Internet of Things (IoT)

Report of project
“Time synchronization for programmable matter based modular robots”

subject «Modular Robot Time Protocol (MRTP)»

Performed by:
Student:
signature Bogdan Gorelkin
«___» _____2021 y.

signature Hassan Hijazi
«___» _____2021 y.

Checked:
Full Professor
Head of the Master Internet of Things
signature Abdallah Makhoul
«___»_____2021 y.

Montbéliard 2021

Table of contents

The purpose of this work

The main goal of this work is to compare the synchronization speed of modular robots using two algorithms. In the first case, we are getting a graph of the synchronization speed using the algorithm of *Modular Robot Time Protocol (MRTP)* [1]. In the second case, we are trying to improve MRTP algorithm using *Simple Algorithm for Improving Time Synchronization in Wireless Sensor Networks* [2]. Result of this work is to compare those two graphs and based on obtained result make a conclusion which one is better.

1. Introduction

Modular reconfigurable robots form autonomous distributed systems in which modules coordinate their activities in order to achieve common goals. Every module has its own computation and communication capabilities, notion of time, sensors and actuators. Connectivity between modules can be rearranged to adapt the global shape of the robot to the application.

Modular robots work like a colony of ants. Each ant works towards the common goal of the colony. Their work requires coordination and precision. If at least one ant works asynchronously, then the work of the entire colony is threatened. The synchronization algorithm is implementing for modular robots composed of identical modules that communicate together using neighbor to-neighbor communications. Coordination among a group of modules relies on the existence of a common notion of time. The assembled form from the modules must remain synchronized even after a long time of work. If the robots are out of sync, it can lead to a wrong work of the entire "Transformer"

Every module has its own notion of time provided by its own hardware clock *(Figure 1)*. Since common hardware clocks are imperfect, local clocks tend to run at slightly different and variable frequencies, drifting apart from each other over time. Consequently, a distributed time synchronization is necessary to keep each module’s local clock synchronized to a global timescale. The offset of two clocks denotes the time difference between them whereas the skew of two clocks denotes their frequency difference. Network-wide time synchronization protocols aim to keep a small offset between local clocks and a global reference time. In most of the existing protocols, devices exchange time-stamped messages in order to estimate the current global time. Since time keeps going during communications, modules have to correctly compensate for network delays in order to evaluate the current global time upon reception of synchronization

messages. Although it is non-trivial to accurately estimate communication delays, especially in presence of unpredictable delays (due for example, to queueing or retransmissions), it is crucial in order to achieve high precision performance.

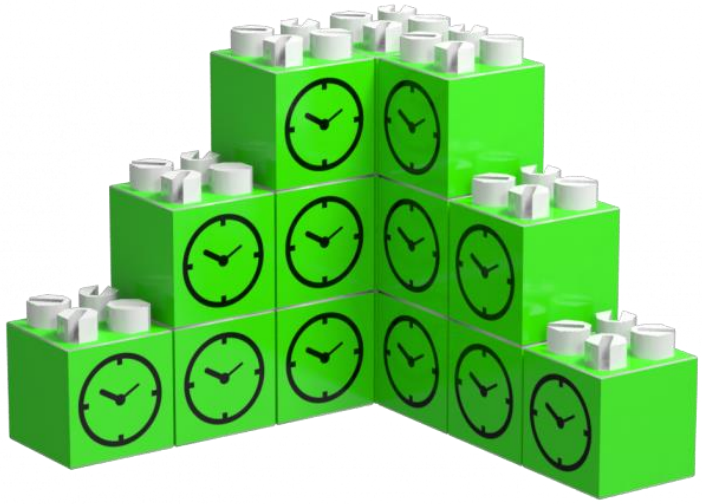


Figure 1 – Each module has a local time

In this work, we are implementing the Modular Robot Time Protocol (MRTP), which in the future we will try to improve and increase its accuracy. The main advantage of the proposed protocol is applicable to work without a global clock signal.

2. Modular Robot Time Protocol

Modular Robot Time Protocol (MRTP) – is the first protocol for modular robots that provides an accurate lowskew global timescale without dedicated hardware [1]. MRTP is a network-wide time synchronization protocol for modular robots. This protocol achieves its performance by combining several mechanisms:

- central time master election, which is like an ant-coordinator of the Paratrechina longicornis colony controls the work of his subordinates and in some case is replaced by another coordinator;
- low-level time-stamping;
- clock skew compensation using linear regression.

The algorithm can be roughly divided into two stages. These stages are necessary for the correct operation of the entire algorithm.

2.1. Algorithm of Modular Robot Time Protocol

During first step the system is initializing and fulfilling:

1. Election of a central module as the time master (TM);
2. Construction of a spanning-tree;
3. Initialization of the global clock.

2.1.1. Time Master Election

An approximate-center module is elected as the time master. The center is the set of all modules that minimize the maximum hop distance to any other module. Placing the time master close to the center of the system reduces the time of the synchronization phases and increases the overall precision because cumulative estimations are made every hop. Any center election algorithm could be used to elect a central module. In [1], the ABC-Center algorithm[3] was used. In our work, we manually set the center, since the main task of our work is not an exact repetition of the work done, but testing the idea of improving the MRTP algorithm.

