The Babel DV Routing Protocol

Leonardo Maccari

December 7, 2017

Primer on Distance Vector [1]

- Distance vector routing allows routers to automatically discover the destinations reachable inside the network
- Distance vector routing is completely distributed, meaning that no node has the full knowledge of the whole network topology.
- The shortest path is computed based on metrics or costs that are associated to each link.

[1] http://cnp3book.info.ucl.ac.be/2nd/html/principles/network.html

RT definition:

Each router maintains a routing table R, for each destination d it includes the following attributes :

- R[d].link is the outgoing link for packets to destination d
- R[d].cost is the sum of the metrics of the links that compose the shortest path to reach destination d
- R[d].time is the timestamp of the last distance vector containing destination d

DV Principles

- Each router regularly sends its distance vector over all its interfaces.
- The distance vector is a summary of the router's routing table including for each d, the cost of the path.
- In principle, this is all that is required for DV routing to work

```
Every N seconds:
 v=Vector()
 for d in R[]:
    # add destination d to vector
    v.add(Pair(d,R[d].cost))
 for i in interfaces
    # send vector v on this interface
    send(v,interface)
```

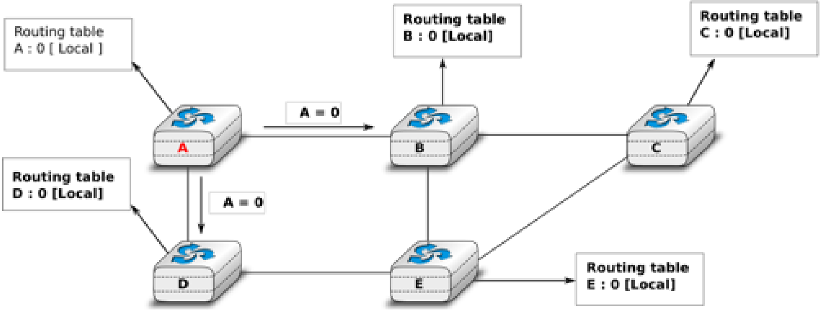
- When a router boots, it does not know any destination in the network and its routing table only contains itself.
- It thus sends to all its neighbours a distance vector that contains only its address at a distance of 0.
- When a router receives a distance vector on link l, it processes it as follows.

When receiving a DV message on link l:

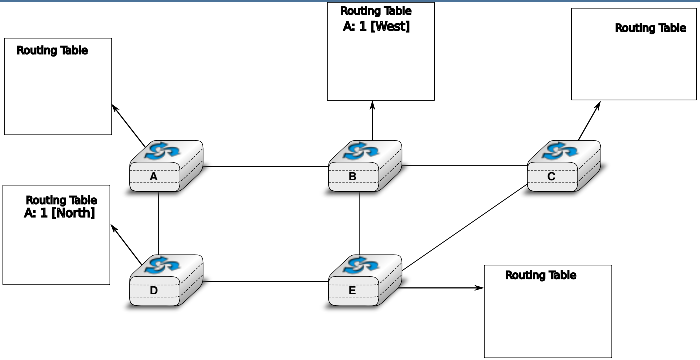
- The router iterates over all addresses included in the distance vector.
- If the distance vector contains an address d that the router does not know, it inserts d in its routing table, with:
  - R[d].link = l
  - R[d].cost = sum between the distance indicated in the distance vector and the cost associated to link l.
- If the destination was already known by the router, it only updates the corresponding entry in its routing table if either :
  - the cost of the new route is smaller than the cost of the already known route ((V[d].cost+l.cost) < R[d].cost)
  - the new route was learned over the same link as the current best route towards this destination (R[d].link == l)

- The first condition ensures that the router discovers the shortest path towards each destination.
- The second condition is used to take into account the changes of routes that may occur after a link failure or a change of the metric associated to a link.

