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:

```c
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*, the *unicast interval* as well as the *broadcast 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

The final 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.

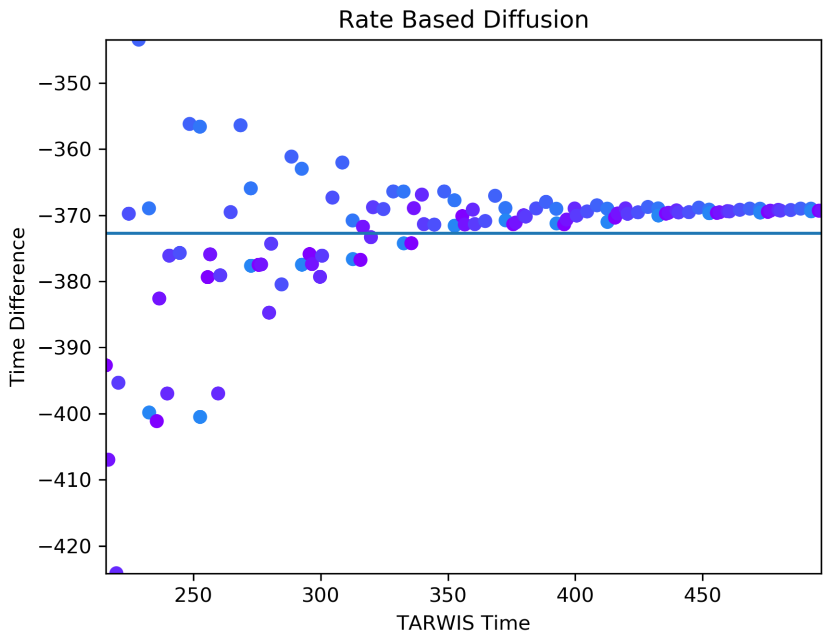

Fig. 1. Test with $r = 0.5$,both intervals 10s, with 10 nodes

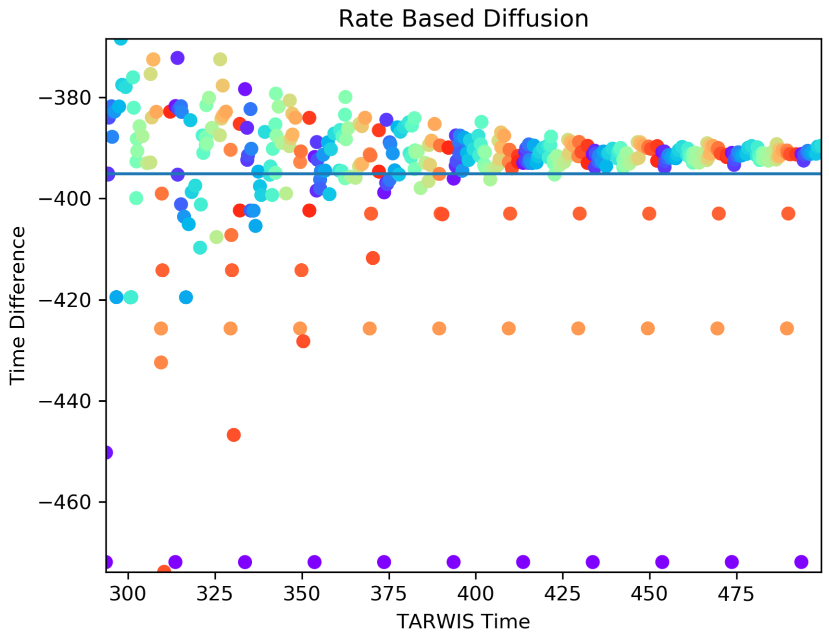

Fig. 2. Test with $r = 0.5$,both intervals 10s, with 40 nodes

In these charts the time difference is the difference between the node time and the TARWIS time. The TARWIS time thus represents some global reference. You can also see a line which is the mean over all measurements.

At first a time interval of 10 Seconds was chosen for both intervals. Later on there were tests with lower time intervals though the result weren't satisfying, so the time intervals were kept during all tests. For r-values smaller than 0.5 the conversion still was happening, but at a slower rate. If the r-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, or they did not pick up the unicasts from their neighbours, thus they were not able to send them back and the offsets would stay to 0 for each neighbour, so the clock won't change.

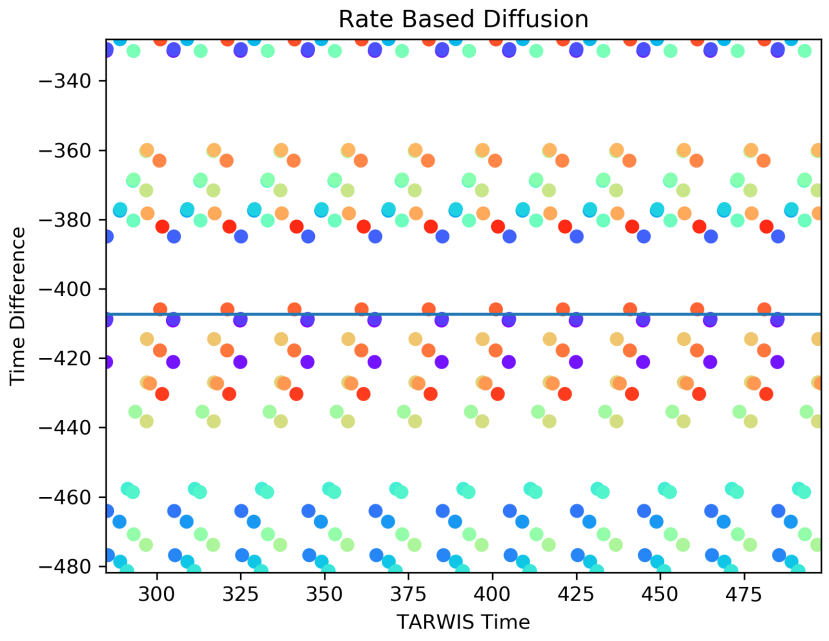

Fig. 3. Test with Xmac

It makes intuitively sense that an r-value of 0.5 would show the best results which was confirmed by our test.

5 Conclusions

There are a lot of challenges when implementing this algorithm. First of all it is difficult to find good times for the both intervals since conditions like environment can change every time, theses intervals ideally need to be different every time. Second there were also technical challenges. The clock time for example only has a certain range and switches from its highest value to its lowest value when overflowing and vice versa. When working with such nodes it is essential to know those things in order to succeed. In light of that the results were more or less as expected.

5.1 Improvements

Even though the synchronization worked well, there are still a lot of improvements that could be made. For example the clock time is not taken into consideration in the code. Further 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.