Netwrok Project Report

5112409040 Xuan Luo, Siyi Yang, Haoran Wang 511309077

June 20, 2014

Abstract

In the project, we fulfill two network protocols. The first one is the basic broadcasting protocol, which broadcasts the packet to neighbors no matter what packet the *sensor* receives . The second protocol supports routing, and dynamically updates the forwarding table. Besides, we explain the principle and motivation of second protocol, and prove its robustness to position change.

1 Basic Protocol: Naive Broadcast Protocol

The first protocol we fulfilled just broadcasts every packet the *sensor* receives. We think this protocol is not efficient enough. It neither keeps the routing mechanism. Since the idea is simple and not exciting, we focus our attention on introducing the second protocol. Our implementation of this protocol resides in the "broadcast version" folder. The rest sections are all about second dynamic protocol.

2 Second Protocol: Dynamic Routing

The second protocol supports routing and forwarding among sensors. Besides, it also adaptively changes the routing table according to routing information of adjacent *sensors*.

3 Data Structure On *Sensor*

Sink is the mote to transfer the received sensed message to the serial port on PC. Each *sensor* keeps two routing tuples. Each routing tuple contains two elements, *First Hop* and *Length*. *Length* is the number of hops in shortest path to *sink*, it is set to *Infinity*(which is equal to number of Nodes) if there is no path to *sink*. *First Hop* is the next node in shortest path to *sink*, which is set to 0 if no path exists.

One of the two tuples is *Current Tuple*, which is determined by *Update* Operation. The other is *Previous Tuple*, which is exactly the *update tuple* before last *Reset*.

The reason we don't record the whole routing table is that there is only one destination in this project, which means that we can just record one tuple to single destination.

This design saves storage space for embedded device.

4 Forwarding Algorithm

We use *Previous Tuple* in forwarding.

Once the sensor receives a packet, it will judge whether the packet is invalid. It is invalid to one *sensor* if the data comes from *First Hop* of the *sensor*. If *sensor* receives invalid data, it just ignores the data. Otherwise, the packet is valid. So the *sensor* forwards packet to *First Hop* of *Previous Tuple*.

The motivation is that invalid packet exists only when the information routing table is wrong due to the change of position. By ignoring these packets, we can save precious resources for supporting other operations like *Broadcasting* and *Forwarding* valid packet.

5 Routing Algorithm

5.1 Relevant Operation

Three operations are designed in this algorithm.

Reset: First, set *Previous tuple* to be the *updated tuple*. Second, set the *Current Tuple* to initial value,namely *Distance* is infinite, *Next* is 0.

Update: Once the *sensor* receives routing tuples from adjacent nodes, it updates its *Current Tuple*. The updating rule is to select the shortest path to *sink* and update both *First Hop* and *Length*.

Broadcast: Broadcast its *Previous Tuple*.

5.2 Description of Algorithm

Rule-1:For every fixed time(1 second in this protocol), every node, both *sink* and *sensor*, does *Broadcast*.

Rule-2:Every *sensor* does *Reset* after every two *Broadcast*,i.e. *sensor* does *Reset* every 2 seconds.

Rule-3:Once a *sensor* receives a routing tuple, does *Update*.

(The motivation and principle is showed in the proof and next section.)

5.3 Proof Of Routing Algorithm

Claim: Once the position is stable, if every node in graph has done at least $k+1$ *Reset*s, then all the *sensors*, which are k hops away to the sink, will set the routing table, right, i.e. set *Fisrt Hop* ans *Length* in the *Previous Tuple* right. All the *sensor* that aren't reachable to *sink* within k hops has *Length* at least $k+1$. And these two properties still holds in the rest of algorithm.

The Claim is saying that once the position among nodes is relatively stable, then the routing table will eventually be right.

Proof of Lemma: Clearly it's true for $k = 0$.

Assume it's true for $k-1$, then for *sensor* that is exactly k hops away from *sink*, after one more *Reset*, it will set the right routing table, as all *sensor* with less than or equal to $k-1$ distance are correct. For *sensor* that is out of k hops, the distance will be larger than k.

6 Problem & Solution & Advantages of our protocol

6.1 *Update* happens between *Reset* and *Broadcast*

If *Update* happens between *Reset* and *Broadcast*, the *sensor* can only broadcasts *lengh = Infinity*, and this *Update* is actually missed. The solution is to add *atomic* around *Broadcast* and *Reset*. In this way, we can make sure that the updated routing tuple does broadcast to other nodes.

6.2 *Sensor* might not receive all adjacent routing tuples

To guarantee the robustness of our protocol, we must make sure that the *sensors* receives all routing tuples from neighbors. If *sensor* does *Reset* and *Broadcast* every 1 seconds, we might lose some tuple in one second and receive duplicate tuples in another second. The solution is to do *Reset* every 2 seconds. In this way, we can *Update* all neighbors' tuples and *Broadcast* before *Reset*, which guarantees the robustness.

6.3 Count to Infinity

We may ignore infinitive count in this protocol. Because *sensor* broadcasts routing information in a fixed time.

In existing algorithm, a *sensor* does *broadcast* every time its routing information changes. If counting to infinity happens, the involved two *sensors* will successively *broadcast*, which badly influence the performance other operations like forwarding on these two *sensors*.

However, in our protocol, it will not influence the frequency of *Broadcast* as we do it every 1 seconds. What's more, since *Infinity = number of nodes*, the sensors won't count much.