2.1.2. Breadth-First Spanning Tree Construction

Creating of a breadth-first spanning-tree rooted at the time master guarantees that modules at distance d_{TM} hops of the time master in the physical configuration, are at distance d_{TM} hops in the tree. Logical neighbors in the tree are neighbors in the physical configuration. At this point, every module knows its parent and children in the tree [4].

```
when START() is received do    % only the distinguished process receives this message %
(1)   send GO(−1) to itself.

when GO(d) is received from p_j do
(2)   if (parent_i = ⊥)
(3)      then parent_i ← j; children_i ← ∅; level_i ← d + 1;
(4)           expected_msg_i ← |neighbors_i \ {j}|;
(5)           if (expected_msg_i = 0)
(6)              then send BACK(yes, d + 1) to p_parent_i
(7)              else for each k ∈ neighbors_i \ {j} do send GO(d + 1) to p_k end for
(8)           end if
(9)      else if (level_i > d + 1)
(10)             then parent_i ← j; children_i ← ∅; level_i ← d + 1;
(11)                  expected_msg_i ← |neighbors_i \ {j}|;
(12)                  if (expected_msg_i = 0)
(13)                     then send BACK(yes, level_i) to p_parent_i
(14)                     else for each k ∈ neighbors_i \ {j} do send GO(d + 1) to p_k end for
(15)                  end if
(16)             else send BACK(no, d + 1) to p_j
(17)          end if
(18)  end if.

when BACK(resp, d) is received from p_j do
(19)  if (d = level_i + 1)
(20)     then if (resp = yes) then children_i ← children_i ∪ {j} end if;
(21)          expected_msg_i ← expected_msg_i − 1;
(22)          if (expected_msg_i = 0)
(23)             then if (parent_i ≠ i) then send BACK(yes, level_i) to p_parent_i
(24)                                    else p_i learns that the breadth-first tree is built
(25)                  end if
(26)          end if
(27)  end if.
```

page 18 of distributed algorithms book

2.1.3. Global Clock Initialization

Slave Modules

Initially, slave modules estimate the global time with their local time. Slave modules adjust their estimation of the global time during synchronization phases, in the second step of MRTP. When a new time master is elected, modules keep their previous estimation of the global time but do not keep previous corrections of the clock skew.

Time Master

Since time cannot run backward, clocks in advance of the global timescale have to slow down or to wait during synchronization process, and clocks behind the global timescale have to jump to it. To make time synchronization convergence faster, the global time is initially set to an estimation of the most advanced global time in the system using the convergecast-timemax algorithm. The pseudocode for any module M_i is shown *Figure 2*.

Input: M_p // parent in the tree
$Children$ // set of children in the tree

1 **Initialization** of M_i at time t_{init}:
2 $offset \leftarrow G^{M_i}(t_{init}) - L^{M_i}(t_{init})$;
$Wait \leftarrow Children$;
3 **if** $M_p = \perp$ **then**
4 // convergecast-max-time terminates
5 **else if** $Wait = \emptyset$ **then**
6 send $m = \text{BACK}(\_,\_)$ to M_p;
// M_p will receive BACK($Y^{M_i}(t_s^m) = L^{M_i}(t_s^m) + offset^{M_i}(t_s^m), L^{M_p}(t_r^m)$) at the application layer. $Y^{M_i}(t_s^m)$ is inserted by M_i at the data-link layer, just before transmission start. M_p will insert $L^{M_p}(t_r^m)$ upon reception, at the data-link layer.
7 **end**

8 **When** m $= \text{BACK}(Y^{M_c}(t_s^m), L^{M_i}(t_r^m))$ **is received by** M_i **from** M_c such that $M_c \in Children$ **do**:
9 $Y^{M_c}(t_r^m) \leftarrow Y^{M_c}(t_s^m) + T^m_{transfer}$;
10 $offset \leftarrow max(offset, Y^{M_c}(t_r^m) - L^{M_i}(t_r^m))$;
11 $Wait \leftarrow Wait - \{M_c\}$;
12 **if** $M_p = \perp$ **then**
13 // convergecast-max-time terminates
14 **else if** $Wait = \emptyset$ **then**
15 send $m' = \text{BACK}(\_,\_)$ to M_p;
// As explain in comment line 6, M_p will receive BACK($Y^{M_i}(t_s^{m'}), L^{M_p}(t_r^{m'})$) at the application layer.
16 **end**

Figure 2 – Convergecast-timemax algorithm

2.2. Periodic Synchronization

The second stage is needed to maintain synchronization during the operation of the robot colony:

- the time master (TM) periodically synchronizes the slave modules.

The time master holds the global timescale and periodically initiates synchronization phases. During each synchronization phase, the time master

broadcasts the current global time along the edges of the spanning-tree built in the first step. $\tilde{G}$ (t), an estimation of the global time is disseminated through the spanning-tree, module-bymodule, starting from the time master. At each hop, the transmitted time is updated to take into account communication delays and time of residence in intermediate modules. Slave modules use a linear model to compensate for clock skew.

The time master starts synchronization phases by sending all its children the actual global time. Details of the synchronization process of any slave module M_i is shown in *Figure 3*.

Input: M_p // parent in the tree
$Children$ // set of children in the tree
w // maximum number of synchronization points used for linear regressions

Initialization of M_i:
$a \leftarrow 1.0$; $b \leftarrow 0$; $W \leftarrow \emptyset$;

When m = SYNC($\tilde{G}(t_s^m)$,$L^{M_i}(t_r^m)$) **is received by** M_i **from** its parent M_p **do**:
$\tilde{G}(t_r^m) = \tilde{G}(t_s^m) + T_{transfer}^m$;
if $|W| = w$ **then**
  $W \leftarrow W - \{\underset{\tilde{G}(t)}{\text{argmin}}\ W(<\tilde{G}(t), L(t)>)\}$;
end
$W \leftarrow W \cup <\tilde{G}(t_r^m), L^{M_i}(t_r^m)>$;
computeLinearRegression(a, b, W);
for each $M_c \in Children$ **do**
  send $m' =$ SYNC(_, _) to M_c;
  // M_c will receive SYNC($\tilde{G}(t_s^{m'}), L^{M_c}(t_r^{m'})$) at the application layer. $\tilde{G}(t_s^{m'}) = \tilde{G}(t_r^m) + a^{M_i}(W^{M_i}(t_s^{m'})) * (L^{M_i}(t_s^{m'}) - L^{M_i}(t_r^m))$ is inserted at the data-link layer, just before transmission start. M_c will insert $L^{M_c}(t_r^{m'})$ upon reception, at the data-link layer.
end

Figure 3 – Synchronization protocol for a slave module, M_i

This model is schematically shown in *Figure 4*. At each hop, the transmitted time is updated to take into account communication delays and time

of residence in intermediate modules. Slave modules use a linear model to compensate for clock skew.

