Ιονιο Πανεπιστημιο

Κατανεμημένα Δικτυοκεντρικά Συστήματα

Κωνσταντίνος Ξυπολιτόπουλος

2 Μαΐου 2018

1 Εισαγωγη

1.1 Ζητουμενο

Το κεντρικό ζητούμενο της εργασίας είναι η κατασκευή ενός δικτύου αποτελούμενο από 100 κόμβους όπου με κατανεμημένο τρόπο θα μπορούμε να κάνουμε συγχρονισμό των εσωτερικών ανεξαρτήτων ρολογιών του κάθε κόμβου στο δίκτυο βάση της μέσης τιμής όλων των κόμβων στο δίκτυο. Η παρούσα αναφορά είναι γραμμένη με τον συνοδευόμενο κώδικα της υλοποίησης της εις νου, ο κώδικας είναι πλήρως σχολιασμένος με λεπτομερείς περιγραφές της ροής εκτέλεσης των κατανεμημένων αλγορίθμων και διαφόρων άλλων πιο λεπτομερών λειτουργιών της συγκεκριμένης υλοποίησης. Για τον κώδικα μπορείτε να αναφερθείτε στο παράρτημα που βρίσκεται στο τέλος της παρούσας αναφοράς.

1.2 Στοιχεια Υλοποιησης

Η κατασκευή του δικτύου είναι πλήρως παραμετροποιημένη, δηλαδή με την αλλαγή κάποιων απλών αριθμών στο αρχείο παραμετροποίησης .ini του omnet++ μπορούμε εύκολα να αλλάξουμε στοιχεία όπως η ακτίνα συνδεσιμότητας (radius of connectivity ή rc) των κόμβων, το πλήθος των κόμβων, τον seed του ψευδο τυχαίου αλγορίθμου παραγωγής αριθμών, και άλλα.
Οι συνδέσεις μεταξύ των κόμβων γίνονται βάση της ευκλείδειας απόστασης μεταξύ των κόμβων, αν η απόσταση δύο κόμβων είναι μικρότερη από την παράμετρο rc τότε θα δημιουργηθεί σύνδεση, αν είναι μεγαλύτερη, τότε δεν θα δημιουργηθεί σύνδεση. Όλοι οι κόμβοι της προσομοίωσης ξεκινούν με τιμή του ρολογιού τους στο 0, έπειτα κάθε 1 δευτερόλεπτο προσθέτουν στο ρολόι τους μια τυχαία τιμή από το 0 εώς το 1 (τυχαία με ομοιόμορφη κατανομή). Επίσης κάθε 50 δευτερόλεπτα τρέχει ο αλγόριθμος συγχρονισμού των ρολογιών.

