Εργασία TinyOS

1η Φάση

Ημερομηνία Παράδοσης : 24 Νοεμβρίου 2017

Επεξεργασία και Διαχείριση Δεδομένων σε Δίκτυα Αισθητήρων

Διδάσκων : Α. Δεληγιαννάκης

Σταματάκης Γεώργιος - 2013030154

Σκυβαλάκης Κωνσταντίνος - 2013030034

GitHub exercise repo : `https://github.com/gstamatakis/SensorsTinyOS`

Βοηθητικό Πρόγραμμα

Για να φτιάξουμε τα δικά μας *topology.txt* αρχεία γράψαμε έναν κώδικα σε Python ο οποίος δέχεται ώς ορίσματα το **D** και το **r** τα οποία υποδηλώνουν το μέγεθος (dimension) του grid που θα δημιουργηθεί και την εμβέλεια (range) των αισθητήρων αντίστοιχα.

Αρχικά, δημιουργούμε το grid ώς έναν $D \times D$ πίνακα που περιέχει τιμές στο σύνολο $\{0, 1, ..., D^2 - 1\}$ οι οποίες τιμές αντιστοιχούν στο TOS_NODE_ID του κάθε κόμβου. Όπως ζητείται από την εκφώνηση, με το παρακάτω arrangement ο κάθε κόμβος j θα ανήκει στη γραμμή j/D και στη στήλη $j\%D$. Η μορφή του grid για $D = 4$ θα είναι η παρακάτω.

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

Αφότου δημιουργήσαμε το grid μας δημιουργήσαμε και μια συνάρτηση που υπολογίζει τους γείτονες κάθε κόμβου την οποία ονομάσαμε $find\_neighbors(m, i, j, r, d)$ και της οποίας τα ορίσματα αντιστοιχούν σε :

- m : ο πίνακας του grid που φτιάξαμε πριν
- i : το index γραμμής του κόμβου του οποίου τους γείτονες αναζητάμε
- j : το index στήλης του κόμβου του οποίου τους γείτονες αναζητάμε
- r : η εμβέλεια των αισθητήρων
- d : η διάσταση του grid

Για να αποφανθεί η συνάρτηση αν κάποιος άλλος κόμβος είναι εντός εμβέλειας του αισθητήρα που εξετάζουμε και κατά συνέπεια να τον προσθέσει ως γείτονα λαμβάνει την ευκλείδεια απόσταση τους (με βάση τα index) ώς εξής :

$$d(p, q) = \sqrt{(i - x)^2 + (j - y)^2} \tag{1}$$

Όπου p, ο κόμβος του οποίου τους γείτονες αναζητάμε και q ο κόμβος με τον οποίο ελέγχουμε σε κάθε χρονική στιγμή αν είναι γείτονας του p. Τα (x, y) είναι τα αντίστοιχα indices του κόμβου q.

Τέλος, αφού το πρόγραμμα έχει υπολογίσει τους γείτονες όλων των κόμβων, τους γράφει σε ένα αρχείο *topology.txt* το οποίο και χρησιμοποιούμε στο simulation μας.

Πρόγραμμα 1

Αφαίρεση κώδικα

Κύριος στόχος μας στην ανάπτυξη του κώδικα ήταν η οικονομία στο μέγεθος και στην αποστολή μηνυμάτων κατά το βέλτιστο δυνατό, μιας και η λειτουργία του πομποδέκτη του αισθητήρα κοστίζει σε μπαταρία πολύ περισσότερο απ΄ότι μερικές παραπάνω γραμμές κώδικα.

Αρχικά, αφαιρέσαμε ότι κομμάτια κώδικα δεν μας χρειάζονταν, όπως κομμάτια κώδικα που δεν σχετίζονται με την υλοποίηση του TAG καθώς επίσης και κομμάτια κώδικα τα οποία δεν πρόκειται να εκτελεστούν ποτέ. Πιο συγκεκριμένα, αφαιρέσαμε τα modules του serial (interfaces, events, timers, κλπ.) που σχετίζονται με επικοινωνία του αισθητήρα μέσω σειριακής θύρας κάτι το οποίο δε μας χρειάζεται. Ακόμη, αφαιρέσαμε ότι σχετίζεται με την λειτουργία των LEDs (events, timers, κλπ.) μιας και σε simulation mode δεν πρόκειται επίσης να μας χρειαστούν.

