Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Επεξεργασία και Διαχείριση Δεδομένων σε Δίκτυα Αισθητήρων
ΠΛΗ 511
Εργασία στο TinyOS
Αναφορά 1ου Προγράμματος

Ιωαννίδης Χρήστος (2018030006)
Παπαματθαιάκη Ηλέκτρα-Δέσποινα (2018030106)

1. Εισαγωγή

Στο πλαίσιο αυτής της εργασίας αναπτύξαμε κώδικα στο TinyOS για την προσομοίωση ενός δικτύου αισθητήρων. Οι αισθητήρες παράγουν τυχαίες τιμές, και η συναθροιστική συνάρτηση που υπολογίζεται από κάθε ομάδα είναι πάντα ο μέσος όρος. Η ομάδα εξαρτάται απο εναν τυχαιο αριθμο που δημιουργει ο κομβος βασης (κόμβος 0), υπάρχουν group 1 έως το 3. Η λειτουργία του δικτύου ακολουθεί το πρωτόκολλο TAG.

2. Υλοποίηση προγράμματος topology

Το βοηθητικό πρόγραμμα δημιουργήθηκε στην γλώσσα προγραμματισμού Python με σκοπό την προσομοίωση ενός grid από κόμβους με μέγεθος και εμβέλεια ορισμένη από τον χρήστη.

Καλείται μέσω στο Terminal με χρήση της εντολής:

python ./mySimulation.py D R

(e.g. python ./mySimulation.py 7 3.5)

όπου D είναι ο αριθμός που δίνεται ως παράμετρος και R ο αριθμός κινητής υποδιαστολής (η εμβέλεια).

Δημιουργείται ένα grid DxD με D^2 κόμβους, με κάθε κόμβο να έχει μοναδικό αναγνωριστικό από 0 έως D^2-1 και να τοποθετείται στη θέση j/D για τη γραμμή και j%D για τη στήλη. Οι οριζόντιες και κάθετες αποστάσεις μεταξύ των κόμβων θεωρούνται ίσες με 1.

Κάθε κόμβος εντοπίζει τους γείτονές του, βρίσκοντας όσους είναι σε απόσταση μικρότερη ή ίση με την εμβέλειά του. Οι πληροφορίες για τους γείτονες χρησιμοποιούνται για τη δημιουργία ενός αρχείου τοπολογίας, παρόμοιου με το topology.txt, που απεικονίζει τις συνδέσεις μεταξύ των κόμβων στο grid.

Σαν έξοδο έχει ένα αρχείο με όλα τα ζευγάρια κόμβων που είναι συνδεδεμένα μεταξύ τους, καθώς και τον θόρυβο του καναλιου. Παρακάτω, παρατίθεται ένα μέρος της εξόδου σαν παράδειγμα.

NODE_ID1 NODE_ID2 CHANNEL_NOISE

0 8 -50.0
8 0 -50.0
0 15 -50.0
15 0 -50.0
0 22 -50.0
22 0 -50.0
0 29 -50.0
29 0 -50.0
0 9 -50.0
9 0 -50.0

3. Πρόγραμμα 1.

Το βασικό πρόγραμμα υλοποιήθηκε, ακολουθώντας τα παρακάτω βήματα:

3.1: Διαγραφή Περιττών Κομματιών

Σε αυτό το στάδιο, πραγματοποιήθηκε η διαγραφή κομματιών κώδικα που δεν είναι απαραίτητα για τη λειτουργία του προγράμματος καθώς δεν συμβαδίζουν με τον τρόπο λειτουργίας του TAG. Αρχικά, καθώς δεν είχαμε σκοπό να μεταφέρουμε δεδομένα από τον κόμβο μηδέν σε κάποιο υπολογιστή όλη η λειτουργία της επικοινωνίας μέσω Serial Port αφαιρέθηκαν από τον κώδικα. Στη συνέχεια, εφόσον ο κώδικάς μας δεν θα εκτελεστεί σε πραγματικό δίκτυο αισθητήρων, αφαιρέσαμε τυχόν κώδικα που αφορούσε τη λειτουργία των LEDs. Οι δύο αυτές αλλαγές δεν επηρεάζουν την λειτουργία του TAG αλλά απλοποίησαν τον κώδικα.