2 Συγχρονισμος Ρολογιων

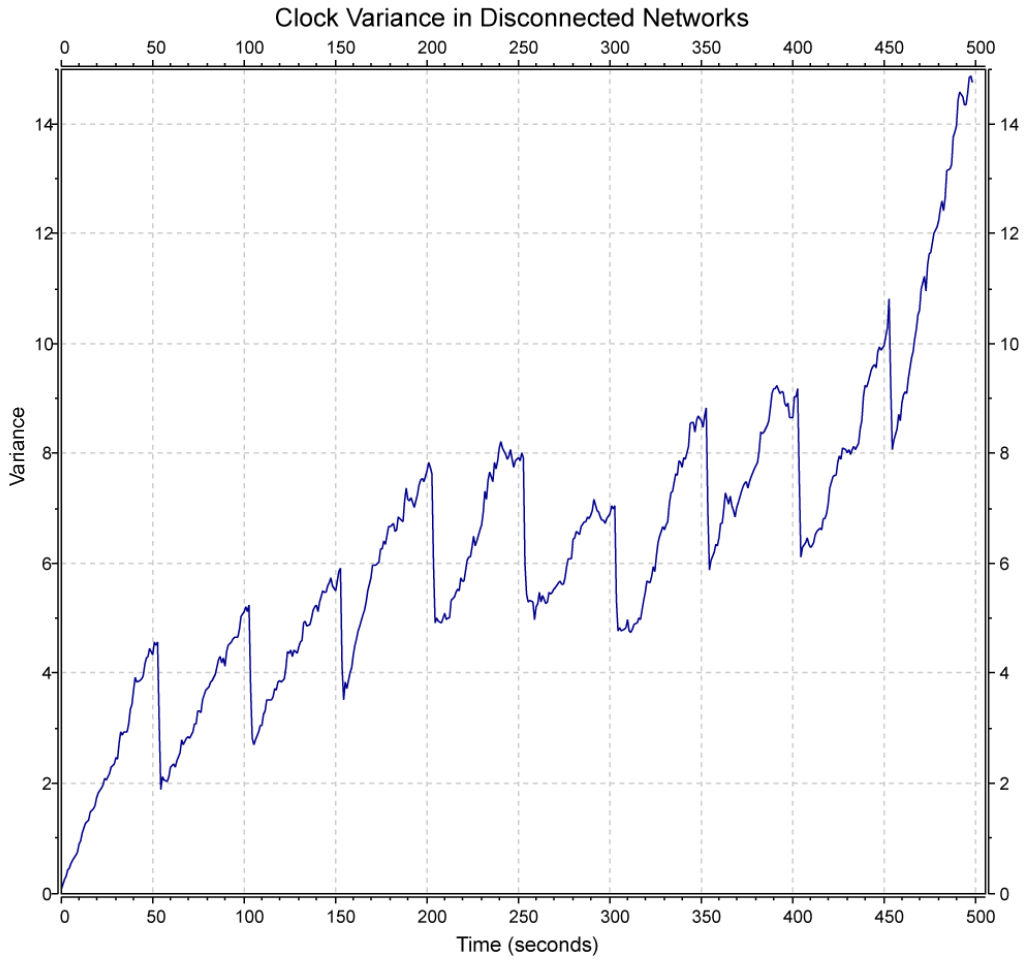
Για το ζητούμενο του συγχρονισμού των ρολογιών όλων των κόμβων στο δίκτυο, χρησιμοποιώ δύο πλήρως κατανεμημένους αλγορίθμους, ο πρώτος είναι ο echo, που έχει την αρμοδιότητα δημιουργίας ενός προσωρινού spanning tree για το δίκτυο, ώστε οι κόμβοι να μπορέσουν να στείλουν την πληροφορία των ρολογιών τους στον κόμβο sink.
Στην φάση όπου οι κόμβοι αρχίζουν και στέλουν μήνυμα στους πατέρες τους, έχω υπερφορτώσει την κλάση cMessage του omnet++ για να μπορεί να με-

ταφέρει πληροφορία, συγκεκριμένα μια μεταβλητή double όπου κάθε κόμβος αποθηκεύει το άθροισμα των ρολογιών του εαυτού του και όλων των παιδιών του, φυσικά οι κόμβοι που είναι φύλλα του γράφου απλά στέλνουν μόνο το δικό τους ρολόι αφού δεν έχουν παιδιά. Άρα ο sink στο τέλος της εκτέλεσης του αλγορίθμου echo θα καταλήξει με το άθροισμα των ρολογίων του συνόλου των κόμβων στο δίκτυο, αλλά δεν μας φτάνει μόνο αυτό αφού θέλουμε να βρούμε την μέση τιμή των ρολογίων, όχι το άθροισμα τους. Για να βρούμε τον μέσο όρο και αφού ήδη έχουμε το άθροισμα μας λείπει ένα στοιχείο, ο αριθμός των κόμβων. Τον αριθμό των κόμβων τον βρίσκουμε με την μέθοδο που περιγράφεται στην επόμενη παράγραφο, και δεν βοηθάει μόνο στον υπολογισμό του μέσου όρου, άλλα παίζει κεντρικό μέρος και στον έλεγχο συνεκτικότητας του δικτύου. Ο συγχρονισμός των ρολογίων γίνεται με μια μετασχηματισμένη έκδοση του αλγορίθμου flooding, με κάποιες πρόσθετες δομές ελέγχου ώστε να είμαστε σίγουροι ότι κάποιος κόμβος δεν θα ενημερώσει το ρολόι του πολλαπλές φορές σε κάθε κύκλο συγχρονισμού.

