ΗΜΜΥ Πολυτεχνειου Κρητης

ΠΛΗ 516-Επεξεργασια και Διαχειριση σε Δικτυα Αισθητηρων

Χειμερινο Εξαμηνο 2018-2019

Εργασία tinyOS: Φάση 1

Φοιτητές
Καλογερακης Στεφανος
Παγκαλος Μανωλης

Διδάσκων
Α. Δεληγιαννακης

Βοηθητικό πρόγραμμα

Σε αυτό το κομμάτι της εργασίας σκοπός ήταν η ανάπτυξη ενός προγράμματος που να δημιουργεί δυναμικά μια τέτοια τοπολογία. Επιλέξαμε την γλώσσα προγραμματισμού Java και το λογισμικό Intellij IDEA για την ανάπτυξη αυτού του προγράμματος. Οι είσοδοι του συστήματος είναι δύο παράμετροι, D και R. Το D αναφέρεται στη διάμετρο της τοπολογίας και το R είναι η ακτίνα εμβέλειας του κάθε κόμβου. Συγκεκριμένα, οι κόμβοι είναι ομοιόμορφα μοιρασμένοι σε σχήμα τετραγώνου πλευράς D με τον κάθε κόμβο j να έχει τοποθετηθεί όπως ζητείται απο την εκφώνηση σε γραμμή j/D και στήλη $J \bmod D$ με τους κόμβους μεταξύ τους να απέχουν 1 μονάδα. Στην συνέχεια ακολουθεί ενδεικτικά τοπολογία με τιμή παραμέτρου $D = 4$. Οι τιμές που είναι τοποθετημένες στο grid αντιστοιχούν στο TOS_NODE_ID

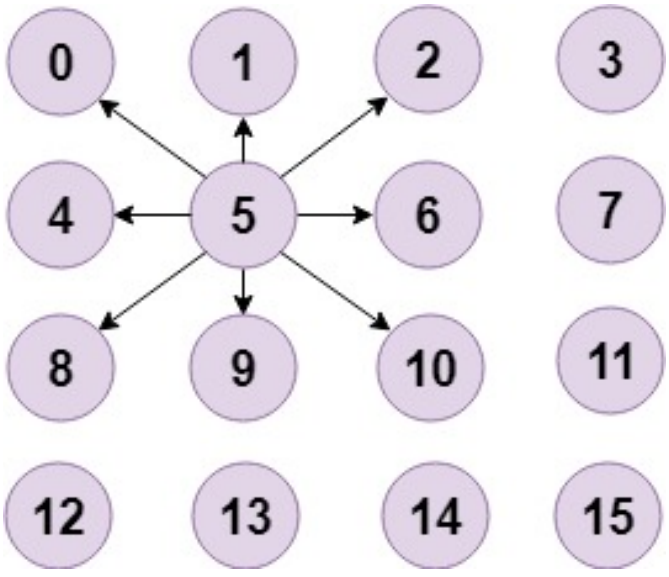
$$grid = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

Προγραμματιστικά, για την δημιουργία αυτού του grid δημιουργήσαμε έναν πίνακα $2 * 2$ όπου σε κάθε θέση καταγράφεται το ID του κόμβου. Σκοπός είναι να εντοπίσουμε για κάθε κόμβο, βάσει της εμβέλειας, ποιους άλλους κόμβους έχει γείτονες. Αυτό για να το υπολογίσουμε ελέγξαμε για τις συντεταγμένες του καθενός την ευκλείδεια απόσταση d που απέχουν όλοι οι υπόλοιποι κόμβοι με τον εξής τύπο:

$$d(curNode, neighNode) = \sqrt{(i-x)^2 + (j-y)^2}$$

Όπου x, y οι συντεταγμένες του εκάστοτε γειτονικού κόμβου και i, j οι συντεταγμένες του κόμβου που εξετάζουμε.

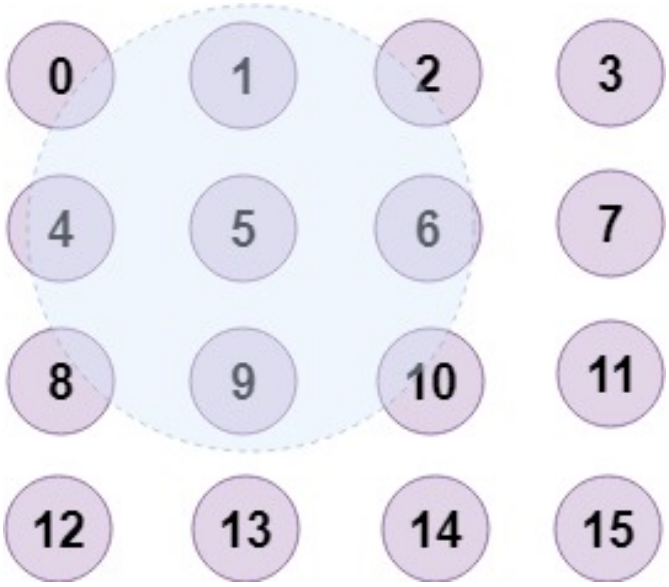
Αν αυτή η απόσταση είναι μικρότερη ή ίση με την ακτίνα της εμβέλειας R τότε οι δύο κόμβοι είναι γειτονικοί και αυτή η πληροφορία καταγράφεται. Σε διαφορετική περίπτωση συνεχίζουν οι αναζητήσεις. Στην ενδεικτική τοπολογία θα αναζητήσουμε ως παράδειγμα τους γείτονες του κόμβου 5.