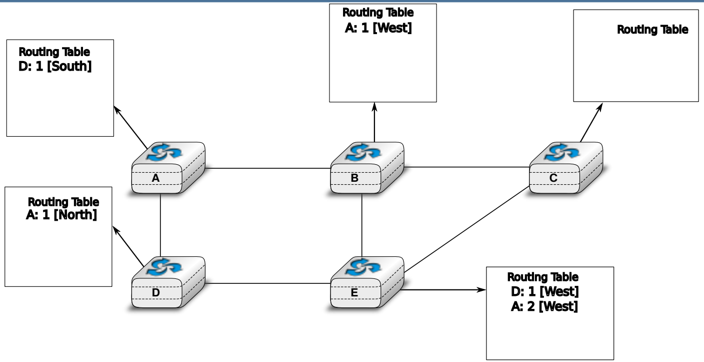
```
def received(V,l): # received vector V from link l
  for d in V[]
    if not (d in R[]): # new route
      R[d].cost=V[d].cost+l.cost
      R[d].link=l
      R[d].time=now
    else: # existing route
      if (( (V[d].cost+l.cost) < R[d].cost) or
            (R[d].link == l) ):
       # Better route or update existent route
       R[d].cost=V[d].cost+l.cost
       R[d].link=l
       R[d].time=now
```



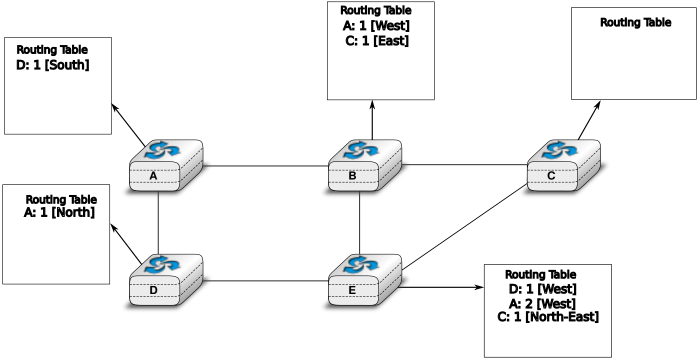
Assume that A is the first to send its distance vector [A=0].



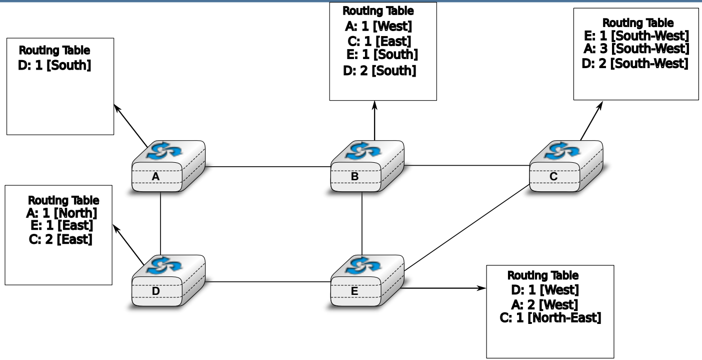
B and D process the received distance vector and update their routing table with a route towards A.



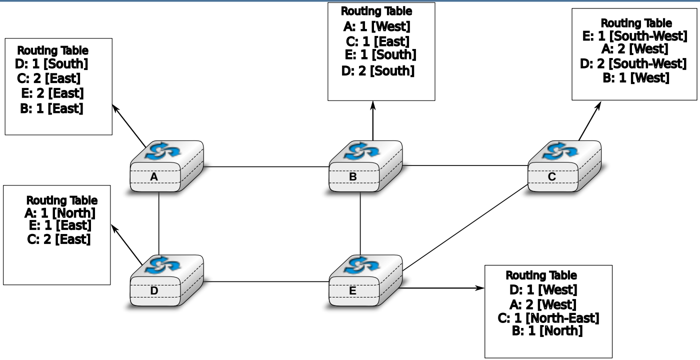
D sends its distance vector [D=0,A=1] to A and E. E can now reach A and D.



