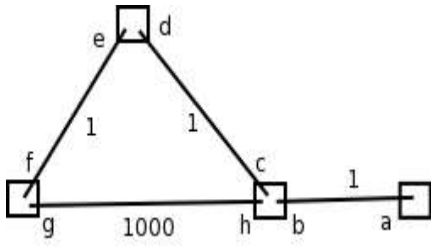
Design Flaws in OMDP Addressing Strategy

Erik van Zijst
erik@marketxs.com
MarketXS
December 19th, 2003

Counting-To-Infinity Problem

The implementation of version 0.4.1 that addresses individual interfaces instead of nodes as a whole appears to suffer from counting-to-infinity problems and possibly routing loops. Consider the network as depicted below. We've given every interface an address in the range [a-h] and specified the link costs as well. Link(g-h) is very slow, while the others are fast.
After the distance tables have converged, node(b,c,h) has a path to node(a) through interface b with cost 1. Node(d,e) goes to node(a) via interface d with cost 2 and node (f,g) goes to node(a) through interface f with cost 3.



Node(f,g) also advertises its route to a to node (b,c,h) because it's not in the path to a. Note that ExBF's isInPath algorithm backtracks the path from a through the head-of-path and checks for occurences of neighbor h. Since h is not in the path a-b-c-d-e-f, node(f,g) sends vector DV[g,h](a, b, 3) to node(b,c,h). (Format: DV[g,h](a,b,3) is the distance vector exchanged between g and h, describing destination a with head-of-path b and cost 3.) Node(b,c,h) therefore has 2 paths to a, one though interface h (cost 1003) and the other through interface b (cost 1). The preferred interface is b.

Now what happens when link(a-b) dies?

Node(b,c,h) switches preferred interface from b to h for destination a and advertises the new path cost (previously 1, now 1003). The new vectors DV[c,d](a, b, 1003) and DV[h,g](a, b, 1003) are sent to node(d,e) and node(f,g) respectively, because neither d, nor g is in the path a-b.
Node(d,e) now changes its cost to a from 2 to 1004, while the head-of-path, nor the preferred interface have changed. Node(f,g) changes its cost_via[g] from 1001 to 2003, but doesn't adjust its preferred interface (f) until it hears from node(d,e).

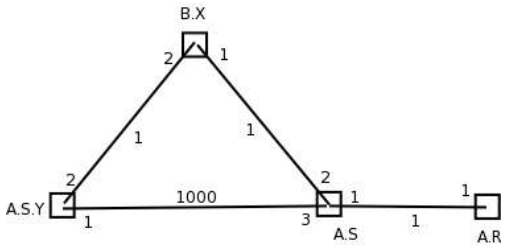
One can see this appears to lead to a complicated storm of distance vector exchanges that might lead to counting-to-infinity, an undetected loop or possibly it converges normally after all.

The problem seems a side effect of not being able to see the relation between interface addresses. Interface b, c and h are treated like independent entities, while they are related to one another in the way that they form a single physical router node

and are connected to each other with cost 0. Because of this, node(f,g) is unable to see that the route to a should not be advertised to node(b,c,h) as it would cause a loop. When node(f,g) does the ExBF-isInPath-check is finds the reverse path to be a-b-c-d-e-f,. which doesn‘t include h, the endpoint of the neighbor link.

