Karlsruhe Institute of Technology (KIT)
Institute of Telematics

TELEMATICS TECHNICAL REPORTS

Extended Virtual Ring Routing (eVRR)

Sebastian Mies
Institute of Telematics, Karlsruhe Institute of Technology (KIT), Germany
mies@kit.edu

June, 30th 2011

TM-2011-2

ISSN 1613-849X

`http://doc.tm.uka.de/tr/`

Institute of Telematics
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany

Technical Report

Extended Virtual Ring Routing (eVRR)

The original Virtual Ring Routing Protocol (VRR) published by Matthew Caesar et al. [1] was inspired by the Chord protocol. The difference is that VRR does not create an overlay. Instead, it builds paths in the underlay topology to create a virtual ring. In other words, VRR embeds a virtual ring into the physical network. Each node in the physical network graph selects an arbitrary, unique number (or identifier) represented by a bit-string of fixed size l. The virtual ring comprises all nodes in ascending order according to their identifier. On the ring, each node has s successors and s predecessors and builds a convex address space.

Naturally, the virtual ring does not reflect the closeness in the network graph. Virtual neighbors may be distributed all over the network. What VRR does is setting up paths between the virtual nodes. VRR employs so called discovery packets to do this. A discovery packet tries to find a node that is a virtual neighbor, i. e., successor or predecessor, of the node. VRR does this greedily, VRR routes the discovery message greedily towards their best closest neighbors according to the virtual ring. If VRR finds a match, VRR establishes a virtual path between the nodes. This the routing-table entries of VRR reflect these paths leading to the virtual neighbors on intermediate nodes.

Building the ring is an incremental process. Initially a node does only sees its immediate, physical neighbors and its identifiers. Thus, the first successor and predecessor is chosen according to the these, currently known, identifiers. Subsequently, each node sends out discovery messages to look for closer nodes. Building up new paths and, therefore, discovering closer nodes in each step until the virtual ring is built up and correct.

When the virtual ring is built, a node can post messages to any other node by forwarding messages along the virtual ring. Intuitively, will lead to a maximal path-length of $\frac{N}{2}$ (N is the number of nodes on the ring), because of the maximal hop-count on rings. However, VRR uses direct neighbors and routing table entries as *short-cuts* in the virtual ring. When there is a shorter route to a node on the ring stored in the routing table it greedily chooses the node as the next hop of the message. This mimics the properties of the finger-table in the Chord protocol to minimize the virtual ring hop count and reduces stretch.

The way the original VRR builds up the routing tables (using discovery packets) is difficult to control and follow. Additionally it is known that it suffers from *loopy cycles* that need to be fixed by flooding. Furthermore, handling link setup and routing table maintenance using discovery packets is complex in a dynamic, asynchronous network scenario.

In search of a simpler, and more controllable solution, the idea came up to use the well-known idea behind sequenced distance vector protocols. The distance vectors create the routing tables for the virtual ring. This section describes and evaluates this extended VRR protocol design in detail.

1.1 Sequenced Distance Vector (sDV) Protocols

Before the introduction of the extended VRR, this section provides a better understanding on sequenced distance vector protocols. One of the earliest sequenced distance vector protocols was the Destination-Sequenced Distance Vector (DSDV) routing [3]. Later, Babel [2] refined DSDV to make it more powerful and more efficient. The main idea behind these protocols is, that each destination node announces its distance vector together with a sequence number. This sequence number increases on any routing update by the destination (or originate) node. When a node receives distance vectors from its neighbors it adapts the distance of an entry, if the distance is smaller, or, the sequence number is greater. This principle solves the problems that come with distance vector protocols: count-to-infinity problem and routing loops. This is the case, because a worse route is only accepted by other nodes when the origin node has chosen a new sequence number. This also causes that all nodes have to choose a new sequence number until the protocol re-converges to a stable state in case of network dynamics. However, note, in case of VRR, such dynamics are only introduced when one of the s neighbors in the virtual ring of a node change.

