CCNx 1.0 Acyclic Core Object Routing Functional Specification

Marc Mosko[1]*

Abstract
We described the the Acyclic Core Object Routing (ACORN) protocol, which provides multicast like delivery of Interests over a CCN network using core based trees. The description is for implementation guidance, we do not offer protocol analysis, or proofs of correctness, liveness, or safety. Each publisher of a namespace joins the namespace's tree, which forms an acyclic graph. Among those publishers, a distributed election determines which node will become the core of the tree. The core of the tree advertise itself using a distance vector protocol, so all nodes in the network now the core's identity. When a subscriber to the namespace wishes to issue an Interest in to it, it unicasts the Interest towards the core, until it hits the first node that is a member of the tree. The Interest then floods the tree until a node responds with a Content Object to consume the Interest.

Keywords
Content Centric Networks – Named Data Networks

[1] *Palo Alto Research Center*
*__Corresponding author__: marc.mosko@parc.com

Contents

Introduction

The Acyclic Core Object Routing (ACORN) protocol provides multicast like delivery of Interests over a CCN network using core based trees. This paper is for implementation guidance, we do not offer protocol analysis, or proofs of correctness, liveness, or safety. Each publisher of a namespace joins the namespace's tree, which forms an acyclic graph. Among those publishers, a distributed election determines which node will become the core of the tree. The core of the tree advertise itself using a distance vector protocol, so all nodes in the network now the core's identity. When a subscriber to the namespace wishes to issue an Interest in to it, it unicasts the Interest towards the core, until it hits the first node that is a member of the tree. The Interest then floods the tree until a node responds with a Content Object to consume the Interest. ACORN is built entirely over CCN-based Interest signaling, it does not use an external routing protocol.

The ACORN protocol uses a distance vector advertisement of cores because those advertisements do double duty to also elect the core. Each node is a potential core, so generates its own distance vector advertisement. As that advertisement progresses through the network, each node evaluates if it is the best advertisement for the core. If it is, the node forwards the advertisement, otherwise it suppressed it and forwards the better advertisement. ACORN's distance vector protocol uses a link cost based distance. The link cost could be "1" for a min-hop protocol or could be a measure of delay or bandwidth, or some other metric in a normed vector space. A degenerate case is when there is only one publisher for a namespace. In this case, ACORN operates like a unicast distance vector protocol. There are no JOIN control messages.

The core-based multicast nature of ACORN is similar to Bidirectional Protocol Independent Multicast (BIDIR-PIM) [1]. The BIDIR-PIM Rendezvous Point Address (RPA) is what we call the "core" of the tree. We do not discuss the use of a Designated Forwarder (DF), but that mechanism should be used on multiple access links. ACORN differs from BIDIR-PIM as it incorporates a distance vector protocol for core advertisements and uses those advertisements to elect the core.

1. Overview

We use a core-based multicast. The protocol uses trees and one tree per advertised name prefix. We will later change to small number of directed acyclic graphs (DAGs) that forward multiple prefixes. In core multicast, one node is elected the multicast core and other nodes form a tree to the core. When a node wishes to join the group, it unicasts a JOIN messages towards the core. At the first member router found along the unicast path, the JOIN messages are aggregated so only one JOIN flows towards the core. When a node leaves a group, it sends a LEAVE message. JOINs also soft-state timeout.

We use a distance vector protocol to elect the core and

establish the unicast routes. Each publisher in the multicast group is a potential core. It sends a DV advertisement with its centrality, routerid, and the name prefix. The "centrality" is a graph-wide metric of a node's centrality in the graph (it could also be considered a priority). A node only forwards the best DV advertisement, so a connected component will converge to only one DV advertisement per prefix for the best core. If the graph becomes partitioned, new cores form automatically due to the DV advertisements and merge again when components connect.

Interest messages for the group flow over the multicast tree. When we use trees, we can send the interest out all JOINed interfaces (not just towards the core) except the ingress interface.

We plan to use an Eigenvalue Centrality[1] measure, as we believe that is can be implemented using only 1-hop messaging with an iterative protocol and not require any global state. If an implementation does not use a centrality measure, all nodes can use a centrality of "1", or treat centrality as an administrative setting for core priority.

2. Namespaces

ACORN uses the following namespaces to store routing state. The first namespace stores the set of prefixes that ACORN should advertise. The `/%C1.M.S.localhost` namespace is defined as only on the local node, and is, for example, used to store state in a repository.