Proposed Solution

We have seen problems with both addressing schemes. When addresses are given only to router nodes, rather than to interfaces, the problem arises that when summarization is applied, two neighbors might be summarized into the same address, making it impossible to maintain different routes through both links.
When changing to a per-interface address however, the above problem is fixed, but introduces the counting-to-infinity problem as described above. In reaction to this we propose a hybrid solution that addresses local interfaces independently, while remote routers are identified by a single address. In general, the interface address is merely an extension to the router address and is used only in the preferred-hop field of the distance table. Consider the same network again, but with router addresses and interface address extensions on top of them. This is illustrated below. When the distance tables converge, A.S initially has distance table $DT_{A.S}$: {(A.S, ?, ?, 0), (A.R, A.S[1], A.S[1], 1)} and advertises to A.S.Y: $DV_{A.S,\ A.S.Y}$: {(A.S, ?, 0), (A.R, A.S, 1)} which leads to A.S.Y‘s distance table $DT_{A.S.Y}$: {(A.S.Y, ?, ?, 0), (A.S, A.S.Y[1], A.S.Y[1], 1000), (A.R, A.S.Y[1], A.S, 1001)}. Node A.S also advertises its route to A.R and A.S to B.X, but because B.X summarizes both those entries into the same A wildcard A.*, the distance table contains: $DT_{B.X}$: {(B.X, ?, ?, 0), (A.*, B.X[1], B.X[1], 1)}. Also note that neighbor A.S.Y is also summarized into A.*, resulting in a single wildcard entry that matches all nodes in the network. Since this wildcard is the same as the one received from A.S and costs are equal, the preferred hop is set to the interface with the lowest number, in this case B.X[1]. Note that when B.X sends a message to A.S.Y, it will travel through A.S with a total cost of 1001. This is an extreme example of the stretch-effect introduced by summarization. After B.X advertised to A.S.Y with $DV_{B.X,\ A.S.Y}$: {(B.X, ?, ?, 0), (A.*, B.X, 1)}, node A.S.Y‘s distance table becomes $DT_{A.S.Y}$: {(A.S.Y, ?, ?, 0), (A.S, A.S.Y[1], A.S.Y[1], 1000), (A.R, A.S.Y[1], A.S, 1001), (B.*, A.S.Y[2], A.S.Y[2], 1)}. The final distance table of A.S will be $DT_{A.S}$: {(A.S, ?, ?, 0), (A.R, A.S[1], A.S[1], 1), (B.*, A.S[2], A.S[2], 1), (A.S.Y, A.S[3], A.S[3], 1000)}.



Now when link(A.R[1] – A.S[1]) fails, A.S, looses its only path to A.R and advertises the changes: $DV_{A.S,\ A.S.Y}$: {(A.S, ?, 0), (A.R, ?, -1), (B.*, A.S, 1), (A.S.Y, ?, -1)} and $DV_{A.S,\ B.X}$: {(A.S, ?, 0), (A.R, ?, -1), (B.*, ?, -1), (A.S.Y, A.S, 1000)}. A.S.Y, that had only a single path to A.R, via A.S looses its path to A.R, while B.X sees no change in its distance table as it never kept track of A.R, and the A.* domain as a whole remains reachable.

Clearly, this particular example network does not suffer from the counting-to-infinity

problem decribed before, however we have not proven the approach to be safe in every topology.

The consequence of keeping track of keeping track of interfaces by adding an extension to the router‘s address is that although addresses are still identified individually, all interfaces are automatically always in the same domain. This is opposed to version > 0.4.1 where every interface can be a totally independent address in a different domain. The new approach does make the distance table smaller and more simple to maintain.

Algorithm Pseudo Code

The proposed modified mechanism is described in the following pseudo code fragments.

Incoming Routing Message

```
# function invoked for received routing messages
function receiveVectors(vectors, rec_iface) {
  foreach(vector in vectors) {
    dest = summarize(vector.dest, router_addr);
    head = summarize(vector.head, router_addr);
    if( ! (dest matches router_addr) ) {
      record = dt.get(dest);
      if(record == null) {
        record = dt.addNewRecord(dest);
      }

      if(vector.cost == -1) {
        record.cost = -1;
      } else {
        record.cost_via[rec_iface] = vector.cost + linkCost(rec_iface);
      }

      if(vector.dest == interface.peerAddress) {
        # we will be the head of path towards our neighbors
        record.head = router_addr;
      } else {
        record.head_via[rec_iface] = head;
      }
    }
  }
  if(pathsChanged()) {
    updateDT();
    advertiseVectors();
  }
}
```

Unicast Message Switching

```
function receiveUnicastMessage(message) {
  if(message.dest == router_addr) {
    notifySubscribers(message);
  } else {
    dest = summarize(message.dest, router_addr);
    record = dt.get(dest);
    if(!record || !record.hop) {
      # no route to destination; discard message
```

```
    } else {
      send(message, record.hop);
    }
  }
}
```

Advertising Routes

```
function advertiseRoutes() {
  foreach(interface) {
    define vectors;
    foreach(record in dt) {
      if(isInPath(interface.peerAddress, record.dest)) {
        vectors.add(record.dest, ?, -1);    # unreachable
      } else {
        vectors.add(record.dest, record.head, record.cost);
      }
    }
    send(vectors, interface);
  }
}

# function to see if addr is in the path to dest.
function boolean isInPath(addr, dest) {
  if(dest == addr) {
    return true;
  }
  record = dt.get(dest);
  if(record.head == ?) {
    return false;     # end-of-path reached
  }

  return isInPath(addr, record.head);
}
```