Σημαντική αφαίρεση κομματιού κώδικα ήταν τα μηνύματα notify και ότι σχετίζεται με αυτά όπως (interfaces, events, queues, κλπ.). Ο λόγος για τον οποίο τα αφαιρέσαμε είναι διότι είμαστε σε simulation mode και γνωρίζουμε ότι κανένας από τους κόμβους δεν πρόκειται να πεθάνει. Μιας και κανένας κόμβος δεν πεθαίνει δε χρειάζεται να ξαναγίνει re-routing του δέντρου και αλλαγή πατεράδων. Ως συνέπεια, οι κόμβοι δε χρειάζεται να στέλνουν notify μηνύματα στους πατεράδες τους. Τέλος, ένας κόμβος καταλαβαίνει ότι ένας άλλος κόμβος είναι παιδί του από το μήνυμα των μετρήσεων που του στέλνει, καθώς κάθε μήνυμα στο TinyOS έχει στοιχεία του αποστολέα στην κεφαλήδα του πακέτου και ο παραλήπτης κόμβος διατηρεί μια λίστα με τα παιδιά του από τη διαδικασία του routing που γίνεται στην αρχή, οπότε και αναγνωρίζει αν το μήνυμα είναι από κάποιο παιδί του.

Προσθήκη κώδικα

Στόχος μας ήταν να ελαχιστοποιήσουμε τα μηνύματα που χάνονταν και να προσπαθήσουμε παρόλο τις απώλειες να διατηρήσουμε ένα σχετικά σωστό τελικό αποτέλεσμα στη ρίζα. Ο κώδικας που προσθέσαμε είναι για τα εξής κομμάτια :

- Παραγωγή τυχαίων readings στο διάστημα $[K, K + 20]$, όπου $K = TOS\_NODE\_ID$
- Αποστολή και λήψη των query μηνυμάτων
- Κατασκευή μικρής μνήμης Cache για κάθε κόμβο
- Συγχρονισμός timers

Πιο συγκεκριμένα, έχουμε θέσει τον κάθε κόμβο να παράγει μια μέτρηση κάθε 60sec που είναι και η διάρκεια της εποχής. Αρχικά, για να σιγουρευτούμε ότι όλα πήγαιναν καλά και μόνο για λόγους δοκιμής, είχαμε θέσει η μέτρηση που παρήγαγαν οι κόμβοι να είναι ίση με 10 και προφανώς ως αποτέλεσμα περιμέναμε να δούμε διασπορά(var) ίση με 0 και μέση τιμή (avg) ίση με 10. Αφού σιγουρευτήκαμε ότι το routing είχε γίνει σωστά, αφήσαμε τους κόμβους να παράγουν τυχαίες τιμές στο προκαθορισμένο διάστημα.

Στη συνέχεια, δημιουργήσαμε τα δικά μας Active Messages για την ανταλλαγή επερωτήσεων (queries) στους κόμβους του δέντρου και τη συλλογή των δεδομένων με βάση το TAG. Δημιουργήσαμε τα απαραίτητα tasks και timers για την επίτευξη αυτής της διαδικασίας. Η πληροφορία που προσθέτουμε εμείς στα routing μηνύματα ειναι το senderID και το curdepth, δηλαδή το επίπεδο του κόμβου που στέλνει το μήνυμα. Στα query μηνύματα στέλνουμε το count που είναι το πλήθος των κόμβων που συμμετέχουν μέχρι στιγμής στο aggregation, το sum που είναι το συνολικό άθροισμα μέχρι στιγμής και τέλος το sum_squared το οποίο το χρησιμοποιούμε για τον υπολογισμό του variance. Οι παρακάτω τύποι δείχνουν αναλυτικότερα τον υπολογισμούς :

$$avg = \frac{sum}{count}$$
$$var = \frac{sum\_squared}{count} - avg^2 \tag{2}$$

Αυτοί οι υπολογισμοί πραγματοποιούνται μόνο από την ρίζα ενώ όλοι οι υπόλοιποι κόμβοι απλώς προωθούν ένα μήνυμα στον πατέρα τους με τις τιμές που έχουν συλλέξει μέχρι στιγμής.