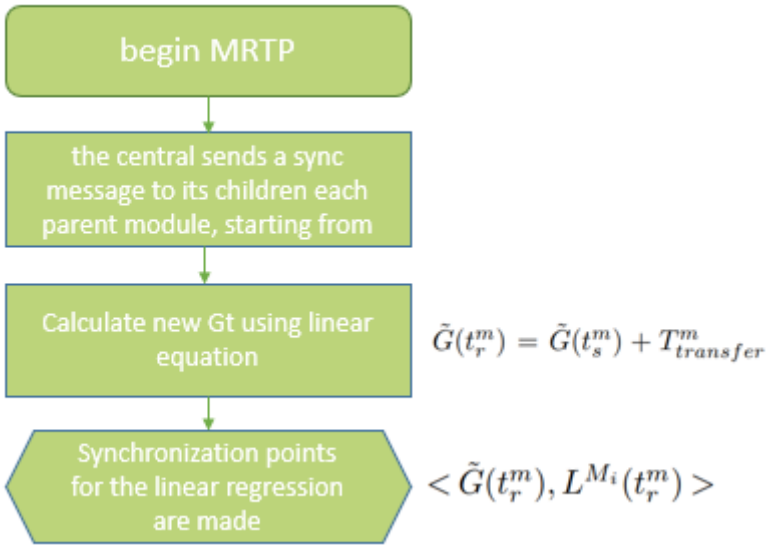


Figure 4 – Schematic of Periodic Synchronization

2.2.1. Time-stamping and Global Time Estimation

The synchronization process uses a single type of message *SYNC*. Every *SYNC* message *msg* is time-stamped twice at the data-link layer: the sender, M_j, inserts $\tilde{G}(t_s^m)$ just before transmission start and the receiver, M_k, inserts $L^{M_k}(t_r^m)$ upon complete reception. When M_i receives a $SYNC(\tilde{G}(t_s^m), L^{M_i}(t_r^m))$ message *m* from its parent, M_i computes $\tilde{G}(t_r^m) = \tilde{G}(t_s^m) + T_{transfer}^m$, an estimation of the global time at the reception of the synchronization message (line 4). $<\tilde{G}(t_r^m), L^{M_i}(t_r^m)>$ forms a synchronization point that contains both M_i's local clock value and the estimation of the global time at nearly the same real-time. M_i can estimate its synchronization error by $G^{M_i}(t_r^m) - \tilde{G}(t_r^m)$.

2.2.2. Global Clock Adjustment

M_i computes $a^{M_i}(W^{M_i}(t))$ and $b^{M_i}(W^{M_i}(t))$such that $\tilde{G}(t) \sim a^{M_i}(W^{M_i}(t)) * L^{M_i}(t) + b^{M_i}(W^{M_i}(t))$ using leastsquares linear regression based on $W^{M_i}(t)$, a window of the last *w* synchronization points (line 9). $a^{M_i}(W^{M_i}(t))$ denotes M_i 's estimated skew relative to the global time, and $b^{M_i}(W^{M_i}(t))$ its estimated offset at time *t*. This mechanism compensates for clock skew and enables modules to be synchronized less frequently without degrading the synchronization precision *(Figure 5)*.

$$\underbrace{\tilde{G}(t)}_{Y} \sim \underbrace{a^{M_i}(W^{M_i}(t))}_{a} * \underbrace{L^{M_i}(t)}_{x} + \underbrace{b^{M_i}(W^{M_i}(t))}_{b}$$

Figure 5 – Clock skew compensation

In order to preserve time monotonicity, our protocol prevents $G^{M_i}(t)$ from running backward: ∀t, ∀t′, $t \geq t'$, $G^{M_i}(t) = max(G^{M_i}(t'), a^{M_i}\left(W^{M_i}(t)\right) * L^{M_i}(t) + b^{M_i}\left(W^{M_i}(t)\right)$. If a new computed model leads to an estimated global time behind the maximum time already reached by $G^{M_i}(t)$, then $G^{M_i}(t)$ is blocked until the new model reaches this maximum time. Otherwise, $G^{M_i}(t)$ jumps into the future.

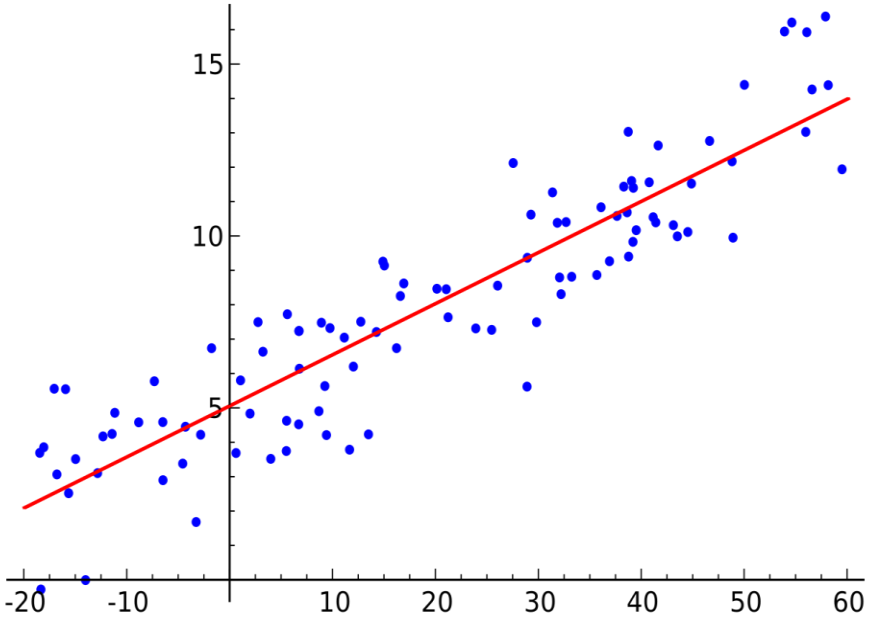


Figure 6 –

2.2.3. Global Time Dissemination

M_i then sends a *SYNC* message m′ to each of its children M_c in the tree (line 11). At the data-link layer, Mi inserts $\tilde{G}\,(t_s^m) = \tilde{G}(t_r^m) + a^{M_i}\left(W^{M_i}\left(t_s^{m'}\right)\right) * \left(L^{M_i}\left(t_s^{m'}\right) - L^{M_i}(t_r^m)\right)$ into m′, just before it starts to transmit the chunk over the communication medium. This compensates for the time of residence at module M_i, assuming M_i clock skew to be constant and equal to $a^{M_i}(W^{M_i}(t_s^{m'}))$ during this time. M_c inserts its local time $L^{M_c}\left(t_r^{m'}\right)$ into the incoming message at the data-link layer, immediately after M_c pulls the synchronization message from the interface buffer. At M_c's application layer, m' contains $\tilde{G}\left(t_s^{m'}\right)$ and $L^{M_c}\left(t_r^{m'}\right)$. M_c then repeats the same synchronization process than M_i.