Παρατηρήσαμε όμως, ότι στον κώδικα που παρέχεται στην εκφώνηση κατά την διαδικασία του Routing ο κάθε κόμβος έστελνε στον πατέρα του ένα **`NotifyParentMsg`**, με το οποίο ειδοποιούσε τον πιθανό του πατέρα ότι είναι το παιδί του, κάτι το οποίο δεν συνάδει με το πρωτόκολλο του TAG, καθώς σε αυτό δεν ενημερώνεται ο ένας κόμβος για το ποιοί το έχουν σαν πατέρα, συνειπώς όλος ο κώδικας αυτός έπρεπε να αφαιρεθεί.

3.2 Υλοποίηση Routing

- Αρχικά ορίστηκε ένας timer **`RoutingFinishedTimer`** ο οποίος καθορίζει το πόση ώρα θα κρατήσει η διαδικασία του routing, όπου στο χρονικό διάστημα μέχρι να χτυπήσει θα πρέπει όλοι οι κόμβοι να γνωρίζουν το βάθος τους καθώς και ποιός είναι ο πατέρας τους. Καθώς μια εποχή διαρκεί 60 δευτερόλεπτα, ορίστηκε σχετικά αυθαιρετα ο χρόνος των 90 δευτερολέπτων προκειμένου να έχει ολοκληρωθεί η όλη διαδικασία.

  ```
  call RoutingFinishedTimer.startOneShot(90*1024);
  ```

  Παρόλα αυτα λαμβάνοντας υπόψη το γεγονός ότι το interval ανάμεσα σε 2 routing μηνύματα είναι τάξεις μεγέθους μικροτερο απο τα 90sec του timer, πολυ πιθανόν να μπορούσε να ελεγχθει μια μικρότερη τιμή, αλλα δρώντας συντηρητικα και με στόχο να μην αναλωθούμε πολύ σε αυτο το αφήσαμε ως έχει.

- Ένας ακόμα βασικός timer που χρησιμοποιείται είναι ο **`RoutingMsgTimer`**, ο οποίος καθορίζει πότε θα κάνει broadcast ο κάθε κόμβος το routing message του. Ξεκινάει από τον κόμβο 0 και συνεχίζει με κάθε κόμβο ο οποίος αποκτά πατέρα μέχρι να γίνει fire από όλους τους κόμβους του δικτύου.

- Στο ***SimpleRoutingTree.h*** έγιναν οι εξής αλλαγές για το Routing message, το οποίο έχει την εξής μορφή:

```
typedef nx_struct RoutingMsg
{
   nx_uint16_t senderID;
   nx_uint8_t depth;
   nx_uint8_t RANDOM_NUM;
} RoutingMsg;
```

Στη δομή το **`senderID`** και **`depth`** είναι αντίστοιχα το ID και το βάθος του αποστολέα. Το **`RANDOM_NUM`** είναι ο τυχαίος αριθμός που δημιούργησε το κόμβος βάσης για τον καθορισμό των διαφορετικών ομάδων (group 1 εως 3 όπως αναφέρθηκε προηγουμένως αλλά θα αναλυθει εκτενέστερα παρακάτω η υλοποίηση του).

- Η επεξεργασία των δεδομένων του πακέτου γίνεται μέσα στο **`receiveRoutingTask()`**, όπου αν ένας κόμβος λάβει ένα τέτοιο πακέτο, σε περίπτωση που δεν έχει πατέρα, ορίζει το ID του αποστολέα σαν πατέρα του, το depth του κατά 1 μεγαλύτερο από τον καινούριο του πατέρα και τέλος με βάση το **`RANDOM_NUM`** υπολογίζει σε ποιό γκρουπ ανηκει. Σε περίπτωση που ο αποστολέας έχει ήδη πατερα το αγνοεί.

```
if ( (parentID<0)||(parentID>=65535))
           {
ParentID=call RoutingAMPacket.source(&radioRoutingRecPkt);
               curdepth= mpkt->depth + 1;
               RANDOM_NUM=mpkt->RANDOM_NUM;
               group=TOS_NODE_ID%RANDOM_NUM;

               if (TOS_NODE_ID!=0)
               {
                   call RoutingMsgTimer.startOneShot(TIMER_FAST_PERIOD);
               }
           }
```

3.3 Υλοποίηση Measurement

Με την ολοκλήρωση της διαδικασίας του routing, αρχικοποιείται όπως αναλύθηκε και παραπάνω ο **RoutingFinishedTimer**:

```
event void RoutingFinishedTimer.fired(){
       roundCounter=0;
       call
SendMeasurementTimer.startPeriodicAt((-curdepth*TIMER_FAST_SEND_MEASUREMENT_PERIOD+
(call RandomMeasurement.rand16() %
(TIMER_FAST_SEND_MEASUREMENT_PERIOD/2))),TIMER_PERIOD_MILLI);

   }
```

Μέσα στο event καλείται ο **SendMeasurementTimer** και είναι αυτός που θα καθορίσει τον χρόνο στον οποίο θα στείλει δεδομένα ο κάθε κόμβος. Στην ουσία ανάλογα με το βάθος του κάθε κόμβου μετατίθεται και ο χρόνος που θα στείλει κατά:

curdepth*TIMER_FAST_SEND_MEASUREMENT_PERIOD

Η περίοδος της κάθε εποχής (**TIMER_PERIOD_MILLI**) είναι ορισμένη στα 60 δευτερόλεπτα.