(1) `/%C1.M.S.localhost/acorn/prefixes`

The format of the prefixes payload is:

```
/prefix/one    [priority]
/other/prefix [priority]
```

The priority is an integer that determines the nodes priority to become the core. by default, all nodes have a priority "0". A positive value is preferred.

The second namespace is for the exchange of routing messages. The "prefixes" file specifies the the prefixes the node is a publisher for. The `/%C1.M.S.localnet` namespace is defined as only propagated on the local 1-hop network.

(2) `/%C1.M.S.localnet/acorn/acorn`

In operation, nodes send out advertisements in Interest messages under

(3) `/%C1.M.S.localnet/acorn/<routerid>`
`/serialized_message`

The serialize_message is either appended as the terminal name component of the Interest or as payload of the Interest. The serialized_message is a signed payload of the Interest, and makes use of CCN's cryptographic identities. The message includes a nonce as using a scoped sequence number as `hostpart:sequencepart`.

The Interest messages sent in Namespace 3 are essentially push notifications. A node does not expect a Content Object response to those advertisements. The Interest Lifetime could be set to "0" or some small value. Because each message in ACORN carries a monotonic msg_seqnum, no two interests should ever duplicate. The msg_seqnum is to protect against in-network message re-ordering.

ACORN protocol messages carried in an Interest may be split across multiple Interest messages. In this case, each message carries the same msg_seqnum and also a fragment id and fragment count for proper reassembly. In the following, we do not show the fragment id and count.

3. Protocol Details

A route advertisement is the tuple {prefix, routerid, centrality, seqnum, distance, flag}. The flag is "add' or "del" to add or delete the route. Route advertisements have an implicit soft-state timeout. Entries in a "del" message stay in the advertisement table like "add" entries and are re-transmitted until they timeout.

An advertisement is {routerid, msg_seqnum, {routes}}. The "msg_seqnum" is to correctly order all messages. It is a local sequence number from the sending node "routerid". routerid is synonymous with "core". "routes" is a set of routing advertisements. Each route in routes carries its own route advertisement seqnum, which is used as part of the loop-free invariant.

We will assume only local increasing integer time (e.g. measured in nanos).

Definition 1. (Route Advertisement Comparison) *A route advertisement A is preferable to B for the same exact prefix and both are "add" operations, and if $A \leq B$. At some node, where the computed timeout of A is A.timeout and for B is B.timeout and the local time is now, then if A is fresh than B is stale or if both are fresh, then A is preferable to B if it lexicographically sorts before B on the tuple (centrality, route rid, seqnum, distance). The comparison is show in Alg. 1.*

At a given node, let T_a be the timeout for advertisement a, let C_a be the centrality, let RID_a be the router id, let SN_a be the sequence number, and D_a be the distance. Then an advertisement A is preferable to B if and only if:

A route A is strictly preferable to B if and only if $A \leq B$ with $D_A < D_B$ in the last condition. We denote strictly preferable as $A < B$.

A JOIN message is the tuple {routerid, msg_seqnum, prefix, {forwarders}}.

The field routerid is the issuing nodes identity. The field msg_seqnum is the message sequence number for the JOIN message. The field prefix is the name prefix being joined. The set forwarders is the set of routerids that the current node declares as its nexthops, and are thus its upstream nodes. This

[1] `http://en.wikipedia.org/wiki/Centrality#Eigenvector_centrality`

Algorithm 1 Route Advertisement Comparison(A, B)

if $T_A > now$ **and** $T_B <= now$, **or** both being in the future **then**
  if $C_A > C_B$ **then**
    return true
  end if,
  if $C_A = C_B$ **and** $RID_A > RID_B$ **then**
    return true
  end if
  if $C_A = C_B$ **and** $RID_A = RID_B$ **and** $SN_A > SN_B$ **then**
    return true
  end if
  if $C_A = C_B$ **and** $RID_A = RID_B$ **and** $SN_A = SN_B$ **and** $D_A \leq D_B$ **then**
    return true
  end if
end if
return false

set is similar to BIDIR-PIM designated forwarder, except ACORN allows a node to nominate multiple.

A JOIN message is an implicit LEAVE for a previous forwarder for routerid if the forwarder is not listed in the current forwarders set.

We use a global soft-state timeout for JOIN messages, but one could add the soft-state timeout as a parameter to JOINs. We use T_{join} of 2 seconds, and a timeout multiple of T_{flush} of 4.5s (i.e. on 2rd loss, remove JOIN). However, on unreliable links, it may be preferable to use a smaller T_{join} to send enough messages to keep the JOIN alive.

