of the LPP beacons when using the Low Power Probing MAC protocol. [3]

The **polite-announcement** will be used when evaluating the Contiki collect protocol (see § 2.5). It is particularly suitable when using the Contiki nullmac MAC layer protocol in the Cooja simulator. The nullmac MAC layer protocol is a simple pass-through protocol, and doesn't perform any medium access control.

The polite-announcement back-end works as follows. Polite-announcement is parameterized by two values: an announcement time period t_{start} and t_{max}. When the value to announce changes, the announcement primitive starts sending out an announcement packet with the announcement period t_{start} (for example, t=8 seconds). The period doubles after each send (see Figure 9-a), with the period being limited to t_{max}.
The send action itself is not immediate, but is instead handled by the Rime ipolite primitive[5]. This polite algorithm is also parameterized by a time interval, in this case the same as the announcement period. During the first half of the time interval, the sender listens for other transmissions. The packet is sent at a random time during the second half of the interval, as shown in Figure 9-b.
The reason for this delayed sending is to potentially cancel the sending if a similar packet is received by the node itself. This, however, will never occur, given the way the collect protocol uses the announcement primitive.

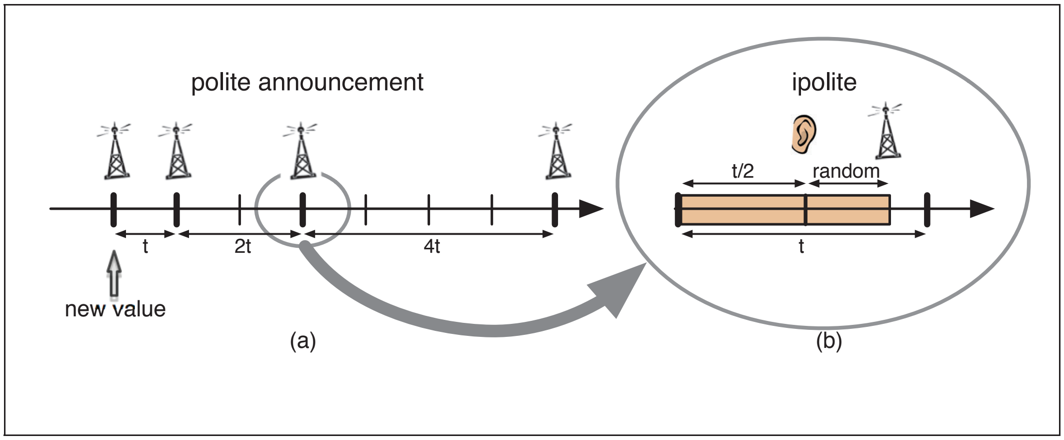

Figure 9 – The polite-announcement algorithm

2.5 The Contiki collect protocol

The Contiki collect protocol implements a hop-by-hop reliable data collection mechanism. Data is sent via a tree topology to a sink node.

An overview of the Contiki collect protocol is given in Figure 10. It illustrates the response of the collect protocol to a number of events: an incoming data packet, an acknowledgment or time-out of a sent data packet, an incoming announcement packet, and the sending of a message by the application using the collect protocol. This figure will also be referred to later in the text.

Note: the description in this section is based on the following Contiki file versions: collect.c v1.28, neighbor.c v 1.16. Since then, the collect protocol has been improved with a packet buffer, allowing more than one packet to be forwarded at the same time.