Επειδή κατα την διάρκεια υπολογισμού τελικού αποτελέσματος παρατηρηθηκε οτι δεδομένα από αρκετούς κόμβους δεν έφταναν κάν στην βαση διαπιστώθηκε μετα απο αρκετες δοκιμές (απόπειρα μείωσης θορύβου κπλ) οτι ο βασικός υπαίτιος ήταν τα collisions μεταξύ μηνυμάτων. Αυτό το πρόβλημα λύθηκε με την εισχώρησης λιγης τυχαιότητας στον χρόνο αποστολης:

**(+(call RandomMeasurement.rand16() %
(TIMER_FAST_SEND_MEASUREMENT_PERIOD/2)))**

Όταν γίνεται “fire” ο **SendMeasurementTimer**, γίνεται post ένα task που ονομάζεται **RandNumMaker()**. Σε γενικές γραμμες, αυτη η συνάρτηση είναι υπεύθυνη για την δημιουργία των μετρήσεων και την αποστολή τους με την χρήση του σωστού struct για την ελαχιστοποίηση του μεγέθους των αποστολών.

Η δημιουργία μετρήσεων γίνεται με τις παρακάτω γραμμες κώδικα:

```
        //keep old measurement
        old_measurement=measurement;
        //create random measurement
        if(measurement==-1){
            measurement = (call RandomMeasurement.rand16() % 100);
            old_measurement=measurement;
        }
        else {
            //+-20% of old measurement
            uint8_t upperBound = (int)old_measurement + 0.2*old_measurement;
            uint8_t lowerBound = (int)old_measurement - 0.2*old_measurement;

measurement = call RandomMeasurement.rand16()%(upperBound-lowerBound + 1) +
lowerBound;
            //check so that the node doesnt pass the value bounds
            if(measurement<1){
                measurement=1;
            }
            if(measurement>100){
                measurement=100;
            }
            old_measurement=measurement;
        }
```

Μέσα σε κάθε κόμβο αποθηκεύεται η παλιά του μέτρηση (**`old_measurement`**) προκειμένου να κάνει την καινουρια να έχει τιμή +-20% της παλιάς.

Προκειμένου να υπολογιστεί ο μέσος όρος χρειαζόμαστε ένα άθροισμα μετρήσεων και τον αριθμό των διαφορετικών μετρήσεων, αυτο υλοποιείται με την χρήση των μεταβλητών:

```
uint16_t aggregate_measurement=0;
uint8_t count=0;
```

Ή μάλλον επειδή έχουμε μέχρι 3 διαφορετικά γκρουπ:

```
uint16_t aggregate_measurement=0;
uint16_t aggregate_measurement1=0;
uint16_t aggregate_measurement2=0;

uint8_t count=0;
uint8_t count1=0;
uint8_t count2=0;
```

Καθώς διαθέτουμε τρεις διαφορετικές δυνατές ομάδες, για να είμαστε αποδοτικοί όσον αφορά το μέγεθος των μηνυμάτων, απαιτούνται επίσης τρία διαφορετικά structs:

```
typedef nx_struct MeasurementMsg
{
   nx_uint8_t includedGroups;
   nx_uint8_t count;
   nx_uint16_t measurement;
} MeasurementMsg;
typedef nx_struct MeasurementMsg1
{
   nx_uint8_t includedGroups;
   nx_uint8_t count;
   nx_uint16_t measurement;
   nx_uint8_t count1;
   nx_uint16_t measurement1;
} MeasurementMsg1;
typedef nx_struct MeasurementMsg2
{
   nx_uint8_t includedGroups;
   nx_uint8_t count;
   nx_uint16_t measurement;
   nx_uint8_t count1;
   nx_uint16_t measurement1;
   nx_uint8_t count2;
   nx_uint16_t measurement2;
} MeasurementMsg2;
```