A LEAVE message is the tuple (routerid, msg_seqnum, prefix), where the fields have the same meaning as in a JOIN message. It is functionally equivalent to a JOIN message with an empty forwarders set. All neighbors can remove routerid from their child list for prefix when they see a LEAVE message with increasing msg_seqnum.

Procedure 1 (ADD_PREFIX)**.** *When a router i becomes a publisher for a prefix p, that prefix is added to the PUBS table* $P^i \leftarrow P^i \cup \{p\}$.

If $p \notin G^i$*, node i create a Route* $r \leftarrow (p, i, C^i, ++SN_i^i, 0, add)$ *and adds it to* A_p^i*. If r is the preferred route, TRANSMIT r.*

The node executes RECV_JOIN(i, p, 0) ■

Procedure 2 (REMOVE_PREFIX)**.** *When router i stops being a publisher for prefix p, it removes that prefix from the PUBS table* $P^i \leftarrow P^i - \{p\}$.

Node i will create a Route $r \leftarrow (p, i, C^i, ++SN_i^i, 0, del)$ *and adds it to* A_p^i*. Node i will TRANSMIT r.*

The node executes SEND_LEAVE(i, p) ■

Procedure 3 (LINKUP($link$))**.** *Set* LC_{link}^i *to link cost. Begin sending Advertisements on the link.* ■

Procedure 4 (LINKDOWN($link$))**.** *When a link goes down, the link cost for each link should be set to* ∞*, which will invalidate all neighbors on that link. For each route that uses an adjacency on the link, the node should re-advertise those routes with the new costs. Alg. 2 shows the procedure.* ■

Table 1. Notation

P^i	the PUBS table at node i. It contains a list of all routed prefixes under which the node publishes
G_p^i	the GROUPS table at node i. It is a map from routed prefix p to GroupState for which the node is a forwarder. That is, for which there are JOIN messages.
A_{pk}^i	the set of routes known at node i for prefix p from node k. Advertisements have a soft-state timeout.
S_p^i	the successor set at node i for prefix p, being the set of loop-free next hops for that prefix. These next-hops should all be on a shortest path to the same core (routerid).
SN_j^i	the current route sequence number at node i for neighbor k (SN_i^i is node i's seqnum).
MN_j^i	the current message sequence number at node i.
LC_j^i	the link cost of link j at node i.
LN_j^i	the set of neighbors ($routerid$) on link j.
C^i	the centrality of node i. The centrality is determined by an external protocol, or it may be used as a priority for becoming a core.
J_p^i	the set of JOIN prefixes, that is the set of prefixes for which the current node is a forwarder. It contains the tuple (j, f, t) where j is the neighbor, f is the interface, and t is the soft-state timeout.
T_{join}	the period of JOIN messages, using a random forward offset (i.e. T_{join} = [2, 2.5] seconds).
T_{flush}	the soft-state flush timeout for a JOIN message, we use 5.25s, slightly longer than the maximum interval of 2 JOIN messages.

Procedure 5 (COST_DECREASE($i, link$))**.** *When the cost of a link decreases, a node should update* LC_{link}^i *to the new cost. Decreasing a link cost cannot cause a loop invariant violation, so no further action is necessary.* ■

Procedure 6 (COST_INCREASE($i, link$))**.** *When the cost of a link increases, a node should update* LC_{link}^i *to the new cost. An increasing link cost may cause a loop invariant violation, as a node may inadvertently choose one of its children as a successor because its cost grew and the child has not yet increased its cost. If the increase of link cost affects any node in the successor set, then node i cannot change its successors for any such affected prefix until it receives a new route advertisement, which will have a larger seqnum.* ■

Procedure 7 (RECV_ADVERTISEMENT(i, a, k))**.** *When node i receives an advertisement a from neighbor k, it should ensure that* $a.msg\_seqnum > SN_k^i$*, otherwise drop the message. It then executes Algorithm 3* ■

Algorithm 2 LinkDown(link)

```
1: for all neighbor k on link do
2:    LC_k^i ← ∞
3:    for all r ∈ A^i via k do
4:       if k ∈ S_{r.p}^i then
5:          ADVERTISE(r)
6:       end if
7:    end for
8: end for
```