1.2 VRR using Sequenced Distance Vectors

Intuitively, VRR does not have much in common with known distance vector protocols on the first glance. E. g., while VRR selectively sends discovery messages, distance vector protocols flood the network with routing information. However, abstractly speaking, what VRR does with discovery packets is “selective flooding” where only nodes close to a virtual location on the ring receive certain routing information. Using this insight an approach using a selective distance vector protocol becomes feasible.

The main data structure of eVRR is the routing table. The eVRR protocol keeps this table sorted ascending by the destination identifier (id). Table 1.1 summarizes the items of a routing table entry. It comprises the destination identifier (id), distance to

Name	Description
id	Destination identifier
dist	Distance to destination, $= 0$ when id is owned by the node
neighbor	Flag, whether the node is a neighbor
nexthop	NID of the next hop neighbor
seq	Sequence number
activity	Timestamp of the last activity
changed	Flag, wheter the entry has changed since the last update

Table 1.1 – eVRR routing table entry

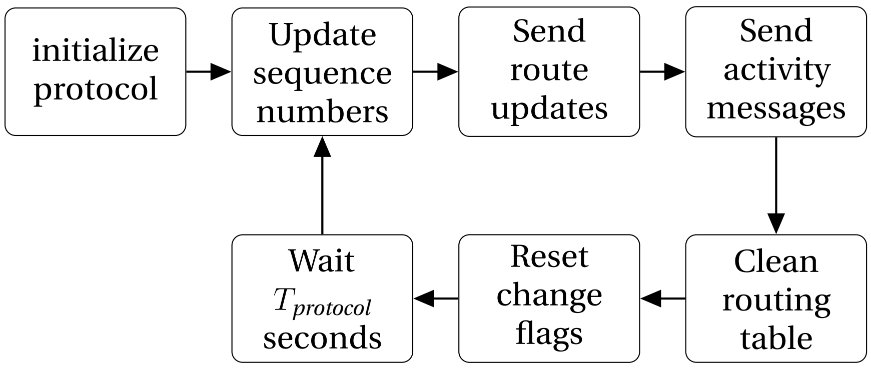

Figure 1.1 – Main loop of eVRR

the destination (dist) which is zero, when the identifier is owned by the node itself. The neighbor-flag denotes whether the routing table entry is from one of the node's direct neighbors. The sequence number (seq) is self-explanatory. Finally, each entry holds a timestamp of its last activity (activity) because of update or activation message, described later. The changed-flag indicates changes since the last protocol round.

Like CMP the extended VRR (eVRR) protocol is based on rounds, i. e., each $T_{protocol}$ seconds it sends routing update messages to neighbor nodes. Received messages are processed directly on arrival. Figure 1.1 illustrates eVRR's mainloop in a flow-chart.

First, the protocol initializes it routing table and fills its own identifiers together with a fresh sequence number and changed-flag set. Then it enters the main loop triggered each $T_{protocol}$ seconds. In the second step, eVRR extracts routing update messages from its routing table for each neighbor.

Send Route Updates to Neighbors

Route updates are extracted from the node's routing table as described in Algorithm 1. This algorithm iterates over the complete routing table entries (for loop in line 5). When a entry came from a neighbor x (check in line 6), i. e., the nexthop has the NID of this neighbor x, the s adjacent routing table entries in both directions on the ring are considered as potential route updates. Lines 7–9 iterate are responsible for interating over these routing table entries. A potential route update is added, if 4 conditions apply. First, the identifiers of the neighbor and the entry must differ (line 10). Second, the routing table must have been changed since the last update iteration of the main-loop (line 11). This ensures that neighbors do not receive duplicate route updates. Third, the entry should not have been received from the neighbor (line 12). This mechanism is called *split horizon* in literature. Finally, no update should be sent twice to a neighbor, which is ensured in line 13. If all conditions apply a route update is extracted from the entry. More precisely, a route update comprises the identifier (id), distance (dist), and sequence number (seq). All neighbors need to know the identifiers owned by the node. Thus, route updates for all entries with dist $= 0$ are added in line 22–25.

Using this algorithm known routes of known adjacent virtual neighbors are exchanged. These routes converge to a single path between each adjacent node. As mentioned before, received route updates from neighbors are processed immediately on arrival. Thus, the processing of route updates by neighbors is described before continuing in the main loop.