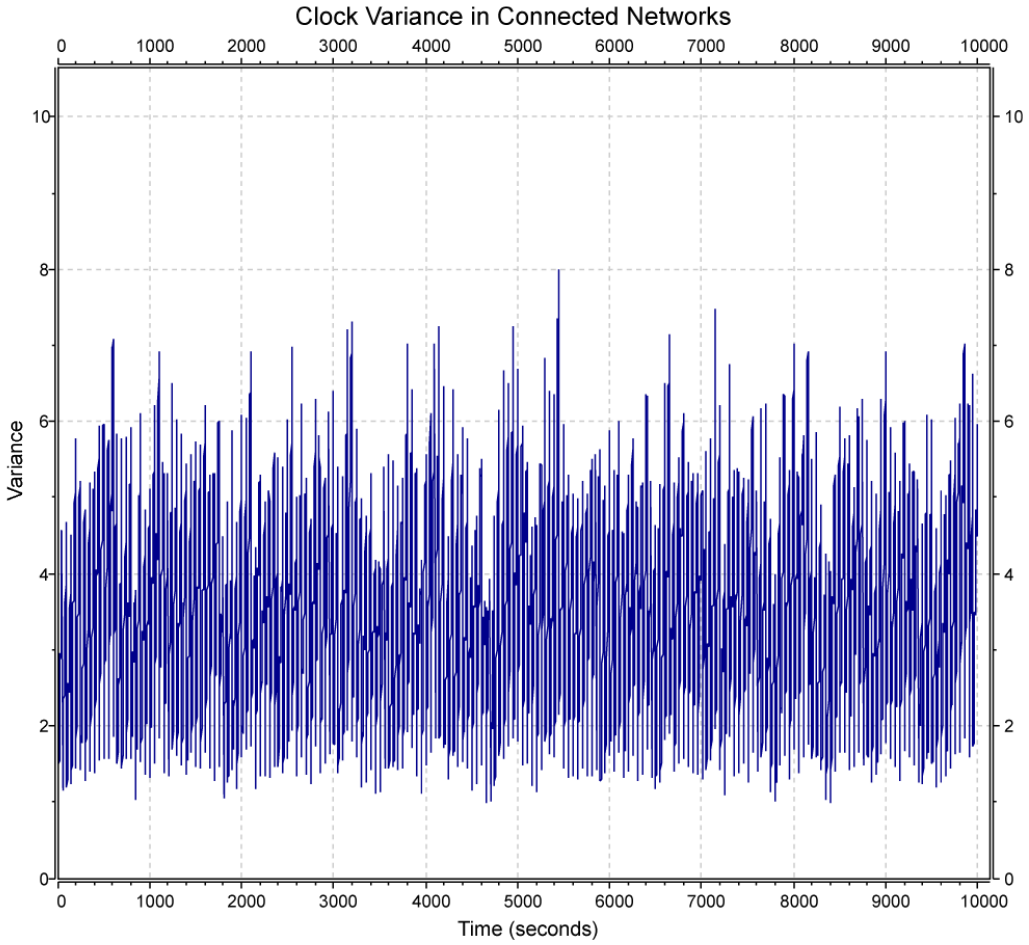
3 Ελεγχος Συνεκτικοτητας Δικτυου

Για το πρόβλημα του ελέγχου συνεκτικότητας δικτύου, υλοποίησα μία πολύ απλή μέθοδο, χρησιμοποιώντας τα μηνύματα του αλγορίθμου echo. Κάθε φορά που ένα παιδί στέλνει το μήνυμα echo στον πατέρα του, ο πατέρας μετράει το παιδί του σε μια μεταβλητή μέτρησης. Όταν ο πατέρας ακούσει τους αντίλαλους από όλα τα παιδιά του, στέλνει την δική του ηχώ στον πατέρα του, αλλά στο μήνυμα αυτό συμπεριλαμβάνει τον αριθμό των παιδιών που έχει, ώστε όταν ο πατέρας αυτού λάβει το μήνυμα να μετρήσει σαν παιδιά του τον κόμβο που του έστειλε την ηχώ, αλλά και τον αριθμό των παιδιών που έχει αυτός.
Τελικά όταν φθάσουν όλα τα μηνύματα στον κεντρικό κόμβο sink, αυτός θα έχει μετρήσει όλους τους κόμβους που μπορεί να φτάσει με τον αλγόριθμο echo, έπειτα συγκρίνει αυτό τον αριθμό με την παράμετρο δημιουργίας του δικτύου numNodes, αν είναι ίσοι τότε το δίκτυο είναι συνεκτικό, αν όχι σημαίνει ότι υπάρχουν κάποιοι κόμβοι όπου δεν μπορούν να φταστούν με κάποιο τρόπο από τον κεντρικό, ή κανένα άλλο στο δίκτυο, άρα το δίκτυο δεν είναι συνεκτικο. Με το που πάρει την απόφαση για την συνεκτικότητα του δικτύου, ο κεντρικός κόμβος εμφανίζε το ανάλογο μήνυμα στην προσωμοίωση ως ένα text bubble. Επίσης με αυτό τον αριθμό τώρα μπορούμε να υπολογίσουμε τον μέσο όρο των ρολογιών που μαζέψαμε από τους κόμβους στο προηγούμενο βήμα.