3. Simple algorithm for improving time synchronisation in wireless sensor networks

Simple Algorithm for Improving Time Synchronization in Wireless Sensor Networks is a simple method for improving time synchronisation suitable for use in ad-hoc sensor networks where processing power is limited. The technique is local and uses Bayesian estimation. A simple simulation shows an improvement of about four times.

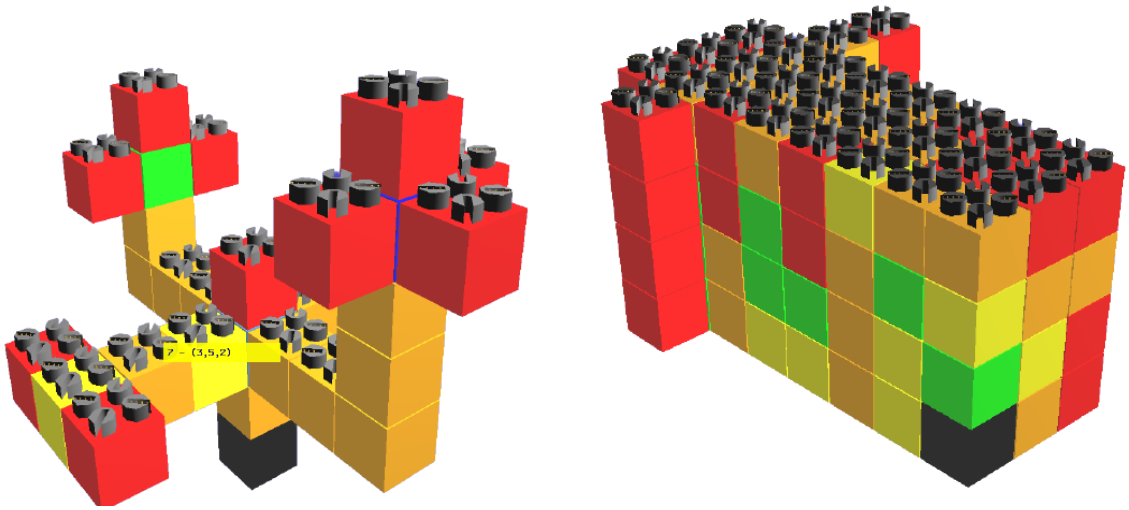
To get an improved estimate $\sim N(T'_i, \sigma'^2_i)$ should be used following equations:

$$\sigma'^2_i = \frac{\sigma'^2_{i-1}\,\sigma^2_i}{\sigma'^2_{i-1}+\sigma^2_i} \qquad (1)$$

$$T'_i = \frac{\sigma^2_i}{\sigma'^2_{i-1}+\sigma^2_i} T'_{i-1} + \frac{\sigma'^2_{i-1}}{\sigma'^2_{i-1}+\sigma^2_i} T_i \qquad (2)$$

4. Implementation of Modular Robot Time Protocol

Because we are working for a modular robots simulator so we were using C++ language to implement those algorithms to run it in VisibleSim. First of all we were working with a shape from *Figure 7*.



7 – a *7 – a*

Figure 7 –config

For the better understanding we colored blocks into different color. Color based on number of children of each blocks in proposed tree.

4.1. Time Master Election

In our work, we manually set the center in the startup function *(Figure)*, since our goal is to compare the synchronization time of two algorithms. First algorithm will follow instruction of MRTP [1], second one will be same, but with additional algorithm using Bayesian's estimation theorem [2].

```
void TimeSyncBlockCode::startup() {
    console << "start";
    // Sample distance coloring algorithm below
    if (module->blockId == 1) { // Master ID is 1
        module->setColor(BLACK);
        module->getLocalTime();
        distance = 0;
        nbWaitedAnwsers = 0;
        for(int i=0; i<module->getNbInterfaces(); i++) {
            if(module->getInterface(i)->isConnected()){
                sendMessage("Go Msg", new MessageOf<int>(GO_MSG_ID, distance),
                    module->getInterface(i), 100, 200);
                nbWaitedAnwsers++;
            }
        }
    } else {
        distance = -1; // Unknown distance
        hostBlock->setColor(LIGHTGREY);
    }
    // Additional initialization and algorithm start below
}
```

Figure 8 – startup function

As you may see on *Figure 7* initial block which is have block id equal to 1 we replaced into BLACK to see on the shape which block is a root of the tree. And then this block start to work as a master of the tree and sends a «Go Msg» to all of its neighbors to build a Breadth-First Spanning Tree.

4.2. Breadth-First Spanning Tree Construction

Hiring blocks starts at the root by sending "Go Msg" to all connected blocks. The implementation code for this part is shown in Figure 9a. When all the neighbors of the block receive the "Go Msg" message, they do the same as the root (parent) and continue sending the "Go Msg" message to all their connected blocks (possible children) except for the sender (the one that has received the Go msg from parent). This continues until the last block (children) is hired. When the block has no children to send the "Go Msg" message, the "Back Msg" message is

sent to the block parent. The implementation code for sending the "Back Msg" message is shown in Figure 9b.