Algorithm 1: Extract routing updates for neighbors

```
   Data: $N$ is the routing table size.
   Data: $e[i]$ denotes the i-th entry of the routing table.
   Data: $t$ is the NID($x$) of neighbor $x$.
   Data: $s$ denotes the number of virtual neighbors in each direction.
   Result: Routing updates $R$ for node $x$
1  begin
2      if $N = 0$ then return;
3      $I \leftarrow \emptyset$; $R \leftarrow \emptyset$;
4      for $i \leftarrow 0$ to $N - 1$ do
           // consider only entries received from node $t$
5          if $e[i]$.nexthop $= t$ then
               // collect routes close to routes of node $t$
6              if $i \geq k$ then $l \leftarrow i - s$;
7              else $l \leftarrow i + N - s$;
8              for $j \leftarrow 0$ to $2 \cdot s + 1$ do // add entry if ...
9                  $u \leftarrow (e[i]$.id $\neq e[l]$.id$) \wedge (e[i]$.changed$)$;   // not the same, changed, and ...
10                 $u \leftarrow u \wedge (e[i]$.nexthop $\neq t)$;   // entry came from another node and ...
11                 $u \leftarrow u \wedge (I \cap \{e[l]$.id$\} = \emptyset)$;   // not already added
12                 if $u = true$ then // add route entry
13                     $R \leftarrow R \cup \{(e[l]$.id$, e[l]$.dist$, e[l]$.seq$)\}$; $I \leftarrow I \cup \{e[l]$.id$\}$;
14                 end
15                 $l \leftarrow (l + 1) \mod N$;
16             end
17         end
18     end
       // add the node's identifiers
19     for $i \leftarrow 0$ to $N - 1$ do
20         if $(e[i]$.dist $= 0) \wedge (I \cap \{e[i]$.id$\} = \emptyset)$ then $R \leftarrow R \cup \{(e[i]$.id$, e[i]$.dist$, e[i]$.seq$)\}$;
21     end
22 end
```

Processing Route Updates by Neighbors

Algorithm 2: Processing of a route update

```
   Data: $R$ are the route updates from neighbor $x$ with $f = \mathsf{NID}(x)$.
   Data: $E$ is the routing table, while $e[i]$ denotes the i-th entry of the routing table.
1  begin
2      forall $u \in R$ do
           // unpack items from routing update triple
3          $id \leftarrow u_1$; $dist \leftarrow u_2$; $seq \leftarrow u_3$;
           // ignore blacklisted identifiers
4          if $id$ is blacklisted then continue;
5          if no entry to $id$ exists in $E$ then // create new entry
6              $i \leftarrow E$.addNewEntry();
7              $e[i]$.id $\leftarrow id$; $e[i]$.seq $\leftarrow seq$; $e[i]$.dist $\leftarrow dist + 1$; $e[i]$.neighbor $\leftarrow (dist = 0)$;
8              $e[i]$.nexthop $\leftarrow f$;
9              $e[i]$.activity $\leftarrow currentTime$; $e[i]$.changed $\leftarrow true$; $tableChanged \leftarrow true$;
10             if $id$ in neighborhood then $vneighborsChanged \leftarrow true$;
11         else
12             $i \leftarrow E$.getEntryOf($id$);
               // Refresh activity of neighbors
13             if $e[i]$.neighbor then $e[i]$.activity $\leftarrow currentTime$
14             if $dist + 1 < e[i]$.dist $\wedge$ $seq = e[i]$.seq then // adapt to lower distance
15                 $e[i]$.dist $= dist$; $e[i]$.nexthop $\leftarrow f$;
16                 $e[i]$.changed $\leftarrow true$; $tableChanged \leftarrow true$;
17             else// adapt new sequence number plus whole update
18                 $e[i]$.seq $\leftarrow seq$; $e[i]$.dist $\leftarrow dist + 1$; $e[i]$.nexthop $\leftarrow f$;
19                 $e[i]$.activity $\leftarrow currentTime$; $e[i]$.changed $\leftarrow true$; $tableChanged \leftarrow true$;
20             end
21         end
22     end
23 end
```