Σχήμα 1: Υπολογισμών γειτονικών τιμών κόμβου 5

Βάσει των υπολογισμών που εξηγήθηκαν προηγουμένως θα προκύψει ότι για ακτίνα εμβέλειας

$R = 1.5$ ο κόμβος 5 μπορεί να επικοινωνεί και με τους διαγώνιους κόμβους καθώς απέχουν $\sqrt{2} = 1.41 < 1.5$.



Σχήμα 2: Γείτονες κόμβου 5

Επομένως, για τον κόμβο 5 θα καταγραφούν ως γειτονικοί κόμβοι οι 0,1,2,4,6,8 και 10.Η ίδια διαδικασίας θα ακολουθηθεί και για όλους τους άλλους κόμβους. Στο τέλος θα έχουμε πληροφορία για τους γείτονές κάθε κόμβου τα οποία καταγράφονται σε ένα *topology.txt* αρχείο.

Αξίζει να σημειωθεί ότι το συγκεκριμένο πρόγραμμα δεν καλείται δυναμικά από κάποιο σημείο του simulation που περιγράφεται στην συνέχεια μετά από υπόδειξη του διδάσκοντα. Πρέπει να καλεστεί από εξωτερική πλατφόρμα εκτέλεσης Java πραγματοποιώντας κατάλληλες αλλαγές όπως περιγράφηκαν προηγουμένως για την τελική δημιουργία *topology.txt* αρχείου της προτίμησης του χρήστη.

Σε κάποιες περιπτώσεις υπήρξε πρόβλημα κατά το import του βοηθητικού προγράμματος σε λογισμικά όπως το Eclipse αλλά ακολουθώντας τις υποδείξεις του λογισμικού επιλύνεται το πρόβλημα. Σε περίπτωση εμφάνισης οποιουδήποτε προβλήματος συμβατότητας ο κώδικας και οι εκδόσεις που δουλέψαμε είναι διαθέσιμα και από το repository του github που ακολουθεί.
Στην περίπτωση που δεν λειτουργήσουν τα συγκεκριμένα βήματα στα παραδοτέα αρχεία υπάρχει και ένα .jar αρχείο με το όνομα SupportApp.jar το οποίο μπορεί να τρέξει μέσω terminal με την εντολή $java - jar < FileFullPath > .jar$ **και το logfile αρχείο θα βρίσκεται στον φάκελο του user**

Βοηθητικό Πρόγραμμα:
$https://github.com/skalogerakis/Processing\_And\_Management\_of\_Wireless\_Sensor\_Networks$

Πρόγραμμα Φάσης 1

Αφαίρεση κώδικα

Αρχικά μετά την κατανόηση του κώδικα ο πρώτος στόχος μας ήταν να επέμβουμε και να κάνουμε κάποιες διορθώσεις προκειμένου η εκτέλεση να συμβαδίζει με την αρχή λειτουργίας του TAG.

Υπήρχαν κάποια κομμάτια στον κώδικα τα οποία δεν εκτελούνταν ποτέ και συνεπώς δεν είχαν κάποιο σκοπό στον κώδικα. Περιττά ήταν τα σημεία όπου καθοριζόταν η λειτουργία των LED τα οποία δεν υπάρχουν στην προσομοίωση. Ακόμη, περιττά ήταν τα σημεία όπου για την αποστολή πληροφοριών χρησιμοποιούνταν Serial Port καθώς πρόκειται για σύνδεση αισθητήρων ενσύρματα, πράγμα που επίσης δεν υφίσταται στην προσομοίωση. Αυτά τα κομμάτια παρόλο που δεν αντιτίθενται στην λειτουργία του TAG τα αφαιρέσαμε για την μείωση του κώδικα.