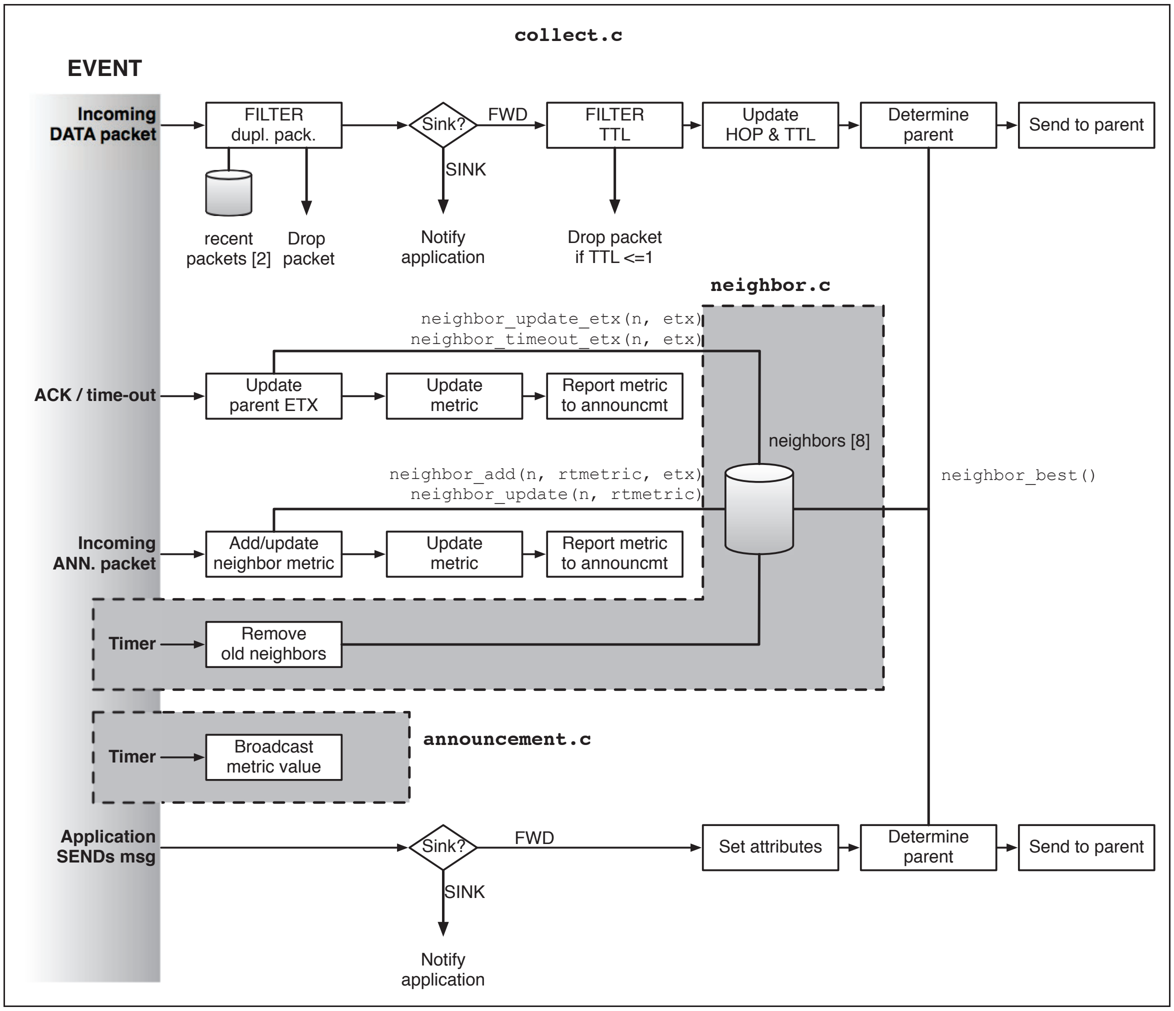

Figure 10 – The Contiki collect protocol: event processing

2.5.1 Components

The protocol consists of a number of high-level components:

- **Routing (tree creation) —** The nodes self-organize in a tree topology, with data always being sent up the tree until it reaches the top node. The sink node is assigned to be the top of the tree, all other nodes are initially tagged as leaf nodes. Gradually, spreading outward from the sink node, nodes update their position in the tree.
- **Neighbor discovery and management** — In a separate background process, neighbors announce their presence by periodically sending out announcement packets. These announcements are used to populate the neighbor table with neighbors.

- **Link estimation** — Based on link level acknowledgments for data packets sent by the node, the ETX value of each neighbor in the neighbor table is updated on each acknowledgment or time-out.
- **Duplicate packet filtering and packet aging** — Packets get forwarded by nodes until they reach the sink node. To protect against forwarding duplicate packets, a node checks a packet to be forwarded against a limited history of forwarded packets. If it has recently been forwarded, the packet is dropped. Additionally, to prevent packets from roaming through the network forever, the packet is dropped if it exceeds a certain maximum number hops.

On a side note, the Contiki collect protocol does not contain any loop detection mechanism. Routing loops generally occur when a node chooses a new route that has a significantly higher ETX than its old one, perhaps in response to losing connectivity with a candidate parent. If the new route includes a node which was a descendant, then a loop occurs.[18] It would be interesting to investigate and implement a number of practical solutions to alleviate this. For example, the collect protocol could be modified to include the current node's routing cost gradient in the packet header, and prevent the receiving node from forwarding packets with a lower gradient. [18]

2.5.1.1 Routing (tree creation)

The routing mechanism to transport data originating from any node to the sink node builds a routing tree (see also § 2.3).

All nodes are organized in a virtual tree, with their position in the tree defined by a 16-bit rtmetric (route metric) value. The sink node, at the top of the tree, has an rtmetric value 0. Child nodes further down the tree have an increasing rtmetric value. The parent of a node is the best neighbor of node, i.e. the node that minimizes the expected number of transmission to the sink.

The tree is created dynamically by updating the rtmetric value at particular events. Initially all nodes have the maximum rtmetric value (this maximum is currently set to 255, not all 16 bits are used), except for the sink node which has value 0 assigned by the application using the collect protocol. The announcement packets sent out for neighbor discovery (see § 2.5.1.1) report the rtmetric value of the announcing node. This neighbor's rtmetric value is stored in the neighbor table of each receiving node.

The node's own rtmetric value is then calculated based on the rtmetric value of the best neighbor node. The best neighbor node is the node that provides the path with the fewest expected transmissions to the sink node. This is the case for the neighbor that minimizes the sum of the neighbor rtmetric and the expected number of transmissions (ETX) from the node to that neighbor.

$$rtmetric = \underset{n \in N}{\arg\min}(rtmetric_n + ETX_n) \qquad \text{(eq. 1)}$$

The rtmetric value is updated on the following events: designation of a node as sink node, an incoming announcement packet, the acknowledgement of a data packet and the time-out of a data packet.

The rtmetric calculation process gradually trickles down from the neighbors of the sink node to the leaf nodes. Once the process has stabilized, the rtmetric of a node represents the expected number of total transmission for a packet to arrive at the sink node.

2.5.1.2 Neighbor discovery and management

See Figure 11.

Neighbor discovery and management is located in a separate code module (core/net/rime/neighbor.c). This is also the location where the neighbor table is managed.

Neighbor discovery is organized using the Rime announcement primitive, see § 2.4.3. The announcement primitive periodically sends out announcement packets (specifically tagged as such) on a separate logical channel. Announcements are characterized by an (ID, value) tuple and are disseminated to local area neighbors. Incoming announcement packets do not traverse the complete Rime stack, but instead are intercepted at the MAC layer. The MAC layer then notifies any registered processes (based on the announcement ID) about the incoming announcement.

The collect protocol is such a registered process, and it uses the incoming announcement to populate a neighbor table. The announcement specifies the sending neighbor address, the ID (which is set to the channel number but of no further use), while the value represents the routing metric used for tree creation and routing. Since the neighbor table is limited in size (currently set to 8 neighbors), it is important that 'old' neighbors do not occupy the table for too long. Thereto, a timer triggers periodic (i.e., every second) scanning of the neighbor table and removes all neighbors which haven't been heard from during the previous 120 scans (i.e., roughly 120 seconds).

2.5.1.3 Link estimation

See Figure 11.

The ETX values for each node's neighbors are stored in the neighbor table, and are calculated each time a data packet is sent to a neighbor. When the sent packed is ACK'ed, the number of transmissions that was required to deliver the packet is reported to the link estimator.

Each of the last eight transmission counts is kept in the table. The link ETX value from a node to a neighbor is then the average over these eight transmission counts. If a transmission times out, the maximum number of transmission (e.g., 4) is reported and added to the current history entry (instead of overwriting it).

