Rate-based Synchronous Diffusion

Internet of Things

Balz Aschwanden, David Boesiger, Jovana Micic, Raoul Norman Grossenbacher

University of Bern
{balz.aschwanden, david.boesiger, jovana.micic,
raoul.grossenbacher}@students.unibe.ch

1 Protocol Introduction

Very often time synchronization of all sensors is required in Wireless Sensor Network (WSN). Since each node has its own clock, it is needed to synchronize clocks in order to support synchronized sleep and duty cycles among nodes.

Rate-Based Diffusion Protocol (RDP) aims to synchronize the nodes in the network to the average value of the clocks in the network. Rate-Based Diffusion Protocol has two main phases:

1. Neighbourhood Discovery Phase

   In this phase, each node has to periodically broadcast a packet with its ID and sequence number to get to know neighbours. All recognized neighbours are saved in neighbour table. Additionally, with each neighbour we have to save the time offset between the node's time and the neighbours times. Broadcast is determined by time the node waits after starting broadcasting. This parameter value will be discussed later in further sections.

2. Convergence Phase

   In convergence phase, each node periodically go through neighbours table and update own time using following formula:

   $$t_i = t_i - r * (t_i - t_j)$$

   Basic idea is to adapt time of the node to the neighbours node time using some r-value. R-value needs to be $0 < r < 1$. Results of choosing different r-values will be discused in further sections. In this phase, unicast messages are used to determine the offset between the clocks.

Algorithm 1 is showing the pseudo code for Rate-based Diffusion Protocol.

Algorithm 1 Diffusion algorithm to synchronize the whole network

```
1: Do the following with some given frequency
2: for each sensor n_i in the network do
3:     Exchange clock times with n_i's neighbours
4:     for each neighbour n_j do
5:         Let the time difference between n_i and n_j be t_i - t_j
6:         Change n_i's time to t_i-r_ij(t_i-t_j)
7:     end for
8: end for
```

2 Methods

In the following part we will show implemented code for receiving and sending unicast messages. The algorithm is very similar to RTT synchronization. A unicast message in our code contains the following variables:

- a boolean to know if the message has passed one round
- the ids of sender and receiver
- the clock time values of both nodes

First the clock time from the sender is inserted and the message is sent to a neighbour. That neighbour then inserts its time change the boolean and sends the packet back. The code is equivalent to the above expect the following lines. The send_uc method is used for both:

```
static void send_uc(uint8_t receiverId, clock_time_t receiverTime,
    uint8_t isRequestForTime) {
  rimeaddr_t addr;
  static struct unicastMessage ucReply;
  ucReply.senderId = node_id;
  ucReply.senderTime = clock_time();
  ucReply.receiverTime = receiverTime;
  ucReply.isRequestForTime = isRequestForTime;

  addr.u8[0] = receiverId;
  addr.u8[1] = 0;
  packetbuf_copyfrom(&ucReply, sizeof(ucReply));
  unicast_send(&ucConn, &addr);
}
```

Then the offset the that neighbour is calculated and saved into the neighbour table. To calculate this offset we simply add half of the RTT to the neighbour time we receive back and compare it to our own time.

```
static clock_time_t calc_offset(clock_time_t senderTime, clock_time_t
    receiverTimeOld)
{
    clock_time_t curr = clock_time();
    clock_time_t rtt = curr - receiverTimeOld;
    clock_time_t neighborTime = senderTime + rtt/2;
    clock_time_t offset = curr - neighborTime;
    return offset;
}
```

After a certain waiting for a certain time to make sure to receive all unicasts back, the sum off all offsets is calculated, multiplicated with the r-value and substracted from the clock time. This way the clock time needs to be adjusted only once per round.

```
static void converge(int numIter)
{
   clock_time_t offset = 0;
   static int i;
   for (i = 0; i < neighborArrayOccupied; i++) {
      offset += neighborTable[i].offset;
      neighborTable[i].offset = 0;
   }

   clock_time_t newtime = clock_time() - (int) offset * rMultiplier /
       rDevider;
   clock_set(newtime);
}
```

3 Experimental setup

There were two phases of experiment. In the first phase we tested our program using Telos nodes and in the second phase we uploaded our code to TARWIS platform. We tried different values for*r-value* and r*unicast interval*. In addition, we run the code with different MAC protocols. In the first version of the program we used default NullMAC protocol and in the second we used X-MAC protocol.

By protocol algorithm, *r-value* needs to be value from range of zero to one. We chose to test our code for five different r-values: 0.25, 0.5, and 0.75.

For *unicast interval* value we chose values from 5 to 10 seconds.

4 Results and Analysis

First an Time Interval of 10 Seconds was chosen for both after the broadcast message and another after the unicast messages. Tests showed that this worked good so the time interval wasn't changed on later tests. There was one test with

a lower Time Interval though the result wasn't satisfying. The results showed that and *r-value* of 0.5 is optimal though there were 3 nodes not responding to the algorithm when testing with 40 nodes.