Στην συνέχεια, διορθώσαμε τα κομμάτια που ήταν ενάντια στην λειτουργία του TAG. Αρχικά, αφαιρέσαμε τελείως τα μηνύματα και τις συναρτήσεις Notify προς τον πατέρα εφόσον αυτό δεν ορίζεται στην λειτουργία του TAG. Βασιστήκαμε όμως στον κώδικα που περιείχαν τα events και τα task για την υλοποίηση των μηνυμάτων Notify προκειμένου να δημιουργήσουμε τα δικά μας Distribution μηνύματα τα οποία μεταφέρουν συναθροιστικές τιμές στον πατέρα τους. Επίσης, αφαιρέσαμε ένα κομμάτι του κώδικα στο receiveRoutingTask όπου ένας κόμβος που είχε πατέρα έψαχνε για κάποιον άλλο καλύτερο πατέρα. Αυτό δεν ισχύει στο TAG καθώς ακολουθεί την λογική First-Heard-First. Τέλος αφαιρέσαμε κάποιες LostTask συναρτήσεις αφού ο κώδικας δεν έμπαινε ποτέ ενώ και για το TAG από την στιγμή που χαθεί ένα μήνυμα θεωρούμε ότι απλά χάθηκε.

Στην τελική υλοποίηση αφήσαμε κάποιους ελέγχους στους οποίους το simulation δεν μπαίνει ποτέ, καθαρά για λόγους πληρότητας του κώδικα και αναφέρεται χαρακτηριστικά σε κάθε περίπτωση με σχόλια.

Προσθήκη κώδικα

Για την επιτυχή υλοποίηση του TAG σύμφωνα με την εκφώνηση έπρεπε να διαφοροποιήσουμε αρκετά τον κώδικα έχοντας τα παρακάτω στο μυαλό μας προς υλοποίηση

* Υλοποίηση μια φορά του Routing
* Σωστή αποστολή και λήψη μηνυμάτων με λιγότερες δυνατές συγκρούσεις
* Σωστός συγχρονισμός timers
* Διατήρηση προηγούμενων τιμών παιδιών σε περίπτωση απώλειας μηνυμάτων

Πιο συγκεκριμένα, με την αρχή της εκτέλεσης του προγράμματος γίνεται το Routing το οποίο ρυθμίσαμε να καλείται μια φορά κατά την αρχή της εκτέλεσης και να δημιουργεί την τοπολογία όπως έχει περιγραφεί προηγουμένως. Για το Routing υπήρχε ήδη ο κώδικας και δεν πραγματοποιήσαμε πολλές αλλαγές. Να σημειωθεί σε αυτό το σημείο, ότι για διευκόλυνση μας στον συγχρονισμό της συνέχειας, διαθέτουμε **επιπλέον 5 δευτερόλεπτα μόνο για το Routing** από την χρόνο λειτουργίας του προγράμματος προκειμένου να έχουν ολοκληρώσει όλοι οι κόμβοι με την συγκεκριμένη διεργασία και να είναι έτοιμοι για την αποστολή μηνυμάτων. Άρα ο τελικός χρόνος εκτέλεσης θα είναι **900 + 5 δευτερόλεπτα**. Παραθέτουμε το σημείο του κώδικα όπου ρυθμίζουμε το Routing να εκτελεστεί μια φορά και να διαρκεί 5 δευτερόλεπτα

```
call RoutingComplTimer.startOneShot(5000);

/** Routing happens once at the start*/
if (TOS_NODE_ID==0)
{
    dbg("SRTreeC", "###########START ROUTING##############\n\n");

    call RoutingMsgTimer.startOneShot(TIMER_FAST_PERIOD);

}
```

Σχήμα 3: Καθυστέρηση 5 δευτερολέπτων για Routing

Στην συνέχεια της υλοποίησης, θέσαμε τους κόμβους να παράγουν μετρήσεις και να πραγματοποιούν συναθροίσεις κάθε $60000ms$ (αντιστοιχεί σε τιμή ελάχιστα μικρότερη των 60 δευτερολέπτων). Η μέτρηση κάθε αισθητήρα είναι ένας τυχαίος ακέραιος αριθμός στο εύρος 0 έως 50 ενώ η δημιουργία της πραγματοποιείται μέσω μιας γεννήτρια τυχαίων αριθμών.

