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 discussed 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-$r_i j(t_i$-$t_j)$
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.

```
00 rimeaddr_t addr;
01 static struct unicastMessage ucReply;
02 ucReply.senderId = node_id;
03 ucReply.receiverId = neighborTable[i].id;;
04 ucReply.senderTime = clock_time();
05 ucReply.receiverTime = 0;
06 ucReply.isRequestForTime = 1;
07
08 addr.u8[0] = ucReply.receiverId;
09 addr.u8[1] = 0;
10 packetbuf_copyfrom(&ucReply, sizeof(ucReply));
11 unicast_send(&ucConn, &addr);
```

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:

```
03 ucReply.receiverId = ucMessageReceived.senderId;
05 ucReply.receiverTime = ucMessageReceived.senderTime;
06 ucReply.isRequestForTime = 0;
```

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.

```
clock_time_t offset = 0;
int offset_count = 0;
for(i = 0; i < neighborArrayOccupied; i++)
{
    clock_time_t offset_tmp = neighborTable[i].offset;
    neighborTable[i].offset = 0;
    if(offset_tmp > 0)
    {
        offset += offset_tmp;
        offset_count++;
    }
}

if(offset_count > 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: 0.1, 0.25, 0.5, 0.75 and 0.9. For *unicast interval* value we chose values from 1 to 5 seconds.

4   Results and Analysis

5   Conclusions

References

[RE1] Author: Article/Book: Other info: (date) page numbers.