Παρατηρούμε ότι πάντοτε, ειδικά για μεγάλες τιμές του D όπου το πλήθος των κόμβων αυξάνεται αισθητά, χάνονται πακέτα. Για να ελαχιστοποιήσουμε αυτές τις απώλειες δημιουργήσαμε μια μικρή μνήμη Cache σε κάθε κόμβο η οποία κρατάει τις τελευταίες μετρήσεις των παιδιών του και τις χρησιμοποιεί όταν χαθεί κάποιο μήνυμα από αυτά. Έτσι, παρόλο τις απώλειες επιτυγχάνεται μεγαλύτερη ακρίβεια τελικών αποτελεσμάτων.

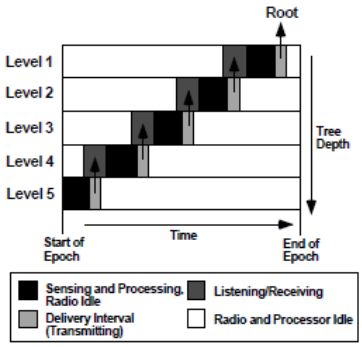
Τέλος, ένα αρκετά challenging task ήταν ο κατάλληλος συγχρονισμός των timers. Η σωστή αποστολή και λήψη μηνυμάτων απαιτεί καλό συγχρονισμό διότι αν ένας κόμβος δεν στέλνει τις σωστές χρονικές στιγμές (περιοδικά) το μήνυμα του υπάρχει κίνδυνος να χαθούν μηνύματα και να αλοιωθούν τα τελικά αποτελέσματα.

$$start = -MUL * (curdepth * TFP + TVM * TNI + rand()\%10 * MUL);$$
$$start = -MUL * ((curdepth - 1) * TFP + TVM * TNI + rand()\%10 * MUL); \tag{3}$$

Όπου :

$$TFP : TIMER\_FAST\_PERIOD = 200$$
$$TVM : TIMER\_VFAST\_MILI = EPOCH/1000 \tag{4}$$
$$TNI : TOS\_NODE\_ID \in \{1, 2, ...\}$$

Οι παραπάνω τύποι μας δίνουν τον χρόνο εκκίνησης των μετρητών όλων των κόμβων εκτός του root για δυο διαφορετικές περιπτώσεις. Η πρώτη περίπτωση αφορά το ενδεχόμενο που βρισκόμαστε ακόμα στα πρώτα στάδια του προγράμματος που οι κόμβοι δεν έχουν όλοι αποκτήσει πατέρα (routing), ενώ η δεύτερη περίπτωση αφορά το ενδεχόμενο ο κόμβος να έχει πατέρα οπότε και αλλάζει ελαφρώς ο τύπος. Σημαντικό επίσης είναι να αναφέρουμε ότι οι περιοδικοί timers με αρνητική τιμή θα έχουν διάφορα σειριακά events μέχρις ωτου κάνουν "catch up" στην τωρινή κατάσταση και επίσης να πούμε ότι όλοι οι timers αναδιπλώνονται. Ακόμη, κλιμακώνουμε τον χρόνο start για να γινει μικροτερη η πιθανοτητα να χαθει μήνυμα. Τέλος, η προσθήκη της rand() είναι μια ακόμη προσπάθεια για ελαχιστοποίηση συγκρούσεων. Αρκετά βοηθητικό στον συγχρονισμό ήταν το παρακάτω σχήμα του TAG.



Σχήμα 1: Partial state records flowing up the tree during an epoch.

Screenshots Κώδικα

Παρακάτω παραθέτουμε μερικά κομμάτια κώδικα τα οποία θεωρούμε σημαντικά.

```
#ifndef SIMPLEROUTINGTREE_H
#define SIMPLEROUTINGTREE_H

#define MAX_CHILDREN 25

enum {
    SENDER_QUEUE_SIZE = 5,
    RECEIVER_QUEUE_SIZE = 3,
    FAKE_READINGS = 0,
    AM_QUERYMSG = 30,
    AM_ROUTINGMSG = 22,
    MUL = 3                     //Scale up/down start time
    EPOCH = 60000,
    TIMER_FAST_PERIOD = 200,
    TIMER_VFAST_MILI = EPOCH / 1000
};

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

typedef struct ChildValue {
    uint16_t senderID;
    uint16_t sum;
    uint16_t sum_squared;
    uint8_t count;
} ChildVal;

/**
 * Avg = sum/count
 * Var = (sum_squares/count) - (Avg)^2
 */
typedef nx_struct QueryMsg{
        nx_uint16_t sum;
        nx_uint8_t count;
        nx_uint16_t sum_squared;
}
QueryMsg;

#endif
```

Σχήμα 2: SimpleRoutingTree.h library file.

