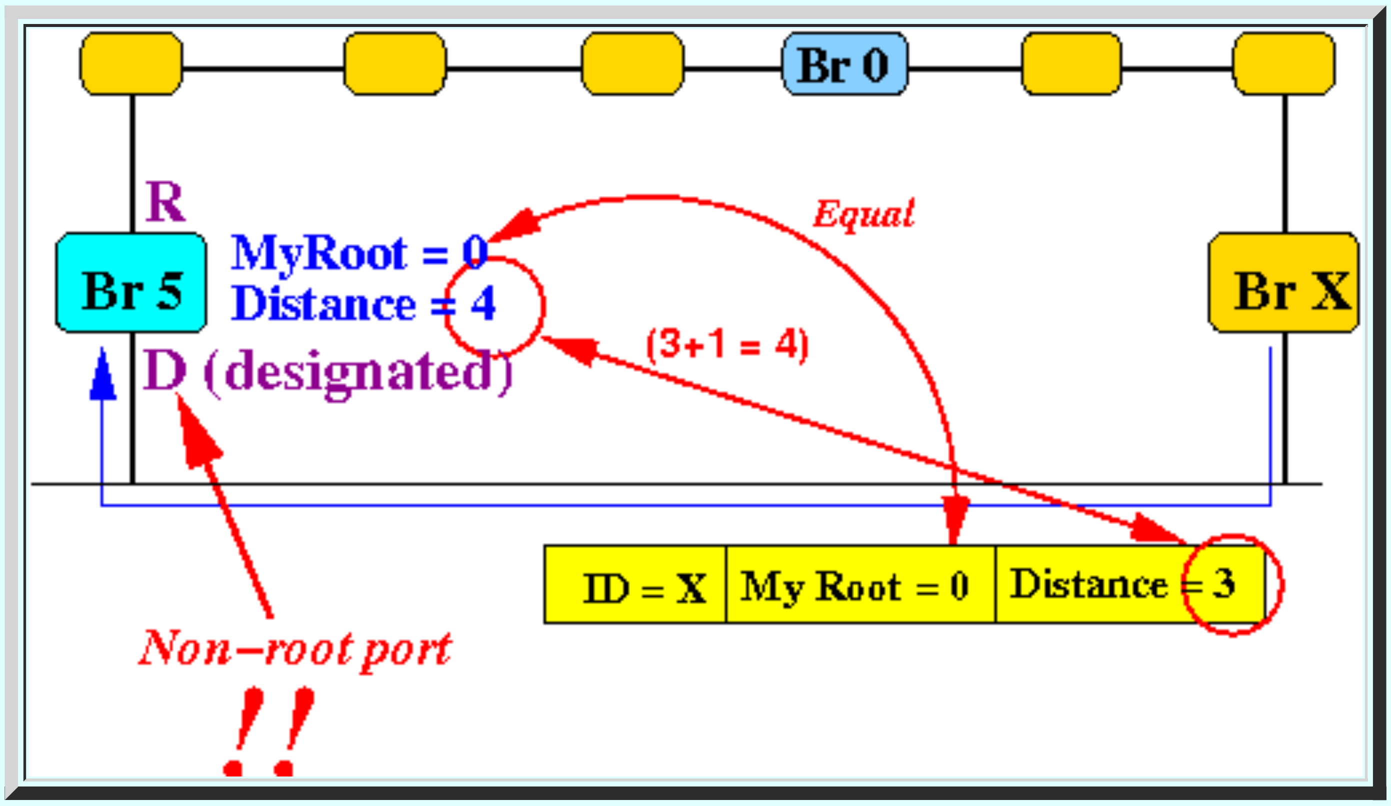
- **How to** break an ***even* cycle**: by **block** the ***non*-root port**

```
** Bridge receives a configuration message on port Port;
```

```
/* ======================================================
   How to detect the last node in an even cycle
   ====================================================== */
if ( ( RootID in the control message == myRoot ) AND
     ( Distance in message == myDistance - 1 )   )
then
{
    if ( Status(Port) != Root )
       Status(Port) = BLOCKED;        // Block the non-root port !
}
```

- **Example:**

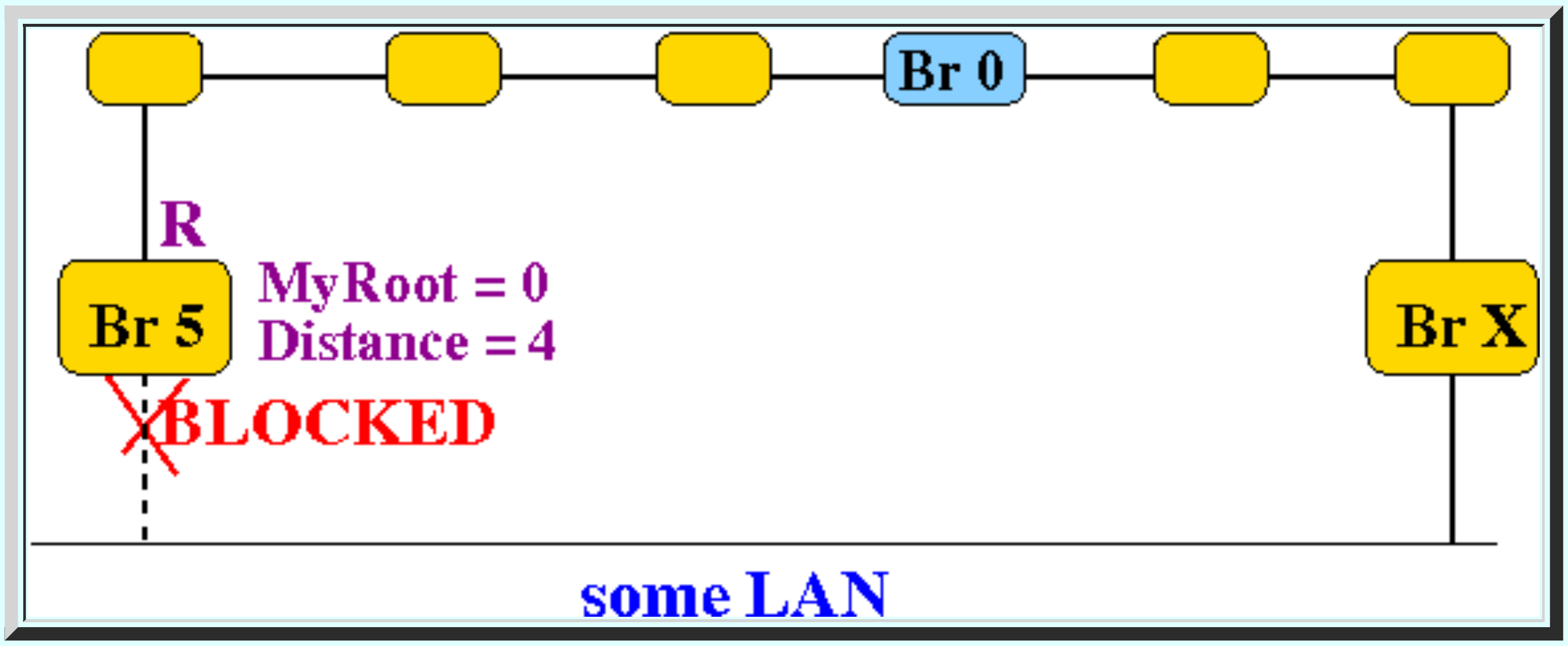
  - The **bridge 5** just received a **configuration message** with ***equal* distance** to **root node** on a ***non-root* port**:

    Now the **bridge 5** can **tell** that:

    - ***It*** is the **last node** in an ***even*** **cycle** !!!

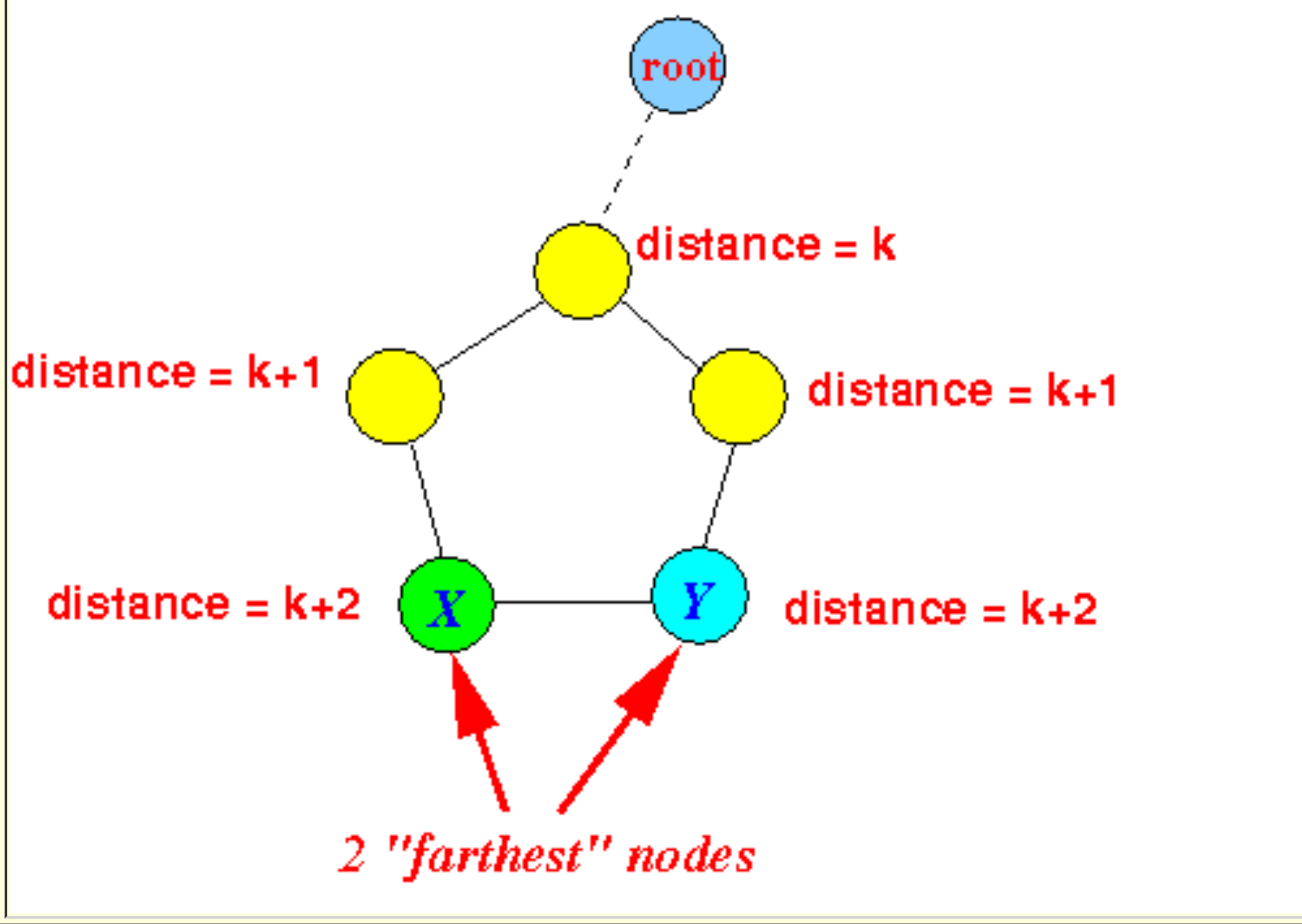
  - **Bridge 5** will ***block*** the ***receiving*** **port**:

The ***cycle*** is now **broken** !!!

- **Detecting *odd* cycles**
  - **Observation:**
    - In an ***odd* cylce:**
      - There will be ***two* bridges** that have the **same *maximum* distance** to the **root bridge**

        Example:

      I.e.:

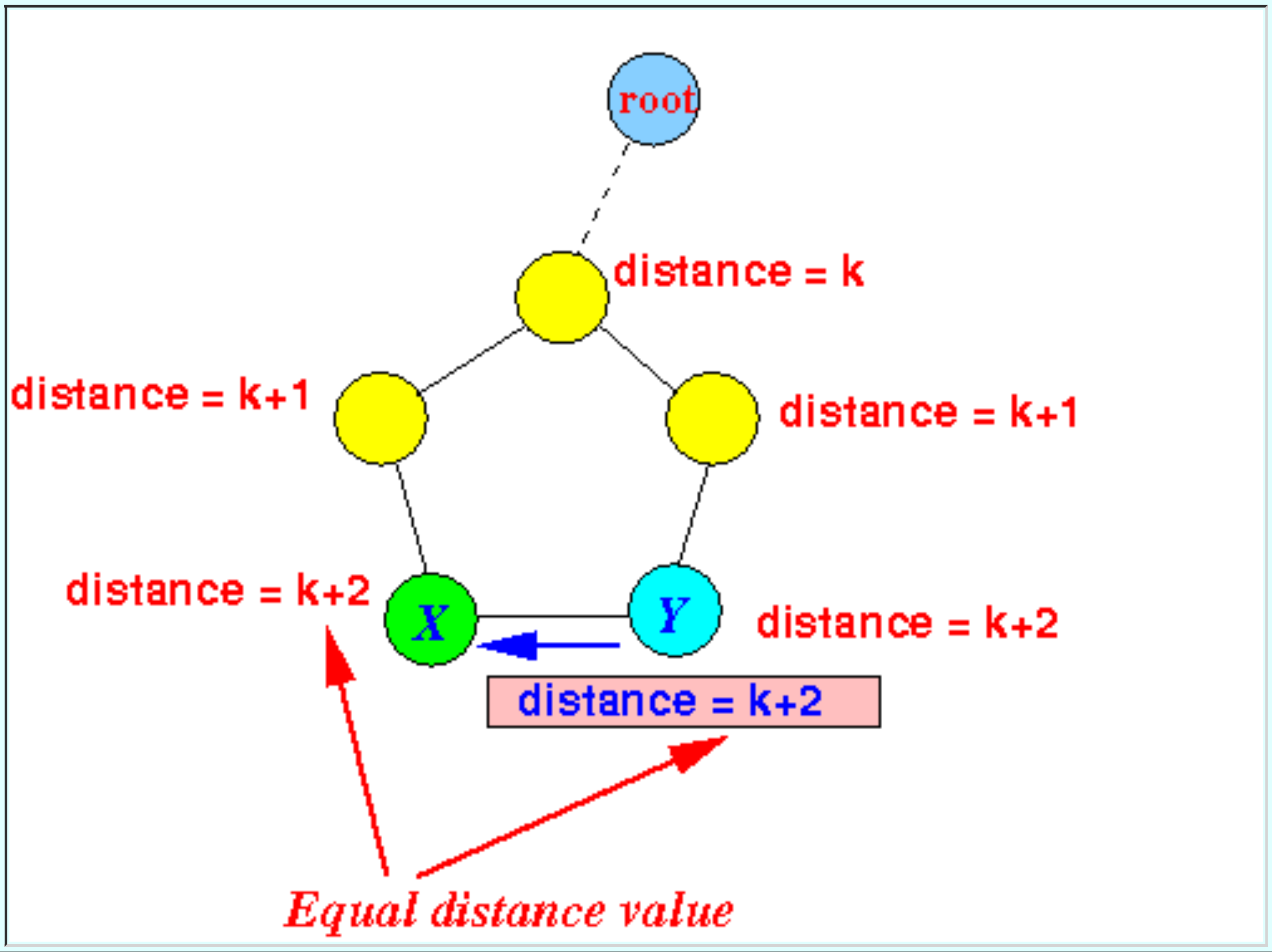
      - There are **2 *last* nodes** in the **cycle**

  - **How to determine** if a **bridge** is a ***last* node** in an ***odd* cycle**:
    - A ***last* node** in an ***odd* cycle** will receive a **configuration message** with:

      ```
      Distance in configuration message == myDistance
      ```

      to the ***same* root bridge**

    - **Example:**



- **Caveat**
  - **Note:**

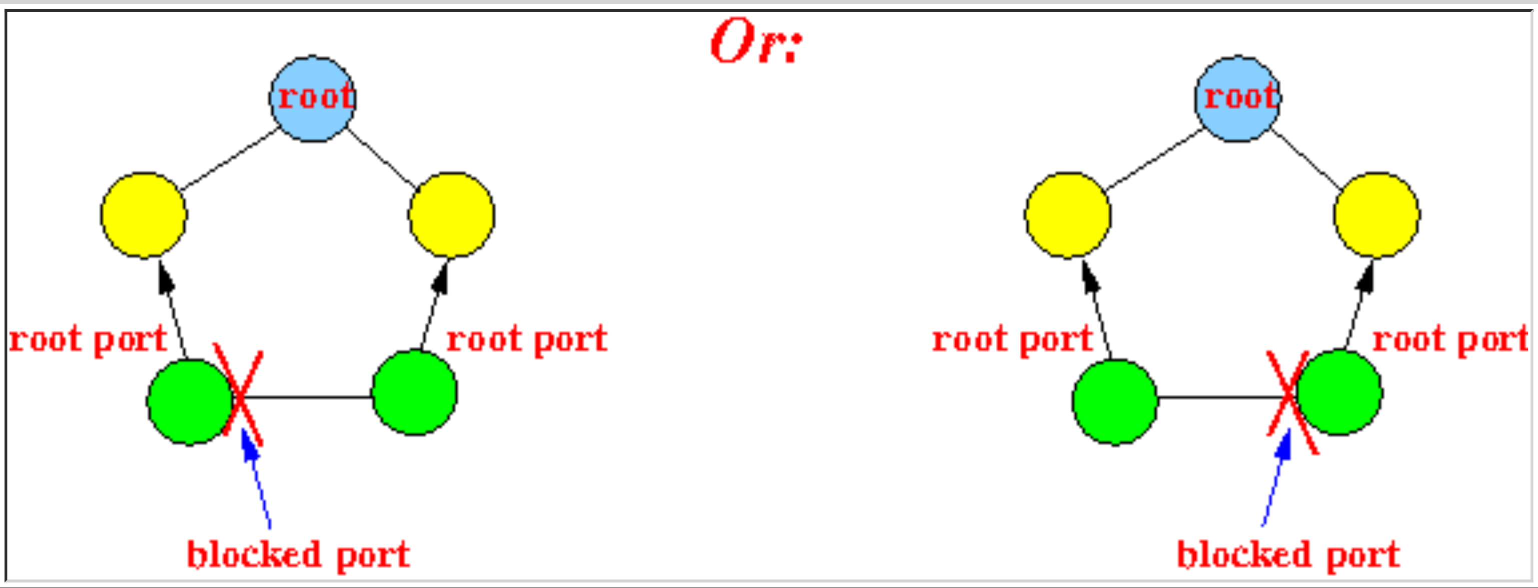
    - There are **2 last nodes** in an ***odd* cycle**

    - ***Both* last nodes** can (and will) **determine** that ***it*** is (one of) a ***last* node** in an ***odd* cycles**

    ***Caveat*:**

    - **Only *one*** of the **bridges** must **block** its **port** !!!