C sends its distance vector [C=0] to B and E

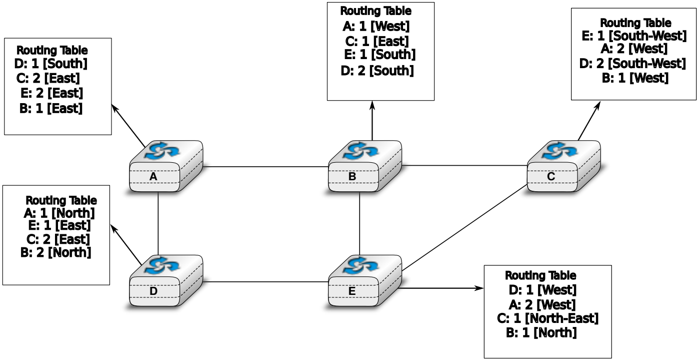


E sends its distance vector [E=0,D=1,A=2,C=1] to D, B and C. B can now reach D and E, C can reach D and A (with a 3-hop path)



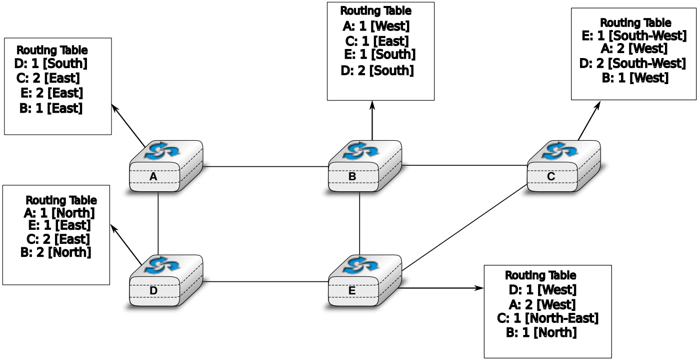
B sends its distance vector [B=0,A=1,C=1,D=2,E=1] to A, C and E. A, B, C and E can now reach all destinations.

DV: Example



A sends its distance vector [A=0,B=1,C=2,D=1,E=2] to B and D.

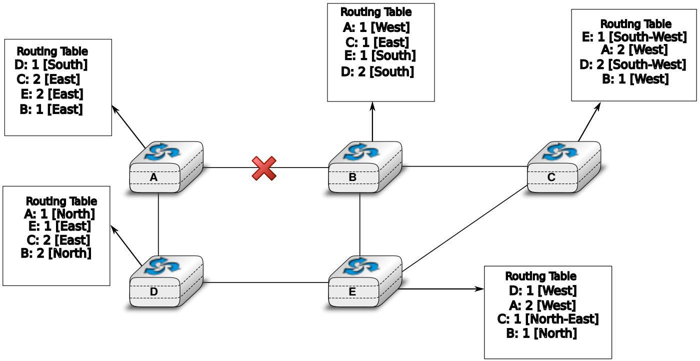
DV: Example



At this point, all routers can reach all other routers

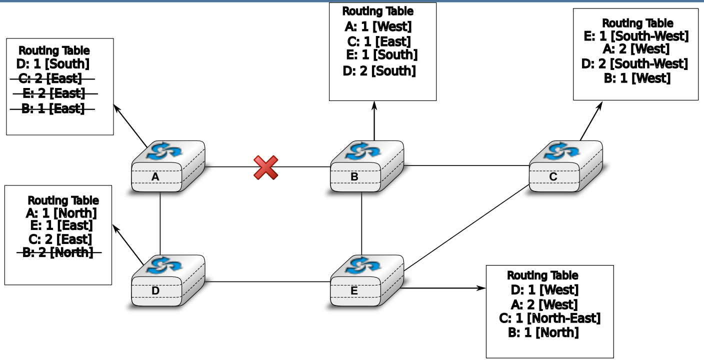
- As all routers send their distance vector every N seconds, the timestamp of each route should be regularly refreshed
- No route should have a timestamp older than N seconds, unless the route is not reachable anymore
- To cope with transmission errors, routers periodically check the timestamp of each route and remove the routes that are older than 3xN seconds

- When a route expires, the router must first associate an ∞ cost to this route and send its distance vector to its neighbours to inform them.
- The route can then be removed from the routing table after some time (e.g. 3xN seconds), to ensure that the neighbouring routers have received the bad news, even if some distance vectors do not reach them due to transmission errors.