Οι συναθροίσεις που μας ζητήθηκε να υλοποιήσουμε για την πρώτη φάση είναι η MAX και AVG. Είναι πολύ σημαντικό κάθε μήνυμα να μην περιέχει περιττή πληροφορία αλλά μόνο την ελάχιστη απαραίτητη για τον υπολογισμό των εκάστοτε συναρτήσεων. Στην προκειμένη περίπτωση απαιτείται πληροφορία για την μέγιστη μέτρηση που έχει παρατηρηθεί, το άθροισμα όλων των μετρήσεων και το πλήθος τον μετρήσεων που λαμβάνουν μέρος στην συνάθροιση. Η μέση τιμή θα υπολογιστεί με την διαίρεση του αθροίσματος των μετρήσεων προς το πλήθος των αισθητήρων που συμμετείχαν στη συνάθροιση. Συνεπώς, τα μηνύματα που στέλνονται από έναν κόμβο προς τον πατέρα του, Distribution Messages, περιέχουν τρία ορίσματα όπως υποδηλώνεται και από τις σχέσεις παρακάτω.

$$avg = \frac{sum}{count} \ \& \ max$$

Παραθέτουμε την δήλωση του structure για τα Distribution Messages.

```
typedef nx_struct DistrMsg{
    nx_uint16_t count;
    nx_uint16_t sum;
    nx_uint16_t max;
} DistrMsg;

/**Initially tried to create multiple arrays in SRTreeC.nc for each element
Better this way less memory is consumed*/
typedef nx_struct ChildDistrMsg{
    //nx_uint16_t parentID;
    nx_uint16_t senderID;
    nx_uint16_t count;
    nx_uint16_t sum;
    nx_uint16_t max;
} ChildDistrMsg;

#endif
```

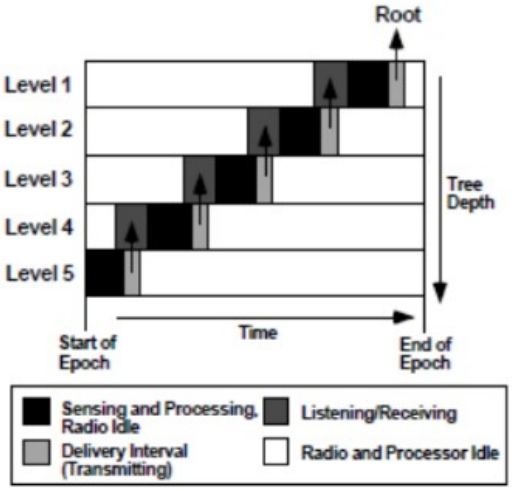
Σχήμα 4: Δήλωση DistrMsg και ChildDistrMsg

Αναφορικά με την διατήρηση των παλιών τιμών των παιδιών είναι χρήσιμο σε κάθε στάδιο να κρατάει κάθε κόμβος πληροφορίας από τις τελευταίες τιμές που έλαβε από τα παιδιά του κυρίως για να αντιμετωπίσει την περίπτωση της απώλειας μηνυμάτων. Αυτό το υλοποιήσαμε δημιουργώντας έναν πίνακα για κάθε κόμβο θεωρώντας ότι ο μέγιστος αριθμός παιδιών που μπορεί να έχει ένας αισθητήρας είναι 20. Αυτή η θεώρηση έγινε για να δώσουμε στον πίνακα τέτοιο αριθμό θέσεων καθώς στο tinyOS δεν υπάρχει δυνατότητα δυναμικής δέσμευσης μνήμης. Παραθέτουμε εδώ τα σημείο του κώδικα όπου γίνεται η ανάθεση τιμών.

```
/**Check if received a message*/
if(len == sizeof(DistrMsg))
{

    DistrMsg* mr = (DistrMsg*) (call DistrPacket.getPayload(&radioDistrRecPkt,len));

    /** Add new child to cache*/
    for(i=0; i< MAX_CHILDREN ; i++){
        if(source == childrenArray[i].senderID || childrenArray[i].senderID == 0){
            childrenArray[i].senderID = source;
            childrenArray[i].count = mr->count;
            childrenArray[i].sum = mr->sum;
            childrenArray[i].max = mr->max;
            break;

        }
        /**
            Still haven't found from the children array the right
            destination
        */

    }

}
else
{
    dbg("SRTreeC","receiveDistrTask():Empty message!!! \n");
    return;
}
```

Σχήμα 5: Διατήρηση τιμών σε Child Cache

Ένα αρκετά σημαντικό κομμάτι της υλοποίησης ήταν ο συγχρονισμός των timers προκειμένου να μεταδίδουν οι κόμβοι περιοδικά επερωτήσεις σύμφωνα με το TAG(βλ. Σχήμα 6) και να υπάρχουν οι λιγότερες δυνατές συγκρούσεις.