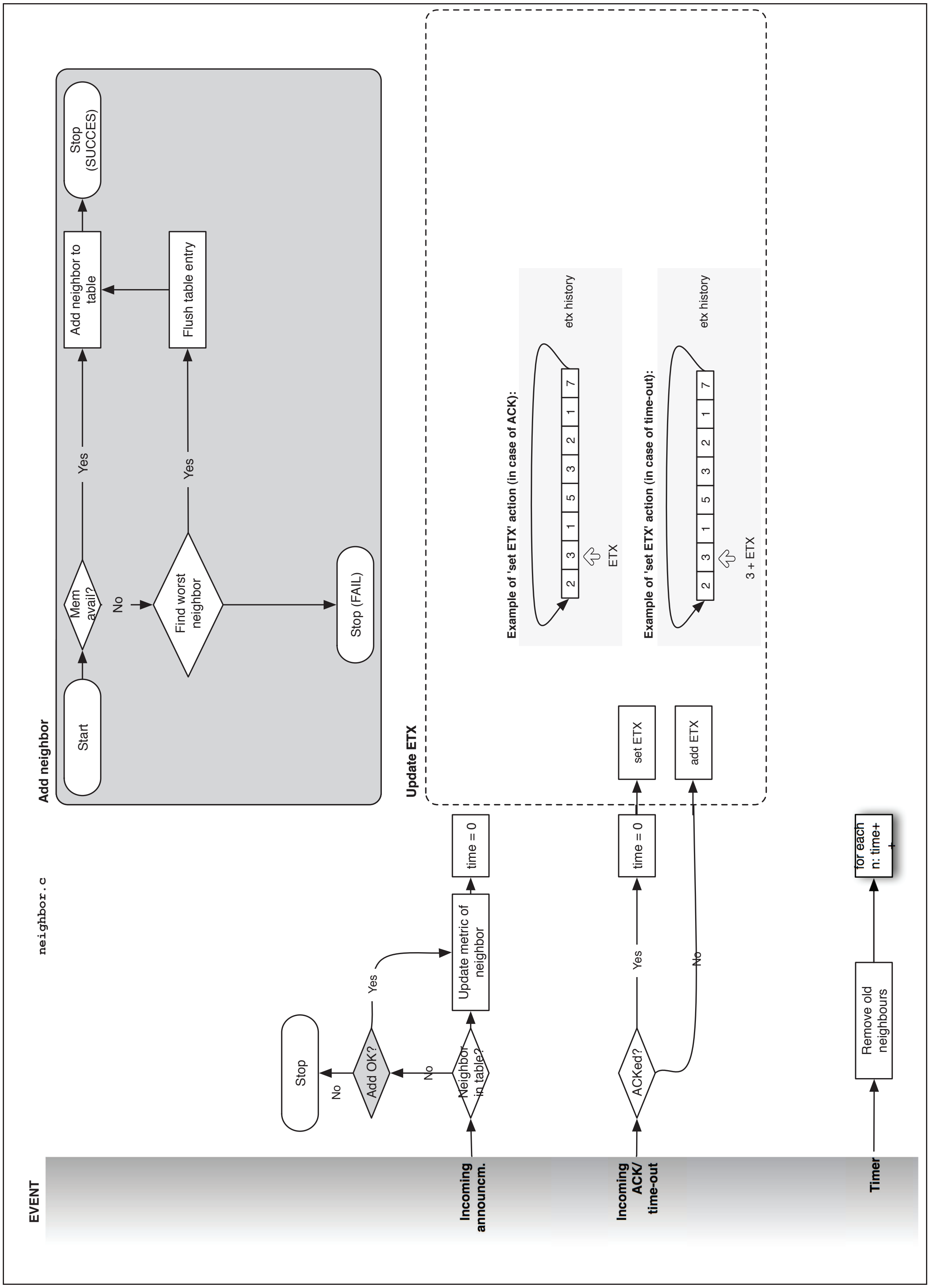

Figure 11 – The Contiki collect protocol: link estimator and neighbor management

2.5.1.4 Duplicate packet filtering and packet aging

See also Figure 10.

Duplicate packets can be created upon retransmission when the ACK is lost. Without duplicate packet elimination, these will be forwarded, possibly causing more retransmissions and more contention, and wasting energy. To protect against forwarding duplicate packets, a node will not forward a packet that it has recently forwarded. Thereto it keeps a small history of recently forwarded packets (currently 2 packets), which are uniquely characterized by their packet ID (EPACKET_ID) and originating node (ESENDER). If a node receives a packet that has the same ID and originator address, the packet is dropped.

Additionally, to prevent packets from roaming through the network forever, the packet is dropped if it exceeds a certain maximum number hops. Thereto each packet has a time-to-live (TTL) attribute, which is initialized to 10 and gets decremented each time a packet is forwarded. On receiving a packet with a TTL value of 1 or lower, the node drops the packet.

2.5.2 Protocol attributes

The following attributes (i.e. fields) are attached to a packet sent using the collect protocol:

- **EPACKET_ID.** Each packet originating from a node gets a 4-bit packet ID (also called sequence number). Together with the originating address stored in the ESENDER attribute, it uniquely identifies the packet.
- **ESENDER**. The address of the node initializing the send of the packet.
- **HOPS**. This attribute represents the current hop count. It is initialized to 1, and will be incremented on each *forward* by a node.
- **TTL**. The time-to-live represent the maximum number of hops the packet can make. It is initialized to 10. On each forward by a node, the TTL value is decreased by 1. If a node receives a packet with TTL equal to (or lower than) 1, the packet is discarded. This prevents packets from traveling through the network forever.
- **MAX_REXMIT**. Used by the underlying reliable unicast Rime layer (runicast). This value represents the maximum number of link-level transmissions to send or forward the packet to a neighbor. If the maximum number of transmissions is reached, the packet times out. Upon time-out of a packet, the packet is dropped, the neighbor ETX data is updated, and the rtmetric is updated as well.

Note that there is no destination address attribute, since for the collect protocol all data is send to the sink, i.e. the node with the routing metric 0.

The following table lists the attributes attached to a packet by the collect protocol and underlying Rime layers (shown from left to right).