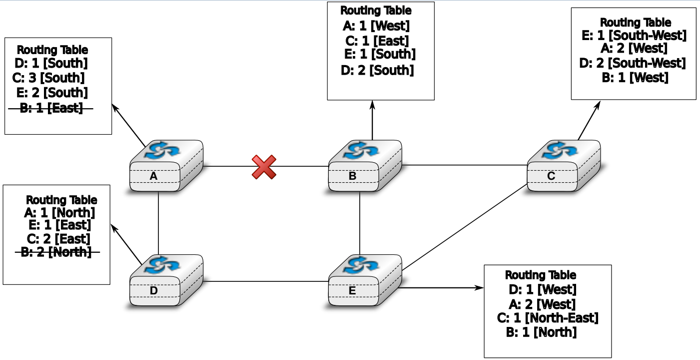


link between routers A and B fails.

DV: Example

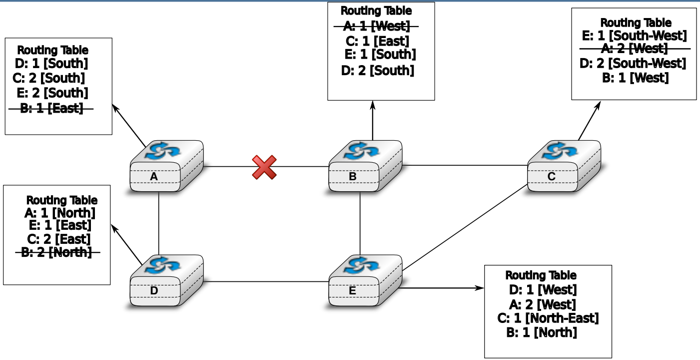


A sends its distance vector [A=0,B=∞,C=∞,D=1,E=∞] D knows that it cannot reach B anymore via A

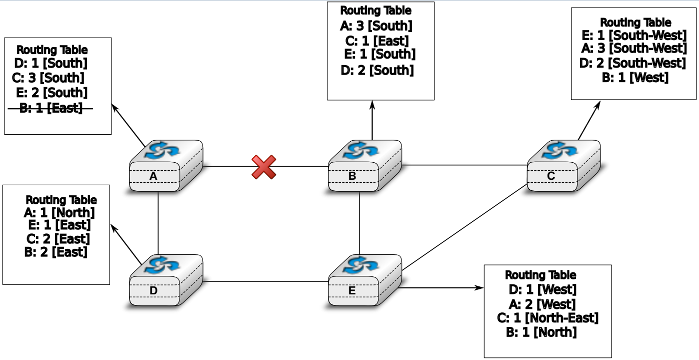


D sends its distance vector [D=0,B=∞,A=1,C=2,E=1] to A and E. A recovers routes towards C and E via D.

DV: Example

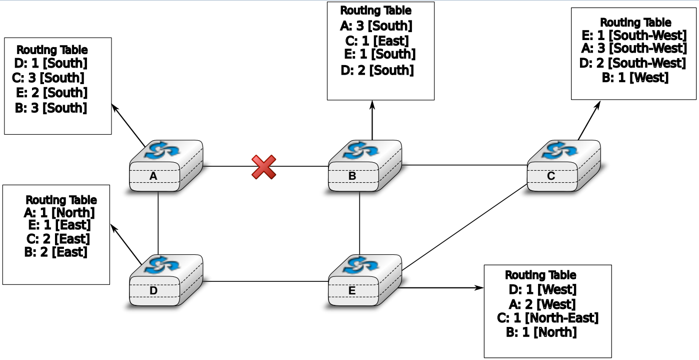


B sends its distance vector [B=0,A=∞,C=1,D=2,E=1] to E and C. C learns that there is no route anymore to reach A via B.