Σχήμα 6: Υλοποίηση συγχρονισμού σύμφωνα με TAG

Στο TAG πρώτα μεταδίδουν τα παιδιά και μετά οι γονείς, η ροή της πληροφορίας γίνεται ομοιόμορφα προς τα πάνω έως ότου φτάσει στην ρίζα. Για να πετύχουμε την λογική του TAG ρυθμίσαμε το πότε θα χτυπήσει για πρώτη φορά ο μετρητής κάθε κόμβου με βάση το επίπεδο στο οποίο βρίσκεται. Επίσης, για τον ορισμό αυτού του χρόνου προκειμένου να ελαττώσουμε όσο γίνεται την σύγκρουση μηνυμάτων γειτονικών κόμβων λάβαμε υπόψη και το ID του κάθε κόμβου, ούτως ώστε να μην μεταδώσουν δύο κόμβοι του ίδιου επιπέδου την ίδια στιγμή. Παραθέτουμε εδώ το κομμάτι του κώδικα που ρυθμίζουμε τον συγχρονισμό.

```
event void RoutingComplTimer.fired(){


    slotTime = EPOCH/(MAX_DEPTH+1);
    subSlotSplit = (MAX_DEPTH);

    subSlotChoose = (MAX_DEPTH - curdepth);


    /**
        Synchronize timers. Divide first the epoch in
        slots as defined by TAG, based on max depth.Then,
        devide every slot in sub-slots based again on max_depth
        and current depth and use TOS_NODE_ID to avoid collision
        between messages. *25 was used after externsive testing.
        Also tried to multiply with random value but was not
        effective in some cases.
    */


    /**
        Altered synchronization. The previous version would lose
        1 epoch due to time constraints and had issues when max_depth = curdepth.
        What changed is that we added 1 extra slot and subslot at each epoch so that we are
        done before the time elapses.
    */

    /*
        The epoch time has delay at the beginning as starting time is defined by max depth and
        curdepth. Now maxdepth is considered to be 14. In some cases for smaller topology files
        we could assign max depth to a smaller value ex. 4. But even in that case the time that is
        "wasted" at the beginning is compensated and all 15 epochs run as requested.
    */

    startPer =  (slotTime / subSlotSplit * subSlotChoose) + TOS_NODE_ID * 25;


    //dbg("SRTreeC", "START %d\n", startPer);

    call DistrMsgTimer.startPeriodicAt(startPer, EPOCH);
}
```

Σχήμα 7: Υλοποίηση συγχρονισμού

Ο συγχρονισμός μας γίνεται βάσει της σχέσης που φαίνεται και στο σχήμα 7

$$startPer = (\frac{slotTime}{subSlotSplit} * subSlotChoose) + TOS\_NODE\_ID * 25$$

Όπου έχουμε:

$slotTime : \frac{EPOCH}{MAX\_DEPTH + 1}$

$subSlotSplit : MAX\_DEPTH$

$subSlotChoose : (MAX\_DEPTH - curdepth)$

Από τις ονομασίες και μόνο των μεταβλητών δίνεται μια εικόνα για το τι προσπαθούμε να υλοποιήσουμε αφού αρχικά διαιρούμε την εποχή σε έναν αριθμό από slots(slotTime) βάσει του Maxdepth(υποθέσαμε ότι η τιμή του είναι 14) ο οποίος είναι ο χρόνος που πρέπει να έχει μεταφερθεί όλη η πληροφορία από ένα επίπεδο στο πιο πάνω επίπεδο του δέντρου. Κάθε τέτοιο slot χωρίζεται σε sub-slots όπου περιέχουν κάθε κόμβο του επιπέδου αυτού και πάλι βάσει του Maxdepth. Το ID κάθε κόμβου πολλαπλασιάζεται με μία σταθερά ώστε όλοι οι κόμβοι του ίδιου επιπέδου να έχουν διαφορά ασφαλείας αναφορικά με στο πότε θα χτυπήσει για πρώτη φορά ο μετρητής του καθένα και να ελαχιστοποιείται έτσι η απώλεια μηνυμάτων.