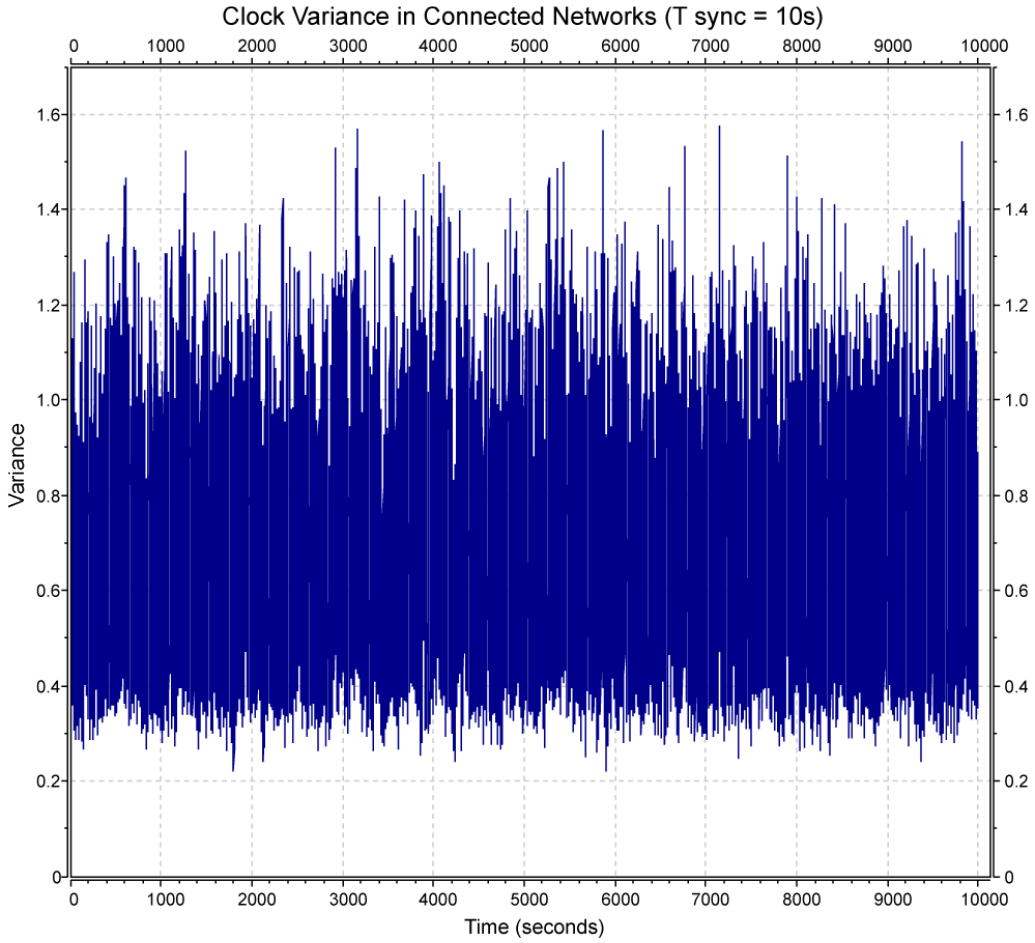
4 Διακύμανση(σ^2) Ρολογιών



Παρατηρούμε τη διακύμανση της τιμής του συνόλου των ρολογιών σε ένα δίκτυο που δεν είναι συνεκτικό, ξεκινάει από χαμηλές τιμές αλλά αυξάνει σταθερά με την πάροδο του χρόνου, επίσης βλέπουμε τις χαρακτηριστικές βουτιές στην διακύμανση κάθε φορά που γίνεται ο συγχρονισμός των ρολογιών των κόμβων που είναι μέρος του μεγάλου συνεκτικού δικτύου, αλλά η γενική αύξηση στην τιμή της διακύμανσης παραμένει ακόμα και μετά τον κάθε κύκλο συγχρονισμού (50 δευτερόλεπτα) και γίνεται εμφανής αμέσως. Λόγος του φαινομένου είναι ότι ο συγχρονισμός των ρολογιών δεν επηρεάζει τα ρολόγια που είναι εκτός του κυρίου συνεκτικού δικτύου, οπότε η διακύμανση συνολικά θα συνεχίσει να αυξάνεται ανεξαρτήτως του ότι κάποιοι κόμβοι συνεχίζουν να συγχρονίζονται.