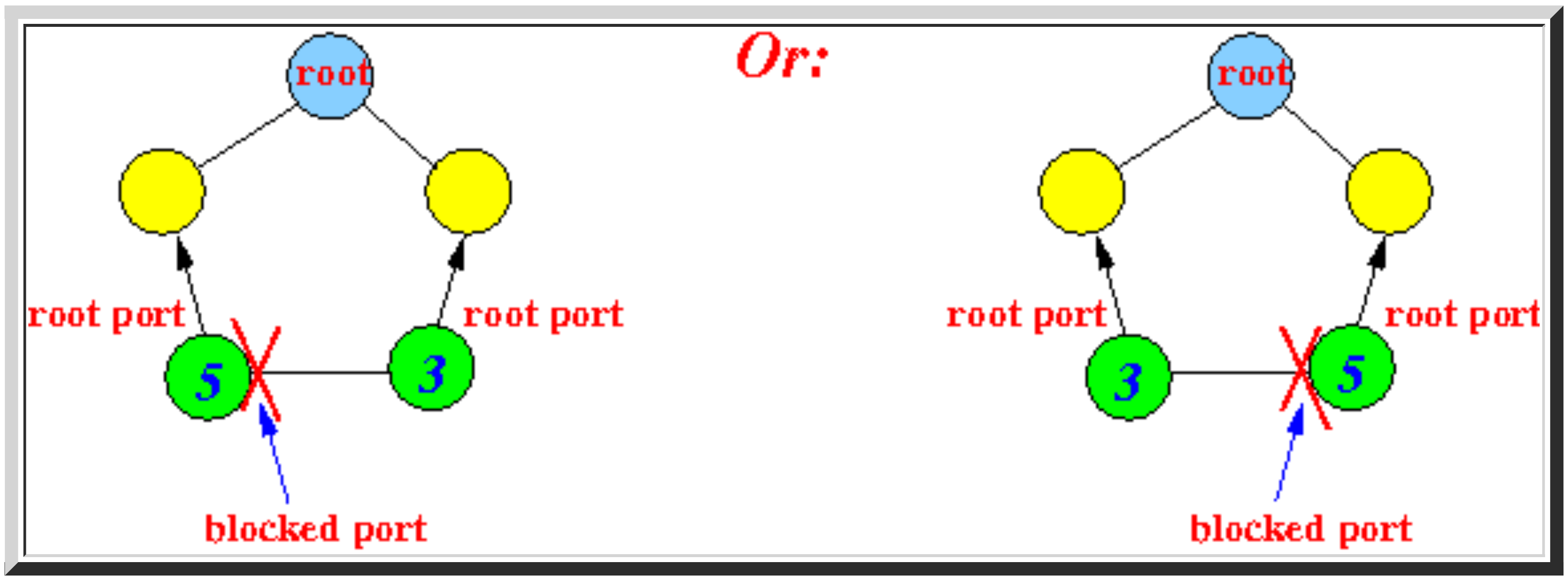
    Graphically:

    (You **must *not*** block ***both* bridges**, **or else** you will **disconnect** some **LAN** !!!)

- **Tie breaker rules used in the Spanning Tree algorithm**
  - **IEEE 802.11D rule** to **determine** the **block port**:
    - The **bridge** with a ***large* ID value** must make its ***non-root* port** into a ***blocked* port**

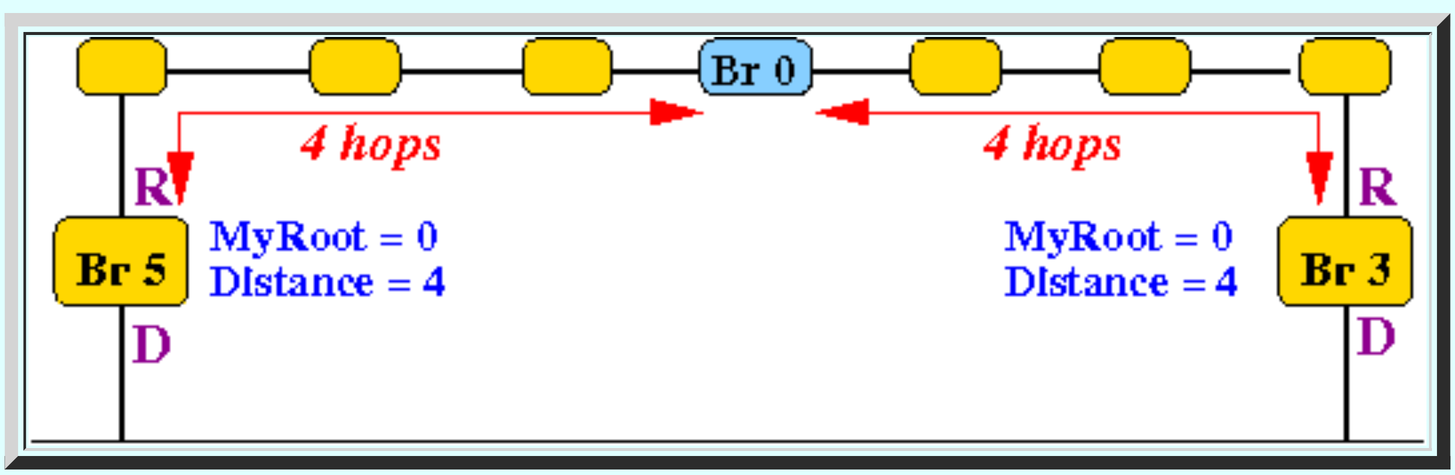
    Example:



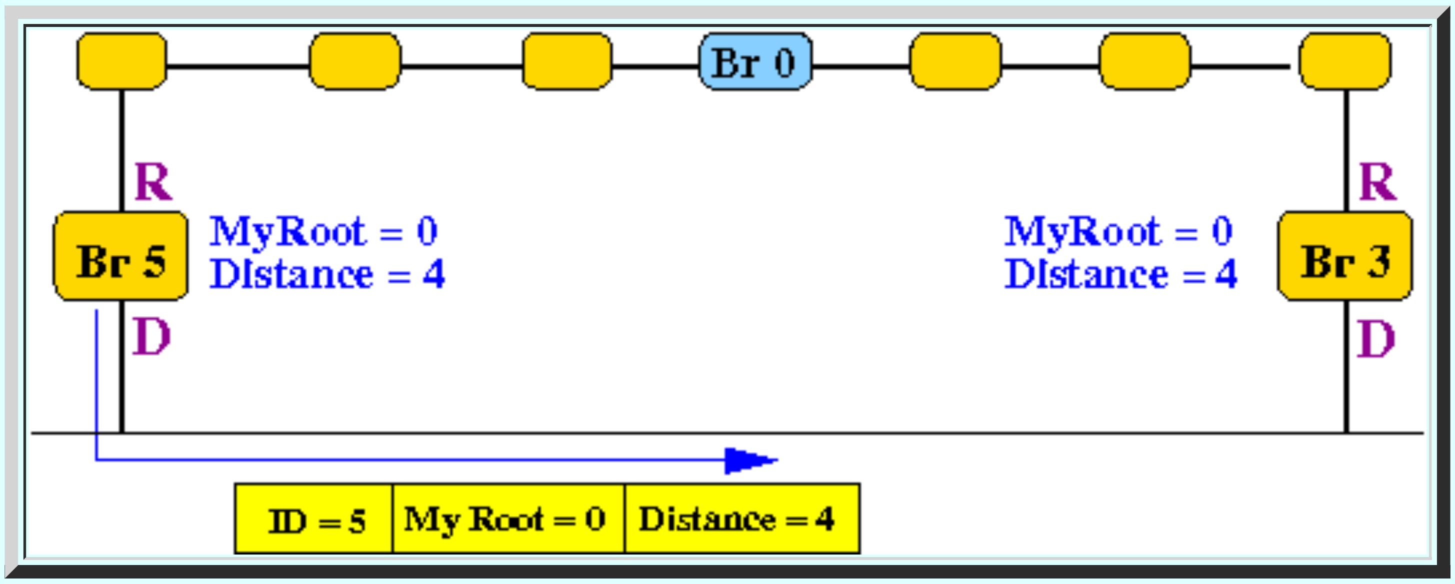
- **Handling *odd* cycles**
  - **Algorithm is pseudo code:**

```
Bridge received a configuration message Msg on port Port;


/* =================================================
   How to detect the last node in an odd cycle
   ================================================= */
if ( ( Msg(RootID) == myRoot )               AND
     ( Msg(Distance) == myDistance ) )
{
   /* =================================================
      Block the port of the bridge with the large ID
      ================================================= */
   if ( myID > Msg(SourceID) )
     Status(Port) = BLOCKED;
}
```

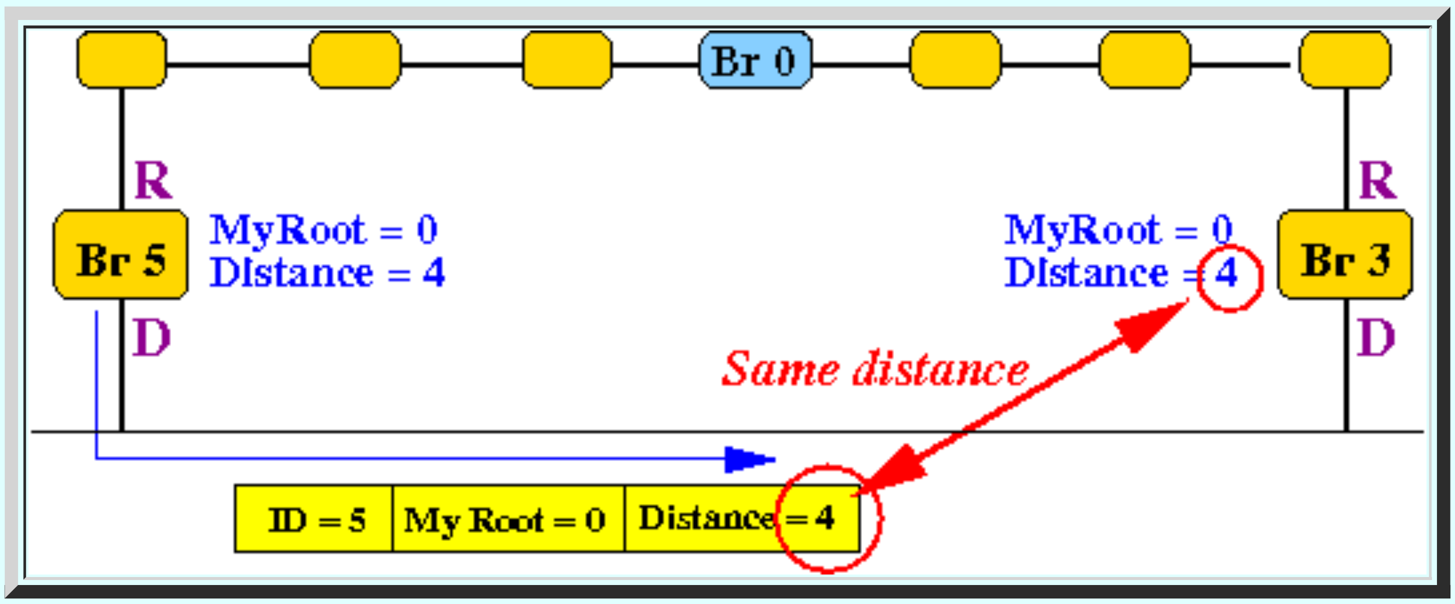
  - A ***concrete*** **example:**
    - **Bridge 5 *and* bridge 3** are **4 hops** away from the **root bridge**:



- When **bridge 3** receives a **configuration message** from **node 5**:



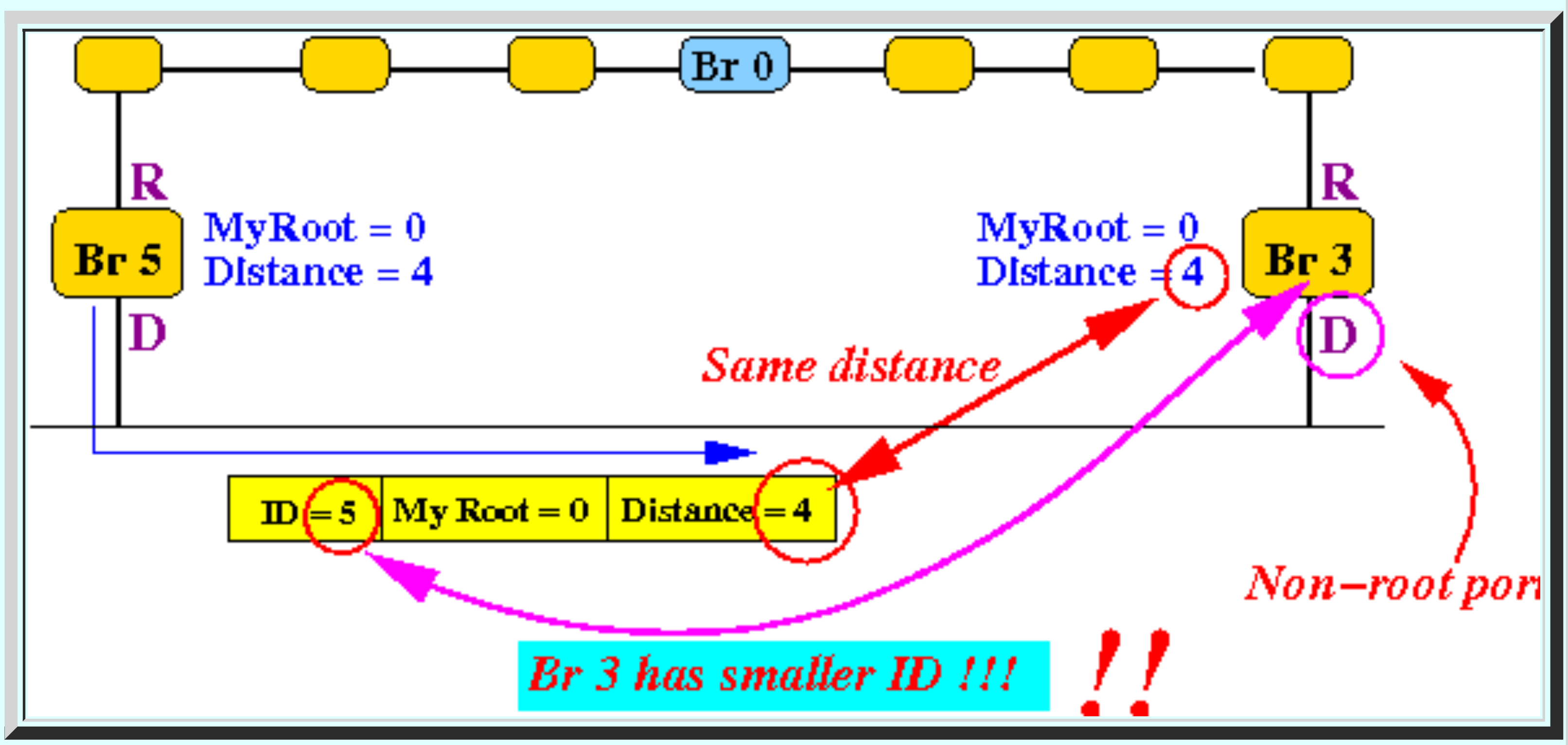
Brigde 3 will find out that ***it*** is ***one*** of the ***last* nodes** in an ***odd* cycle**:



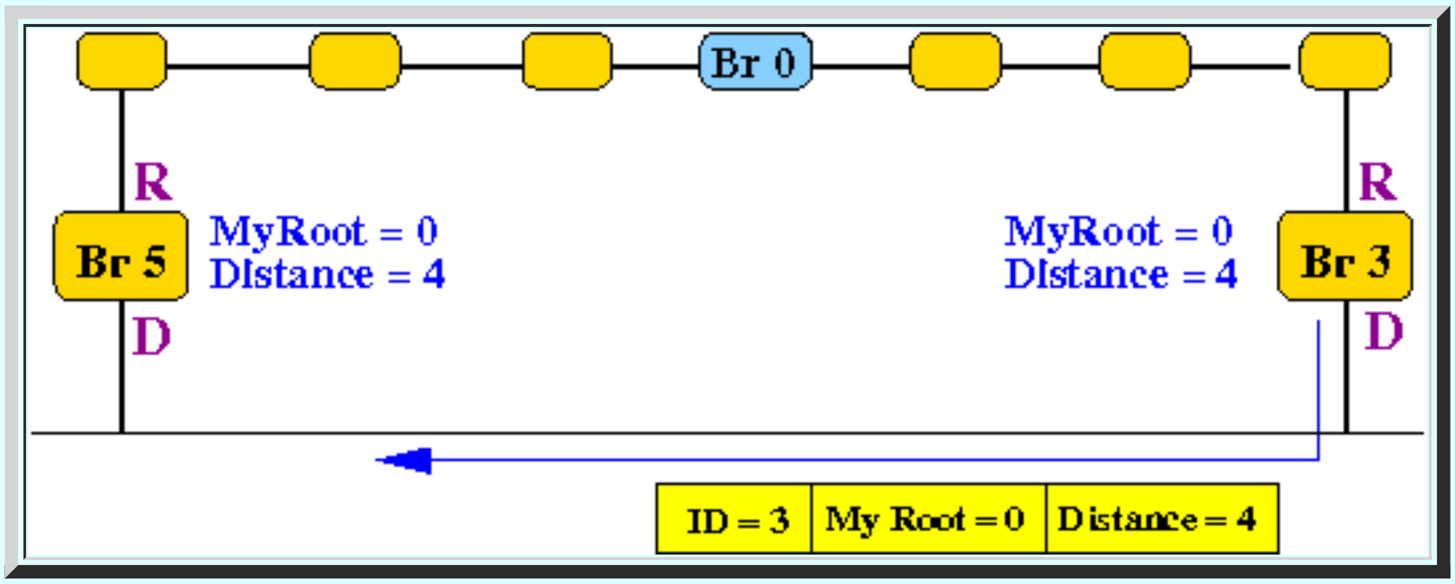
However:

- **Brigde 3** will ***ignore*** the **message** because its **ID** is ***smaller*** than **bridge *5***

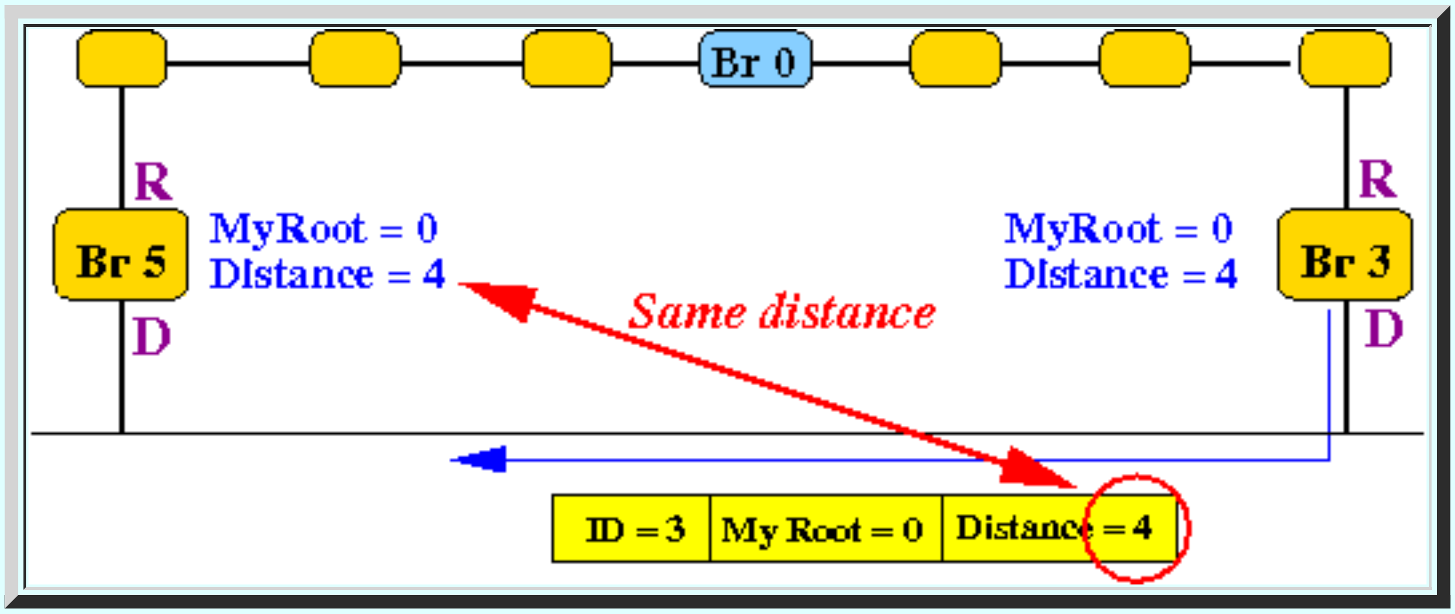
Graphically:



- ***But***, **when bridge 5** receives a **configuration message** from **node 3**:



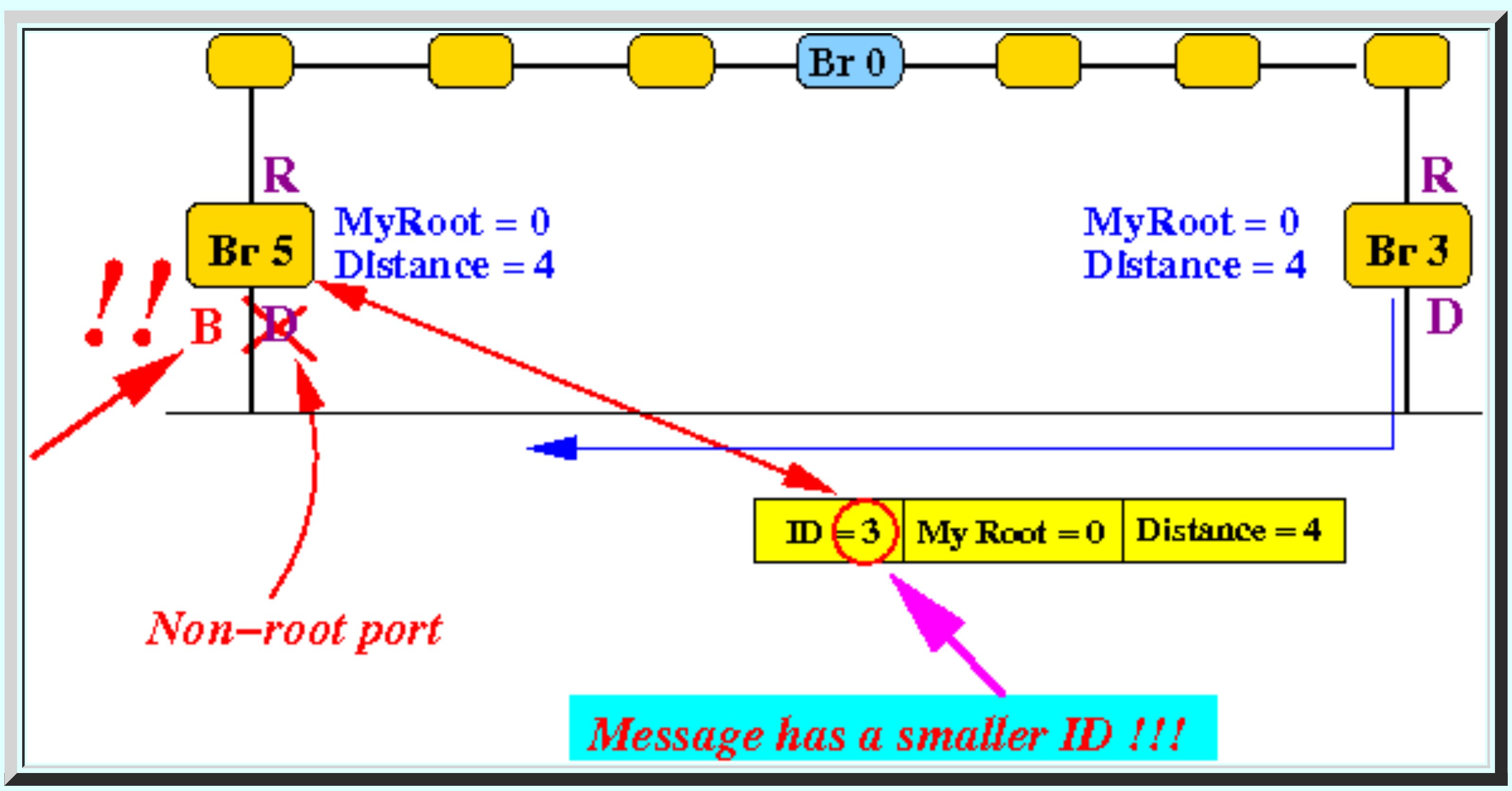
Brigde 5 will find out that ***it*** is ***one*** of the ***last* nodes** in an ***odd* cycle**:



In ***this* case**:

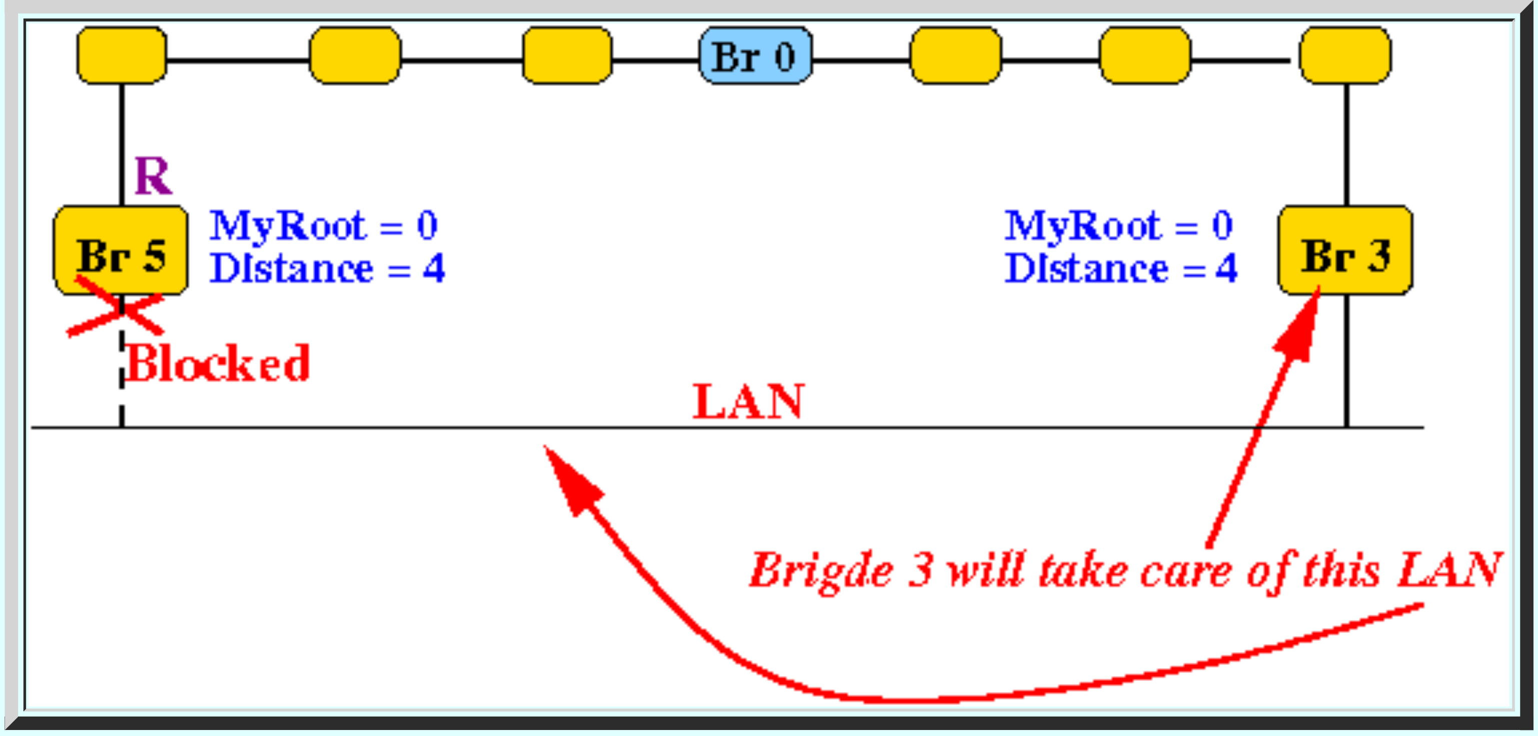
- **Brigde 5** will **change** the **status** of the **incoming port** to ***blocked***

Graphically:



because its **ID** is ***larger*** than **bridge *3***

- ***Result:***



The **cycle** is **also** **broken**

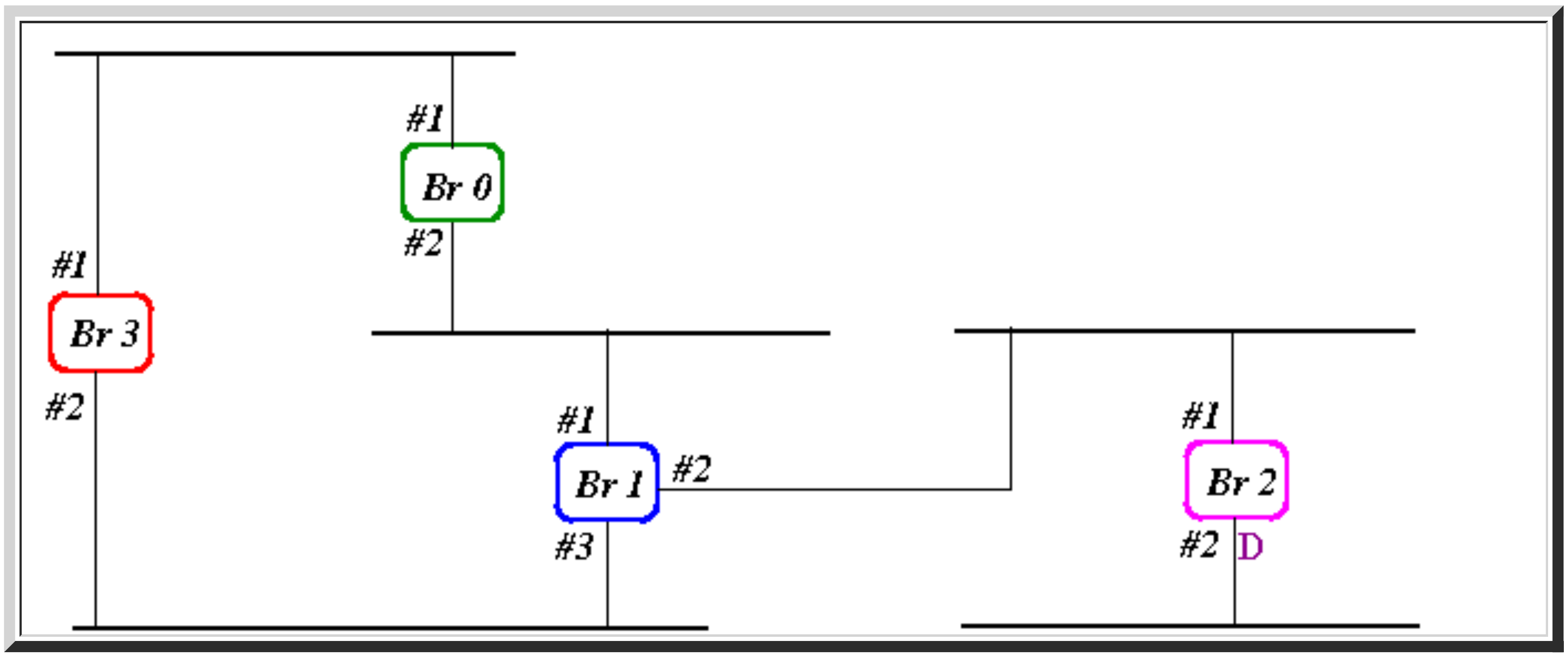
The Spanning Tree Algorithm and an example