In principle, processing route updates works in the same way as in sequenced distance vector protocols (cf., Section 1.1). Algorithm 2 processes route updates. First, each route update is unpacked to its components, identifier, distance, and, sequence number in line 3. If the route update is on a blacklist, the entry is skiped in line 4. The blacklist is used to ensure that removed entries do not return to the routing table by the routing table update described later. If no entry is available in the routing table, the algorithm inserts a new entry according to its identifier to keep the table sorted by identifiers in line 5–10. The algorithm sets all fields: the sequence number, nexthop, neighbor flag, activity timestamp, and, changed flag. Futhermore, it checks if the entry is a new virtual neighbor. If this is the case, it sets the *vneighborsChanged* flag.

If the routing table already contains a entry for the identifier, the activity timestamp is refreshed, if the node is a neighbor in line 13. This ensures, that routing tables of neighbors never time out. Subsequently, if the sequence number of both, route update and table entry is the same *and* the distance of the update is less than the distance of the entry, the entry adapts to the route update in line 15–16. If the sequence number is greater than the number in the table entry the entry adapts to the update even when the distance is greater in line 18–19. This mechanism effectively prevents the count-to-infinity problem because only a new sequence number from the destination node can increase the distance.

With sending and processing of routing updates, eVRR builds routes to $2 \cdot s$ virtual neighbors. However, a lot of residue in the routing table is left behind when these routes become stable. The residue results from the incremental process of eVRR. As eVRR starts to know a route to the best neighbor known and incrementally converging to closer virtual neighbors. Each step leaves a old route behind that is not needed anymore. Even more problematic, the old routing information introduces asymmetries that cause routing loops in the greedy routing scheme.

To remove the problematic residue the routing table is cleaned from old entries. Before this can work, eVRR needs to provide information about active routes. To do this, eVRR needs to route messages.

Routing using eVRR

A message routed with eVRR comprises the source and destination identifier, time-to-live, hop count and payload. Furthermore, it contains the shortest distance to the destination seen during routing. This ensures, that a message does not run in cycles when the virtual ring is in an inconsistent state or route asymmetries exist. eVRR employs greedy routing on the ring. Thus, the routing mechanism is simple. All messsages are routed into the direction of the closest identifier to the destination in the routing table. More concrete, eVRR uses Algorithm 3 to determine the next-hop using the node's routing table. This algorithm additionally considers routing table entries that have been active the last $3 \cdot T_{protocol}$ seconds or the distance is zero, i. e., a direct route always considered as best-choice.

Using this routing primitive, eVRR can route messages to other nodes. Furthermore, it

Algorithm 3: Finding the next hop to a destination identifier id

```
   Data: $E$ is the routing table, $e[i]$ denotes the $i$-th entry of the routing table.
   Data: $id$ denotes the destination identifier
   Result: The next hop routing entry $nextHopEntry$.
1  begin
2      $dist \leftarrow \infty$;
3      $nextHopEntry \leftarrow nil$;
4      for $i \leftarrow 0$ to $|E| - 1$ do
5          if $dist > |e[i].\mathrm{id} - id| \wedge (e[i].\mathrm{id} = id \vee (e[i].\mathrm{activity} + 3 \cdot T_{protocol}) > currentTime)$
           then
6              $dist \leftarrow |e[i].\mathrm{id} - id|$; $nextHopEntry \leftarrow e[i]$;
7          end
8      end
9      return $nextHopEntry$;
10 end
```

uses the routing to mark routes between virtual neighbors *active*. This means, that the activity entry in the routing table gets refreshed with the current time.

Sending Activity Messages to Virtual Neighbors

Each $T_{protocol}$ seconds, eVRR refreshes the timestamps of the routes leading to virtual neighbors. To this end, eVRR sends activity messages using the same greedy routing algorithm used for common data messages. However, additionally, the activity message refreshes every activity field in the routing table entry, used on the route to the destination node, with the current time. The routing table entry on the source node is not refreshed, to ensure that not responding neighbors get removed from the routing table. When a activity message reaches the destination virtual neighbor, it refreshes the timestamp of its routing table entry leading to the source.