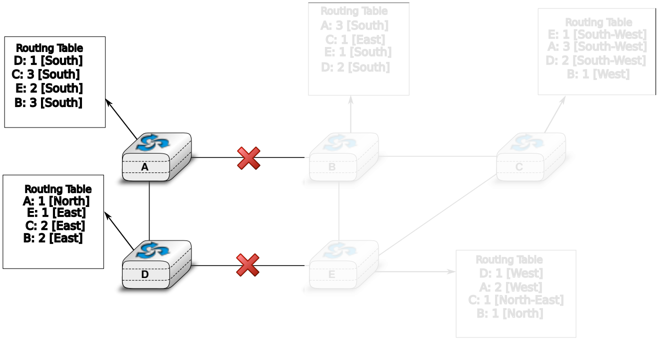


E sends its distance vector [E=0,A=2,C=1,D=1,B=1] to D, B and C. D learns a route towards B. C and B learn a route towards A.

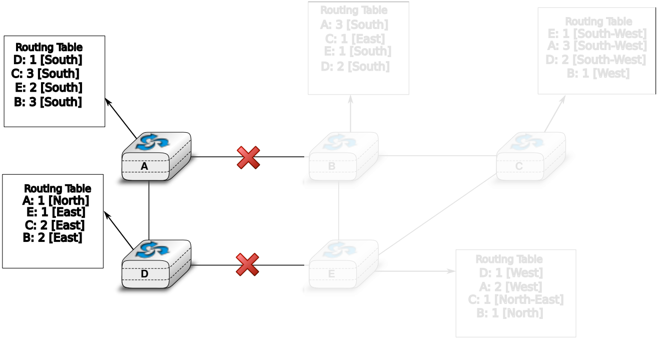
DV: Example



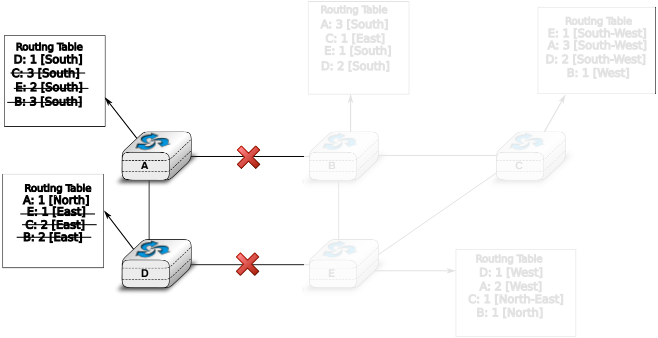
D sends its updated distance vector [A=1,B=2,C=2,D=1,E=1], A recovers the route towards B



Now also the link between D and E fails. The network is now partitioned into two

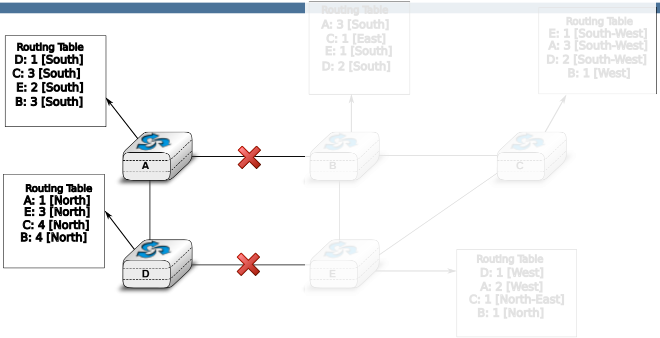


The routes towards B, C and E expire first on router D, D sends $[D=0, A=1, B=\infty, C=\infty, E=\infty]$

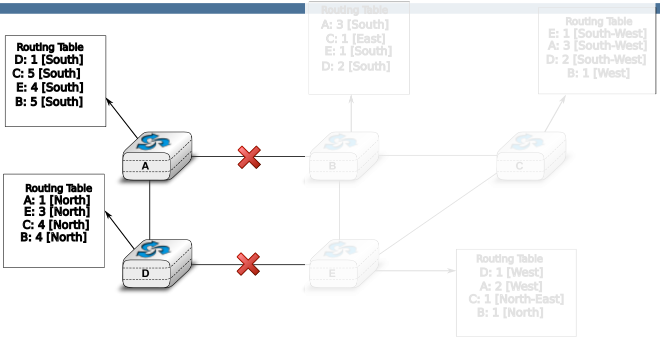


A learns that B, C and E are unreachable and updates its routing table.