- **The IEEE 802.1D Spanning Tree Algorithm**
    - The **(Distributed) Spanning Tree Algorithm** in **pseudo code**:

```
Let: msg = control message received
     p   = port on which the message msg was received


/* ===========================================
   Check for better root
   =========================================== */
if (  msg.rootID < bridge.rootID )
{
   /* =====================================
      Update state variables
      ===================================== */
   bridge.rootID   = msg.rootID;
   bridge.distance = msg.distance + 1;
   status[p]       = R;        // This is the root port

   for ( all ports q ≠ incoming port p )
   {
      status[q] = D;  // Other ports are now designated
   }

   /* =====================================
      Forward new state to neighbors
      ===================================== */
   for ( all ports q ≠ incoming port p )
   {
      send (bridge.ID, bridge.rootID, bridge.distance) on port q;
   }
}
/* ===========================================
   Check for shorter path to root bridge
   =========================================== */
else if (  msg.rootID == bridge.rootID &&
           msg.distance + 1 < bridge.distance )
{
   /* =====================================
      Update state variables
      ===================================== */
   bridge.rootID   = msg.rootID;
   bridge.distance = msg.distance + 1;
   status[p]       = R;        // This is the root port

   for ( all ports q ≠ incoming port p )
   {
      status[q] = D;  // Other ports are now designated
   }

   /* =====================================
      Forward new state to neighbors
      ===================================== */
   for ( all ports q ≠ incoming port p )
   {
      send (bridge.ID, bridge.rootID, bridge.distance) on port q;
   }

}
/* ===========================================
   Check for farthest node in even cycle
   =========================================== */
else if (  msg.rootID == bridge.rootID &&
           msg.distance + 1 == bridge.distance )
{
   if ( status[p] != R )
      status[p] = B;        // Block the incoming port
}
/* ===========================================
   Check for farthest node in odd cycle
   =========================================== */
```

```
    else if (  msg.rootID == bridge.rootID &&
               msg.distance == bridge.distance )
    {
       if ( msg.ID < bridge.ID )
          status[p] = B;         // Block the incoming port
    }
    else
    {
        // Do nothing, ignore a worse configuration
    }
```

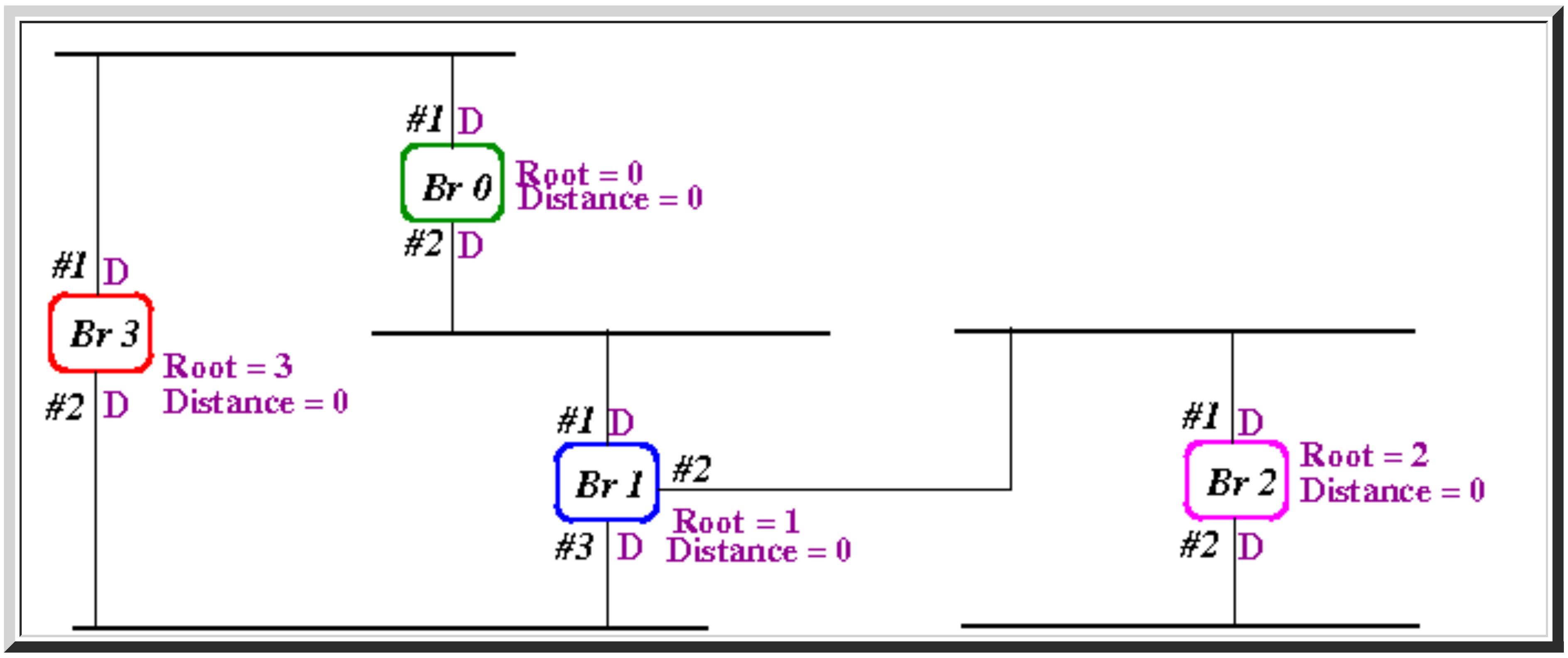
- **Example of the IEEE 802.1D Spanning Tree Algorithm**
  - Consider the following **interconnect LANs**:



  - **Initialization** of the **IEEE 802.1D Spanning Tree Algorithm** (at time of **start up**):

    - ***Each* bridge *assume*** that it is the the **root bridge**

    Result:



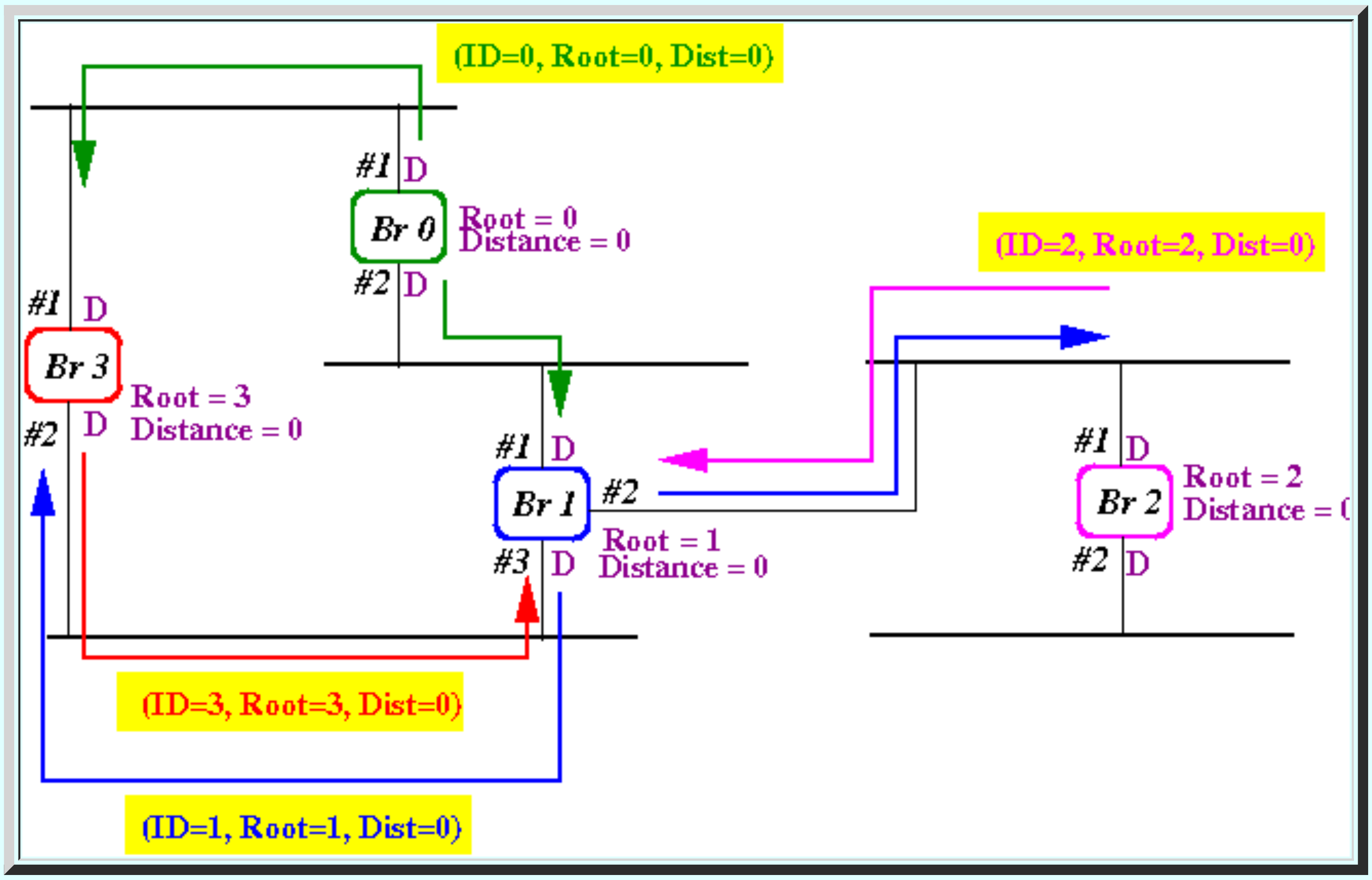
  - **Recall:**

    - The ***Root* bridge *x*** will **periodically** transmit the **configuration message:**

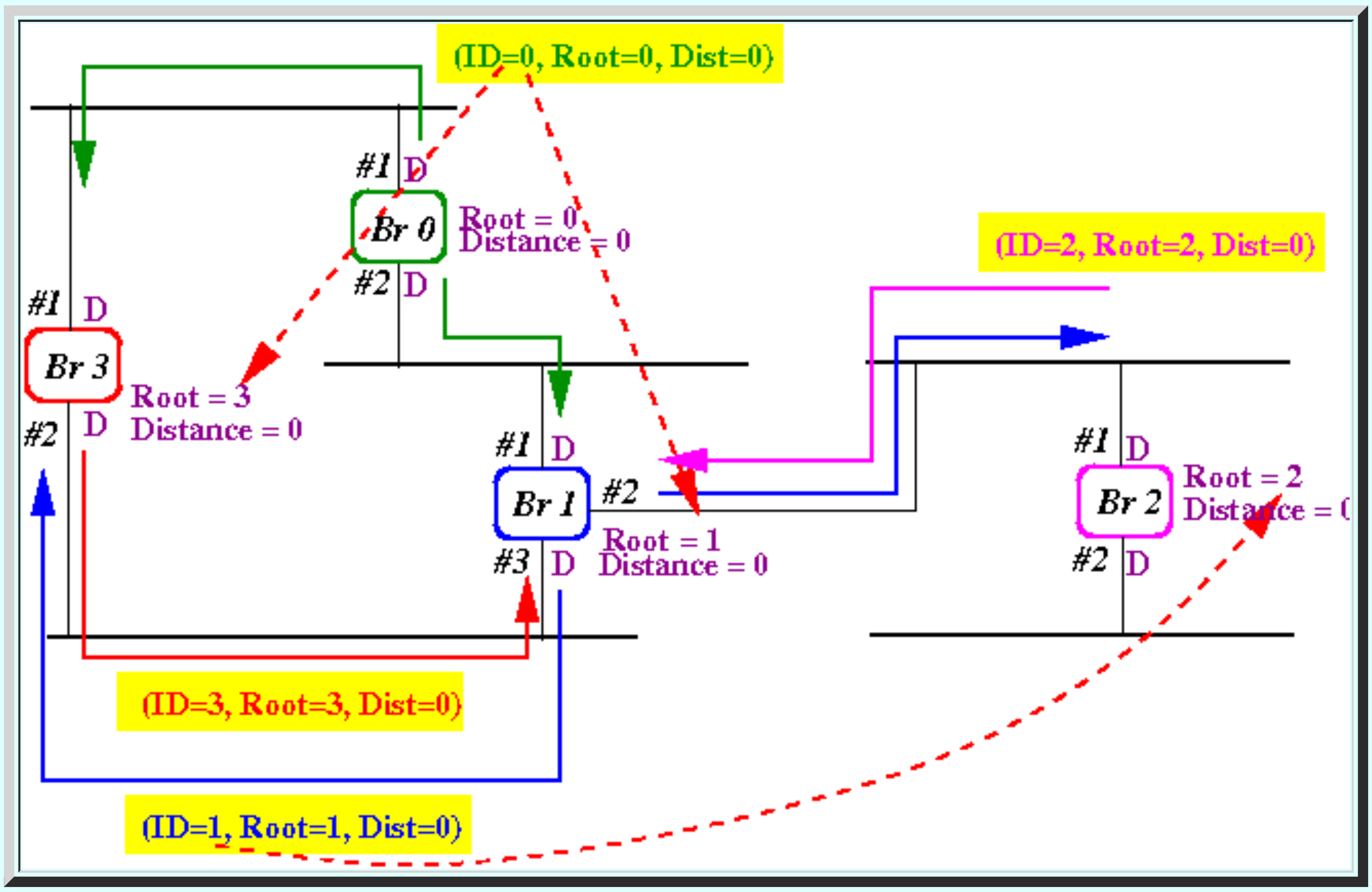
(x, x, 0)

- **Start** of **message exchange:**

  - **Messages** that are **sent**:



  - **Computations** performed by **each node**:



Explanation:

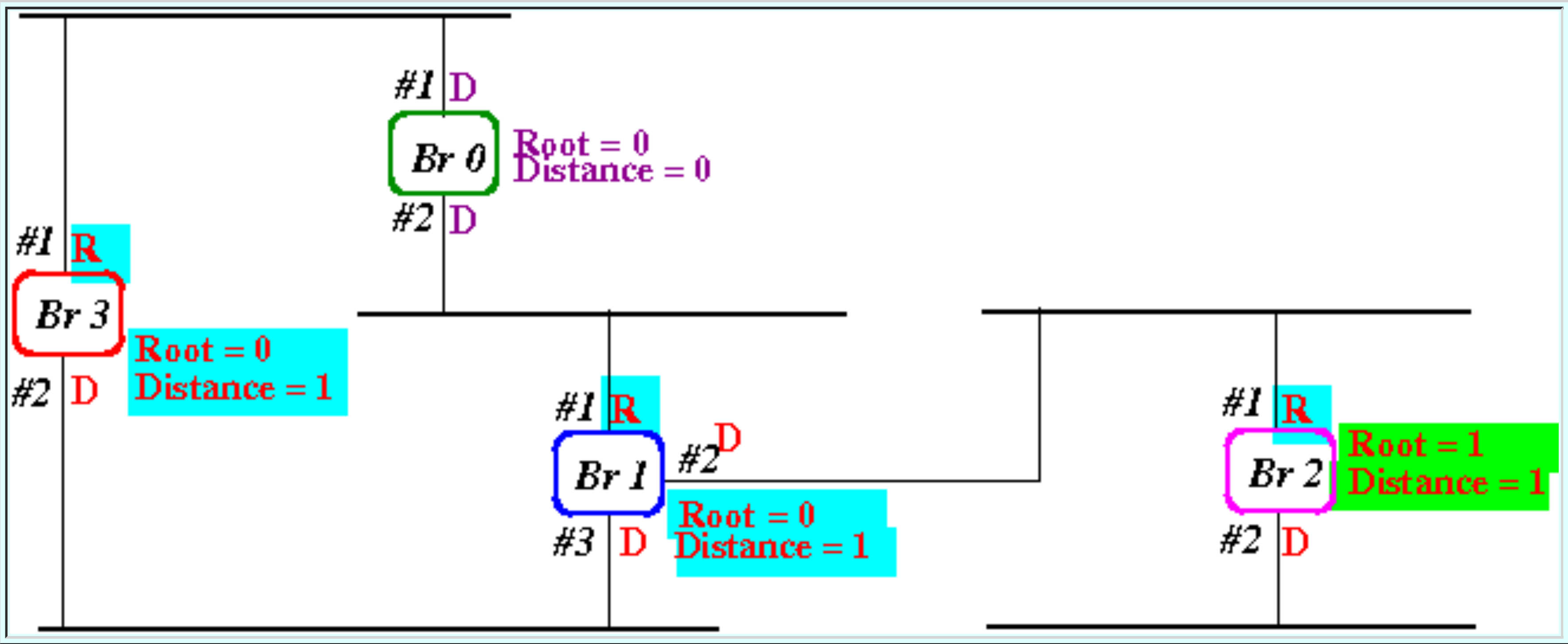
- **Bridge 1** and **bridge 3** found a ***better* root bridge 0**

    - Set **incoming port** to ***root***
    - Set ***other* ports** to ***designated***