Η μεταβλητη **`includedGroups`** είναι μία πολύ σημαντική μεταβλητή η οποία θα μας βοηθήσει να καταλάβουμε όταν διαβάζουμε ένα πακέτο σε ποιά μεταβλητή του struct βρίσκεται πληροφορία για ποιό γκρουπ. Για την επίτευξη αυτού του στόχου χρησιμοποιεί τα τελευταία 3 bits, καθένα απο αυτα αντιστοιχεί σε ενα ξεχωριστό γκρουπ και αποκωδικοποιείται ως εξής:

1. Αρχικά, επιδιώκουμε να αποκτήσουμε πληροφορίες σχετικά με τον αριθμό των γκρουπ που περιέχονται στο πακέτο που λάβαμε, το οποίο ανιχνεύουμε μέσω του μεγέθους του πακέτου.
2. Στην περίπτωση που υπάρχει ένα μόνο γκρουπ μέσα στο πακέτο, τα τρία τελευταία bits θα είναι ένας αριθμός "ασσος". Ανάλογα με τη θέση του στα τρία τελευταία bits, μπορούμε να καταλάβουμε για ποιο γκρουπ διαβάζουμε. Για παράδειγμα, αν ο αριθμός είναι 0b001, σημαίνει ότι διαβάζουμε πληροφορίες για το γκρουπ 1.

3. Σε περίπτωση που υπάρχουν δύο γκρουπ στο πακέτο, τα γκρουπ για τα οποία διαβάζουμε πληροφορία είναι αυτά στα οποία υπάρχει "ασσος" στη θέση τους. Διαβάζουμε πάντα με αύξουσα σειρά γκρουπ. Για παράδειγμα, αν έχουμε 0b101, σημαίνει ότι διαβάζουμε πληροφορία για τα γκρουπ 1 και 3. Η πληροφορία θα διαβαστεί ως εξής: πρώτα για το γκρουπ 1, έπειτα για το 3.

```
aggregate_measurement+=mpkt1->measurement;
count+=mpkt1->count;
aggregate_measurement2+=mpkt1->measurement1;
count2+=mpkt1->count1;}
```

4. Σε περίπτωση που υπάρχουν τρία γκρουπ, τότε απλώς τα διαβάζουμε με την σειρά.

```
aggregate_measurement+=mpkt2->measurement;
count+=mpkt2->count;
aggregate_measurement1+=mpkt2->measurement1;
count1+=mpkt2->count1;
aggregate_measurement2+=mpkt2->measurement2;
count2+=mpkt2->count2;
```

Η διαδικασία αυτή δίνεται στην συνάντηση **`receiveMeasurementTask()`**. Με παρόμοια λογικη γινεται και η κωδικοποίηση τους στην συνάρτηση **`RandNumMaker()`**.

Τέλος, ο υπολογισμός του τελικού αποτελέσματος γινεται στον κόμβο βάσης την ώρα που θα χτυπήσει ο μετρητής για να στειλει δεδομένα:

```
if(TOS_NODE_ID==0){
            if(count==0){
                count=1;
            }
            if(count1==0){
                count1=1;
            }
            if(count2==0){
                count2=1;
            }
dbg("final" , "FINAL RESULT group 1: %u \n", aggregate_measurement/count);
dbg("final" , "FINAL RESULT group 2: %u \n", aggregate_measurement1/count1);
dbg("final" , "FINAL RESULT group 3: %u\n", aggregate_measurement2/count2);



        }
```