Table 2 – Rime packet attributes for the collect protocol and underlying layers. The figure indicates the number of bits used

Attribute	collect	reliable unicast	unicast	broadcast	anonymous broadcast
EPACKET_ID (end-to-end)	4				
ESENDER (end-to-end)	Addr length				
HOPS	4				
TTL	4				
MAX_REXMIT	3				
PACKET_ID (single-hop)		2			
PACKET_TYPE		1			
RELIABLE		1*			
RECEIVER (single-hop)			Addr length		
SENDER (single-hop)				Addr length	

** The RELIABLE attribute is only used internally to optimize radio operation (guiding the decision to either switch off the radio after sending, or to keep the radio on in anticipation of the ACK), but is not attached to the outgoing packet.*

2.5.3 Operation (initialization, sending and receiving)

2.5.3.1 Initialization

When the collect protocol is initialized (by calling `collect_open`), the underlying reliable unicast (runicast) connection is initialized, the rtmetric of all nodes are initialized, and neighbor discovery is started (by registering with the announcement primitive) on a separate channel.

For the protocol to operate correctly, the application should assign one node as the sink node by calling `collect_set_sink` on that node.

2.5.3.2 Sending messages

See Figure 10.

To send data to the sink, an application calls the `collect_send` function. The algorithm operates as follows:

1. First, all collection specific attributes (see § 2.5.2) are set.
2. Then, the node sends the packet to its parent using reliable unicast.

   The underlying reliable unicast will send a packet reliably to a neighbor, i.e. it will try to deliver a message to the neighbor by trying for a maximum number of times until the packet is ACK'ed. If it succeeds, it notifies the upper layer (in this case, the collect protocol). If it doesn't succeed (i.e. the packet has not been ACK'ed after the specified number of times), it times out and notifies the upper layer of the time-out.

The parent node is determined each time upon sending by requesting the best neighbor from the neighbor table. If no best neighbor can be found (i.e., the table is empty), the packet is dropped and the node actively listens for announcements (during a limited period) to detect potential neighbors.

Should the send function have been called on the node that is the sink node, the receive function of the application using the collect protocol is called, and no reliable unicast send takes place.

3. When the sent packet gets ACK'ed by the parent (`node_packet_sent` is called) or times out (`node_packet_timedout` is called), the node's rtmetric and the ETX of the parent is updated.

2.5.3.3 Receiving messages

See Figure 10.

Two types of packets can be received: data packets and announcement packets.

Data packets

When a node receives a collection data packet (via reliable unicast which calls the function `node_packet_received`) several things happen:

- First, duplicate packet filtering is executed: the packet is checked against the last two forwarded packets. If the ID (EPACKET_ID attribute) and originator address (ESENDER attribute) match with any of these, the packet is dropped.
  If not dropped, the ID and originator address of the received packet is stored in the recently forwarded packets table.
- If the node receiving the packet is the sink node, the application using the collect protocol is notified of the reception of a packet. The originator address, packet ID and number of hops the packet has travelled are provided as arguments.
- If the node receiving the packet is NOT the sink node, the packet has to be forwarded.
  - If the TTL value is 1 (or lower), the packet is dropped.
  - The HOP count is incremented, and the TTL is decremented.
  - The packet is forwarded to its parent using the underlying reliable unicast layer. The parent selection is identical as described under "Sending messages" (see § 2.5.3.2).

When a packet has not yet been ACK'ed or timed-out, new packets cannot be forwarded but are dropped instead. In a recent update of the collect protocol, a packet queue has been added. This will not be considered in this work however.

Announcement packets

When a node receives an announcement packet (`received_announcement` is called by the announcement back-end, see § 2.4.3) the following actions are taken:

- The neighbor who sent out the announcement is checked against the neighbor table. If not yet present, the neighbor is added to the table.

Otherwise, the neighbor's rtmetric value is updated in the neighbor table.

If the neighbor table is full, the worst neighbor in the neighbor table is evicted, and replaced by the new neighbor (see Figure 11). The worst neighbor is defined as the neighbor with the highest route metric to the sink.

- The node updates its rtmetric.

2.6 The four-bit wireless link estimator (4B)

The link estimator in the Contiki collect protocol bases its calculation on information from the data link layer only, i.e. link level acknowledgments or time-outs.

The **four-bit wireless link estimator** (4B), proposed by Fonseca et al. [8], provides well-defined interfaces to combine information from the physical, data-link and network layers for link estimation. 4B uses ETX (see § 2.3) as the link quality metric.

In 4B, the interfaces provide 4 bits of information compiled from different layers:

- A **white bit** from the physical layer, denoting the low probability of decoding error in received packets. If the white bit is set, the medium quality is high. If the white bit is not set, then the medium quality may or may not have been high during the packet’s reception.
  As a rule of thumb, the medium quality is considered high when the Link Quality Indication (LQI) of a packet is above 90%. The LQI is a characterization of the strength and/or quality of a received packet, as defined by the IEEE 802.15.4 Standard.[12]
- An **ack bit** from the data link layer, indicating if an acknowledgment is received for a sent packet. If the bit is set, the packet was acknowledged by the data link layer transmission. If not set, the packet may or may not have been received successfully.
- A **pin bit** from the network layer, indicating if the link estimator can remove the neighbor entry from its neighbor table or not. The network layer sets the pin bit for those entries that should not be removed from the table. This makes sense for, for example, the neighbors of the sink node: they would want to keep the sink in their neighbor table at all cost.
- A **compare bit** from the network layer, indicating if the metric value of the neighbor from which a packet was received is better – i.e. lower – than the metric value of one or more entries in the neighbor table. If the bit is set, the network signals that the path through that neighbor is better than a path through at least one neighbor in the neighbor table.

Figure 12 shows 4B mapped to the Rime protocol stack. 4B is typically represented by a triangle to indicate the three layers that it utilizes.