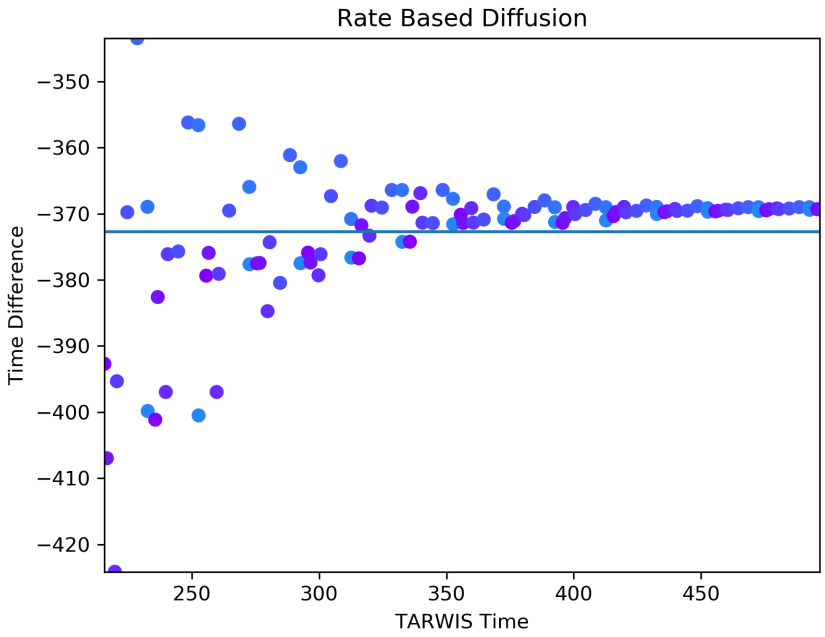

Fig. 1. Test with $r = 0.5$,both intervals 10s, with 10 nodes

The time interval was kept during all tests. For r-value smaller than 0.5 the conversion still was happening, but it was happening at a slower rate. If the value was over 0.5 there was no visible conversion in the charts. When testing with XMAC, no node was responding to one another. This was probably because the nodes weren't able to build their neighbour table.

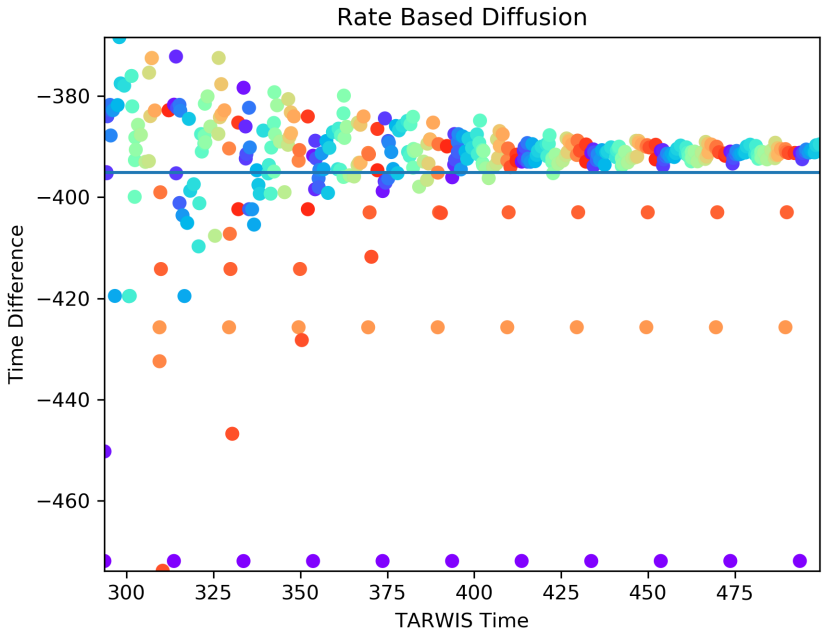

Fig. 2. Test with $r = 0.5$,both intervals 10s, with 40 nodes

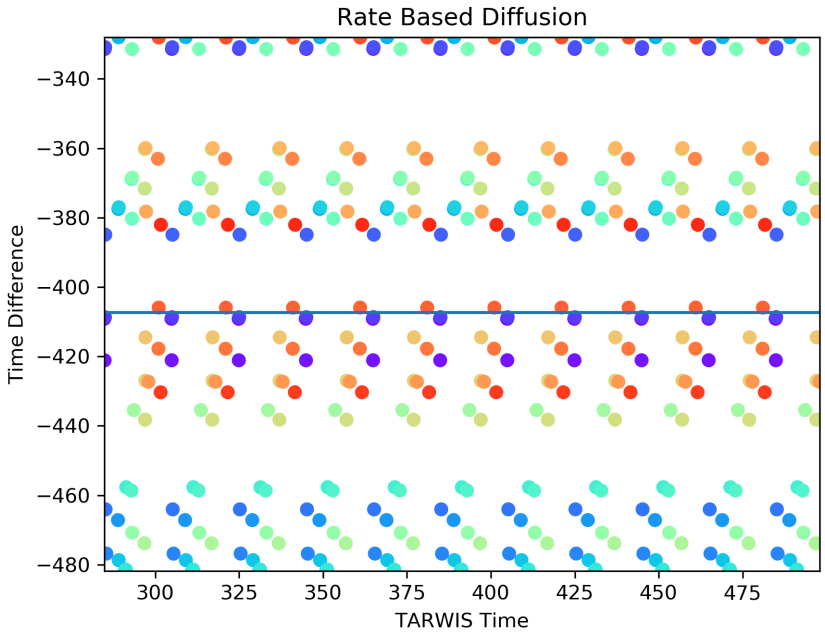

Fig. 3. Test with Xmac

5 Conclusions

It is difficult to find out a good interval for both broadcast and unicast messages. Since the time a packet has to travel between nodes can be affected by multiple sources. The big problem is that those sources e.g. the environment can change every loop, so the interval should be changed as well in order to get satisfying results.

5.1 Improvements

Even though the synchronization worked well, there are still a lot of improvements that could be made. The clock time from a clock on the TelosB nodes only has a ceration range, which means that overflows will happen at some point. Our code does not take this into consideration. Also the calculation for the offset is done with integers, so rounding errors could occur. If there are a lot of unicast messages sent at one point, packets may be lost resulting in a slower performance of the algorithm.