Time Synchronization II

Rate-Based Synchronous Diffusion

Pascal Gadient
Domenico Iapello
Felix Langenegger

University of Bern
Communication and Distributed Systems

1 Protocol Introduction

A sensor network is basically made of some sensor nodes that are linked together. To fulfil the main goals of the sensor network the time synchronisation of the network nodes is important. For example, it is useful to have the same time frames on each node to compare the measurements of each node. Or if the network is a mobile sensor network, the exact localisation of each node could be demanded, which can only be provided, if the nodes have a synchronised time. Additionally, the coordination of duty cycles can only be performed if the nodes are in sync.
The requirements of an accurate time synchronisation algorithm are amongst others, precision, energy efficiency, a small memory footprint, scalability and robustness. Most of all, the time synchronisation is not the main application of the node itself, so the algorithm should not consume most of the energy and processing resources. The same goes for the memory. The time synchronisation algorithm should be scalable in two dimensions. Firstly, it has to be scalable to many different resource-constrained devices, that means it should run on nodes with very small resources as well as on devices with high-end circuitry. Secondly, it should be scalable from small to large sensor networks. In other words, the algorithm should run as robust as possible independently of how many nodes in the network are added at some point.
This project report explains how we solved the problem of time synchronisation based on the Rate Based Synchronous Diffusion Algorithm (RBSDA)[1]. The basic idea of the RBSDA is to know all the visible nodes of one specific node and then to iterate over the known neighbours to determine the time offset via round-trip synchronisation.

2 Methods

First, we will speak in detail about the synchronisation algorithm itself and discuss how it works. Later, we pin-point some programming specialities of the used sensor nodes and finally, we present some debugging habits and methods used to ensure a fully functional implementation.

[1] See lecture slide: V. Time Synchronization - 3.3.1 Diffusion-based Synchronization p. 29-30. This slides are based on the paper: Global Clock Synchronization in Sensor Networks by Qun Li and Daniela Rus[1]

The synchronisation algorithm progresses in three steps: Initially, the sensor node generates an internal representation of the surrounding neighbour nodes. Afterwards, it sends round-trip-time (RTT) request packets to all discovered nodes and stores then each node's received RTT together with the remote timestamp into the previously created neighbour node table representation. Finally, we calculate the median of all skews with respect to the RTTs and adjust therefore the local node time with these results multiplied by a constant factor alpha. We repeat the steps starting with the RTT request as long as we try to stay in sync with the other nodes (in general this is equivalent to the node's lifetime). This process is described in pseudo-code at listing 1.1.

```
// Do the following with some given frequency
  for each sensor n_i in the network do
    Exchange clock times with n_i s neighbours
      for each neighbour n_j do
        Let the time difference between n_i and n_j be t_i - t_j
          Change n_i s time to t_i - r_ij (t_i - t_j)
```

Listing 1.1: Diffusion algorithm (DA) used to synchronise the whole sensor network

On a high-level perspective this converges, because each node adjusts itself towards the time of its neighbours. Hence, the more neighbours are seeing each other, the faster will the synchronization take place. However, if there are some partition-like structures in the network it will still converge, but we will encounter disproportionately long synchronisation sequences. This is a result of a few nodes that form a bridge between two different time clusters, so all time adjustments between both clusters will involve these few nodes. Thus, the impact on the whole network is proportionally small, caused by the median calculations (a big skew divided by the amount of (many) nodes results in a small correction value). Last, but not least, the alpha value only guarantees a convergence while being roughly in the range (0, 10]. Hence, with an alpha value of 10, the synchronization accuracy would be at most 10 seconds, further it would take significantly more time to converge than with smaller values (as explained in section 4). The same goes for an alpha value set to zero. With this constraint, no adjustments would take place at all (skew adjustment would always be zero in this scenario).

A side mark for the RTT calculations: We consider only half the RTT, because the time set by the remote node travels only from the remote node to the corresponding node, i. e. half of the RTT time is used.

Now, we will have a look at some code that handles the skew adjustment in the DA-algorithm in listing 1.2.

```
static void updateMyTime() {
  totalClockSkew = 0;
  activeNeighbourCount = 0;
  for (i5 = 0; i5 < MAX_NEIGHBOURS; i5++) {
    if (neighbours[i5].activeState) {
      activeNeighbourCount += 1;
      totalClockSkew += neighbours[i5].skew / ALPHA;
      neighbours[i5].skew = 0;
      neighbours[i5].rtt = 0;
      neighbours[i5].activeState = 0;
```

```
11      }
12    }
13
14    if (activeNeighbourCount > 0) {
15      totalClockSkew = totalClockSkew / activeNeighbourCount;
16      clockCorrection = clock_time() + totalClockSkew;
17      clock_set(clockCorrection, clockCorrection);
18    }
19  }
```

Listing 1.2: Skew Adjustment

In the listing above we can easily see several specialities that occur in general on embedded devices such as our used sensors: i) Floating-point unit does not exist, hence, we have to use divisions instead of floating-point multiplications as in line 7. ii) Most hardware-specific system calls are not that well documented, we assume this is an outcome of embedded CPU designers, that set the focus on the chip, rather than the diverse operating systems that drive the chip, shown in line 17. iii) C code is used in general, as a result of the small memory and CPU footprint requirement.