4. Παραδείγματα εκτέλεσης κώδικα

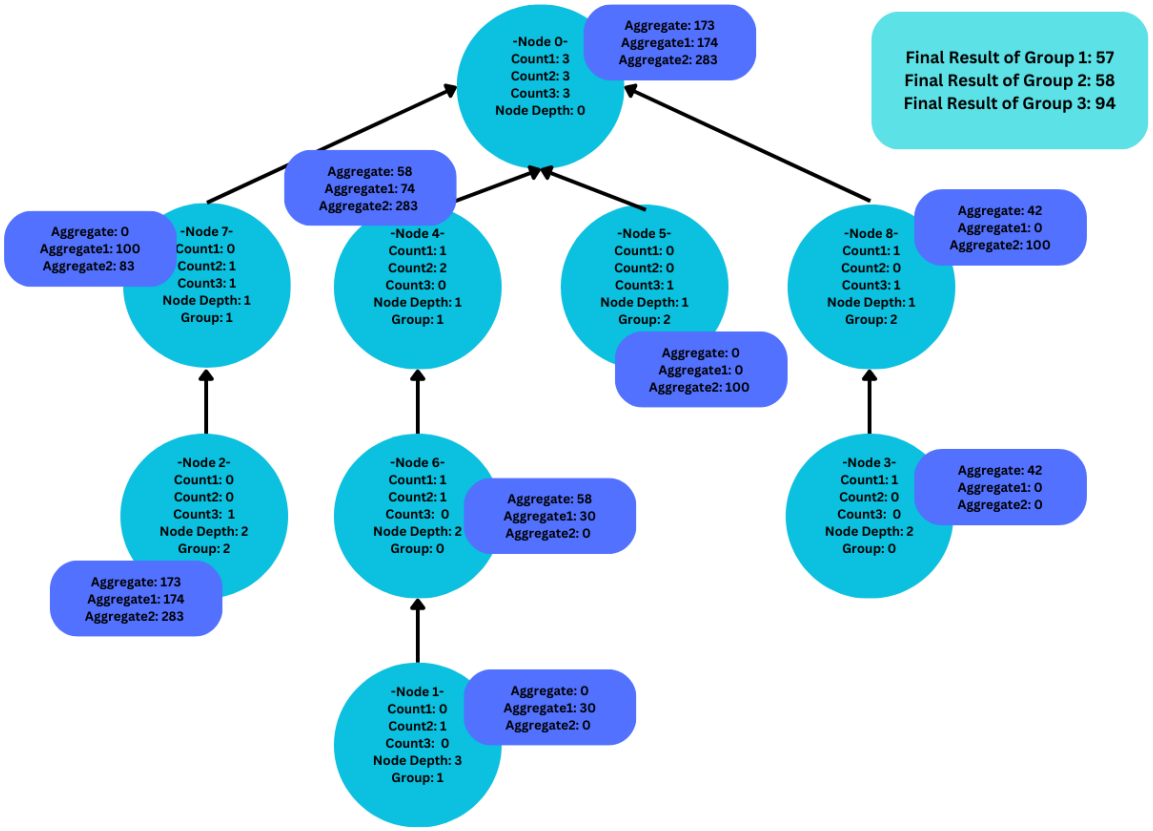
- Παράδειγμα εκτέλεσης με 9 κόμβους(3x3 grid) και range 3:

Για αρχή, για να μπορέσουμε να ελέγξουμε την ορθότητα του κώδικα δουλεύαμε με εννέα (9) κόμβους και εμβέλεια τρία (3) για το λόγο αυτό θα εξηγηθεί ένα τέτοιο παράδειγμα για να διαπιστωθεί η ορθότητα του προγράμματος.