Each parent expects to receive exactly as many "Back Msg" as they sent "Go Msg". As soon as the root has received "Back Msg" the tree is considered to be built.

```
void TimeSyncBlockCode::handleGoMessage(std::shared_ptr<Message> _msg,
                                        P2PNetworkInterface* sender) {
    MessageOf<int>* msg = static_cast<MessageOf<int>*>(_msg.get());
    int d = *msg->getData()+1;

    if (distance == -1 || distance > d) {
        // if already has parent (parent == nullptr)
        // then send back false
        distance = d;
        parent = sender;
        console << "parent: " << parent->getConnectedBlockId() << "\n";
        module->setColor(Colors[distance % NB_COLORS]);
        // Broadcast to all neighbors but ignore sender
        nbWaitedAnwsers = 0;
        for(int i=0; i<module->getNbInterfaces(); i++) {
            if(module->getInterface(i)->isConnected() && module->getInterface(i) != sender){
                sendMessage("Go Msg", new MessageOf<int>(GO_MSG_ID, distance),
                    module->getInterface(i), 100, 200);
                nbWaitedAnwsers++;
            }
        }
        if(nbWaitedAnwsers == 0) {
            // send back true
            if(module->getNbNeighbors()==1){
                local = module->getLocalTime();
                sendMessage("Back Time Msg", new MessageOf<Time>(BTIME_MSG_ID,local),sender, 100, 200);
            }
            vector<int> res;
            res.push_back(1);
            res.push_back(distance);
            sendMessage("Back Msg", new MessageOf<vector<int>>(Back_MSG_ID, res),
                    sender, 100, 200);
        }
    } else {
            // send back false
        vector<int> res;
            res.push_back(0);
            res.push_back(distance);
            sendMessage("Back Msg", new MessageOf<vector<int>>(Back_MSG_ID, res),
                    sender, 100, 200);

    }
    console<<"nbwaited msg= "<<nbWaitedAnwsers<<"\n";
}
```

Figure 9a – Go message

```
void TimeSyncBlockCode::handleBackMessage(std::shared_ptr<Message> _msg,
                                          P2PNetworkInterface* sender) {
    MessageOf<vector<int>>* msg = static_cast<MessageOf<vector<int>>*>(_msg.get());
    console << " received back from =" << sender->getConnectedBlockId()<< "\n";
    vector<int> r = *msg->getData() ;
    nbWaitedAnwsers--;


    if(r[1]==distance+1 || r[1]+1==distance ){
        if(r[0]==1 && r[1]==distance+1){
            children.insert(sender);
        }
        if(nbWaitedAnwsers == 0) {
            if(parent != nullptr){

                sendMessage("Back Msg", new MessageOf<vector<int>>(Back_MSG_ID, {1,distance}),
                        parent, 100, 200);
                local = module->getLocalTime();
                sendMessage("Back Time Msg", new MessageOf<Time>(BTIME_MSG_ID,local),parent, 100, 200
);
            } else {
                for(auto c: children){
                    sendMessage("Test Msg", new Message(TEST_MSG_ID),
                            c, 100, 200);
                }
            }
        }
    }
console<<"nbwaited msg= "<<nbWaitedAnwsers<<"\n";
}
```

Figure 9b – Back message

When the form is nontrivial, that is, there are several possible ways of receiving the message for the block, ambiguity arises. The solution to this problem is achieved by calculating the distance. So the distance aof each block

are sent with the "GO msg" and when received, the received block unpacks the msg and checks if data received in the mssage is equal to its distance plus one, if it is then he will respond back with a YES that means that it is now his children , or with NO it means that it is not. the YES answers are presented by 1 in our code and in case of NO it will not respond back.

4.3. Global Clock Initialization

When the last block replies to its parent, in addition to the YES message, it also reports its local time *(Figure 10)*. As soon as the parent has received this data from all his children, he passes this message further towards the root. The parent does not forward the message until all messages from his children have been received.

When the root has received messages from all children of the tree, it automatically chooses the maximum local time among all its children and sets it as the main one, so that the block with the maximum time waits for the lagging ones. This is due to the fact that we cannot teleport to the past.

```
void TimeSyncBlockCode::handleBtimeMessage(std::shared_ptr<Message> _msg,
                                           P2PNetworkInterface* sender) {
            MessageOf<Time>* msg = static_cast<MessageOf<Time>*>(_msg.get());
            Time t = *msg->getData();
           // t[0]=t[0]+ 0.1; //d0 or dt??
            local = module->getLocalTime();
            console<<"my local time="<< module->getLocalTime() <<"\n";
                if(module->blockId ==1 && nbWaitedAnwsers==0){

                    console<<children.size();
                    console<<"\n"<<local<<"\n";

                    for(auto c: children){
                        sendMessage("Sync Msg", new MessageOf<Time>(SYNC_MSG_ID,local),
                        c, 100, 200);
                    }

                    getScheduler()->schedule( new InterruptionEvent(getScheduler()->now() + 5000000,this->module, 1010));
                    //periodical call function in schedule 5 sec
                }
        }
```