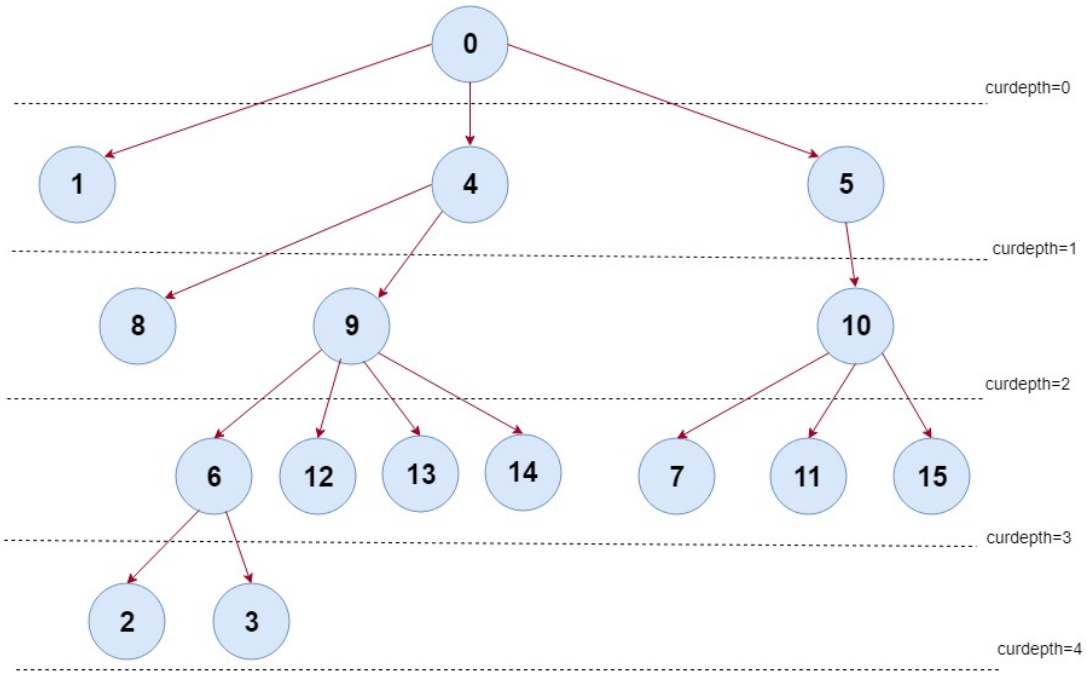
Αποτελέσματα

Σε αυτό το σημείο θα επιβεβαιώσουμε την ορθή λειτουργία του δικτύου. Εκτελώντας την προσομοίωση με πλέγμα $4 * 4$, δηλαδή 16 αισθητήρες το αποτέλεσμα του Routing ήταν το παρακάτω.

```
0:0:10.000000010 DEBUG (0): ###########START ROUTING###############

0:0:10.199600224 DEBUG (5): NodeID= 5 : curdepth= 1 , parentID= 0
0:0:10.199600224 DEBUG (4): NodeID= 4 : curdepth= 1 , parentID= 0
0:0:10.199600224 DEBUG (1): NodeID= 1 : curdepth= 1 , parentID= 0
0:0:10.396743791 DEBUG (9): NodeID= 9 : curdepth= 2 , parentID= 4
0:0:10.396743791 DEBUG (8): NodeID= 8 : curdepth= 2 , parentID= 4
0:0:10.403732276 DEBUG (10): NodeID= 10 : curdepth= 2 , parentID= 5
0:0:10.593994158 DEBUG (14): NodeID= 14 : curdepth= 3 , parentID= 9
0:0:10.593994158 DEBUG (13): NodeID= 13 : curdepth= 3 , parentID= 9
0:0:10.593994158 DEBUG (12): NodeID= 12 : curdepth= 3 , parentID= 9
0:0:10.593994158 DEBUG (6): NodeID= 6 : curdepth= 3 , parentID= 9
0:0:10.603286747 DEBUG (15): NodeID= 15 : curdepth= 3 , parentID= 10
0:0:10.603286747 DEBUG (11): NodeID= 11 : curdepth= 3 , parentID= 10
0:0:10.603286747 DEBUG (7): NodeID= 7 : curdepth= 3 , parentID= 10
0:0:10.791915907 DEBUG (3): NodeID= 3 : curdepth= 4 , parentID= 6
0:0:10.791915907 DEBUG (2): NodeID= 2 : curdepth= 4 , parentID= 6
```

Σχήμα 8: Δεδομένα κόμβων μετά από Routing

Από εδώ μπορούμε να αντλήσουμε όλη την απαραίτητη πληροφορίας για τον σχεδιασμό του δέντρου που προκύπτει. Έτσι, το δέντρο θα έχει την εξής κατανομή.



Σχήμα 9: Δενδρική κατανομή κόμβων

Έχοντας πλήρη εικόνα της τοπολογίας μπορούμε να ελέγξουμε αν τα αποτελέσματα στους ενδιάμεσους κόμβους και το τελικό αποτέλεσμα είναι σωστά και προκύπτουν με την λειτουργία του TAG.