- **Bridge 2** found a ***better* root bridge 1**

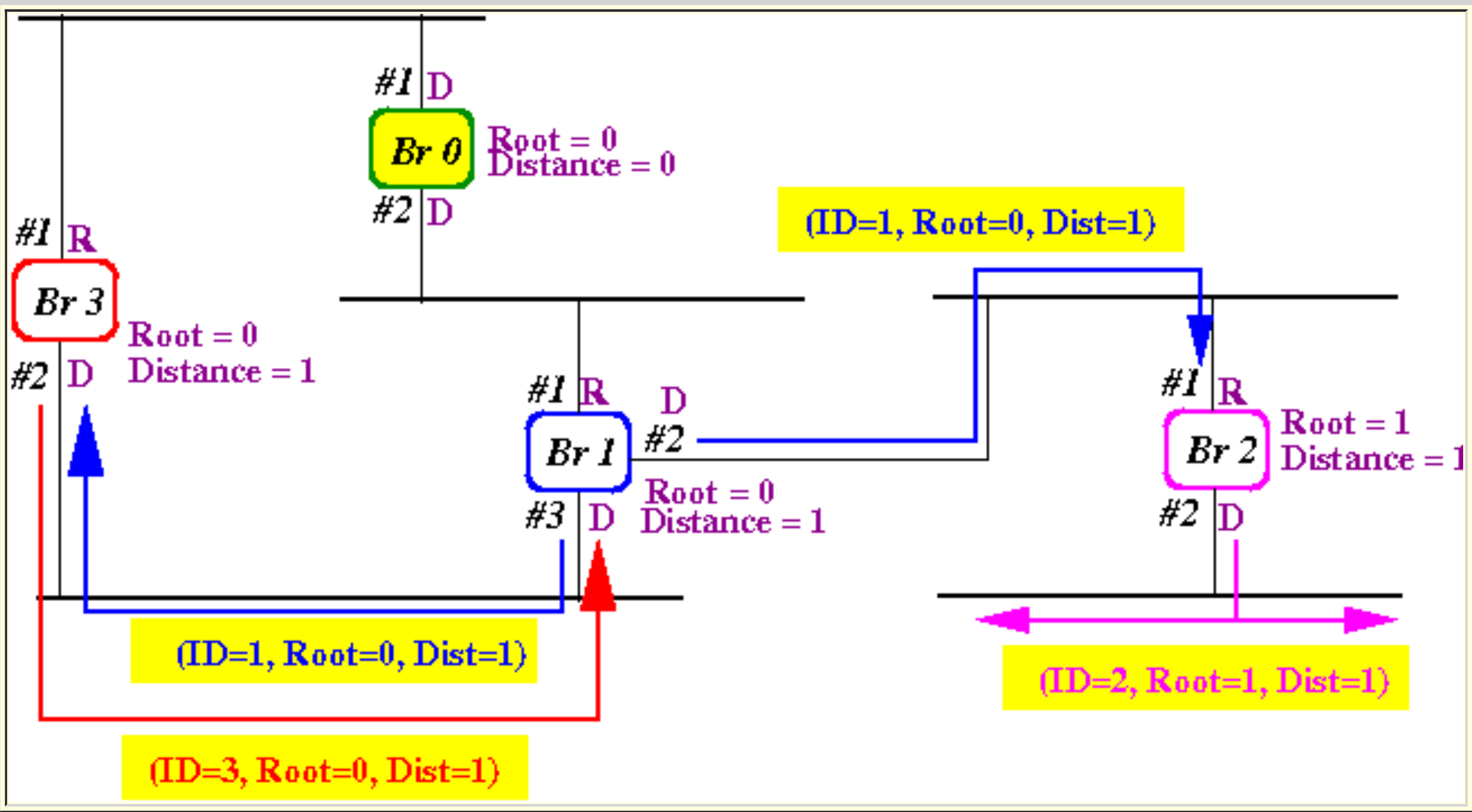
    - Set **incoming port** to ***root***
    - Set ***other* ports** to ***designated***

- **State** after the **computations:**

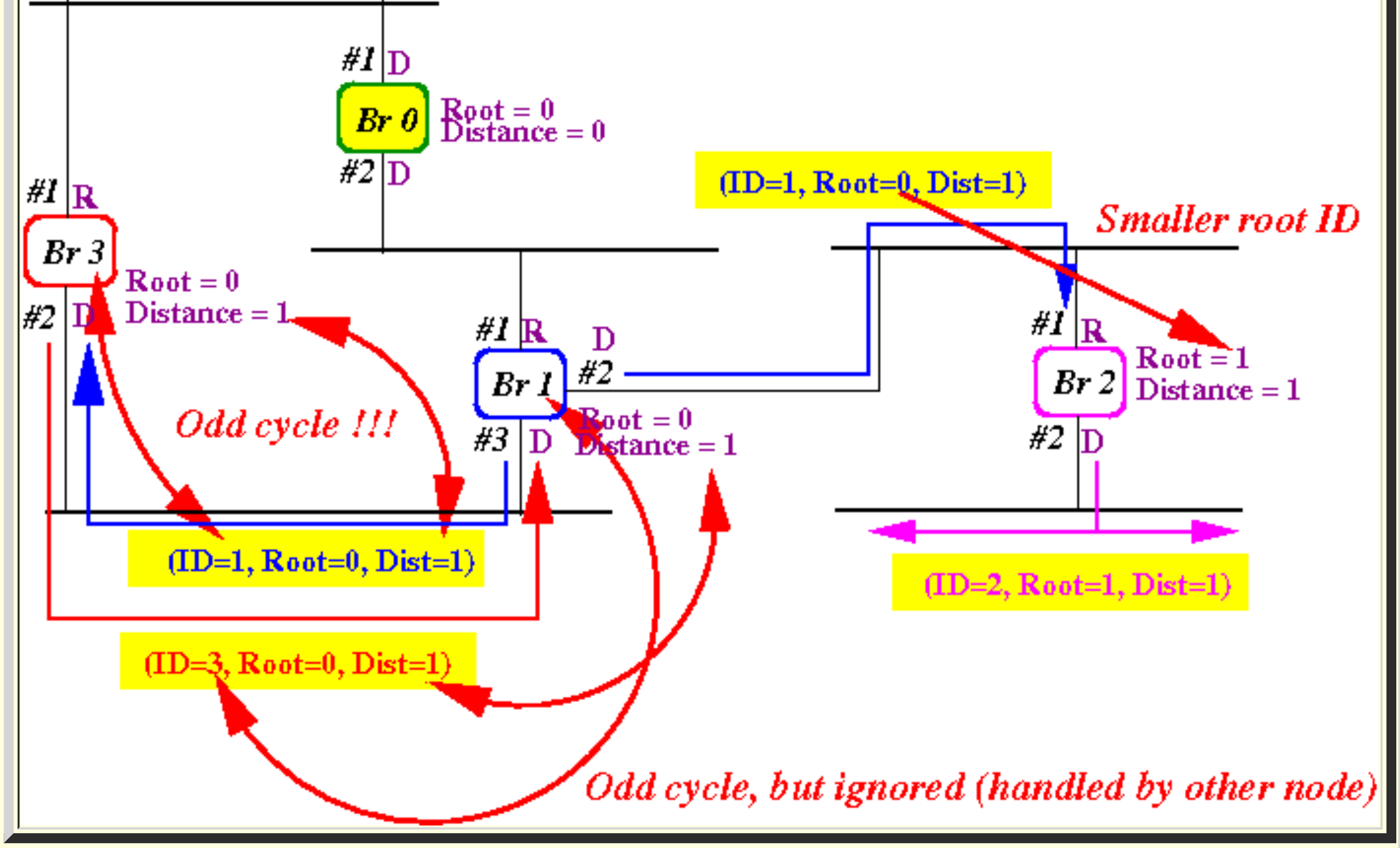


- ***Next***, **bridges** that ***computed*** a ***better* configuration** will ***forward*** a **configuration message** on its ***designated* port**:

    - **Messages** that are **sent:**



- **Computations** performed by **each node**:



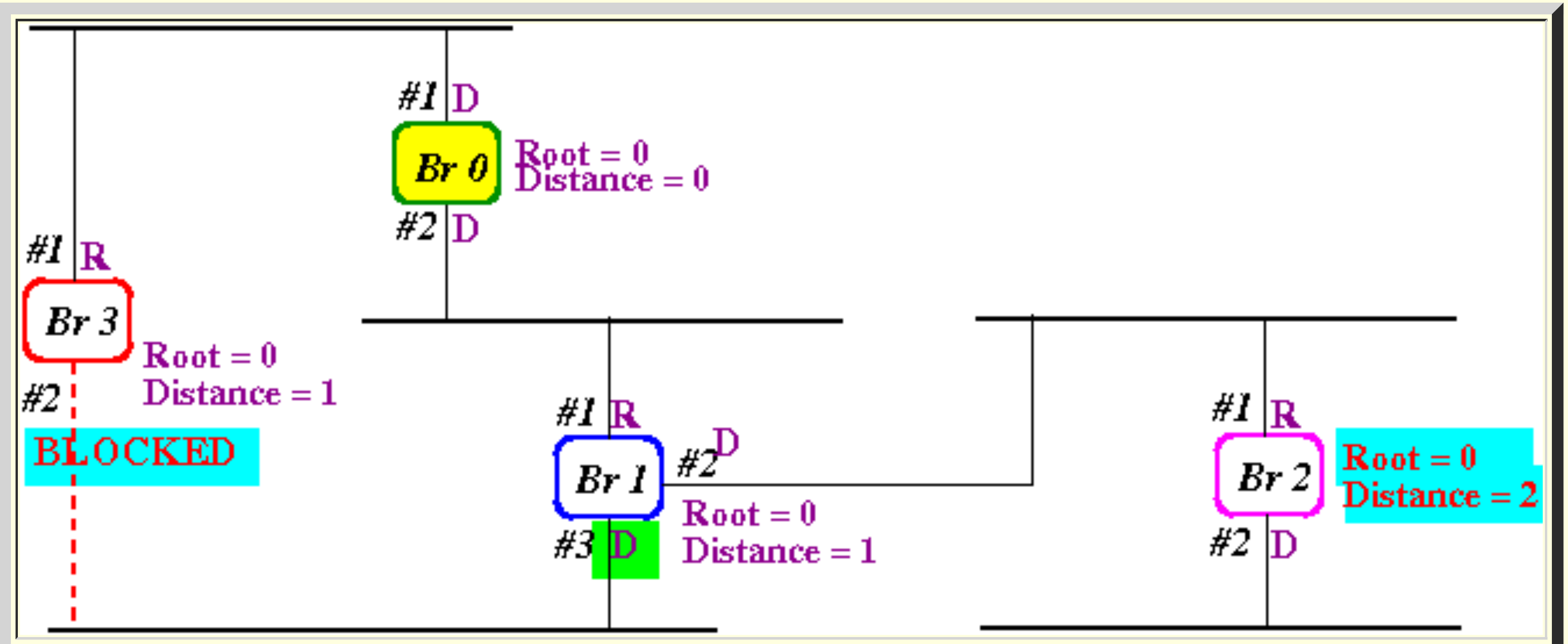
Explanation:

- **Bridge 1** detects an ***odd* cycle**
  - **However**, since the **ID (=3)** in the **message** is ***larger*** than the **bridge (= 1)**, **bridge 1** will do **nothing**

- **Bridge 3** ***also*** detects an ***odd* cycle**
  - **Because**, since the **ID = 1** in the **message** is ***smaller*** than the **bridge (3)**, **bridge 3** will change the **incoming port** to ***blocked***

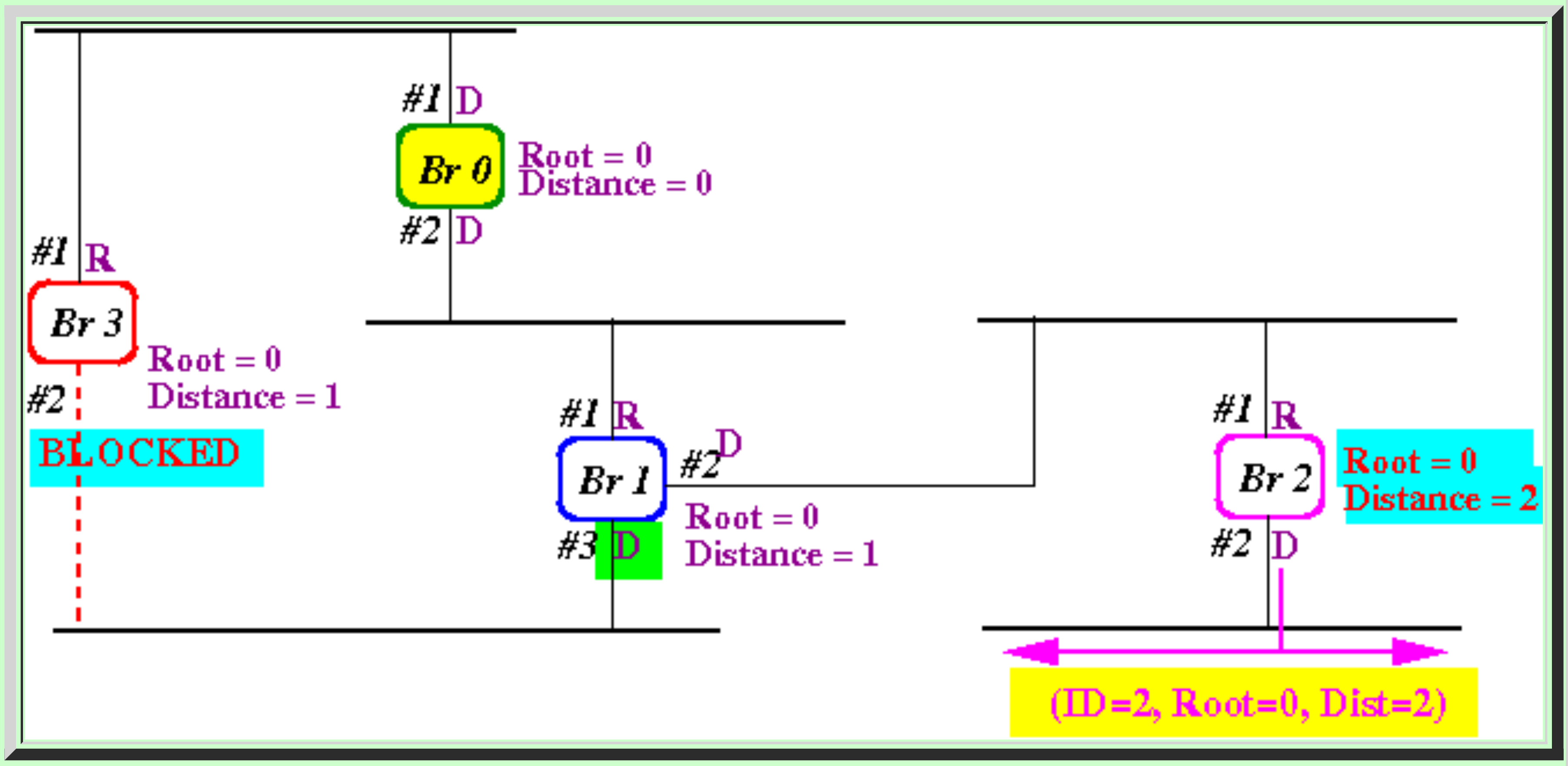
- **Bridge 2** found a ***better* root bridge 1**
  - Set **incoming port** to ***root***
  - Set ***other* ports** to ***designated***

- **State** after the **computations:**



- ***Again***, **bridges** that ***computed*** a ***better* configuration** will ***forward*** a **configuration message** on its ***designated* port**:

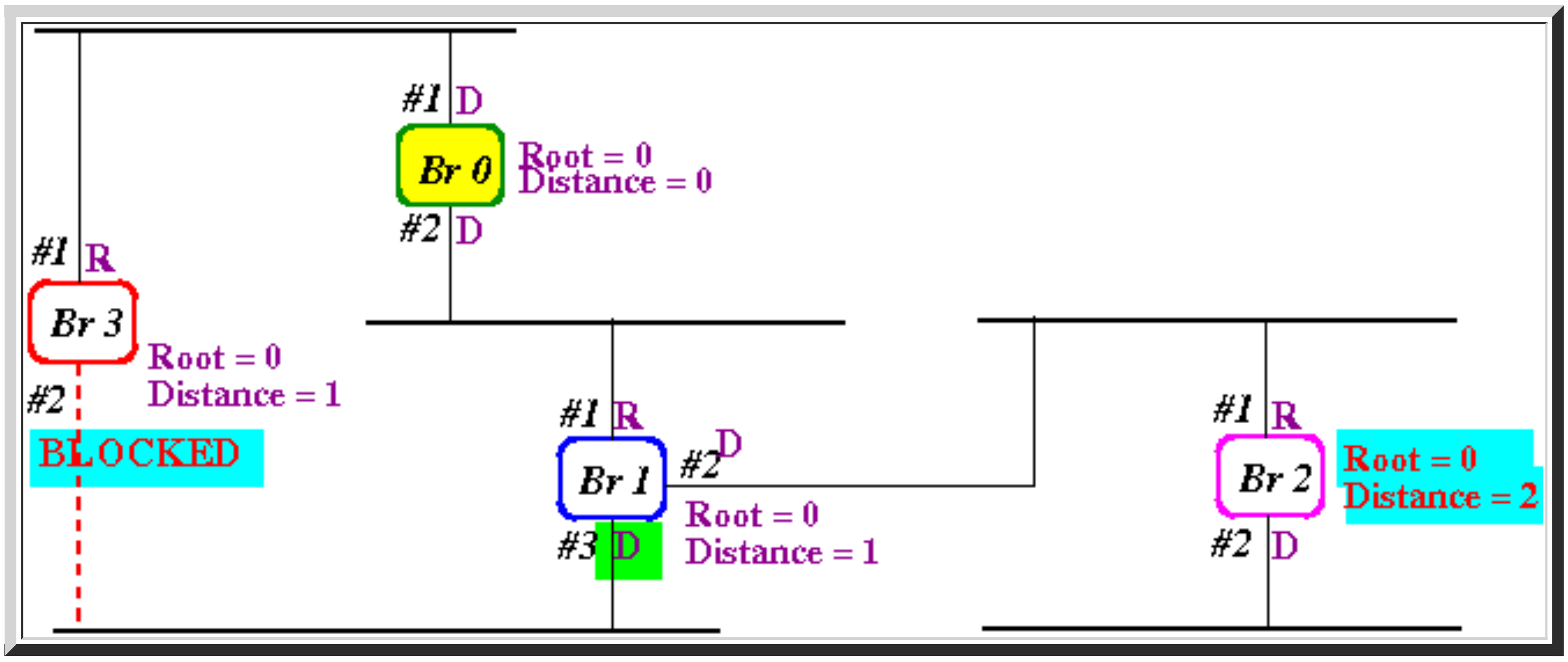
  - **Messages** that are **sent:**

  - No computations are performed....