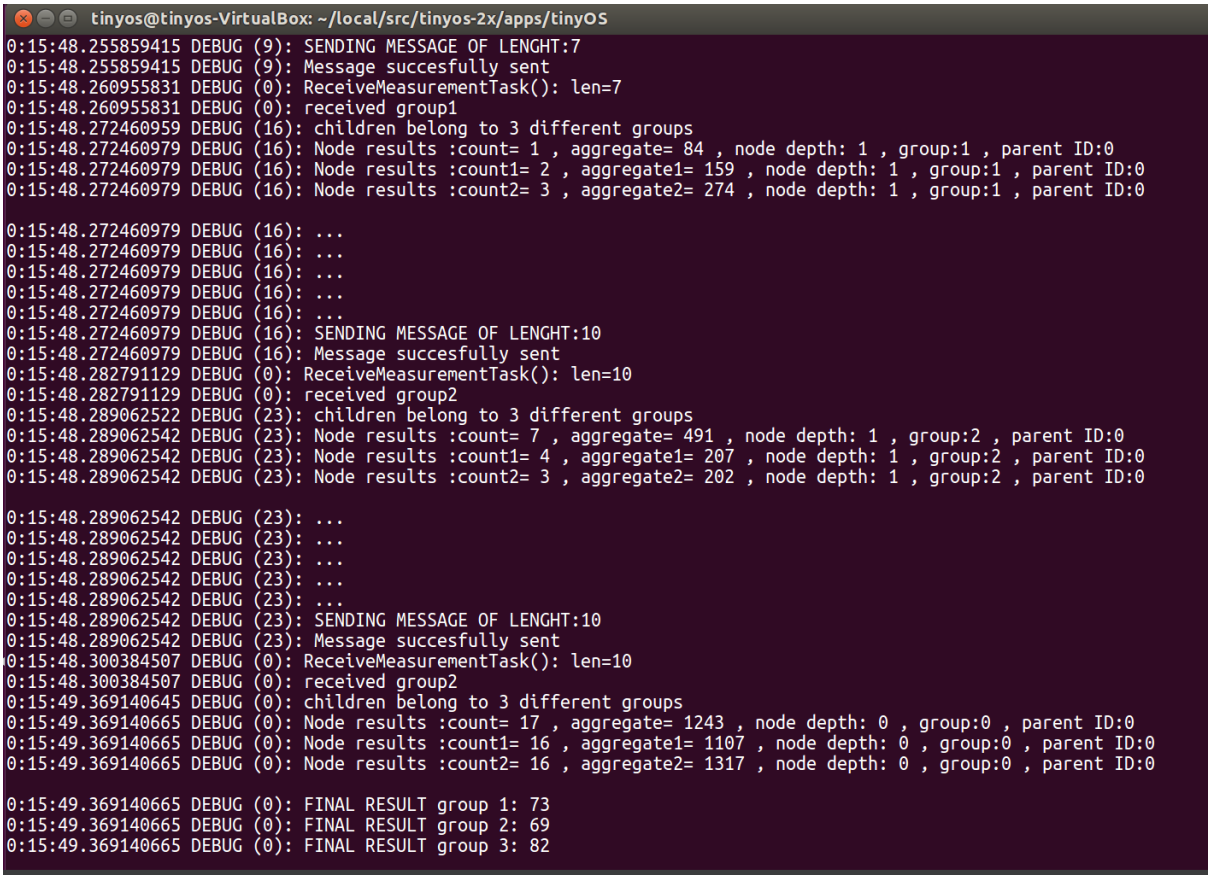
```
0:16:5.576171895 DEBUG (1): children belong to 1 different groups
0:16:5.576171915 DEBUG (1): Node results :count= 0 , aggregate= 0 , node depth: 3 , group:1 , parent ID:6
0:16:5.576171915 DEBUG (1): Node results :count1= 1 , aggregate1= 30 , node depth: 3 , group:1 , parent ID:6
0:16:5.576171915 DEBUG (1): Node results :count2= 0 , aggregate2= 0 , node depth: 3 , group:1 , parent ID:6

0:16:5.576171915 DEBUG (1): ...
0:16:5.576171915 DEBUG (1): ...
0:16:5.576171915 DEBUG (1): ...
0:16:5.576171915 DEBUG (1): ...
0:16:5.576171915 DEBUG (1): ...
0:16:5.576171915 DEBUG (1): SENDING MESSAGE OF LENGHT:4
0:16:5.576171915 DEBUG (1): Message succesfully sent
0:16:5.584045414 DEBUG (6): ReceiveMeasurementTask(): len=4
0:16:5.584045414 DEBUG (6): received group0
0:16:6.766601583 DEBUG (6): children belong to 2 different groups
0:16:6.766601603 DEBUG (6): Node results :count= 1 , aggregate= 58 , node depth: 2 , group:0 , parent ID:4
0:16:6.766601603 DEBUG (6): Node results :count1= 1 , aggregate1= 30 , node depth: 2 , group:0 , parent ID:4
0:16:6.766601603 DEBUG (6): Node results :count2= 0 , aggregate2= 0 , node depth: 2 , group:0 , parent ID:4

0:16:6.766601603 DEBUG (6): ...
0:16:6.766601603 DEBUG (6): ...
0:16:6.766601603 DEBUG (6): ...
0:16:6.766601603 DEBUG (6): ...
0:16:6.766601603 DEBUG (6): ...
0:16:6.766601603 DEBUG (6): SENDING MESSAGE OF LENGHT:7
0:16:6.766601603 DEBUG (6): Message succesfully sent
0:16:6.770568875 DEBUG (4): ReceiveMeasurementTask(): len=7
0:16:6.770568875 DEBUG (4): received group1
0:16:6.902343770 DEBUG (3): children belong to 1 different groups
0:16:6.902343790 DEBUG (3): Node results :count= 1 , aggregate= 42 , node depth: 2 , group:0 , parent ID:8
0:16:6.902343790 DEBUG (3): Node results :count1= 0 , aggregate1= 0 , node depth: 2 , group:0 , parent ID:8
0:16:6.902343790 DEBUG (3): Node results :count2= 0 , aggregate2= 0 , node depth: 2 , group:0 , parent ID:8

0:16:6.902343790 DEBUG (3): ...
0:16:6.902343790 DEBUG (3): ...
0:16:6.902343790 DEBUG (3): ...
0:16:6.902343790 DEBUG (3): ...
0:16:6.902343790 DEBUG (3): ...
0:16:6.902343790 DEBUG (3): SENDING MESSAGE OF LENGHT:4
0:16:6.902343790 DEBUG (3): Message succesfully sent
0:16:6.910781861 DEBUG (8): ReceiveMeasurementTask(): len=4
0:16:6.910781861 DEBUG (8): received group0
0:16:6.989257832 DEBUG (2): children belong to 1 different groups
0:16:6.989257852 DEBUG (2): Node results :count= 0 , aggregate= 0 , node depth: 2 , group:2 , parent ID:7
0:16:6.989257852 DEBUG (2): Node results :count1= 0 , aggregate1= 0 , node depth: 2 , group:2 , parent ID:7
0:16:6.989257852 DEBUG (2): Node results :count2= 1 , aggregate2= 83 , node depth: 2 , group:2 , parent ID:7

0:16:6.989257852 DEBUG (2): ...
0:16:6.989257852 DEBUG (2): ...
0:16:6.989257852 DEBUG (2): ...
0:16:6.989257852 DEBUG (2): ...
0:16:6.989257852 DEBUG (2): ...
0:16:6.989257852 DEBUG (2): SENDING MESSAGE OF LENGHT:4
0:16:6.989257852 DEBUG (2): Message succesfully sent
0:16:6.995574964 DEBUG (7): ReceiveMeasurementTask(): len=4
0:16:6.995574964 DEBUG (7): received group0
0:16:7.853515645 DEBUG (5): children belong to 1 different groups
0:16:7.853515665 DEBUG (5): Node results :count= 0 , aggregate= 0 , node depth: 1 , group:2 , parent ID:0
0:16:7.853515665 DEBUG (5): Node results :count1= 0 , aggregate1= 0 , node depth: 1 , group:2 , parent ID:0