Count to Infinity



If the distance vector sent to A is lost or if A sends its own distance vector ([A=0,D=1,B=3,C=3,E=2]) at the same time as D sends its distance vector, D updates its routing table to use the shorter routes advertised by A towards B, C and E.



After some time D sends a new distance vector : [D=0,A=1,E=3,C=4,B=4]. A updates its routing table and after some time sends its own distance vector [A=0,D=1,B=5,C=5,E=4], etc.

Routers A and D exchange distance vectors with increasing costs until these costs reach ∞. Count to Infinity!

DV may suffer from count to infinity problems in other scenarios if there is a cycle in the network.

- This count to infinity problem occurs because router A advertises to router D a route that it has learned via router D.
- A possible solution is to that router A could create a distance vector that is specific to D and contains the routes that have not been learned via D

```
# one vector for each interface
for l in interfaces:
  v=Vector()
  for d in R[]:
    if (R[d].link != l) :
       v=v+Pair(d,R[d.cost])
  send(v)
```

With the Poison Reverse Variant, a route is sent with ∞ cost

- Babel is a mostly loop-free distance vector protocol based on the Bellman-Ford protocol
- Babel includes a number of refinements that either prevent loop formation altogether, or ensure that a loop disappears in a timely manner and doesn't form again.

- A Babel node periodically broadcasts `H` messages to all of its neighbours;
- It also periodically sends an `IHU` ("I Heard You") message to every neighbour from which it has recently heard a `H`.
- From the information derived from `H` and `IHU` messages received from its neighbour B, a node A computes the cost C(A, B) of the link from A to B.
- cost is normally ETX, but could be something different

- In the rest of the text, S is always the node on which we are building the route for
- Given a route between any two nodes, the metric of the route is the sum of the costs of all the edges along the route: D(B) is the cost of the path from B to S.
- The goal of the routing algorithm is to compute, for every source S, the tree of the routes of lowest metric to S.

- We have seen that with Bellman-Ford loops can be created, and that split-horizon helps to solve them
- In a wireless network you may not have one interface per link, so you can not apply split-horizon
- How do you prevent loops?
- Babel uses a strict Feasibility Condition and sequence numbers to ensure loop-freedom

Key observation: Loops

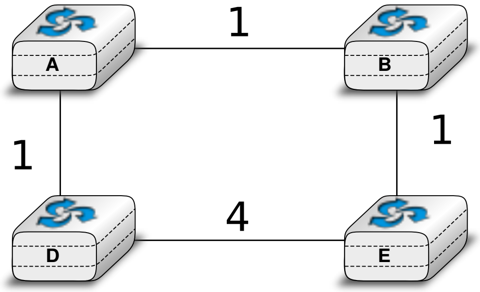
- A looped route is created when a piece of information (a route) travels from node A to B and then back from B to A.
- The associated cost must be larger than the cost of the real shortest path (if there is one)
- Therefore, a routing loop can only arise after a router has switched to a route with a larger metric than the route that it had previously selected.

- A feasibility condition is a condition applied to accept a routing update from a neighbor and helps preventing loops.
- Due to the previous observation, one could decide that a route is feasible only when its metric at the local node would be no larger than the metric of the currently selected route
- announcement carrying a metric D(B) is accepted by A when $C(A, B) + D(B) <= D(A)$.
- If all routers obey this constraint, if A has selected B as its successor, then D(B) $<$ D(A), which implies that the forwarding graph is loop-free.
- This condition is used in the DSDV protocol

- Babel uses a slightly more refined feasibility condition, used in EIGRP
- Call FD(A) the feasibility distance of A: the smallest metric that A has ever advertised for S
- An update sent by a neighbour B of A is feasible when D(B) $<$ FD(A).
- It can be shown that this condition is no more restrictive than the EIGRP one...