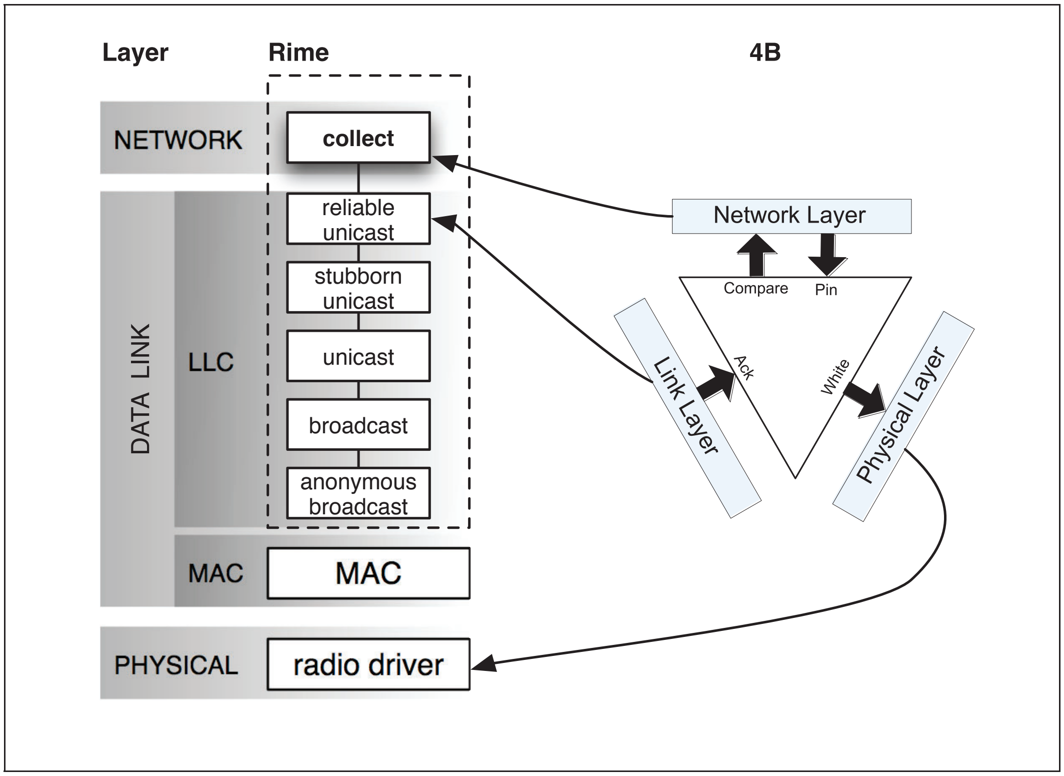

Figure 12 – 4B mapped to the Rime protocol stack

3 Implementation of 4B

In this section we will look at the operation details of the 4B estimator, show how it is different from the collect estimator, and discuss some Contiki implementation notes. A modified reliable unicast Rime primitive will also be introduced to correct for a flaw in the current reliable unicast Rime primitive.

3.1 Where does 4B fit in?

As outlined in § 2.5.1, the Contiki collect protocol is made up of a number of high level components: neighbor discovery & management, link estimation and routing.

The implementation of 4B will impact the first two components (neighbor discovery & management and link estimation). The routing component will be kept identical for the 4B implementation.

3.2 The 4B hybrid estimator algorithm

The 4B link estimator that will be implemented in the Contiki collect protocol is based on the 4B TinyOS implementation (see [17], [18] and [19]).

The 4B link estimator described in the 4B paper[8] and also implemented in TinyOS is a hybrid link estimator: to calculate the link quality it combines the information provided by the three layers with periodic beacons (i.e. broadcast packets without any payload).

Table 3 outlines the high-level differences between the original Contiki collect estimator and the 4B estimator.

Table 3 – High-level difference between the link estimator in the Contiki collect protocol and the 4B estimator

	Contiki collect protocol	**4B**
Uses **data packets** to:	• Estimate bidirectional link quality (using ACK/time-out)	• Estimate bidirectional link quality (using ACK/time-out) • Report metric value (however: not implemented)
Uses **beacons/ announcements** to:	• Broadcast presence (to populate neighbor table) • Report metric value (for routing)	• Broadcast presence (to populate neighbor table) • Report metric value (for routing) • Estimate link quality (inbound only!)

Figure 13 shows where each of the four bits is used in the 4B algorithm. The ack bit is used for sent data packets to calculate the ETX value. The pin, white and compare bit are all used to decide upon insertion of a neighbor in the neighbor table.

As can be seen when comparing Figure 11 with Figure 13 the differences between the Contiki collect protocol estimator and the 4B estimator are:

- a different neighbor eviction/insertion policy,
- usage of data packets, in addition to beacons, to update link quality,
- a different link quality (i.e. ETX) calculation.