    DONE !!!

- **Final configuration:**

  Observed that:

  - Because **port#2** of **Bridge 3** is **blocked**:

    - The **loop** in the **physical newtork** is **broken**

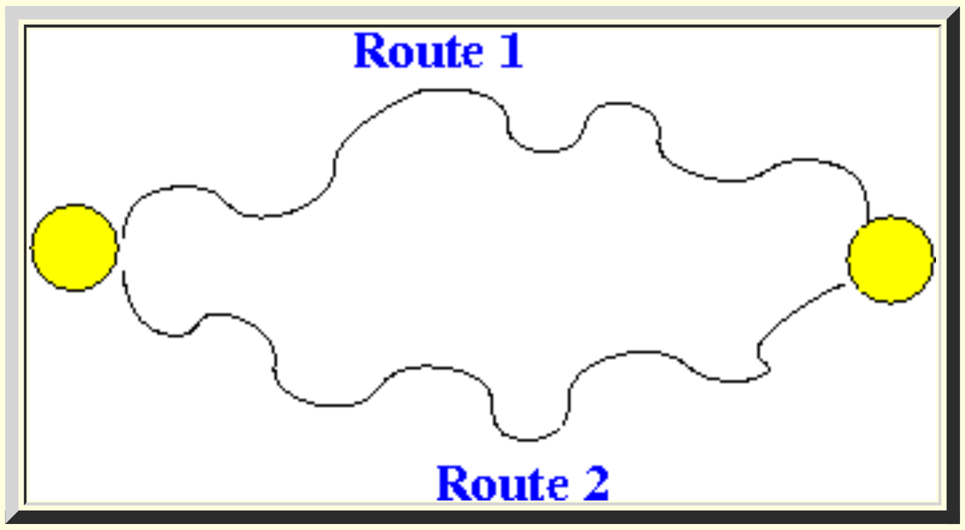
  - The **learning algorithm** of the **switches** (see: **click here**), the **forwarding database (table)** will ***now*** be **constructed correctly**...

    - **Recall** that the **learning technique** failed because of **loops**.....

(The **learning bridge technique** will **work** again)

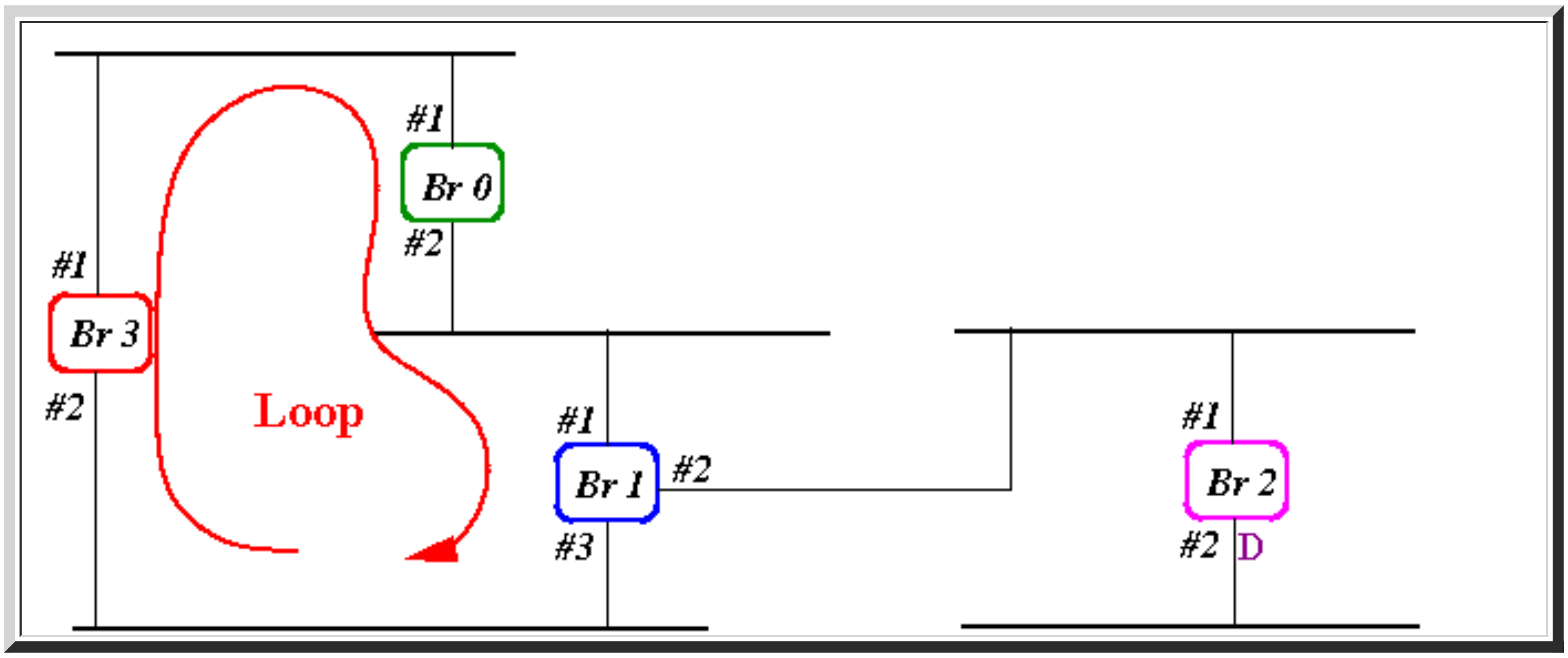
Fault Tolerant Operation of the IEEE 802.1D Spanning Tree Algorithm

- **Fault Tolerance: Recovery from Bridge Failure**
  - **Recall** the ***purpose*** of having **loops** in the **network**:
    - **Loops** are the **necessary evil** that comes with providing ***multiple*** **paths** between a **source** and a **destination**:

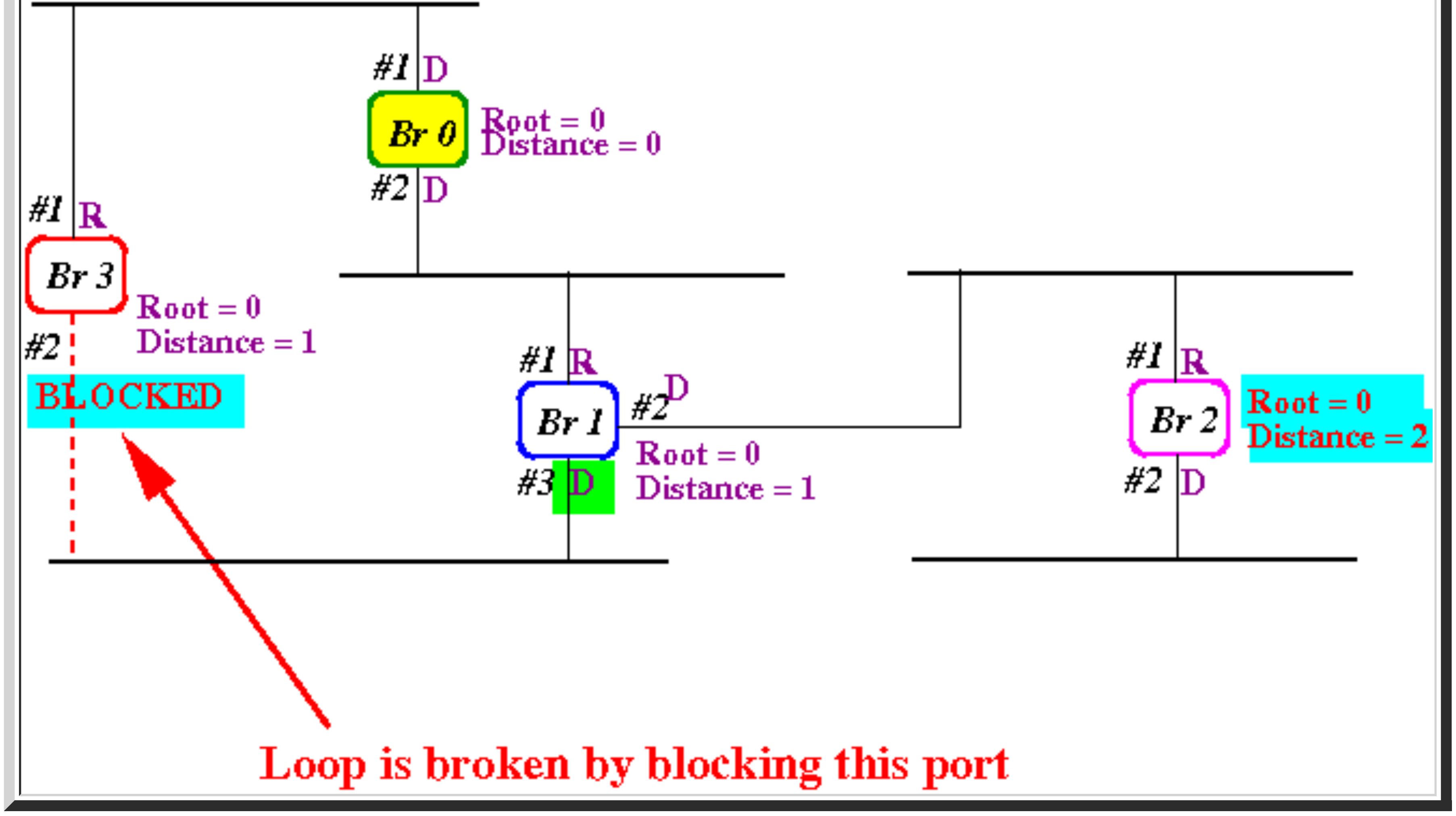


    - In other words: we **need** to have **loops** because we ***wanted*** **fault tolerance**

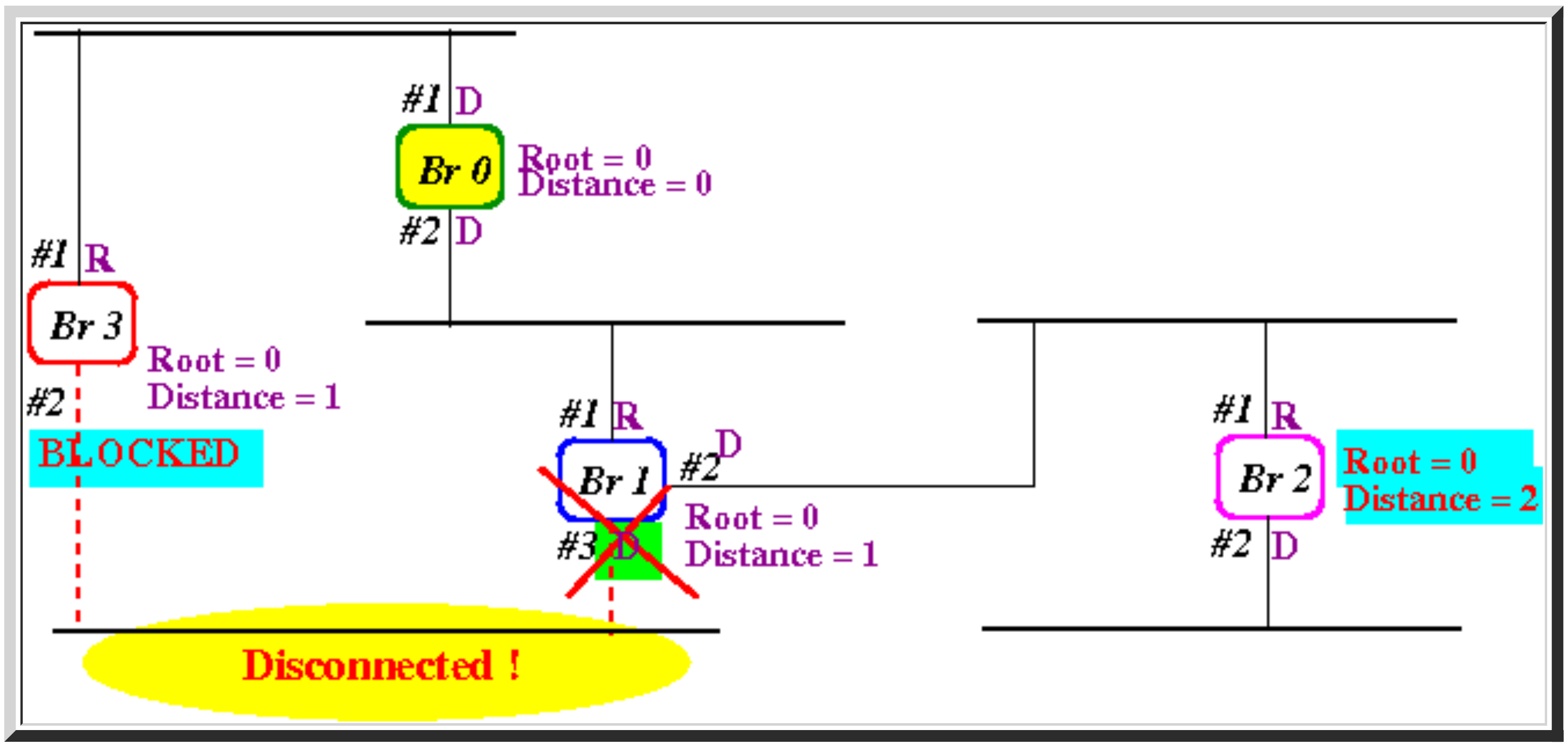
- **Fault tolerant operation**
  - **Recall** the ***physical*** **network** contains a **loop**:



  - We **constructed** a ***logical*** **network** that is **loop-free**:



- Now, **suppose port #3** of bridge **Br 1** fails:

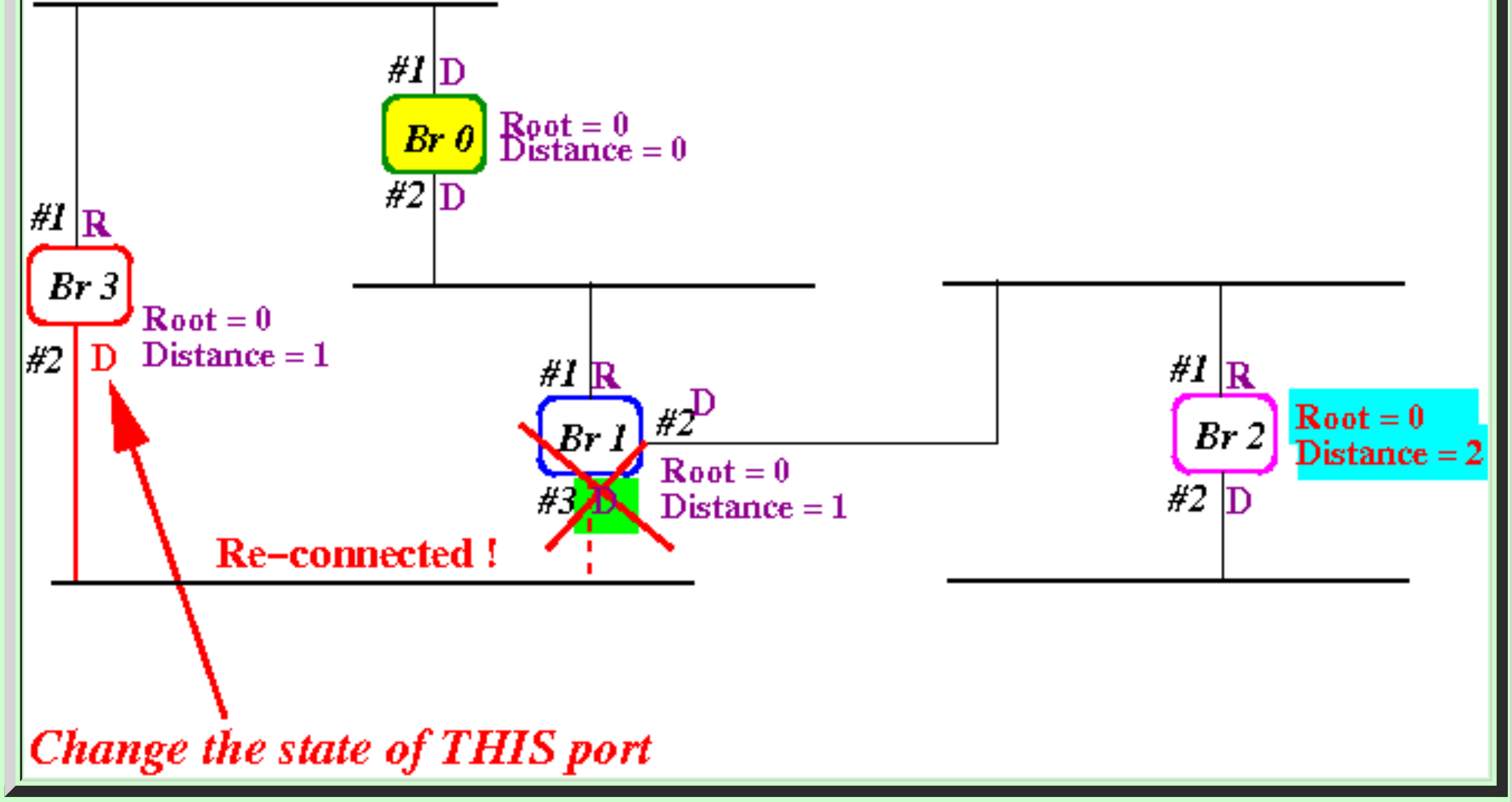


Result:

- **One** of the **LAN segment** is ***disconnected*** from the **rest of the network**...