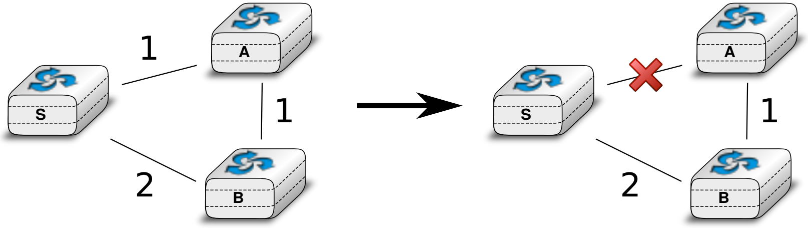
Babel Feasibility Condition

- Suppose that A obeys DSDV-feasibility; then D(A) <= FD(A) (actually D(A) = FD(A) most of the time, and D(A) < FD(A) in the transitory phase when the route has changed but it was not advertised yet)
- Now A receives a feasible update with a metric D(B)
- The update is DSDV-feasible $\rightarrow$ C(A, B) + D(B) <= D(A) $\rightarrow$ D(B) < D(A)
- and since D(A) <= FD(A) then D(B) < FD(A).
- Since the DSDV condition compares with D(B), then Babel condition (that compares with FD(A)) is no more restrictive



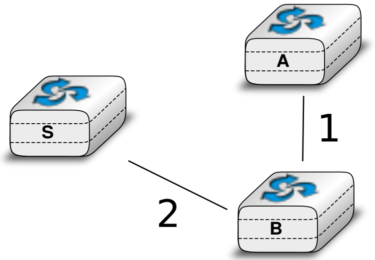
- E-B-A and E-D-A are both feasible for Babel, not for DSDV
- E-B-A is the chosen one, but E-D-A can be used if the first breaks
- not in DSDV

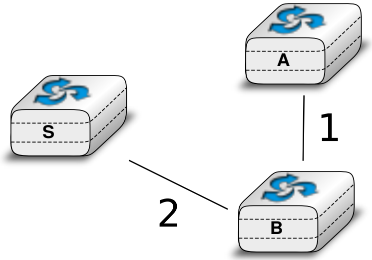
- when A accepts an update from B, D(B) < FD(A) and FD(B) <= D(B)
- then FD(B) < FD(A)
- Since this property is preserved when A sends updates, it remains true at all times
- Metrics are non-incremental, thus, the loop condition can not happen



- The feasibility condition produces starvation when a router remains without feasible routes to choose from
- A-B-S is not feasible!

- Babel solves this issue with sequenced routes, a technique introduced by DSDV
- In addition to a metric, every route in the DV message carries a sequence number, a nondecreasing integer that is propagated unchanged through the network and is only ever incremented by the source;
- a pair (s, m), where s is a sequence number and m a metric, is called a distance.
- A received update is feasible when either it is more recent than the feasibility distance maintained by the receiving node, or it is equally recent and the metric is strictly smaller.
- More formally, if FD(A) = (s, m), then an update carrying the distance (s', m') is feasible when either $s' > s$, or $s = s'$ and $m' < m$.





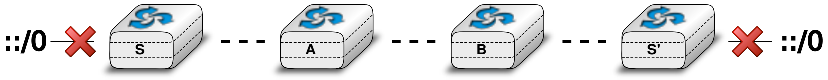
Breakage Time	After S sent a message
FD(A) = (137, 1)	FD(A) = (137, 1)
D(B) = (137, 2)	D(B) = (138, 2)
FD(B) = (137, 2)	FD(B) = (138, 2)
Unfeasible!	Feasible!