Figure 10 – handleBtimeMessage function

4.4. Periodic Synchronization

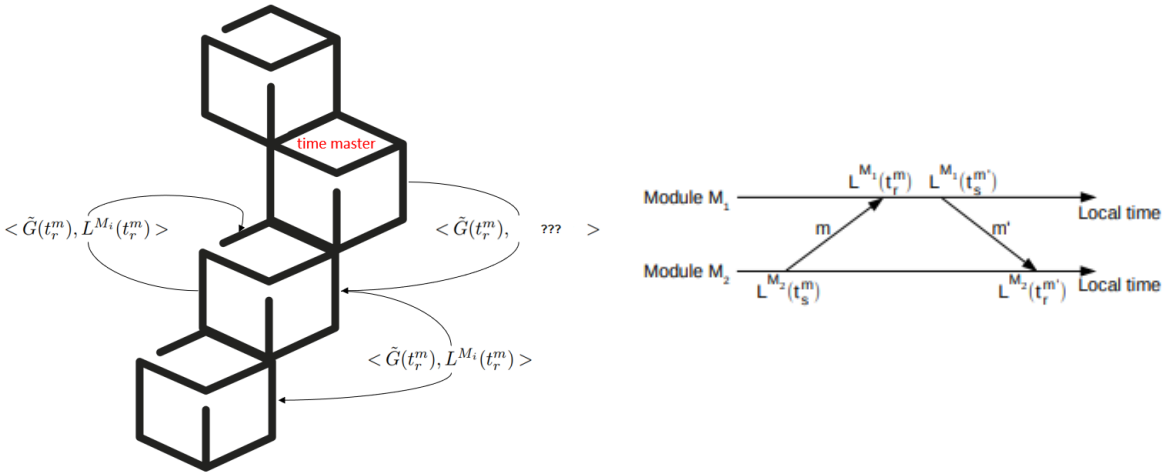
At the moment when the root is familiar with all its children (NbWaitedAnswers == 0), the first synchronization message is sent, which is explained in Figure 10. After that, synchronization is repeated every 5 seconds (5 000 000 microseconds) by calling the "processLocalEvent" function as shown in *Figure 11*.

```
BlinkyBlocksBlockCode::processLocalEvent(pev);

    switch (pev->eventType) {

            /// .... ALL THE OTHER EVENT STUFF

        case EVENT_INTERRUPTION: {
            std::shared_ptr<InterruptionEvent> itev =
                std::static_pointer_cast<InterruptionEvent>(pev);

            switch(itev->mode) {
                case 1010: {
                        minTime = module->getLocalTime();

                        int nbModules = BaseSimulator::getWorld()->getNbBlocks();
                        for(int i=1 ;i<=nbModules ; i++){
                        TimeSyncBlockCode *x = (TimeSyncBlockCode*) BaseSimulator::getWorld()->getBlockById(i)->blockCode;
                        if(x->getGlobalTime()>maxTime){
                            maxTime=x->getGlobalTime();
                            }
                        }
                        //cout<<"MINT IS    =   "<<minTime<<"   and MAXT is =  "<<maxTime<<endl;
                       // if(maxTime>5000000){
                        cout<<minTime<<";"<<maxTime<<endl;
                       // }
                        for(auto c: children){
                        sendMessage("Sync Msg", new MessageOf<Time>(SYNC_MSG_ID,this->module->getLocalTime()),
                        c, 100, 200);
                    }
                            std::shared_ptr<Message> message;
stringstream info;

    // Do not remove line below
    BlinkyBlocksBlockCode::processLocalEvent(pev);
                        getScheduler()->schedule( new InterruptionEvent(getScheduler()->now() + 5000000,this->module, 1010));
                }
            }
        }
    }
}
```

Figure 11 – Periodic Synchronization

4.5. Time-stamping and Global Time Estimation

We are expecting to get the same time for all blocks to make them synchronized. For this we used Linear regression. Linear regression is used to predict the delay during transmitting the global time to each block. This is shown schematically in the *Figure 12a*, the implementation can be seen in the *Figure 12b* and the implementation in *Figure 13a* and *Figure 13b*, respectively. We are computing the linear regression based on sync points contained in the list «W» that can contains 5 points at max and each point is in a form of a struct that contains two variables: its local time and the global time received by the parent. And then it sends the result of the function downstream to its children as shown in *Figure 14*.



12 – a *12 – b*

Figure 12 –

```
void  TimeSyncBlockCode::Computelinearregression(){
    float x_mean=0,y_mean=0;
    float SS_xx=0,SS_xy=0;
    console<<"W: ";

    for(auto sd: W){
        x_mean+=sd.ltime;
        y_mean+=sd.gtime;
        console << sd.ltime << "   " <<sd.gtime <<"\n";
    }
    x_mean/= W.size();
    y_mean/=W.size();
    for(auto sd: W){
        SS_xy+=((float)sd.ltime-x_mean)*((float)sd.gtime-y_mean);
        SS_xx+=pow((float)sd.ltime-x_mean,2);
    }
    a=SS_xy/SS_xx;
    console<<"ss_xx= "<< SS_xx <<"\n";
    b=y_mean-a * x_mean;
    console<<"a = "<<a<<"\n";
    console<<"b = "<<b<<"\n";
}
```

Figure 13a –

```
void TimeSyncBlockCode::handleSyncMessage(std::shared_ptr<Message> _msg,
                                          P2PNetworkInterface* sender) {
        MessageOf<Time>* msg = static_cast<MessageOf<Time>*>(_msg.get());
        console<<"receieved sync from: "<<sender->getConnectedBlockId()<<"\n";
        console<<"recieved is = "<<*msg->getData()<<"\n"<<"and the local= "<<module->getLocalTime()<<"\n";
        sds sd;
        sd.gtime=*msg->getData();
        sd.ltime=module->getLocalTime();
        if(W.size()>4){
            W.erase(W.begin());
        }
        W.push_back(sd);
        if(W.size() > 1) {
            Computelinearregression();
        }
        for(auto c: children){
            sendMessage("Sync Msg", new MessageOf<Time>(SYNC_MSG_ID, getGlobalTime()),
            c, 100, 200);
            }
    }
```

Figure 13b –

Synchronization occurs every 5 seconds. This means that every 5 seconds the maximum time is calculated from all tree to re-set the total time for each block, starting with the time master.