Στο παραπάνω screenshot φαίνονται οι αλλαγές και οι προσθήκες που έχουμε κάνει στο αρχείο SimpleRoutingTree.h όπως ότι θέσαμε ένα μέγιστο αριθμό παιδιών, προσθέσαμε μια λίστα για τις τιμές των παιδιών του κάθε κόμβου και μια για το μήνυμα του ίδιου, αφαιρώντας τις λίστες για τα notify μηνύματα όπως άλλωστε έχουμε πει και σε προηγούμενο στάδιο της αναφοράς.

```
event void ReadingTimer.fired()
{
    reading = FAKE_READINGS ? 10 : rand() % 21 + TOS_NODE_ID;
    dbg("Values", "\nreading: %u", reading);
}
```

Σχήμα 3: Random value generation every time the ReadingTimer is fired.

```
//Add the readings of THIS sensor.
atomic{
        sum = 0;
        sum_squared = 0;
        count = 0;

        for (i=0;i<MAX_CHILDREN && myChildren[i].senderID != 0;i++){
            sum += myChildren[i].sum;
            sum_squared += myChildren[i].sum_squared;
            count += myChildren[i].count;
        };

        sum += reading;
        sum_squared += pow(reading, 2);
        count += 1;
}


//If root print the results
//else forward to the parent
if(TOS_NODE_ID == 0){
    atomic{
            avg = sum / (float) count;
            var = sum_squared / (float) count - pow(avg, 2);
    };
    roundCounter++;

    dbg("Readings", "EPOCH:%u-> sum: %u | count: %u | sum_squared: %u | avg: %.2f | var: %.2f\n",
        roundCounter, sum, count, sum_squared, avg, var);
    dbg("OUT","%.2f %.2f\n",avg,var);

    dbg("SRTreeC", "## ROUND %u ## \n", roundCounter);
} else {
    //Load the values and send it to your father.
    atomic{
            qpkt->sum = sum;
            qpkt->sum_squared = sum_squared;
            qpkt->count = count;
    };
```

Σχήμα 4: Query message contents and computation of aggregate functions.

```
task void receiveQueryTask()
{
    uint8_t len;
    uint16_t msource;
    message_t queryRcvPkt;
    uint8_t i;

    queryRcvPkt = call MsgReceiveQueue.dequeue();
    len = call QueryPacket.payloadLength(&queryRcvPkt);
    msource = call QueryAMPacket.source(&queryRcvPkt);  //Child that sent the message

    if(len == sizeof(QueryMsg)){
        QueryMsg *mpkts = (QueryMsg *) (call QueryPacket.getPayload(&queryRcvPkt, len));

        //Add new child to cache
        for (i = 0; i < MAX_CHILDREN; i++) {
            if (myChildren[i].senderID == 0 || myChildren[i].senderID == msource) {
                myChildren[i].senderID = msource;
                myChildren[i].sum = mpkts->sum;
                myChildren[i].sum_squared = mpkts->sum_squared;
                myChildren[i].count = mpkts->count;
                break;
            }
        }
    }
```

Σχήμα 5: Cache creation and latest values storage.

```
 event void RoutingMsgTimer.fired(){
        message_t tmp;

        RoutingMsg* mrpkt;
PRINTFDBG_MODE
        printfflush();
        printf("RoutingMsgTimer fired!  radioBusy = %s \n",(RoutingSendBusy)?"True":"False");
        printfflush();

        if (TOS_NODE_ID == 0){
    call MsgTimer.startPeriodicAt(0, EPOCH); // Cancel previous submission and query periodically
        }

        if(call RoutingSendQueue.full()){
PRINTFDBG_MODE
                printf("RoutingSendQueue is FULL!!! \n");
                printfflush();

                return;
        }

        mrpkt = (RoutingMsg*) (call RoutingPacket.getPayload(&tmp, sizeof(RoutingMsg)));

        if(mrpkt==NULL){
PRINTFDBG_MODE
                printf("RoutingMsgTimer.fired(): No valid payload... \n");
                printfflush();

                return;
        }

        atomic{
    mrpkt->senderID=TOS_NODE_ID;
    mrpkt->depth = curdepth;
        }

PRINTFDBG_MODE
        printf("NodeID= %d : RoutingMsg sending...!!!! \n", TOS_NODE_ID);
        printfflush();

        call RoutingAMPacket.setDestination(&tmp, AM_BROADCAST_ADDR);
        call RoutingPacket.setPayloadLength(&tmp, sizeof(RoutingMsg));
```

