Iowa's Neighborhood Component

Ted Herman

November 3, 2007

Abstract

This document describes Iowa's Neighbor interface and its implementation, which is an alternative to the LEEP protocol described in TEP 124 of TinyOS. The Neighbor interface is used in Iowa's Timesync protocol and some other applications, which exercise the hierarchical design of the interface. The implementation takes "neighbor" to mean another node such that there is a stable link with communication in both directions between the two nodes. This document describes the interface and implementation of that component. One idea of the implementation (though not yet fully developed) is that no new messages need to be sent in the determination of neighbor health, because components such as Timesync happen already to have periodic beacons.

1 Introduction

Some sensor network applications tolerate dynamic reconfigurations to the network. The protocol architecture therefore has components, perhaps within network or MAC layers, that support dynamic changes to topology. Iowa's support of dynamic topology is limited to a neighborhood service that identified, for each node, the current set of stable links to other nodes. This document explains the interface and implementation methods of this neighbor service. We begin with a description of the interface, and then document the implementation of the protocol underlying the service.

2 The `Neighbor` Interface

The `Neighbor` interface is used by Timesync and other middleware at runtime to determine who the "stable" neighbors are for a given node. The `Neighbor` interface is wired to a TinyOS component that has tables, timer actions, and messages to establish the stability of a link between neighbors. Though it may be an unnecessary complication, the `Neighbor` interface is parameterized: a module using the interface looks like this:

```
module WhateverP {
   uses interface Neighbor;
   ...
   }
```

The `Neighbor` wiring for the `Whatever` module looks like this:

```
    WhateverP.Neighbor -> TnbrhoodC.Neighbor[1];
```

The above declares that `WhateverP` is wired to the "layer 1" usage of the `Neighbor` interface. The idea of this "layer" parameter is that middleware tends to be layered, and the stability of a link could depend on the layer. For example, a layer that rarely communicates over a link, and tends to use the link when there is not much congestion, could find the link acceptably stable, whereas another layer might find the link unstable because it uses the link at times of high contention. The current design assumes that layers are hierarchical, so instability at layer i implies instability at layer j, $j > i$. The current implementation supports at most four layers, numbered 0-3.

The `Neighbor` interface caters to two kinds of users (where "user" means a module that uses `Neighbor`). Users with *beacon behavior* are similar to components such as timesync, and possibly routing: each node periodically sends some group of messages – status messages or values to share. The exact number of messages is unpredictable, but at least one message is sent in each period by each node; different nodes may send different numbers of messages in the same period. The second kind of user has a more predicatable behavior: in any given time interval, all nodes send the same number of messages. We call the first kind of user a *periodic beacon* user, and the second kind of user a *self-clocking* user. Note that all the messages mentioned above are presumed to be addressed to all neighbors (that is, destination is `AM_BROADCAST_ADDR`). These are called *beacon* messages below in interface descriptions.

Notes

- First off, I doubt we can get away with calling this the `Neighbor` interface, because it's likely that some component in TinyOS will claim this name. Any suggestions for a more durable name?
- The implementation is currently inseparable from Iowa's Timesync. This was mainly done to enable a leaner implementation, since Timesync also uses tables for neighbors and to avoid some duplication of structures in memory, timesync and the `Neighbor` component share data structures. We actually planned on making this a separable component; maybe it works, but we didn't test it without Timesync.
- The rest of the document is sketchy, but you might get some general idea of how it works.

Commands and Events

command `error_t regist( uint32_t freq )`

The using module dynamically "declares" its usage of the `Neighbor` interface with the `regist` command. There are two forms of using this command:

1. If the using module has periodic beacon behavior, then `freq` is the period between beacons, in units of 1/8 seconds (that is, `freq` = 16 denotes two seconds).
2. If the using module is a self-clocking user, then `freq` is zero.

command `void sent()`

The using module should call `sent` when it is known that a beacon message was successfully transmitted (this is usually known within the `sendDone` event).

command `void received( uint16_t Id )`

The module wiring to Neighbor should call `received(Id)` upon receiving a beacon message from the node with identifier `Id`.

command `bool present( uint16_t Id )`

The `present(Id)` command can be called to answer the question: is node `Id` currently a neighbor?

event `void join( uint16_t Id )`

The `join(Id)` is signalled when node `Id` joins the neighborhood, that is, when the link between this node and `Id` is judged to be bidirectionally stable.

event `void leave( uint16_t Id )`

Event `leave(Id)` is signalled when the link between this node and `Id` no longer meets the criterion of being bidirectionally stable.

command `bool allow( uint16_t Id )`

The `allow` command is for testing and topology control. You can modify the implementation of `allow` to exclude certain `Id`s from being part of a neighborhood – this allows you to make a testbed with multiple hops, even if all nodes a physically close together.