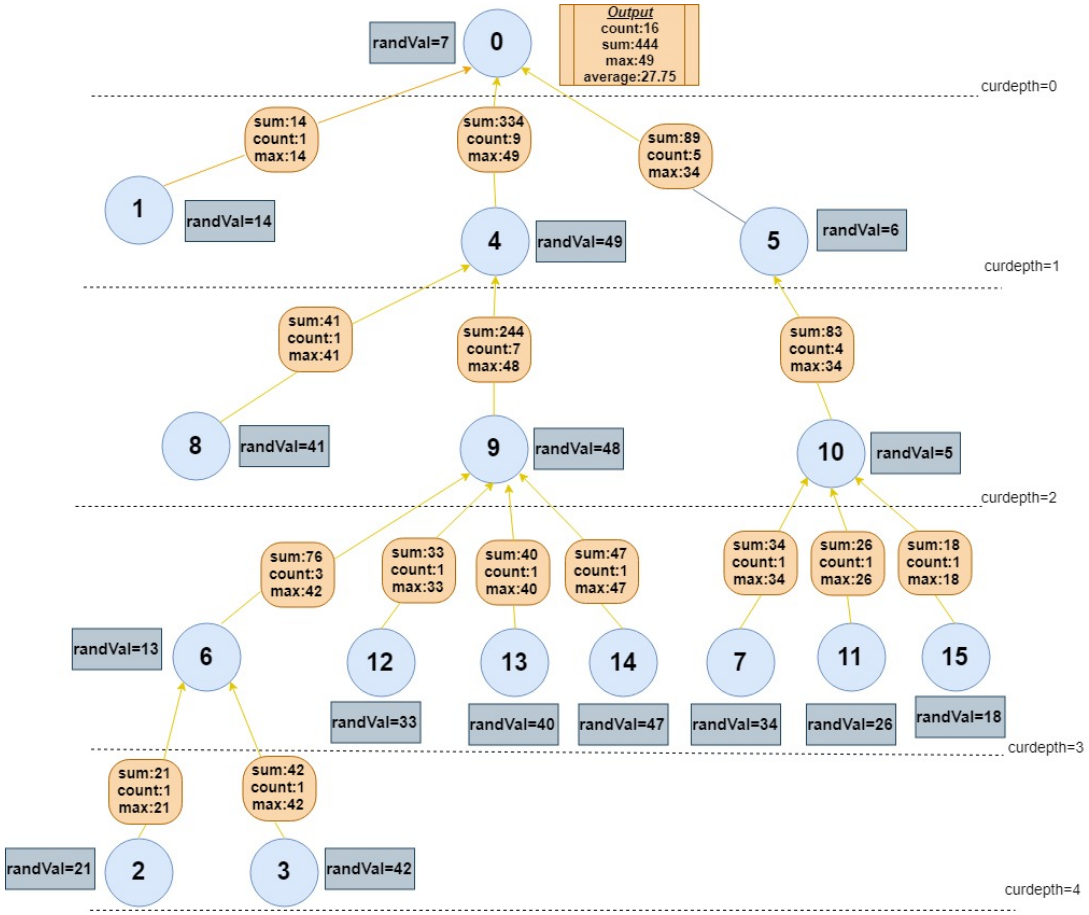
Στην εικόνα 10 παραθέτουμε τα αποτελέσματα από την πρώτη εποχή.

```
0:1:11.425781260 DEBUG (2): Random value generated 21
0:1:11.425781260 DEBUG (2): Node: 2 , Parent: 6, Sum: 21, count: 1, max: 21 , depth: 4
0:1:11.450195322 DEBUG (3): Random value generated 42
0:1:11.450195322 DEBUG (3): Node: 3 , Parent: 6, Sum: 42, count: 1, max: 42 , depth: 4
0:1:11.801757823 DEBUG (6): Random value generated 13
0:1:11.801757823 DEBUG (6): Node: 6 , Parent: 9, Sum: 76, count: 3, max: 42 , depth: 3
0:1:11.826171885 DEBUG (7): Random value generated 34
0:1:11.826171885 DEBUG (7): Node: 7 , Parent: 10, Sum: 34, count: 1, max: 34 , depth: 3
0:1:11.923828136 DEBUG (11): Random value generated 26
0:1:11.923828136 DEBUG (11): Node: 11 , Parent: 10, Sum: 26, count: 1, max: 26 , depth: 3
0:1:11.948242198 DEBUG (12): Random value generated 33
0:1:11.948242198 DEBUG (12): Node: 12 , Parent: 9, Sum: 33, count: 1, max: 33 , depth: 3
0:1:11.972656261 DEBUG (13): Random value generated 40
0:1:11.972656261 DEBUG (13): Node: 13 , Parent: 9, Sum: 40, count: 1, max: 40 , depth: 3
0:1:11.997070323 DEBUG (14): Random value generated 47
0:1:11.997070323 DEBUG (14): Node: 14 , Parent: 9, Sum: 47, count: 1, max: 47 , depth: 3
0:1:12.021484386 DEBUG (15): Random value generated 18
0:1:12.021484386 DEBUG (15): Node: 15 , Parent: 10, Sum: 18, count: 1, max: 18 , depth: 3
0:1:12.128906260 DEBUG (8): Random value generated 41
0:1:12.128906260 DEBUG (8): Node: 8 , Parent: 4, Sum: 41, count: 1, max: 41 , depth: 2
0:1:12.153320323 DEBUG (9): Random value generated 48
0:1:12.153320323 DEBUG (9): Node: 9 , Parent: 4, Sum: 244, count: 7, max: 48 , depth: 2
0:1:12.177734386 DEBUG (10): Random value generated 5
0:1:12.177734386 DEBUG (10): Node: 10 , Parent: 5, Sum: 83, count: 4, max: 34 , depth: 2
0:1:12.236328135 DEBUG (1): Random value generated 14
0:1:12.236328135 DEBUG (1): Node: 1 , Parent: 0, Sum: 14, count: 1, max: 14 , depth: 1
0:1:12.309570322 DEBUG (4): Random value generated 49
0:1:12.309570322 DEBUG (4): Node: 4 , Parent: 0, Sum: 334, count: 9, max: 49 , depth: 1
0:1:12.333984385 DEBUG (5): Random value generated 6
0:1:12.333984385 DEBUG (5): Node: 5 , Parent: 0, Sum: 89, count: 5, max: 34 , depth: 1
0:1:12.490234385 DEBUG (0): Random value generated 7
0:1:12.490234385 DEBUG (0):

##########################Epoch 1 completed#####################
0:1:12.490234385 DEBUG (0): #### Output:
0:1:12.490234385 DEBUG (0): #### [count] = 16
0:1:12.490234385 DEBUG (0): #### [sum] = 444
0:1:12.490234385 DEBUG (0): #### [max] = 49
0:1:12.490234385 DEBUG (0): #### [AVG] = 27.750000
```