Σχήμα 6: Routing message contents.

Αποτελέσματα

Αρχικά, τροποποιήσαμε το αρχείο mySimulation.py και θέσαμε ότι στο simulation που ακολουθεί θα έχουμε 25 κόμβους ($D = 5$) με range=1.5. Επίσης, να σημειώσουμε ότι έχουμε αλλάξει με δικά μας κάποια από τα debug μηνύματα που τυπώνονται για λόγους καλύτερης κατανόησης της ροής του output. Παραθέτουμε μαζί με τα αρχεία της εργασίας ένα αρχείο με όνομα logfile_sim_25.txt το οποίο περιλαμβάνει τα αποτελέσματα της εκτέλεσης τα οποία σχολιάζουμε παρακάτω.

Στην αρχή του $logfile$ υπαρχουν αρκετά όμοια μηνύματα που έχουν να κάνουν σχέση με το routing των κόμβων. Παρακάτω όμως, συναντάμε μηνύματα όπως τα εξής :

$EPOCH : 2 -> sum : 399|count : 20|sum\_squared : 9569|avg : 19.95|var : 80.45$

Τέτοιου τύπου μηνύματα μας δηλώνουν το τέλος μιας εποχής και συνάμα τι αποτελέσματα είχαμε σε αυτόν τον γύρο. Μιας και οι κόμβοι παίρνουν τυχαίες τιμές στο διάστημα $[K, K + 20]$ μπορούμε με χρήση πιθανοτήτων να υπολογίσουμε το αναμενόμενο αποτέλεσμα και να επαληθεύσουμε κατά πόσο είμαστε σωστοί.

Η μέση τιμή που περιμένουμε να έχει ένας κόμβος X_K είναι η εξής :

$$E[X_K] = \sum_{i=K}^{K+20} p_{X_K}(i) \cdot i = \frac{1}{20} \sum_{i=K}^{K+20} i = \frac{K^2 + 41K + 420}{40}$$

Συνολικά για ένα δίκτυο N κόμβων περιμένουμε να έχουμε τελική μέση ίση με την παρακάτω :

$$E[X] = E[X_1 + X_2 + ...X_N] = \frac{1}{N} \sum_{i=1}^{N} E[X_i] = \frac{1}{N} \sum_{i=1}^{N} \frac{i^2 + 41i + 420}{40}$$

Τέλος, έχουμε και μηνύματα της παρακάτω δομής.

$MsgTimerfired!$
$MsgTimer.fired() : Taskposted.$
$sendQueryTask() : QueryAMSend.sendsuccess!$
$QueryReceive.receivefrom\ i$
$QueryAMSend.sendDone : True$

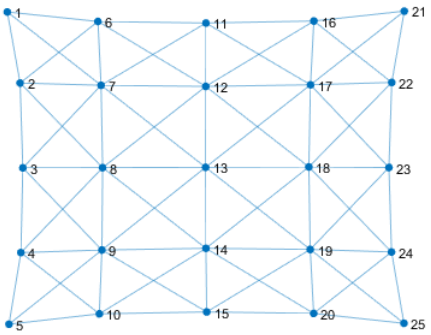
Τέτοιου τύπου μηνύματα μας λένε ότι αρχικά χτυπάει ο timer για να ετοιμάσουμε και να στείλουμε το μήνυμα με τις μετρήσεις του κόμβου και στη συνέχεια κάνουμε post το task της αποστολής του μηνύματος βάζοντας επιτυχώς το μήνυμα στην ουρά αποστολής. Στη συνέχεια, ο πατέρας του κόμβου, i, λαμβάνει το μήνυμα από το παιδί του και κάπου εκεί ολοκληρώνεται η διαδικασία αποστολής. Αφού ο πατέρας λάβει το μήνυμα το ¨ξεπακετάρει' προσθέτει με τη σειρά του τις δικές του μετρήσεις και των υπόλοιπων παιδιών του με σωστό τρόπο και το προωθεί παραπάνω κ.ο.κ. .