3 The `TnbrhoodP` Module

Okay, this is (perhaps the only) part you want to read, how it all works. As mentioned before, the neighborhood component mixes concerns of supporting the `Neighbor` interface with concerns of the Timesync service; here we mainly describe support of the `Neighbor` interface.

State Variables

Let's begin with the state variables of `TnbrhoodP`:

```
uint8_t sendCount[NUM_TYPES];   // how many messages have been sent
uint32_t sendTime[NUM_TYPES];   // last send time for each type
uint16_t sendFreq[NUM_TYPES];   // frequency of sending
neighbor_t nbrTab[NUM_NEIGHBORS];
```

The implementation defines a header-file constant NUM_NEIGHBORS=12 as a bound on the maximum number of neighbors. This bound can be altered, however the TinyOS constraint on payload size prevents us from scaling this too far in the current implementation. The other header-file constant is NUM_TYPES, currently 4, which is the depth of the hierarchy of Neighbor users. The one structure mentioned above is neighbor_t, which has numerous fields for timesync, and the following that help with Neighbor support:

```
uint8_t connected; // indications of connectedness;
uint8_t rcvHist[NUM_TYPES]; // history of receive events
uint16_t Id;      // identifier of the neighbor
```

These three fields are present for each neighbor; an explanation of how these three fields are used:

- connected is a bit array, one bit per level in the hierarchy (see above) of Neighbor users. Bit i in this array is *true* iff there is a stable link to node Id at layer i. The implementation enforces that bit i can be *true* only if bit j is *true* for all $j < i$.
- rcvHist[i] is an 8-bit array for layer i in the hierarchy. This array represents the recent history of beacon messages (with respect to layer i) received by this node from the other node, named Id. This array is treated as a shift register, that is, it is shifted to the left one place to make room for new "history" (thus discarding ancient history). Exactly when and how this happens depends on how layer i registered itself with the Neighbor interface (see below).

Unlike the per-neighbor fields described above, the sendCount, sendTime, and sendFreq fields are arrays with one item for each layer. These are:

- sendCount[i] is a count of the number of beacon messages sent (as observed via Neighbor.sent() commands) since layer i registered. This counter maxes out at 8, that is, if sendCount[i]=8, then adding another message is a no-op.
- sendTime[i] is used if layer i registers as a periodic beacon behavior; each Neighbor.sent() records the current time (in 1/8 seconds) to sendTime[i].
- sendFreq[i] is just the parameter that layer i passed when registering itself.

Note: I've only superficially compared this bit-based implementation with the LEEP implementation, which uses time-averaging to compute link quality. I didn't see that LEEP uses lower-layer concepts for link quality, that is, LQI or RSSI data. It looks like both LEEP and Neighbor (described here) just rely on goodput in both directions across a link. Like LEEP, the Neighbor implementation efficiently senses bidirectional stability by using local broadcast, rather than unicast, so that a single message notifies all neighbors of their potential stability. In a hierarchical usage of Neighbor, it's possible to use different sending power levels at different layers in the hierarchy, but I haven't tried this.

Tasks and Health Maintenance

Currently, the TnbrhoodP modules has three background tasks, but two of these are only for timesync support (and are actually optional there, for skew maintenance). The task we need to describe here is the "aging" task. Periodically, with period HEALTH_CHECK_FREQ (again, a constant in a header file, now equal to one minute), the aging task runs. When fired up, this task examines all current neighbors to cull out any unhealthy neighbors. If some neighbor is found to be unhealthy, the aging task signals the Neighbor.leave() event for the approriate node and layer, so that the application is notified of a neighbor leave event.

The definition of healthy is somewhat involved. With full timesync support, health should imply bidirectional connectivity. Biconnectivity is established by another background task of TnbrhoodP, which periodically transmits a "health" beacon; the content of the health beacon is a message containing a list of Ids that have recently been overheard sending a timesync beacon. The health message is supposed to contain up to NUM_NEIGHBOR such identifiers. Now, if a node p observes a health beacon from neighbor q, and that message has p in its list, then p can claim that the (p, q)-link is bidirectional (well, almost; we really need both nodes to make this observation with respect to each other).

Assuming that the test of biconnectivity passes, we get to the details of what "healthy" means for a given layer i and a neighbor Id. Here are the rules:

- If the number of *true* bits in rcvHist[Id] is at least half the sendCount[i], then the link is healthy;
- Otherwise, if the rcvHist[Id] is empty, the link is healthy – this seems like an error, but the thinking here is to promote fast joining.
- Otherwise, there is a tricky test (with ugly details) to favor cases when only a few messages have been received so far, again to promote fast joining, something like TCP's hack for slow start. Specifically, the following bit patterns are healthy for rcvHist[Id]:

  00000100, 00000010, 00000110, 00001001, 00000001, 00001011, 00001110

  These bit patterns (read how rcvHist evolves below) are healthy because they represent an incipient link which hasn't had time to build much history. The intuition is that if a neighbor has just appeared and no more than two consecutive messages have been dropped, the link is healthy. Thus 00001000 is *not* a healthy pattern.