While contributing to this project, we were in need of many different debugging strategies, because C programming can be very painful, especially for team members that aren't that experienced with low-level programming. So we developed strategies that can be roughly assigned to one of these two classes: Syntax-Supporting Strategies (SSS), as well as Logic-Supporting Strategies (LSS). A prime example of the SSS is the separately maintained "Playground.c" file in which we performed sandbox runs of some complicated code blocks, that upon successful validation eventually got in the main code base. In contrast the LSS consist of mainly three approaches: i) Simplify the code as much as possible (i.e. using structs for message exchanges and internal data representations and trying to reduce code complexity in general). ii) Analysis and verification of the generated logs. iii) Creation of decision-trees for sophisticated code blocks.
Booting up a node results in loading two network protocol stacks, technically speaking, the unicast and the broadcast message handlers. Whilst the broadcast module disseminates the node's id along the neighbour nodes within the main program loop, the broadcast receiver block `recv_bc(...)` recognises these messages and stores the corresponding neighbour in the node's neighbour table with `addNeighbour(unsigned short id)`. Further in the main loop the code requests RTTs via unicast messages from all neighbours stored in the neighbour table with `requestRTT(unsigned short id)`. As soon as they get replied, the `recv_uc(...)` callback method extends the neighbour list with the received data via `saveRTT(unsigned short id, unsigned short rtt, signed long skew)`. After polling each neighbour, the own time gets adjusted by `updateMyTime()`. Accordingly, the saved RTT and skew data gets deleted and the process repeats again within the main program loop. Some more casual methods can be found in the code as well. For example there exist some led enabler and disabler methods with the corresponding callbacks (`enableLED(int color), disableLED(int color)`), as well as print methods for additional debugging like `printNeighborList()`, or the `calcTimeDifference(unsigned short actualTimeOwn, unsigned short rttCorrTimeOther)` method used to calculate the time difference based on the time of two nodes with respect to tick overflows.

3 Experimental Setup & Measurement Procedure

For the lecture 'Sensor Network and Internet of Things' each member got two sensor nodes called 'Telosb' which were assembled by Crossbow[2]. The nodes run an operating system called Contiki.
The assignment for the project was divided into two iterations. The first phase was about to implement a prototype of the algorithm, which we planned to run on four of our nodes. In the second phase we had to adapt the software, so that it can run on the TARWIS network with up to 40 nodes.

In the beginning we inspected carefully each print statement in the console, but as further we came, the less feasible it got. Hence, we began to write parsers for different purposes which can in combination with some gnuplot scripts nicely visualise the outcome of the synchronisation process. The endproducts of these parsers and scripts are shown as figures in the next few sections.

4 Results and Analysis

In this section we present some gathered results from our experiments, as well as some additional explanations and interpretations. The experiments consist of as less as four nodes in local execution and scale between 10 up to 40 nodes for the testbed runs. We run the experiment locally and on TARWIS with these three different configurations: i) Normal run, ii) Fast run, iii) Fast run with some node resets after 55 minutes.
We performed additional measurements with different alpha values apart from the previously mentioned three setups.
Important side mark for TARWIS results: It lies in the nature of a heavily used, commonly shared testbed at a remote location that the results are not every time perfectly deterministic. This may be caused by accidentally performed system maintenance on-site, other users that are requesting much capacity at that time, or workload that uses some shared resources on the specific cluster. Therefore, we had to use larger timeouts and intervals on TARWIS as with the local setup, to obtain valid results.