- If the sequence number of a source is increased periodically, the new sequence number may take a significant amount of time to be propagated.
- Babel instead sends requests when something seems to be broken
- When a node detects that it is suffering from a potentially spurious starvation, it sends an explicit request to the source for a new sequence number.
- This request is forwarded hop by hop to the source (with no regard to the feasibility condition).
- Upon receiving the request, the source increases its sequence number and broadcasts an update, which is forwarded to the requesting node.

- Note that not all such potentially parallel requests will, in general, reach the source, as some will be sent over links that are now broken.
- However, if the network is still connected, then at least one among the nodes suffering from spurious starvation has an (unfeasible) route to the source;
- hence, in the absence of packet loss, at least one such request will reach the source. (Resending requests a small number of times compensates for packet loss.)
- Since requests are forwarded with no regard to the feasibility condition, they may, in general, be caught in a forwarding loop; this is avoided by having nodes perform duplicate detection for the requests that they forward.

- There are cases in which the same prefix is originated by different routers, i.e. the default prefix.
- Babel treats routes for the same prefix as distinct entities when they are originated by different routers: every route announcement carries the “router-id” of its originating router (a unique identifier)
- feasibility distances are not maintained per prefix, but per source, where a source is a pair (router-id, prefix).

- In effect, Babel guarantees loop-freedom for the forwarding graph to every source (pair router-id, prefix);
- But IP packets are routed according to prefixes, not router-ids
- Since the union of multiple acyclic graphs is not in general acyclic, Babel does not in general guarantee loop-freedom when a prefix is originated by multiple routers.
- Anyway, any loop will be broken in a time at most proportional to the diameter of the loop – as soon as an update has "gone around" the routing loop.



- If both gateways fail at the same time, A will switch to B, and B will switch to A
- Then A emits an update with the router-id of S'
- When this propagates to B, B considers the route infeasible (B already invalidated it)
- Same things happen the other way around
- Loops last for the time needed to travel once on the loop itself

- The Type of Service (ToS) or Differentiated Services Code Point (DSCP) is a field of the IPv4 (and IPv6) header.
- It can be used to request different per-hop behaviour when forwarding IP packets with identical source and destination.

[2] https://tools.ietf.org/html/draft-chouasne-babel-tos-specific-00

- Generally, based on the ToS field a node uses different queueing policies (priority, drop probability, etc.).
- It can also be taken into account in addition to the destination address when performing a routing decision.
- A router that has a low-latency default route with high monetary cost might announce it with a "low- latency" ToS, and thus avoid carrying ordinary best-effort traffic over the expensive route.
- This extension allows to use both ToS-specific routes and non-ToS-specific routes handled by the original Babel protocol.

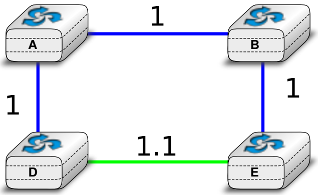
ToS Routes

- A router that performs ToS-specific routing maintains a routing table which instead of being merely indexed by destination prefixes is indexed by pairs of a prefix and a ToS value.
- The router adds a ToS TLV to the Routing Update packets.
- Updates and Requests for ToS-specific routes will be ignored by nodes implementing only the original protocol.
- So the ToS TLV does not propagate on a path that is not made of all ToS-enabled nodes

- This is important, because it guarantees that the wanted quality is preserved along the whole path.
- Similarly, a ToS-enabled node will add the ToS TLV also in Requests packets
- In order to be routed according to a given entry in the routing table, a packet must match not only the destination prefix but also the ToS value.

- A wireless mesh network may use links with interfering channels
- Or, with non-interfering channels

[3] https://tools.ietf.org/html/draft-chroboczek-babel-diversity-routing-01



- Different colors implies different (possibly non-interfering) channel
- In this case, even if A-B-E has a lower cost than A-D-E, it may be convenient to use the A-D-E since it passes through non-interfering channels
- This is especially true if the link metric does take into account only loss (as ETX)