0:16:7.853515665 DEBUG (5): Node results :count2= 1 , aggregate2= 100 , node depth: 1 , group:2 , parent ID:0
```

```
0:16:7.853515665 DEBUG (5): ...
0:16:7.853515665 DEBUG (5): ...
0:16:7.853515665 DEBUG (5): ...
0:16:7.853515665 DEBUG (5): ...
0:16:7.853515665 DEBUG (5): ...
0:16:7.853515665 DEBUG (5): SENDING MESSAGE OF LENGHT:4
0:16:7.853515665 DEBUG (5): Message succesfully sent
0:16:7.864578233 DEBUG (0): ReceiveMeasurementTask(): len=4
0:16:7.864578233 DEBUG (0): received group0
0:16:7.929687520 DEBUG (7): children belong to 2 different groups
0:16:7.929687540 DEBUG (7): Node results :count= 0 , aggregate= 0 , node depth: 1 , group:1 , parent ID:0
0:16:7.929687540 DEBUG (7): Node results :count1= 1 , aggregate1= 100 , node depth: 1 , group:1 , parent ID:0
0:16:7.929687540 DEBUG (7): Node results :count2= 1 , aggregate2= 83 , node depth: 1 , group:1 , parent ID:0

0:16:7.929687540 DEBUG (7): ...
0:16:7.929687540 DEBUG (7): ...
0:16:7.929687540 DEBUG (7): ...
0:16:7.929687540 DEBUG (7): ...
0:16:7.929687540 DEBUG (7): ...
0:16:7.929687540 DEBUG (7): SENDING MESSAGE OF LENGHT:7
0:16:7.929687540 DEBUG (7): Message succesfully sent
0:16:7.932220494 DEBUG (0): ReceiveMeasurementTask(): len=7
0:16:7.932220494 DEBUG (0): received group1
0:16:8.092773458 DEBUG (8): children belong to 2 different groups
0:16:8.092773478 DEBUG (8): Node results :count= 1 , aggregate= 42 , node depth: 1 , group:2 , parent ID:0
0:16:8.092773478 DEBUG (8): Node results :count1= 0 , aggregate1= 0 , node depth: 1 , group:2 , parent ID:0
0:16:8.092773478 DEBUG (8): Node results :count2= 1 , aggregate2= 100 , node depth: 1 , group:2 , parent ID:0

0:16:8.092773478 DEBUG (8): ...
0:16:8.092773478 DEBUG (8): ...
0:16:8.092773478 DEBUG (8): ...
0:16:8.092773478 DEBUG (8): ...
0:16:8.092773478 DEBUG (8): ...
0:16:8.092773478 DEBUG (8): SENDING MESSAGE OF LENGHT:7
0:16:8.092773478 DEBUG (8): Message succesfully sent
0:16:8.095642123 DEBUG (0): ReceiveMeasurementTask(): len=7
0:16:8.095642123 DEBUG (0): received group1
0:16:8.315429707 DEBUG (4): children belong to 2 different groups
0:16:8.315429727 DEBUG (4): Node results :count= 1 , aggregate= 58 , node depth: 1 , group:1 , parent ID:0
0:16:8.315429727 DEBUG (4): Node results :count1= 2 , aggregate1= 74 , node depth: 1 , group:1 , parent ID:0
0:16:8.315429727 DEBUG (4): Node results :count2= 0 , aggregate2= 0 , node depth: 1 , group:1 , parent ID:0

0:16:8.315429727 DEBUG (4): ...
0:16:8.315429727 DEBUG (4): ...
0:16:8.315429727 DEBUG (4): ...
0:16:8.315429727 DEBUG (4): ...
0:16:8.315429727 DEBUG (4): ...
0:16:8.315429727 DEBUG (4): SENDING MESSAGE OF LENGHT:7
0:16:8.315429727 DEBUG (4): Message succesfully sent
0:16:8.324523923 DEBUG (0): ReceiveMeasurementTask(): len=7
0:16:8.324523923 DEBUG (0): received group1
0:16:9.188476582 DEBUG (0): children belong to 3 different groups
0:16:9.188476602 DEBUG (0): Node results :count= 3 , aggregate= 173 , node depth: 0 , group:0 , parent ID:0
0:16:9.188476602 DEBUG (0): Node results :count1= 3 , aggregate1= 174 , node depth: 0 , group:0 , parent ID:0
0:16:9.188476602 DEBUG (0): Node results :count2= 3 , aggregate2= 283 , node depth: 0 , group:0 , parent ID:0

0:16:9.188476602 DEBUG (0): FINAL RESULT group 1: 57
0:16:9.188476602 DEBUG (0): FINAL RESULT group 2: 58
0:16:9.188476602 DEBUG (0): FINAL RESULT group 3: 94
```