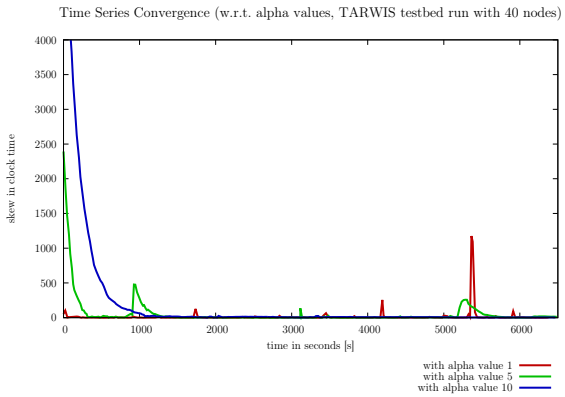

Fig. 1: Time series convergence with respect to alpha values

[2] The datasheet for the node can be found on: `http://www.willow.co.uk/TelosB_Datasheet.pdf`

In figure 1 we can clearly see that the algorithm is able to reach convergence around 250 seconds with a default alpha value of 5. The small amplitudes in the figure are non-deterministic interferences within the sensor network. When we now enlarge the alpha value by a factor of 2, the algorithm performs worse, because the differences are only taken into account at small steps. Hence, the convergence phase takes almost four times as long.

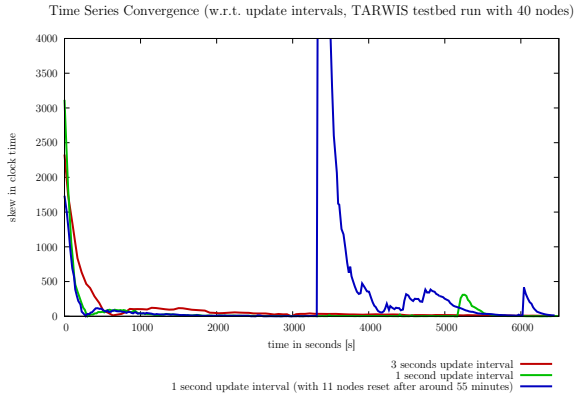

Fig. 2: Time series convergence with respect to update intervals

On the other hand when we decrease the time synchronisation update interval, we can see that the assimilation of the nodes responds in an almost linear way to these changes, i. e. when we set the interval three times as high, the assimilation phase lasts roughly for three times the duration as well. We visualise this in figure 2. In addition, a spike can be identified where some nodes have been reset.

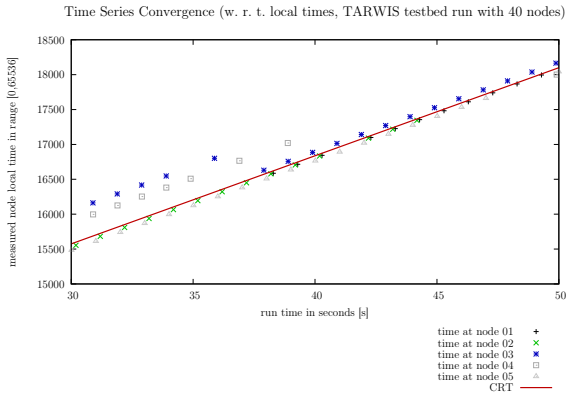

Fig. 3: Time series convergence with respect to some node's local time

Next, we analyse the clock value over time. In figure 3 we see vertically each nodes current clock value at a specific point in time (x-axis). The gaps are representing the random wait of the node, before he starts the next RTT request cycle. The random wait is a result of introducing some randomness, because we want a guarantee that the nodes would not infer each other in a repetitive way. We see that all lines are slowly pointing towards a common center line, we call this the cluster reference time (CRT). This CRT can be determined for each single node with respect to the neighbours and it is perfectly diagonal in any plot per definition. In the end it is our goal to ensure as accurate as possible that the CRT is the same for all nodes. Please consider, that the plots only visualise a small fraction of the synchronisation process inside a specific node (hence, it represents even a smaller field in the whole sensor network).

Nevertheless, we were also interested in the discovery time of the neighbours. We saw in our experiments, that in most cases all neighbours could be identified after 300 seconds. In some specific cases, this interval was much larger for some nodes. These nodes appear to have encountered a crowded channel, or poor signal strength in general.

For all runs we obtained a maximal accuracy around the value of alpha. This can easily be explained: The changes being made to each nodes clock are `(calculated_mean_skew / alpha)` and the device doesn't contain a floating-point unit (FPU), so as an example, all averaged skews smaller than 5 for an alpha set to 5 result in a skew change of 0.

5 Conclusions

We were able to implement the demanded work and the results are quite promising. Nevertheless, the accuracy and the convergence speed depend highly on the used parameters. We identified an alpha value around 1 to be the most robust solution, whereas the update interval should be defined as small as possible to ensure a quick convergence. However, if the interval was set too low, the TARWIS testbed crashed and wasn't able to respond anymore. In our experiments we found an update interval of one second to be the best trade-off between reliability and convergence speed.

References

1. Li, Q., Rus, D.: Global clock synchronization in sensor networks. IEEE Trans. Comput. **55**(2) (February 2006) 214–226