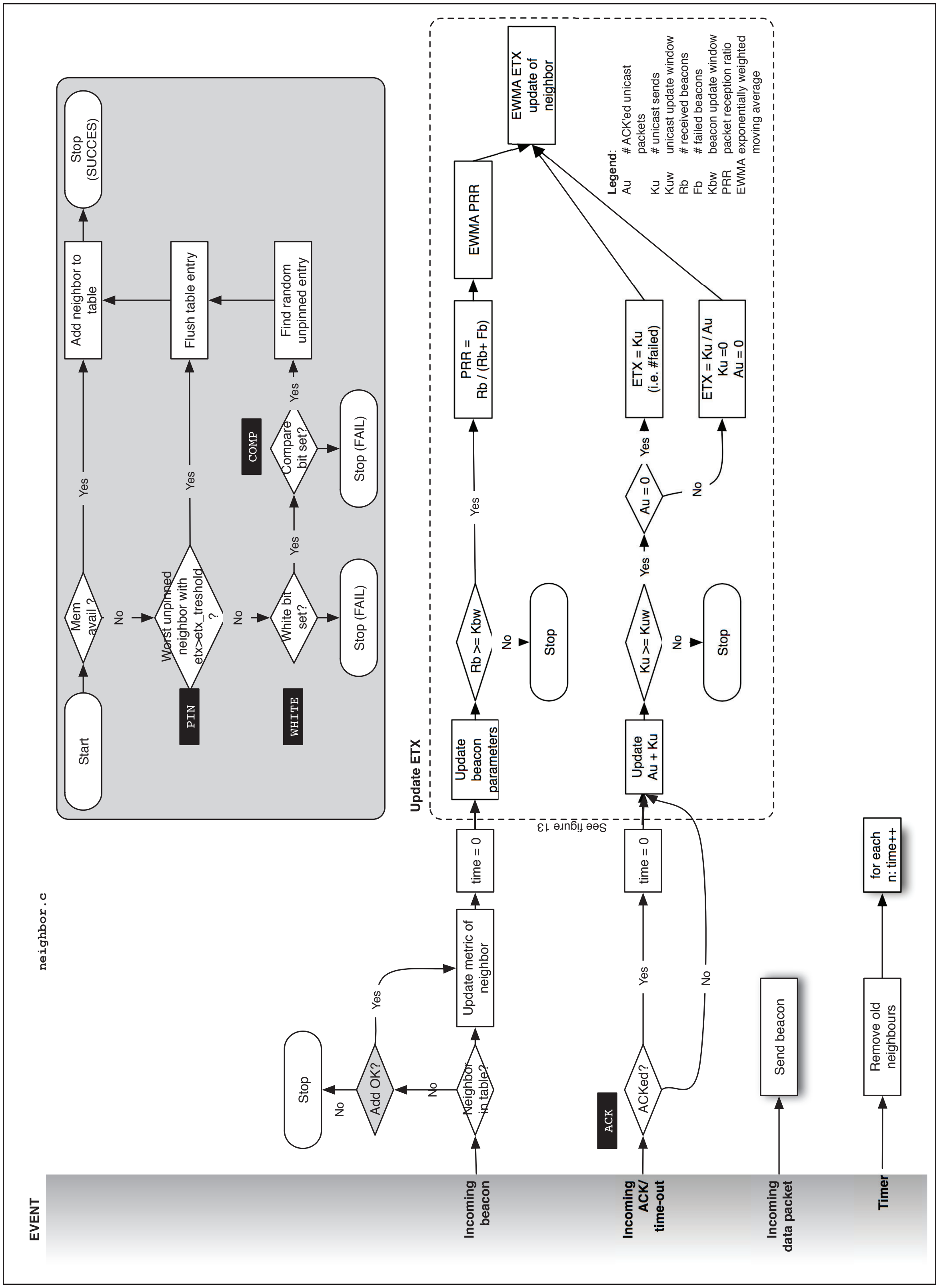

Figure 13 – The 4B link estimator

3.2.1 Neighbor insertion (neighbor table management)

See Figure 14.

When a node receives a beacon (or data packet) from a neighbor that is not present in the neighbor table, and there are no more free entries, the neighbor is evaluated for replacing a current entry in the table:

1. The table is scanned for the (unpinned) neighbor with the worst link quality, i.e. the highest link ETX value. Note that this is not the same as for the original Contiki collect estimator, where the worst neighbor is determined based on the worst path to the sink (i.e. the sum of the link ETX and metric value). In addition, blacklisting[12] is used, i.e. the worst neighbor has to be worse than a certain threshold to be evicted.
2. If no such neighbor can be found, the white bit is used to further decide upon potential insertion. If the white bit is not set, indicating the radio quality is low, the neighbor will not be considered for insertion.
3. If, in addition to the white bit, the compare bit is set for that neighbor, the neighbor will be inserted. The compare bit is reported to be set if the neighbor's metric value is better (i.e. lower) than a neighbor already in the table. If the compare bit is not set, the neighbor will not be inserted.
4. Finally, when the neighbor has been selected for insertion, the algorithm evicts a random, unpinned entry.

Note the difference with the Contiki collect protocol estimator, where the worst neighbor was calculated as the neighbor with the worst path to the sink, i.e. the neighbor with the highest sum of the link ETX *and* metric value.

For 4B, the worst neighbor is first evaluated based *only* on the link ETX values. The metric value is only taken into account later using the compare bit. The reason for this split evaluation, is because of the strict interfaces that are proposed by 4B. Strictly speaking, the link ETX value is only known by the link estimator, whereas the metric value is part of the routing logic, and thus only known by the network layer.

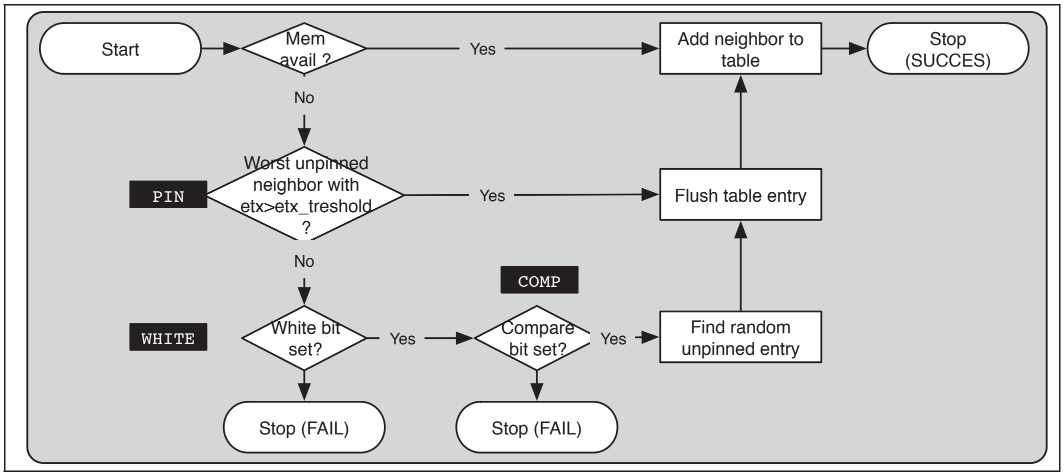

Figure 14 – The 4B link estimator – neighbor insertion

3.2.2 Link quality calculation

See Figure 15.

Before being combined, the ETX values are separately calculated for the sent unicast packets and received beacons.