Hence, activity messages "clamp" functioning routes between virtual neighbors by updating the respective activity timestamps. This allows eVRR to remove unused routes from the routing table in the next step.

Cleaning Up Routing Tables

The activity timestamps in the routing table indicate whether eVRR needs the route for accurate routing and complete virtual ring. Before starting over in the main loop. eVRR removes all entries that are "old", i. e., the activity timestamp is $a \cdot T_{protocol}$ in the past. a denotes the number of protocol intervals a entry is kept in the routing tables. Because each routing table entry may "travel" only one hop in each protocol iteration, it is set to a the maximum expected path-length between two virtual neighbors, e. g., $a = 32$. When a route to a virtual neighbor of a node is removed, eVRR remembers this by setting the $vneighborsChanged$-flag, indicating that the virtual neighborhood has been changed.

After cleaning up the routing table, eVRR checks whether the $vneighborsChanged$-flag is set. If the case, the sequence number is increased, as it indicates a change in the network topology. Finally, all flags indicating changes are reset, i. e., changed-flags in the routing tables, the $vneighborsChanged$-flag, and $tableChanged$-flag. eVRR waits $T_{protocol}$ until it starts over in the main-loop.

1.3 Optional Optimizations

eVRR allows several optimizations to accelerate convergence speed and route optimizations. In contrast to the original VRR, the extended version presented here benefits from all experiences of classical distance vector protocols. However, this section concentrates on two optimizations that make eVRR behave like the original VRR design.

Active Route Teardowns

VRR uses active teardown messages to remove old routes from the routing tables. eVRR in contrast removes routes only when virtual neighbors did not use a route for some time. The idea is, to actively remove routing table entries when neighbors do not need these anymore. This is done using feedback messages from the neighbors. Whenever a routing table entry's nexthop field is changed, the old next-hop neighbor is informed that the route does not point to this node anymore. eVRR does the same with route updates that do not find their way into the routing table.

Each node holds a reference count for each neighbor on each routing table. Whenever the node sends a route update message of this entry to a neighbor that did not already receive the update, it increases the reference count. Futhermore, the reference count is decreased when a neighbor informs the node that the route is not used anymore.

When the reference count drops to zero and is not part of the node's virtual neighborhood, the route is dropped immediately. This accelerates the number of unnecessary routing table entries, therefore, also decreases routing asymmetries. In summary, the delivery ratio during convergence of the network will improve.

Selective Shortest-Paths

As eVRR uses distance vectors, classical shortest-path routes can be obtained. The only addition is a small flag in on the routing table is necessary to indicate that the node should forward route updates on this entry to all neighbors. The activity messages are be used to keep only the shortest route between two selected nodes alive. Thus, eVRR can obtain selective shortest-paths if required.

No Traffic after Convergence

In the design presented here, activity messages are neccessary to keep routes alive on nodes. While this is not problematic on nodes that have good bandwidth and energy ressources, weak nodes may suffer from this. One way of dealing with this problem is stop sending updates when the network has converged. To achieve this, sending route updates is stopped, when a route as been activated a times. The route is then considered stable and not considered in the routing table cleanup. However, when a node recognizes a change in its neighborhood, a trigger is sent to activate activity messages again by resetting the convergence counter.

1.4 Summary

This technical report introduced the extended Virtual Ring Routing (eVRR) protocol. It maps the idea of the original VRR protocol to a simple sequenced distance vector problem. This reliefs the design from unexpected behaviour, like loopy-cycles, and simplifies the protocol operation. The distance vector approach also allows to maintain additional shortest-paths by flooding if necessary. This would require a small extension to the protocol design.

Bibliography

[1] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *Proc. ACM SIGCOMM 2006*, pages 351–362, Pisa, Italy, 2006.

[2] J. C.-P. W. M. Abolhasan, B. Hagelstein. The babel homepage., May 2011, url = http://www.pps.jussieu.fr/ jch/software/babel/,.

[3] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. *SIGCOMM Comput. Commun. Rev.*, 24:234–244, October 1994.