5. Implementation of Simple Algorithm for Improving Time Synchronization in Wireless Sensor Networks

1. Let $T'_1 = T1$ and $\sigma'1 = \sigma1$ where T1 is the time of first node and σ1 is its clock resolution.
2. An upstream node passes its estimated time $T'i-1$ and the standard deviation $\sigma'i-1$ of estimate error to a downstream node.
3. A downstream node measures the local time Ti , extracts the standard deviation σ i of its measurement error from the measurement record and calculates an improved time estimation value T 'i and standard deviation σ 'i from $T'i-1$, $\sigma'i-1$, Ti , σ i by equation (1) and (2).
4. This continues down the chain until the last hop is reached. We should notice that our time synchronization improvement approach is not independent and must be based on existing time synchronization algorithms, which give the original time measurement Ti and set up the measurement record. The synchronization algorithm can be chosen from those discussed before but is not limited to them.

```
Time TimeSyncBlockCode::getTprime() {
        T=getGlobalTime();
        Delta=(Tparent-T);
       // cout<<"loook hon ::  "<<T<<endl;
        Deltaprime=(pow(Deltaparent,2)*pow(Delta,2))/(pow(Deltaparent,2)+pow(Delta,2));
        Deltaprime=sqrt(Deltaprime);
        Tprime=((pow(Delta,2)*Tparent)/(pow(Deltaparent,2)+pow(Delta,2)))+((pow(Deltaparent,2)*T)/(pow(Deltaparent,2)+pow(Delta,2)));
        return round(Tprime);
}
```

Figure 14 – improving algorithm

6. Analysis of the obtained results

As a result of the implementation of this work, it was possible to obtain data on the speed of the time synchronization of the modules. Figures 15a and 15b show the graphs of the synchronization speed for the shape from *Figure 7–a* of the MRTP algorithm and the improved MRTP, respectively.

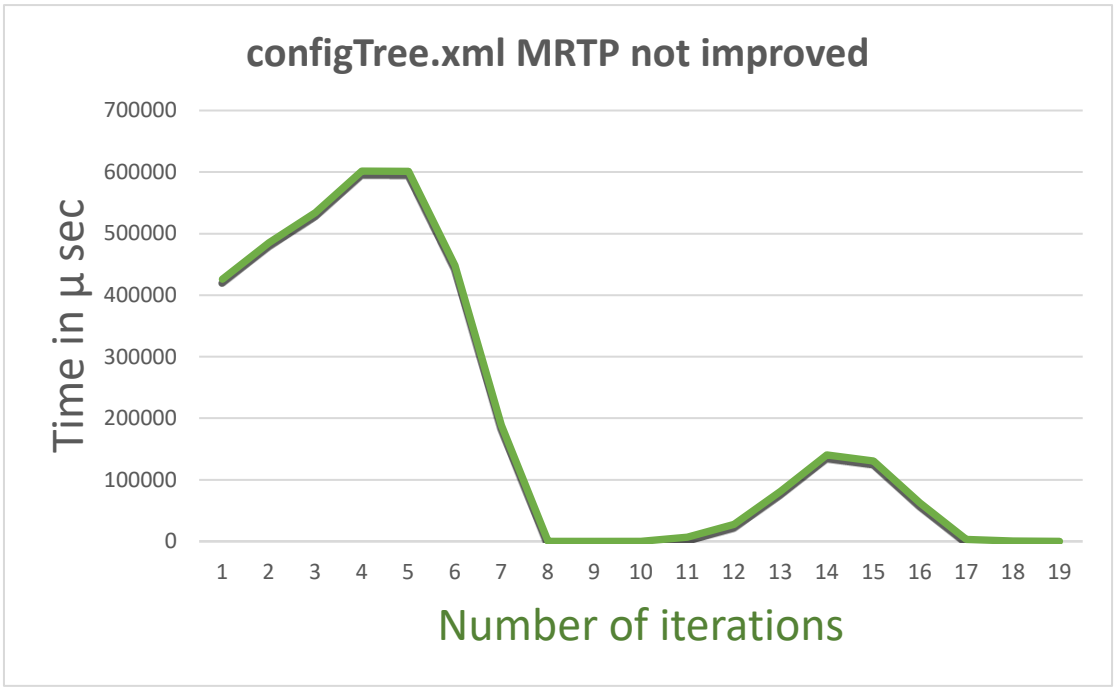


Figure 15a – configTree.xml MRTP not improved

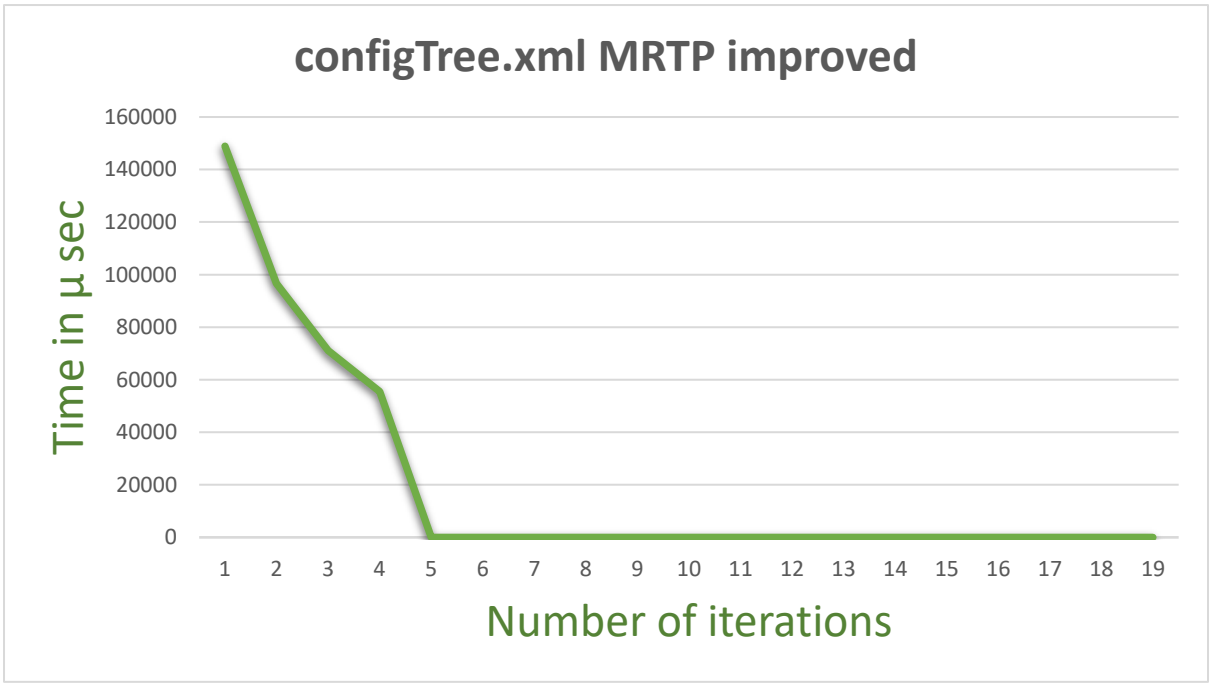


Figure 15b – configTree.xml MRTP improved

Figures 16a and 16b show the graphs of the synchronization speed for the shape from *Figure 7–b* of the MRTP algorithm and the improved MRTP, respectively.