- The unicast ETX value is updated every k_{uw} unicast packets. k_{uw} is called the unicast update window.
  If a out of k_{uw} packets are acknowledged by the receiver, the unicast ETX estimate is

  $$ETX = \frac{k_{uw}}{a} \qquad \text{(eq. 2)}$$

  If a = 0, then the estimate is the number of failed deliveries since the last successful delivery.
- The beacon ETX value is updated every k_{bw} beacons (of which some might be missed). k_{bw} is called the beacon update window.
  The calculation is similar, but involves an extra step. First the packet reception ratio (PRR) is calculated based on the number of received beacons R_b and failed beacons F_b.

  $$PRR_{last} = \frac{R_b}{R_b + F_b} \qquad \text{(eq. 3)}$$

  This instantaneous PRR value is dampened using an exponentially weighted moving average (EWMA) function:

  $$PRR_{new} = \alpha \times PRR_{old} + (1-\alpha) \times PRR_{last} \qquad \text{(eq. 4)}$$

  with alpha being a weighting factor between 0 and 1.
  The resulting PRR value is then inversed to turn it into an ETX value.

  $$ETX = \frac{1}{PRR} \qquad \text{(eq. 5)}$$

- These two streams of ETX values are combined in a second EWMA:

  $$ETX_{new} = \alpha \times ETX_{old} + (1-\alpha) \times ETX_{last} \qquad \text{(eq. 6)}$$

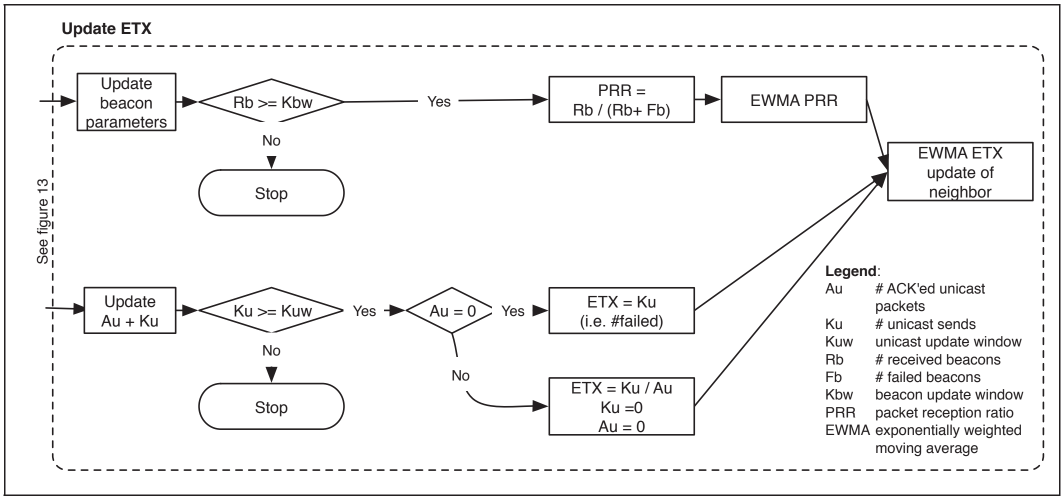

Figure 15 – The 4B link estimator – link quality calculation

Figure 16 illustrates an example calculation (source [8]) for a unicast update window k_{uw}=5 and beacon update window k_{bw}=3. The alpha weighting factor is 0.5. Incoming packets are light boxes, dropped packets are marked with an ‘x’. The link estimator calculates link estimates for each of the two estimators at the times indicated with vertical arrows.

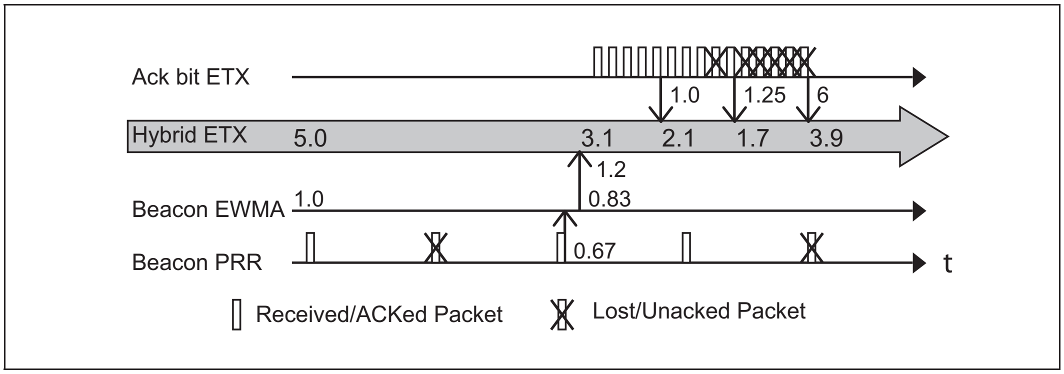

Figure 16 – Example hybrid ETX calculation (source: [8])

3.3 Implementation notes

- **Interface vs. estimator logic** – The four-bit wireless link estimation paper by Fonseca at al.[8] proposes a strictly defined link estimator interface consisting of 4 bits; the paper then evaluates the performance of a hybrid estimator using these interfaces (as described in § 3.2).

  In this work, the hybrid estimator logic has been implemented but not with the strict layer separation due to time constraints. For performance evaluation of the 4B hybrid link estimator logic, the absence of the strict layering is not a problem.
- **Announcement primitive vs. broadcasting** – The original Contiki collect estimator uses the announcement primitive (see § 2.4.3) to broadcast the metric value. In the 4B implementation, we have chosen not to reuse the announcement primitive for broadcasting beacons, but instead write a dedicated beacon broadcasting mechanism using the broadcast Rime primitive (see Figure 8).
- **Location of broadcasting logic** – The 4B beacon broadcasting mechanism is implemented as part of the link estimator module. Since the beacons are also used for broadcasting routing information (the metric value), one could argue that the broadcasting mechanism should be part of or at least controlled by the collect protocol, i.e. the network layer. However, to restrict changes to the collect module, broadcasting has been implemented in the link estimator.
- **Data link layer (runicast) changes** – The Contiki collect protocol builds on the underlying reliable unicast (runicast) Rime primitive (see Figure 8) to send its data packets. The reliable unicast primitive requires a ’maximum number of transmissions’ parameter when sending a packet. The reliable unicast only reports to the collect layer when the packet has been ACK’ed or

when the maximum number of transmission has been reached (i.e. a time-out).