Σχήμα 10: Δεδομένα συναθροίσεων σε μια τυχαία εποχή

Παρατηρούμε ότι χρονολογικά πρώτα μεταδίδουν οι κόμβοι 2 και 3 που είναι οι κόμβοι του επιπέδου 4. Στην συνέχεια μεταδίδουν οι κόμβοι του επιπέδου 3 προς τους γονείς τους . Ο κόμβος 6 που έχει παιδιά μεταδίδει τις τιμές που προκύπτουν μετά από συνάθροιση της μέτρησης του και αυτές των παιδιών του. Οι πληροφορίες επεξεργάζονται σταδιακά και στέλνονται προς τα πάνω μέχρι να φτάσουν στην ρίζα, δηλαδή στον κόμβο 0, ο οποίος θα υπολογίσει και θα εμφανίσει το τελικό αποτέλεσμα της εποχής. Σε αυτό το σημείο να επισημάνουμε ότι τα αποτελέσματα της εκτέλεσης καταγράφονται σε ένα αρχείο logfile.txt και δεν εμφανίζονται στην οθόνη του τερματικού.

Η διαδικασία της συγκεκριμένης εποχής αναπαρίσταται στην παρακάτω εικόνα.



Σχήμα 11: Συναθροίσεις σε δενδρική δομή

Παρατηρούμε ότι κάθε κόμβος χωρίς παιδιά στέλνει σαν άθροισμα και μέγιστη τιμή την μέτρηση που έχει λάβει. Επίσης, για τους κόμβους χωρίς παιδιά το count που στέλνουν είναι 1 καθώς αναφέρονται σε μετρήσεις για έναν μόνο κόμβο. Οι κόμβοι με παιδιά συναθροίζουν τις τιμές και τις στέλνουν πιο πάνω. Για παράδειγμα, ο κόμβος 9 υπολογίζει:

$$SUM = randVal + sum6 + sum12 + sum13 + sum14 = 48 + 76 + 33 + 40 + 47 = 244$$

$$COUNT = 1 + count6 + count12 + count13 + count14 = 1 + 3 + 1 + 1 + 1 = 7$$

$$MAX = max(randVal, max6, max12, max13, max14 = max(48, 42, 33, 40, 47) = 48$$

Διαπιστώσαμε λοιπόν ότι το δίκτυο μας λειτουργεί όπως καθορίζει το TAG ενώ παρατηρήσαμε ότι ακόμα και για μεγαλύτερο grid δεν εμφανίζονται συχνά απώλειες μηνυμάτων που μας οδηγεί στο συμπέρασμα ότι πραγματοποιήθηκε επιτυχώς η διαδικασία του συγχρονισμού.

Καταμερισμός Εργασιών

Στο μεγαλύτερο κομμάτι της άσκησης υπήρξε δίκαιος καταμερισμός εργασιών και συνεργασία στα στάδια της λύσης και από τα δύο μέλη της ομάδας.Το βοηθητικό πρόγραμμα υλοποιήθηκε από τον φοιτητή Στέφανο Καλογεράκη.

Πρόγραμμα φάσης 1: $https://github.com/skalogerakis/Test\_SNR$