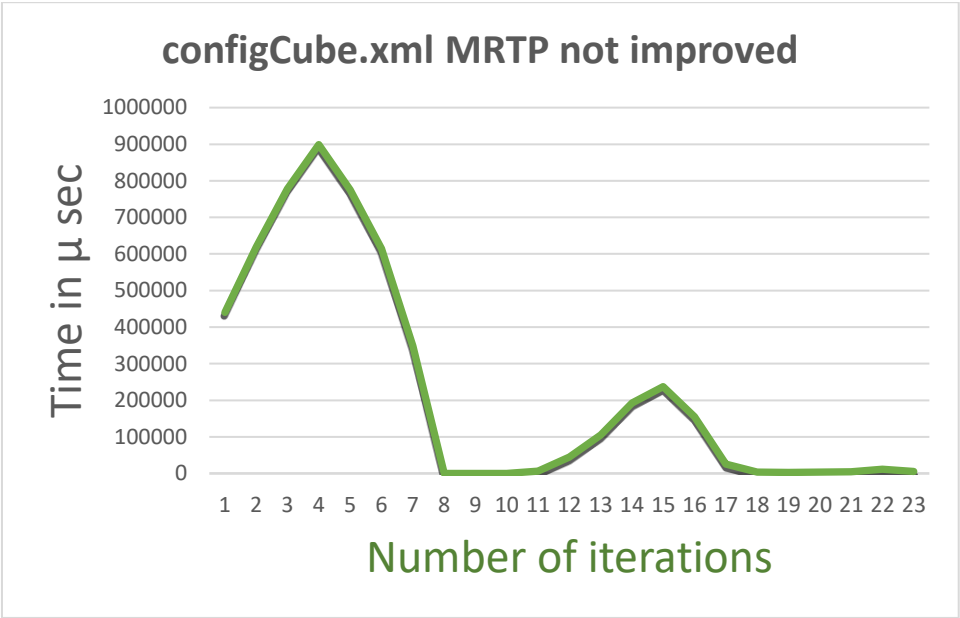


Figure 14 – improving algorithm

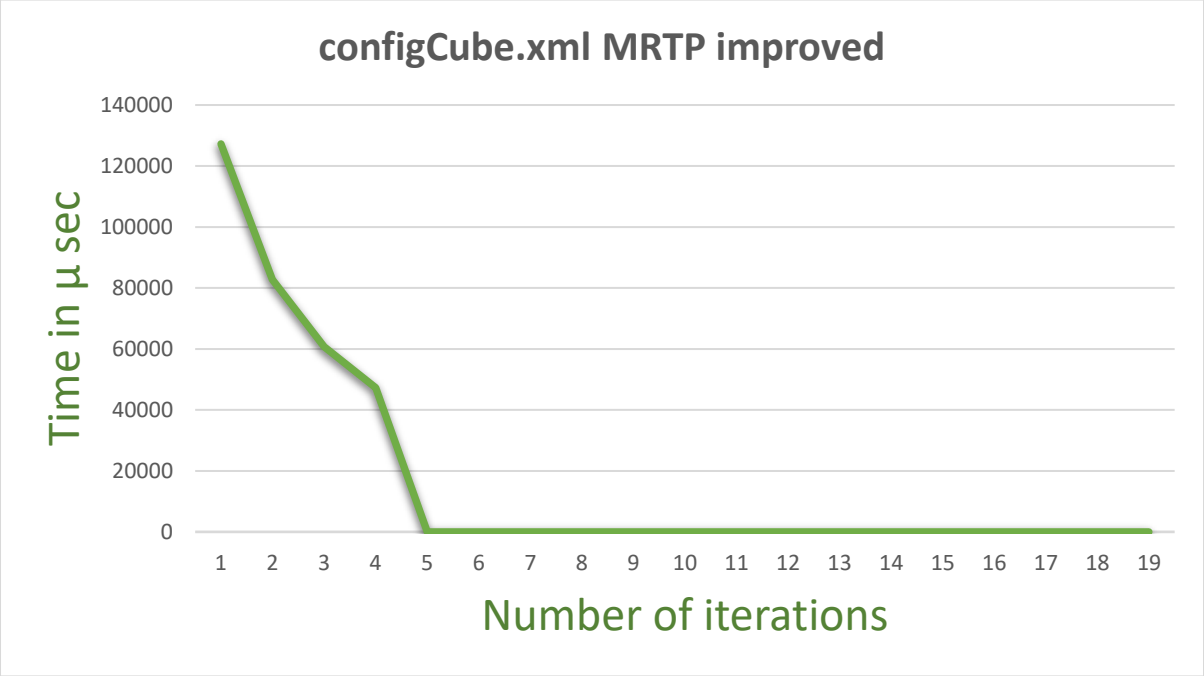


Figure 14 – improving algorithm

Conclusions:

From the performed modeling it can be concluded that the combination of the two algorithms significantly accelerates the synchronization speed for any type of form used.

In addition to modeling in the future, it is proposed to implement this algorithm on BlinkyBlocks in order to test the operation of the allogrhythm in real conditions.

List of sources used:

1. Naz A. et al. A time synchronization protocol for modular robots //2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). – IEEE, 2016. – С. 109-118.
2. Gao Q., Blow K. J., Holding D. J. Simple algorithm for improving time synchronisation in wireless sensor networks //Electronics Letters. – 2004. – Т. 40. – №. 14. – С. 889-891.
3. A. Naz, B. Piranda, S. C. Goldstein, and J. Bourgeois, “ABC-Center: Approximate-center election in modular robots,” in Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems IROS ’15, (Hamburg, Germany), September 2015.
4. Distributed Algorithms for Message-Passing Systems