The 4B estimator, however, requires reporting of an acknowledgment or absence thereof after each packet (re)sent. Thereto, the reliable unicast primitive had to be changed to support notifying the link estimator upon each individual ACK or time-out.

- **ETX vs. EETX** – The 4B TinyOS implementation internally represents the link quality as *extra* expected number of transmissions (EETX). So, an optimal link has a link quality of EETX=0, corresponding to one transmission (ETX=1).

  In the 4B Contiki implementation regular ETX values have been used instead of EETX values.
- **WHITE BIT (LQI)** – For the CC2420 radio, present in the Sentilla Jcreate motes used at VUB, the white bit is set when the Link Quality Indication (LQI) of a packet is above 105. The LQI is a characterization of the strength and/or quality of a received packet, as defined by the IEEE 802.15.4 Standard.[12] The LQI value range is specified[12] to be from 0 – 225. However, the range of LQI returned by the CC2420 radio is from 50 to 110[16]. So a packet with an LQI value greater than 105 indicates a link quality better than 90%.

  In Contiki, the LQI can be queried directly through a sensor interface call.
- **PIN BIT** – In the 4B TinyOS implementation, a neighbor is pinned in two cases: (a) if the neighbor is the sink node, or (b) if the neighbor is elected as parent (when a new parent is chosen, the old parent is unpinned). This prevents the parent or sink from being removed from the neighbor table (see Figure 14).

  In the 4B Contiki implementation, the pin bit logic is not explicitly implemented (i.e. no arbitrary neighbors can be pinned by the collect protocol). However, the parent neighbor is prevented from being evicted.

3.4 Rewrite of the Rime reliable unicast primitive

3.4.1 The problem

As mentioned in § 3.3, the Contiki collect protocol uses reliable unicast (runicast) to send its packet to a single-hop neighbor (see also § 2.4.2).

It turned out that the default runicast implementation is inherently flawed in reporting the number of required transmissions to the upper layer primitive (collect, in our case). This value is critical however to correctly evaluate the performance of both link estimators. Therefore, reliable unicast had to be rewritten to support correct (re)transmission count reporting.

The reason for the flaw is the way in which runicast builds on stubborn unicast (stunicast) to deliver its packets reliably (see Figure 17).

The stunicast primitive sends and resends the packet until the upper layer primitive (runicast) cancels the transmission. Stunicast reports to runicast each time it has (re)sent the packet. Based on this reporting, runicast will let stunicast continue sending, or, if the maximum number of transmissions has been reached, will instruct stunicast to cancel the repeated sending and report a time-out to the upper layer primitive (e.g. collect).

This interaction is flawed, and makes it for example possible for runicast to both report a time-out and an acknowledgment to the upper layer primitive. This is easy to illustrate if we consider a packet that has to be sent reliable with a maximum transmission count of 1:

1. Runicast will instruct stunicast to start sending the packet.
2. Stunicast will send the packet, schedule the next resend, and report the send to runicast.
3. Since runicast keeps track of the maximum number of transmissions (i.e. 1), and since this value has now been reached, it will instruct stunicast to cancel any subsequent sends. Runicast will report a time-out to the upper layer primitive (collect) because the maximum number of transmissions has been reached.
4. However, the packet that was sent might by now be ACK'ed. So, runicast will now also report an ACK to the upper layer primitive, while it has already reported a time-out!

The reason for this bug is that the reporting from stunicast to runicast about the sent packet is too late. Stunicast should report to runicast just *before* resending, so that runicast can cancel the pending resent in case the maximum transmission count has been reached.

3.4.2 The solution

To overcome this bug, I rewrote the runicast primitive while getting rid of the reliance on stunicast. I've called the new implementation **brunicast** (a 'better runicast') and modified the collect protocol to use it.

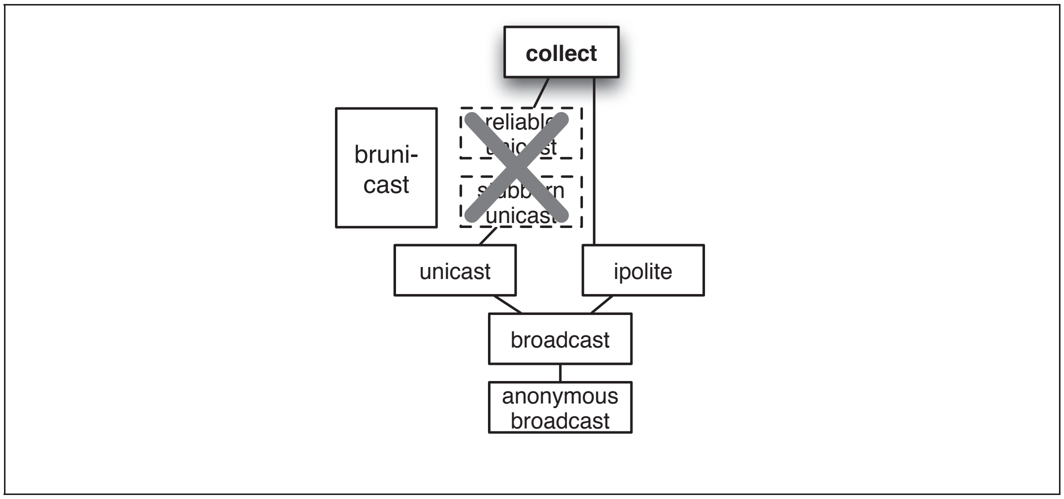

Figure 17 – The brunicast (better runicast) primitive that replaced the flawed runicast and stunicast primitives

The brunicast primitive schedules an *evaluation* instead of a resend. If an acknowledgment packet is received before the evaluation function has been called, the scheduled evaluation is canceled, and the acknowledgment is reported to the upper layer primitive. If no acknowledgment comes in, the evaluation function is called when the timer expires. Upon evaluation, the packet is either resend if the maximum transmission count has not yet been reached, or a time-out is reported to the upper layer primitive in the other case.

I’ve submitted brunicast to replace runicast and stunicast in the official source tree.