Μέσα στο αρχείο $logfile\_sim\_25.txt$ φαίνονται κάποιες γραμμές όπως αυτές στην παρακάτω εικόνα.

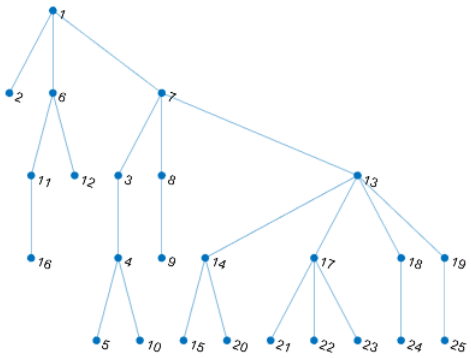
```
0:0:10.204391456 DEBUG (6): ### RoutingReceive.receive() end #####
0:0:10.204391456 DEBUG (5): ### RoutingReceive.receive() end #####
0:0:10.204391456 DEBUG (1): ### RoutingReceive.receive() end #####
0:0:10.204391466 DEBUG (6): PARENT: 0
0:0:10.204391466 DEBUG (5): PARENT: 0
0:0:10.204391466 DEBUG (1): PARENT: 0
```

Σχήμα 7: Routing messages.

Αυτό που μας λέει η παραπάνω εικόνα στις τρεις πρώτες γραμμές είναι ότι οι κόμβοι 6,5 και 1 έλαβαν μήνυμα για να κάνουν routing, δηλαδή να διαλέξουν τον πατέρα τους. Στις επόμενες τρεις γραμμές βλέπουμε ότι οι κόμβοι 6,5 και 1 διάλεξαν και οι τρεις για πατέρας τους τον κόμβο 0, δηλαδή τη ρίζα του δένδρου.



Σχήμα 8: Topology of sensors.



Σχήμα 9: Topology of sensors after routing.

Το παραπάνω σχήμα (Σχήμα 8) δείχνει την τοποθέτηση των κόμβων και τις συνδέσεις μεταξύ όσων είναι γείτονες. Αφού γίνει το routing ο κάθε κόμβος θα αποφασίσει ποιος θα είναι ο πατέρας του από τους γείτονες που έχει προφανώς. Το δένδρο που προκύπτει αφού γίνει το routing φαίνεται επίσης παραπάνω (Σχήμα 9). Να σημειωθεί μόνο ότι η αρίθμηση των κόμβων λόγω MATLAB ξεκινάει από το 1 και φτάνει μέχρι και το 25 ενώ θα έπρεπε να είναι από το 0 μέχρι το 24. Για να είναι απόλυτα ακριβές αυτό που βλέπουμε ουσιαστικά πρέπει να θεωρήσουμε το ID του κάθε κόμβου να είναι αυτό που φαίνεται στα διαγράμματα μείον 1.

Καταμερισμός Εργασιών

Ουσιαστικά το project αποτελείται από 3 κομμάτια το βοηθητικό πρόγραμμα, το πρόγραμμα 1 (το μεγαλύτερο κομμάτι της εργασίας) και την αναφορά. Τα παραπάνω 3 κομμάτια ολοκληρώθηκαν με δίκαιο καταμερισμό και συνεργασία σε κάθε στάδιο. Όλοι οι κώδικες συγγράφηκαν από κοινού ενώ και οι δυο μας προτείναμε διορθώσεις ο ένας στον άλλο σε κάποιες περιπτώσεις.

Βιβλιογραφία

[1] TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks
https://www.usenix.org/legacy/publications/library/proceedings/osdi02/tech/full_papers/madden/madden_html/paperhtml.html

[2] TinyOS Turorial 1
http://cs.uccs.edu/~cs526/mote/tinyos/tutorial/lesson1.html

[3] Modules and the TinyOS Execution Model
http://tinyos.stanford.edu/tinyos-wiki/index.php/Modules_and_the_TinyOS_Execution_Model