We can ***repair*** the **failure** as **follows**:

- We can ***reconnect*** the **LAN segment** by ***unblocking*** **port #2** of **Bridge Br 3**:



- **$64,000 Question:**

    - **How** can we **unblock port #2** of **Bridge Br 3**

- ***Note*:**

    - The **solution must *not*** use ***human* intervention** !!!

Answer:

- Use a ***time out*** mechanism !!!

- **Timeout Mechanism in the 802.1D algorithm**

    - The **time out mechanism**:

        - Each ***state* variables** in a **bridge**:

            - **RootID**
            - **State** of **each port**

          has a **time out counter** associated with it

        - When the **time out *expires***:

- The **variable** is **reset** to its **default value**

- **Algorithm** for the **time out** setting of **default values**:

```
if ( RootID timer expires )
{  /* ===============================
      Reset to the initial state
      =============================== */
   RootID   = myID;
   Distance = 0;

   for ( each port P )
   {
      status[P] = D;
   }
}

if ( state[P] time expires )
{
   state[P] = D;
}
```

- **Note:**

  - ***Every* time out** mechanism needs a **timer *maintainance* procedures**:

    - The **time out *timer*** must be ***reset*** (from **time to time**) to **prevent** the ***time out*** from **occuring** !!!

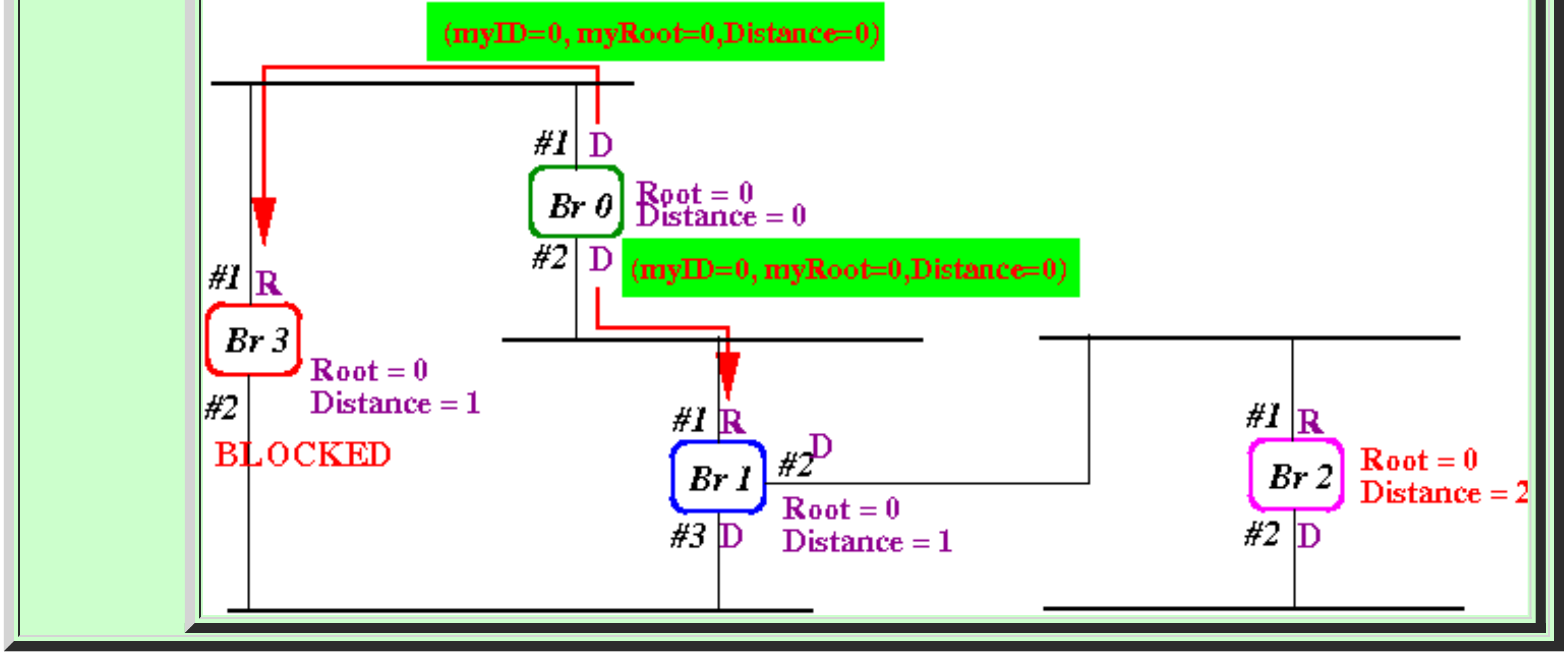
- **Time out maintainance procedure**

  - The **time out *maintenance* procedure:**

    - **Recall:**

      - A ***root* bridge** will **periodically** (= ***before*** the **time out expires**) transmits a **configuration message** !!!

      Graphically:



- The **(Distributed) Spanning Tree Algorithm** in **pseudo code**:

```
Let: msg = control message received
     p   = port on which the message msg was received


/* ===========================================
   Check for better root
   =========================================== */
if (  msg.rootID < bridge.rootID )
{
   /* =====================================
      Update state variables
      ===================================== */
   bridge.rootID   = msg.rootID;
   bridge.distance = msg.distance + 1;
   status[p]       = R;        // This is the root port

   for ( all ports q ≠ incoming port p )
   {
      status[q] = D;  // Other ports are now designated
   }

   /* =====================================
      Forward new state to neighbors
      ===================================== */
   for ( all ports q ≠ incoming port p )
   {
      send (bridge.ID, bridge.rootID, bridge.distance) on port q;
   }

   Reset all time out timers;
}
/* ===========================================
   Check for shorter path to root bridge
   =========================================== */
else if (  msg.rootID == bridge.rootID &&
           msg.distance + 1 < bridge.distance )
{
   /* =====================================
      Update state variables
      ===================================== */
   bridge.rootID   = msg.rootID;
   bridge.distance = msg.distance + 1;
   status[p]       = R;        // This is the root port

   for ( all ports q ≠ incoming port p )
   {
      status[q] = D;  // Other ports are now designated
   }
```

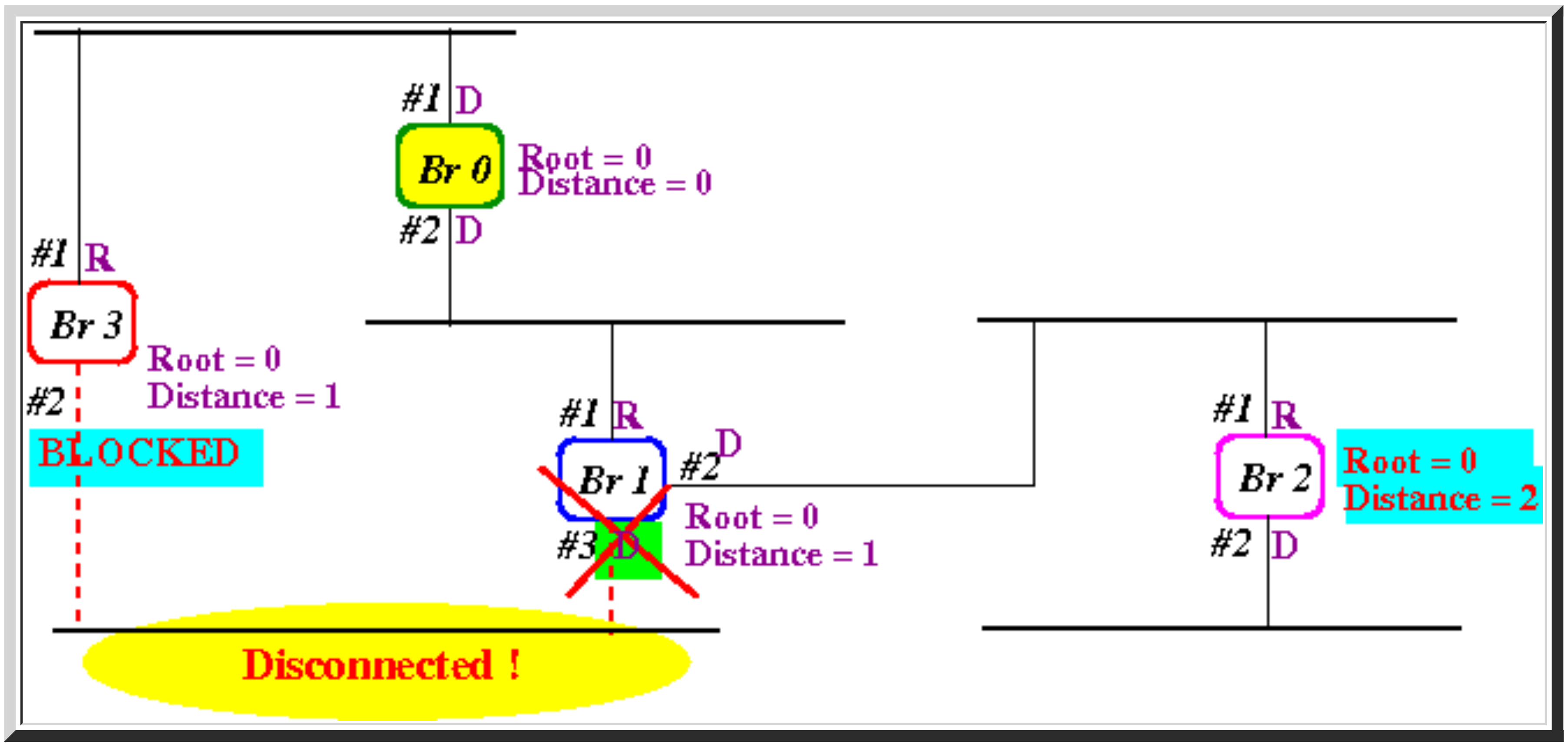
```
      /* ======================================
         Forward new state to neighbors
         ====================================== */
      for ( all ports q ≠ incoming port p )
      {
         send (bridge.ID, bridge.rootID, bridge.distance) on port q;
      }

      Reset all time out timers;
   }

   /* ===========================================
      Check for farthest node in even cycle
      =========================================== */
   else if (  msg.rootID == bridge.rootID &&
              msg.distance + 1 == bridge.distance )
   {
      if ( status[p] != R )
      {
         status[p] = B;         // Block the incoming port
         Reset time out for status[p];          // Refresh blocked port
      }

      /* ==================================================
         Time out counter maintenance

         Node is one of the last nodes in odd cycle:
             Send control message to your neighbor to keep
             port blocked
         ================================================== */
      if ( status[p] == R )
      {
         /* ==========================================
            My old rootID and distance value received AGAIN
            on my Root port !!!!
            **** must be maintenance time ****
            ========================================== */
         for ( all ports q ≠ incoming port p )
         {
            send (bridge.ID, bridge.rootID, bridge.distance) on port q;
         }
      }
   }
   /* ===========================================
      Check for farthest node in odd cycle
      =========================================== */
   else if (  msg.rootID == bridge.rootID &&
              msg.distance == bridge.distance )
   {
      if ( msg.ID < bridge.ID )
      {
         status[p] = B;         // Block the incoming port
         Reset time out for status[p];          // Refresh block port status
      }
   }
   else
   {
       // Do nothing, ignore a worse configuration
   }
```

Effect:

- The **processing** the **configuration control message** will
  - **Recomputes** the ***same* value** for each **state variable**

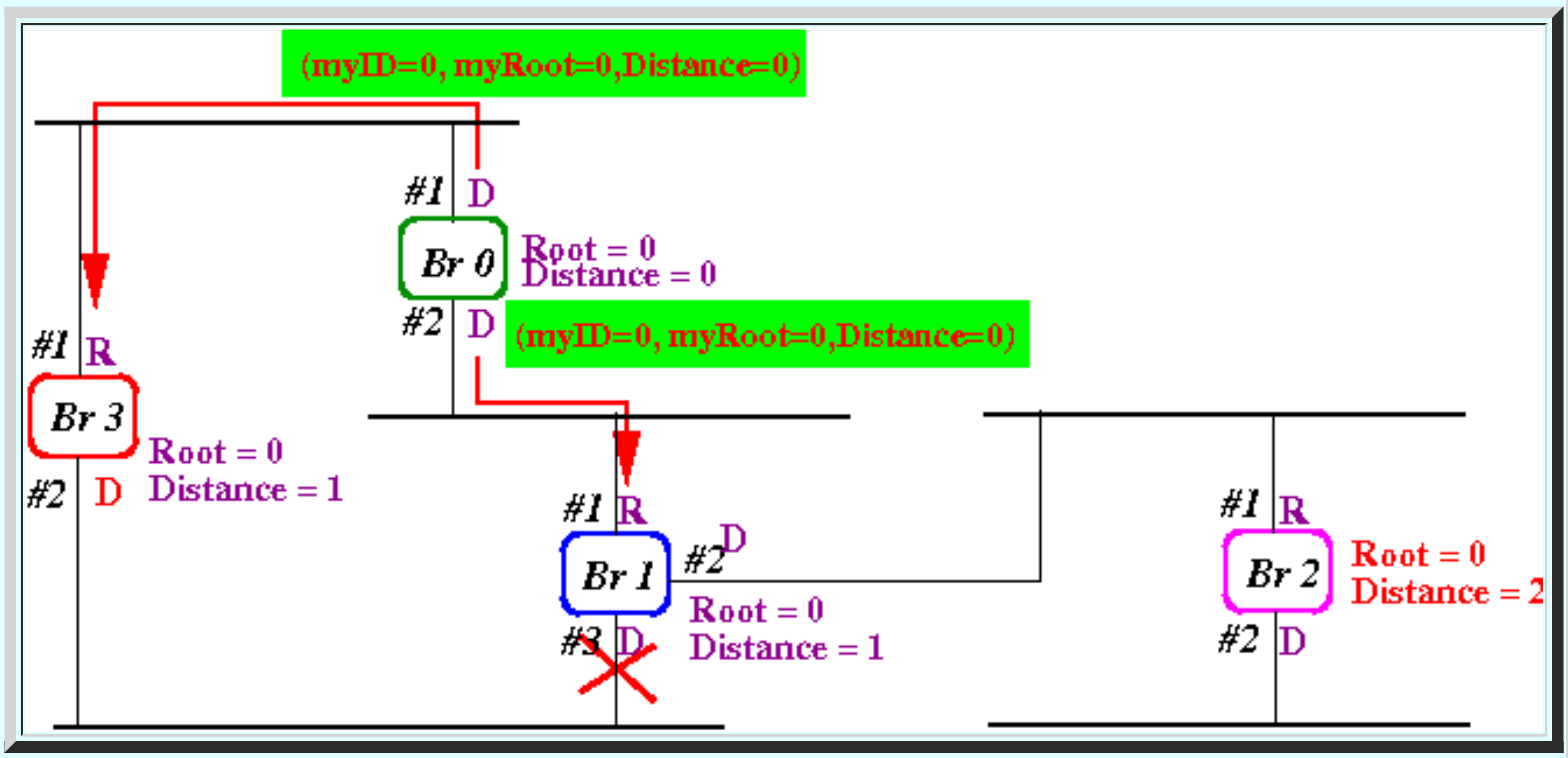
  - The ***time out* conuter** has been ***reset***

- **Recovery example**
  - ***Scenario:*** suppose **port #3** of bridge **Br 1** fails:



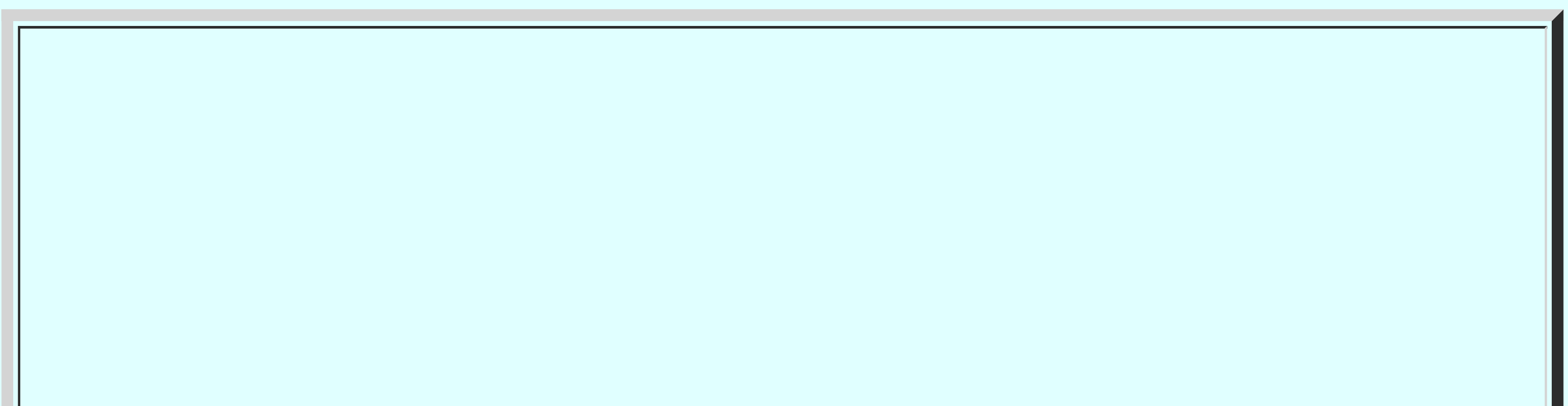
  - The **disconnected LAN** can be **reconnected** as **follows**:

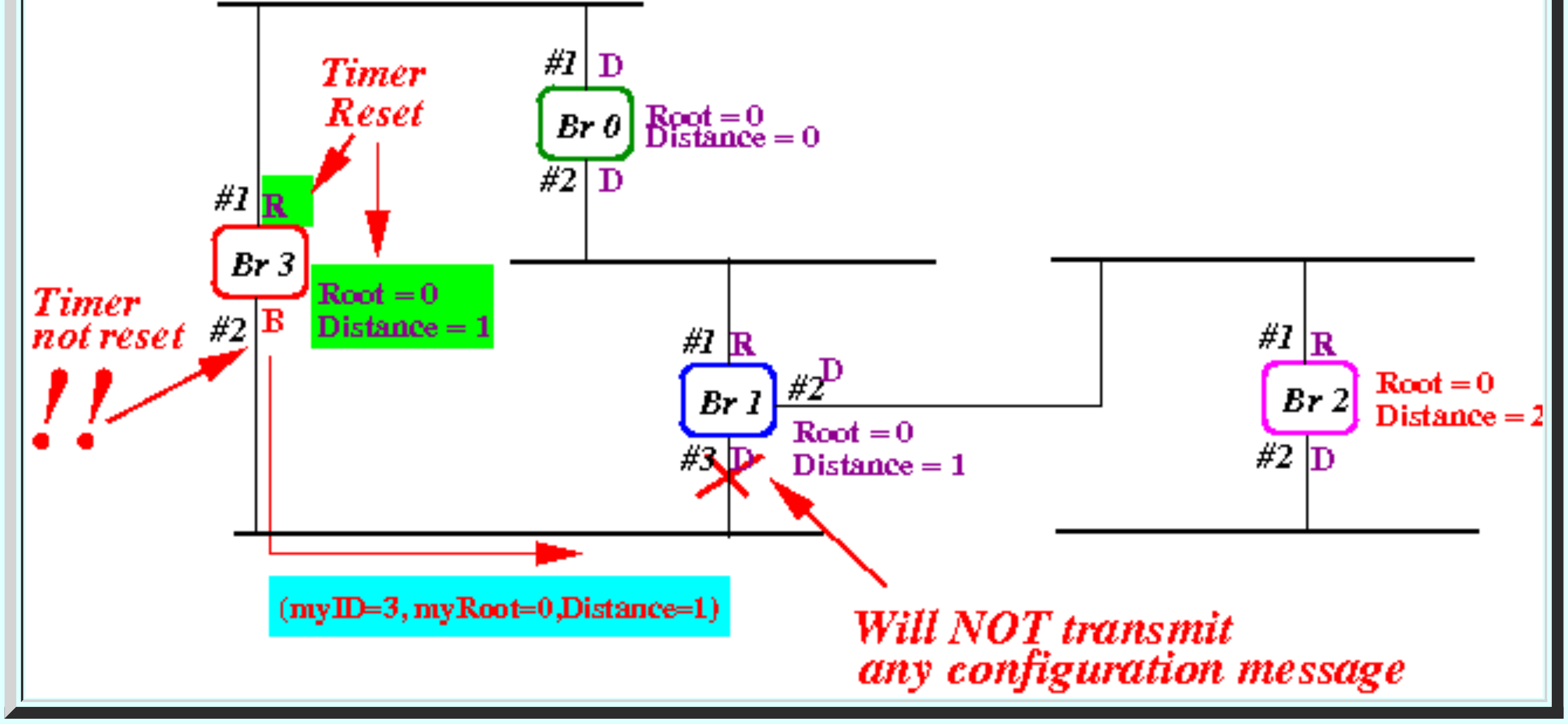
    - The **root bridge** transmits the (periodic) **configuration message**:



    - **Bridge 3** will **renew** the ***all*** **timers** **except** the timer for the **block port**

      (The **block state timer** must be **reset** by a **message** from **bridge 1 !!!**)





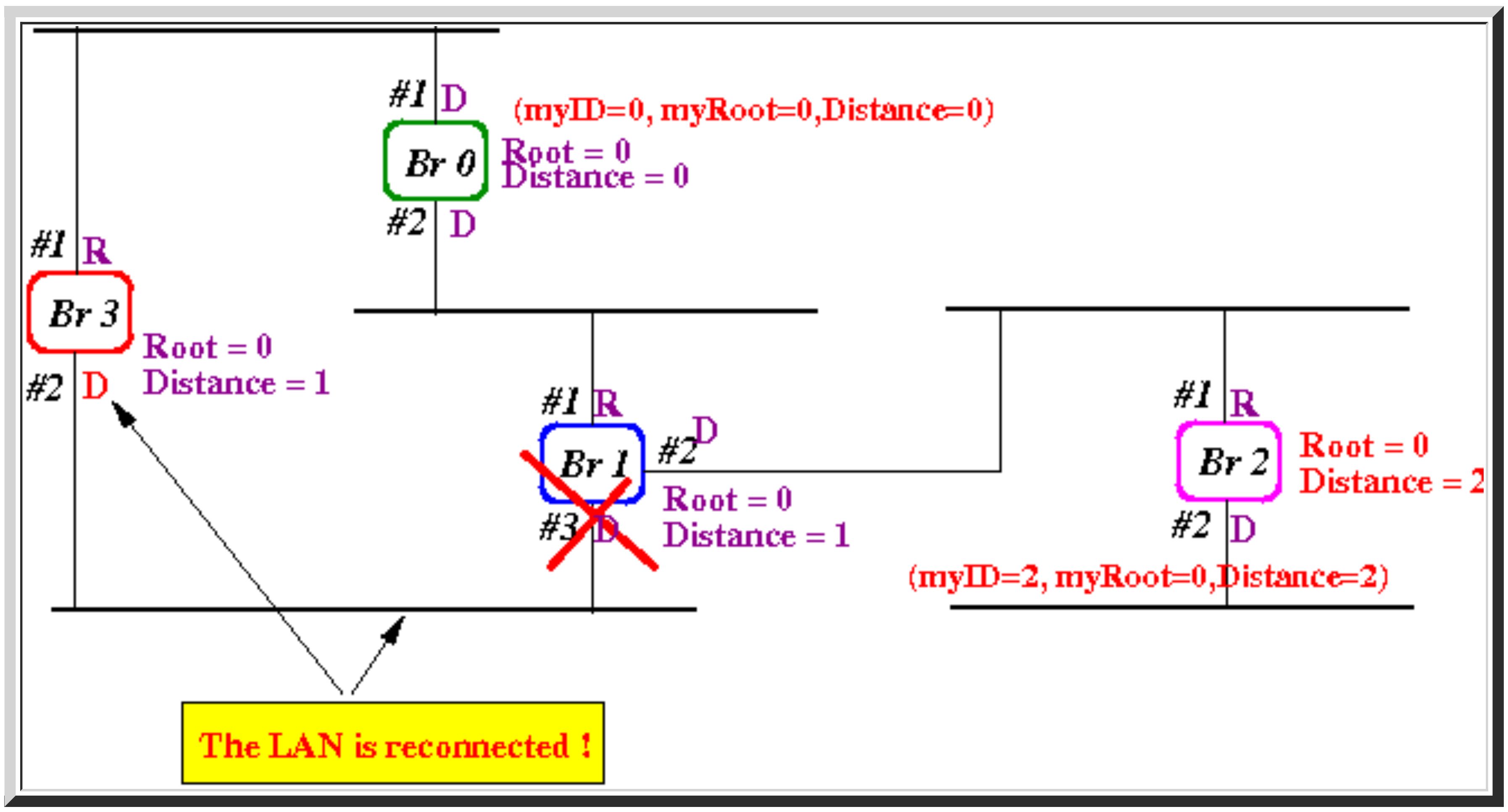
Notice:

- **Bridge 1** can **not** transmit a **configuration message** to **bridge 3** !!!

Result:

- The **timer** for the **state** of **port #2** of **bridge 3** will ***time out*** !!!
  - The **block state** will become ***designated***

- **Result:** the ***new*** **tree** is as follows:



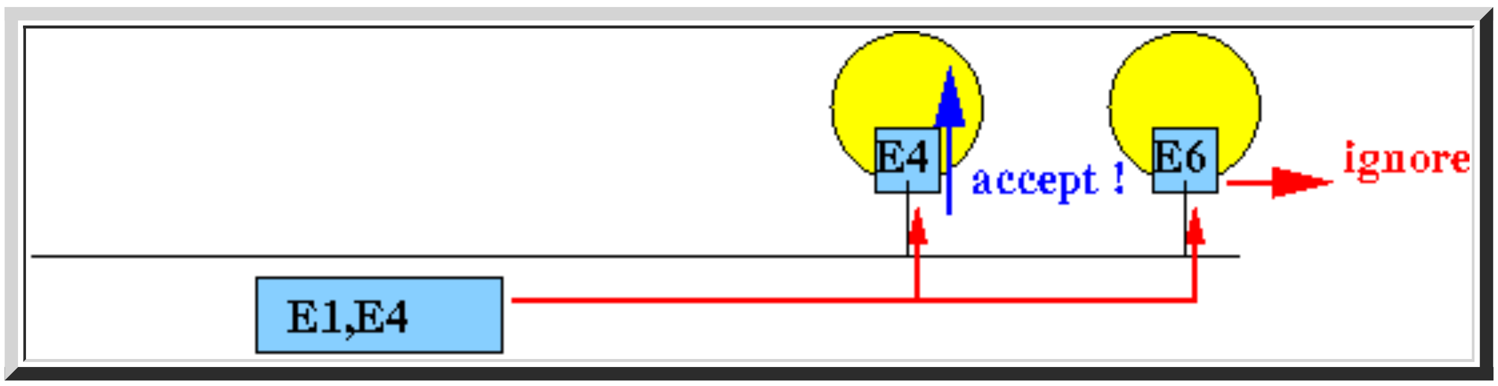
the **LAN** is **re-connected** to the **network** !!!

Promiscuous Interface and Broadcasting

- **Miscellaneous topics**
  - In the final webpage on **Ethernet**, I will discuss **2 random topics**:
    - **Promiscuous/Non-promiscuous ports**
    - **Broadcasting**

- **The *non-promiscuous* Ethernet Interface:**
  - **Recall:**
    - An **Ethernet network interface card (NIC)** has a **unique Ethernet Address**
    - The **Ethernet card** can **recognize** its own **network address** and **accept *only*** the **Ethernet frames** that contains its **network address**
  - **Nomenclature:**
    - ***Non*-promiscuous interface** = a **network interface** that **accepts *only*** frames with its **network address** as **destination**
  - There is ***another* type** of **Ethernet Interface**:
    - The ***promiscuous* interface** (which is found on **bridges** and **Ethernet switches**)

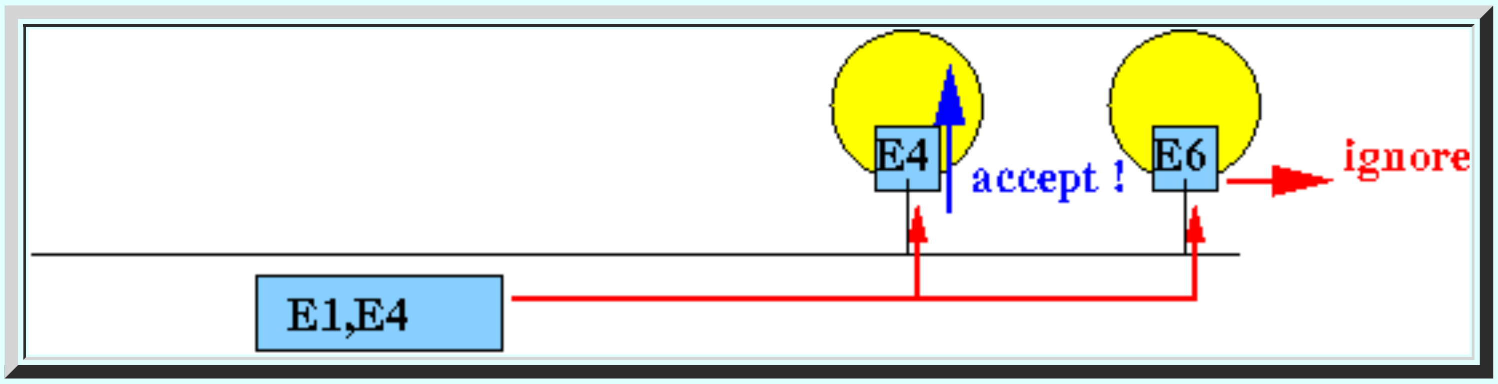
- **Operation of a Non-promiscuous Ethernet Interface:**



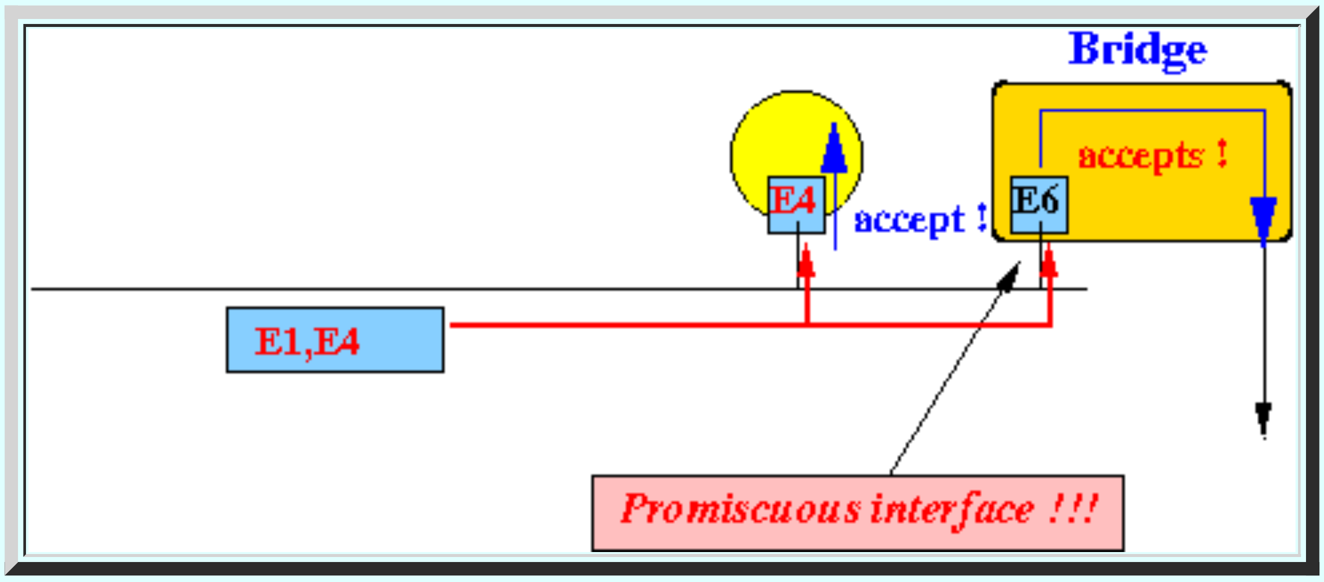
  - A Non-promiscuous port or interface will read the **destination address** in an Ethernet frame.
  - If the **destination address** matches the Ethernet Address of the interface, then the interface will accept (and

process) the Ethernet frame.

- **Operation of a Promiscuous Ethernet Interface:**
  - **Promiscuous port:**
    - **Promiscuous port** = a **network interface** that **accepts *every*** frame (and **process** it)
  - **Difference** between **promiscuous** and ***non*-promiscuous** port illustrated:
    - A ***non*-promiscuous** port **E6** will ***not*** accept a **frame** destined for **E4**:



    - A ***promiscuous*** port **E6** will ***accept* (and process)** a **frame** destined for ***another* destination E4**:



- **Difference between Ethernet bridges/switches and Ethernet hosts**
  - **Facts:**
    - **Ethernet bridges/switches** have ***promiscuous* ports** (because a **bridge** must **process *all* messages** --- even those that does **not** contain the **MAC address** of the **bridge**)
    - **Ethernet hosts (e.g., computer)** have ***non-promiscuous* ports**

- **Broadcasting**

  - **Broadcasting:**

    - **Broadcasting** = transmit a **frame** to ***all* nodes** in the **network**

      (**I.e.**, ***all* nodes** will **accept** the **frame** and **process** it)

  - **Ethernet** (and **token ring**) networks has a **built-in broadcast capability**

    - That's because a **frame** will **reach *all* nodes** in the **network** !!!
    - Some **network features** (discussed later) will **exploit** this inherent **broadcast capability**

  - **Addressing *all* node** in the **network**:

    - **Broadcast address** = a **special network address** used to **identify *every* node** in the **network**

    Broadcast address of **Ethernet**:

    ```
    Ethernet broadcast address = 11111111 11111111 11111111 11111111 11111111 11111111
    ```

  - Example: **broadcasting**