Με την βοήθεια του παρακάτω γραφήματος είναι πιο ξεκάθαρο ότι ο κώδικάς μας τρέχει σωστά. Αρχικά, δημιουργήθηκαν 3 groups, και στο τέλος, δηλαδή στον κόμβο μηδέν εάν προστεθούν τα count το αποτέλεσμα είναι 9. Εάν προστεθούν τα aggregates του κάθε group μπορεί να διαπιστωθεί εύκολα ότι το τελικό αποτέλεσμα του κάθε κόμβου αποτελεί τον μέσο όρο αυτών.



- Παράδειγμα εκτέλεσης με 49 κόμβους και range 3:



Σε ένα παράδειγμα με σημαντικά μεγαλύτερο αριθμό κόμβο μπορεί να παρατηρηθεί και πάλι η ορθή λειτουργία στον κόμβο της βάσης, το count καθώς και τα τελικά αποτελέσματα είναι αυτό που περιμέναμε.

6. Τρόπος εργασίας

- Βοηθητικό πρόγραμμα: Παπαματθαιάκη Ηλέκτρα-Δέσποινα
- Routing: Ιωαννίδης Χρήστος
- Measurement Messages (all nodes same group): Ιωαννίδης Χρήστος-Παπαματθαιάκη Ηλέκτρα-Δέσποινα
- 3 different group implementation: Ιωαννίδης Χρήστος
- Report: Ιωαννίδης Χρήστος-Παπαματθαιάκη Ηλέκτρα-Δέσποινα

Στην εργασία αυτή αποφασίσαμε να εργαστούμε μέσω Github ώστε να διευκολυνθεί η διαδικασία συγγραφής κώδικα και εξ αποστάσεως, και για να βελτιώσουμε το workflow αξιοποιώντας τις διάφορες ευκολίες που προσφέρει η πλατφόρμα αυτή. Μερικές φορές, ήταν περισσότερο βολικό να βρισκόμαστε από κοντά για να βλέπουμε την πρόοδο που έχει γίνει και να γράφουμε κώδικα στον ίδιο υπολογιστή. Παρακάτω μπορείτε να δείτε ένα στιγμιότυπο από την εργασία στον ιστότοπο (το repository μέχρι να ολοκληρωθεί η εργασία, είναι private). Τέλος, αξίζει να σημειωθεί ότι όλα τα μέλη της ομάδας εργαστηκαν ενεργά σε όλα τα μέρη της εργασίας, η λίστα παραπάνω ειναι προσεγγιστική.