These rules are debatable, but they survived our testing.

Sending and Receiving Messages

Layer i calls Neighbor.sent() and Neighbor.received() so that the TnbhrhoodP can keep track of message flow between neighbors.

Invocation of `Neighbor.received(Id)`

The logic here depends on how layer i registered. If `sendFreq[i]` is zero, the call to `Neighbor.received(Id)` reduces to the assignment:

```
nbrTab[i].rcvHist[Id] |= 1;
```

Example: if `rcvHist[Id]` is `0x13`, then the above statement does nothing; but if the value were `0x18` beforehand, then the new value will be `0x19`. The effect of this assignment is that receipt of multiple messages will be ignored (only the first message counts), until the `rcvHist` array, which is a shift register, is next "aged". However, if `sendFreq[i]` is not zero, then before executing the statement above, the following is performed:

```
nbrTab[i].rcvHist[Id] <<= 1;
```

Example: if `rcvHist[Id]` is `0x19` beforehand, then the new value will be `0x32`, or in binary the old and new values are 00011001 and 00110010 respectively. This shift of one place makes room for the next bit; thus if `sendFreq[i]` has been specified, the `rcvHist` array is a more accurate count of the number of messages recently received.

Invocation of `Neighbor.sent()`

Again, the logic here depends on how layer i registered. If `sendFreq[i]` is zero, the code in effect is:

```
// age all bit vectors (one more sent vs record of received)
sendCount[i] = (sendCount[i] < 8) ? sendCount[i]+1 : 8;
for (p=nbrTab;  p<nbrTab+NUM_NEIGHBORS; p++) p->rcvHist[i] <<= 1;
```

This adds one to the counter of sent messages and ages all the receive history arrays. The idea here is that this layer is "self-clocking" and there should be approximately the same number of beacons received as sent. The main idea here is that neighborhood health can be determined without any new kind of message; we only have to be observers of messages sent and received to determine health.

What happens if `sendFreq[i]` is nonzero? Then we are dealing with layer that uses a time-based interval rather than "tit for tat" behavior. Here, the code is:

```
w = call OTime.getCurrentTime();
v = w - sendTime[i]; // v is elapsed time since last send
if (v >= sendFreq[i]) {
   sendCount[i] = (sendCount[i] < 8) ? sendCount[i]+1 : 8;
   for (p=nbrTab;  p<nbrTab+NUM_NEIGHBORS; p++)
      p->rcvHist[i] <<= 1;
   sendTime[i] = w;   // remember new period starting
   }
```

Thus `Neighbor.sent()` only does something when a "new period", according to the registered period length, has been reached. Again the action is to add one to the `sendCount` and shift the `rcvHist` arrays for each neighbor, but doing so only once per period.

4 Timesync and Neighbor

The current implementation uses the notion of "biconnected" as a filter on health, that is, in order for a link to be healthy, it first must pass a biconnectivity test. Here we explain how biconnectivity is evaluated with the help of the Timesync protocol. First, the Timesync protocol calls another interface command, `Tnbrhood.record`, whenever it receives a time sync beacon message. This command creates a table entry, if necessary, which is how `Tnbrhood` becomes aware of who are candidate neighbors. An attribute of a Timesync beacon is the state of the transmitting node, which is either "normal" (meaning that the node has been running long enough to be synchronized) or "initializing" (meaning that the transmitting node hasn't yet synchronized its clock). The state of the transmitted node, and other timestamp values, are recorded by `Tnbrhood` in a table.

Periodically, according to an interval set by Timesync (dynamically), the `TnbrhoodP` module transmits a brief list of all nodes in its table; this is currently calculated to fit into a 24-byte payload, which fits into a TinyOS message. When a node `p` receives such list from node `q`, then `p` checks if `q` is in `p`'s table and also that `q` is a "normal" node: when these two conditions hold, then `p` marks the table entry for `q` as a bidirectional link. This use of a 24-byte payload as a list of all neighbors has good and bad points: it is very economical compared to alternative protocols that send health messages on a per-neighbor basis (one message informs many neighbors of connectivity); but the scheme would perhaps be clumsy if security or large neighborhood sizes exceed the payload constraint for a single message.

Do bidirectional links ever expire? Yes, this is managed in the aging task; when this task signals `Neighbor.leave(Id)` for layer 0 (which is implicitly timesync), then the table entry for node `Id` is deallocated and cleared, available to be reused by another, new node.