Algorithm 3 RECV_ADVERTISEMENT(i, a, k)

```
1: for all route r ∈ a.routes do
2:    if r < A_pk^i then
3:       A_pk^i ← r
4:       if r.flag = del then
5:          remove k from S_p^i.
6:          TRANSMIT r
7:       end if
8:    end if
9:    if r.flag = add then
10:      for all j ∈ S_p^i do
11:         if r < A_pj^i then
12:            remove j from S_p^i.
13:         end if
14:         if r <= A_pj^i then
15:            add k to S_p^i.
16:         end if
17:      end for
18:      if S_p^i is empty then
19:         add k to S_p^i
20:      end if
21:      if k added to S_p^i then
22:         ADVERTISE p
23:      end if
24:   end if
25: end for
```

In Alg. 3, node i first checks if the advertisement r is feasible in line 2 for neighbor k. If the route is strictly preferable, then the current advertisement from node k is updated in line 3. If the update is a deletion, node k is removed from the successor set and the route advertisement is immediately re-transmitted. Beginning on line 9, node i checks if node k should be added to the successor set. Node i compares the new advertisement to all the current successors. If r is strictly preferable to a successor, node i removes that successor from its set in the block starting at line 11. If r is feasible, but not strictly better than an existing successor, it is added to the successor set in the block starting at line 14. If the successor set, in line 18, is empty, then node i adds r to the successor set. If node i modified its successor set, it should then schedule an advertisement for p.

Procedure 8 (ADVERTISE(p)). *For each successor $k \in S_p^i$, find the least restrictive route r in A_{pk}^i for successor k. Least restrictive means largest distance plus link cost to that neighbor. Then, add $A_p^i r$ to advertisement packet, with the advertisement's distance + link cost.* ■

Procedure 9 (RECV_JOIN(j, p, f)). *Node i receives a JOIN message from neighbor j for prefix p on interface f. Set the soft-state timeout $t \leftarrow now + T_{flush}$ (e.g. 2 seconds). Then, add (j, f, t) to J_p^i. If J_p^i was empty, execute SEND_JOIN(p)* ■

Procedure 10 (SEND_JOIN(p)). *For each neighbor $k \in S_p^i$, send p to k. This is done by creating a JOIN message with a forwarder set equal to S_p^i. If the current node is the core, S_p^i is i, in which case no message is necessary.* ■

Procedure 11 (SEND_LEAVE(p)). *So long as the node is not the core, it should send a LEAVE message when it is no longer interested in the prefix p.* ■

Procedure 12 (UNSATISFIED_INTEREST(i, p)). *When node i receives an unsatisfied interest for prefix p, it should forward the Interest to each node in the union of the success set and the child set, less the reverse path. If that is the empty set, the node should discard the interest.* ■

Procedure 13 (ROUTE_TIMER(i)). *Each "active" route in A^i should be re-transmitted on a periodic timer, for example every 2 seconds or 4 seconds, until the lifetime of the route expires. This applies to both ADD and DEL routes. The interest lifetime of the advertisement should be set to just under the ROUTE_TIMER interval.*

If there are no active routes, the router should still send an empty Advertisement, which functions as a HELLO message. ■

We keep a priority queue of timer tasks (sorted by absolute deadline) and set a TimerTask based on the top of the heap. We only scheduler a new timer if there is not one already scheduled or if a new task pops up to the top of the heap.

4. Conclusion

The Acyclic Core Object Routing (ACORN) protocol provides multicast like delivery of Interests over a CCN network using core based trees. Each publisher of a namespace joins the namespace's tree, which forms an acyclic graph. Among those publishers, a distributed election determines which node will become the core of the tree. The core of the tree advertise itself using a distance vector protocol, so all nodes in the network now the core's identity. When a subscriber to the namespace wishes to issue an Interest in to it, it unicasts the Interest towards the core, until it hits the first node that is a member of the tree. The Interest then floods the tree until a node responds with a Content Object to consume the Interest.

The ACORN protocol uses a distance vector advertisement of cores because those advertisements do double duty to also elect the core. Each node is a potential core, so generates its own distance vector advertisement. As that advertisement progresses through the network, each node evaluates if it is

the best advertisement for the core. If it is, the node forwards the advertisement, otherwise it suppressed it and forwards the better advertisement.

Future work includes forming a forest of trees around brittle cut points, so each well-connected component may operate with its own core and the narrow cut points operate as gateways between those components.

References

[1] M. Handley, I. Kouvelas, T. Speakman, and L. Vicisano. Bidirectional Protocol Independent Multicast (BIDIR-PIM). RFC 5015 (Proposed Standard), October 2007.