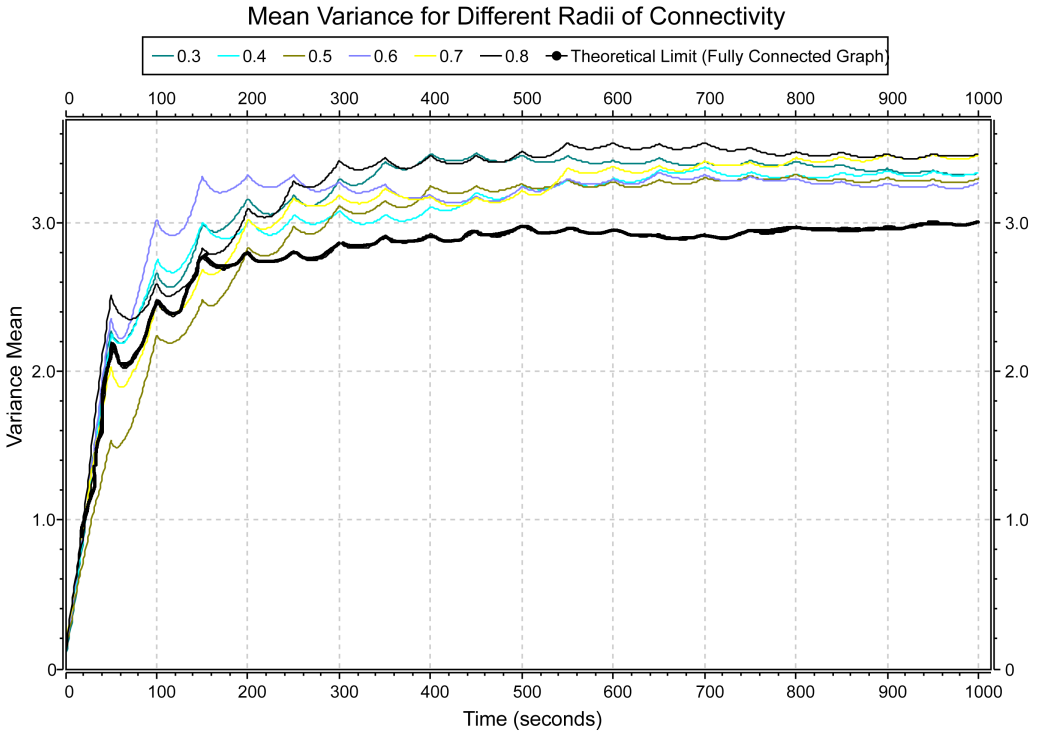


Το συγκεκριμένο γράφημα περιέχει πολλαπλάσιο αριθμό γεγονότων από το προηγούμενο γιατί θέλουμε να δώσουμε έμφαση στο πόσο σταθερή είναι η τιμή της διακύμανσης σε μεγάλα χρονικά διαστήματα (στην περίπτωσή μας, περίπου 3 ώρες εκτέλεσης της προσομοίωσης) όταν μιλάμε για συνεκτικά δίκτυα όπου όλοι οι κόμβοι έχουν πρόσβαση στον συγχρονισμό με τον κεντρικό κόμβο. Η διακύμανση δεν ξεφεύγει ποτέ πάνω από την τιμή 8, και σπάνια πάνω από τις τιμές 6 και 7, το ίδιο μπορούμε να πούμε και για το ότι δεν πέφτει ποτέ κάτω από την τιμή 1.5, και να κυμαίνεται πάντα μέσα σε αυτά τα όρια με μέσο όρο γύρω στο 4



Τώρα, αν το μέγεθος της διακύμανσης στο προηγούμενο γράφημα είναι μεγάλο για τις συγκεκριμένες απαιτήσεις του συστηματός μας, και θα θέλαμε να κυμαίνεται σε πιο χαμηλές τιμές, αυτό που θα μπορούσαμε να κάνουμε έιναι να μειώσουμε την περίοδο T που συγχρονίζουμε τα ρολόγια των κόμβων, από 50 δευτερόλεπτα σε κάποια πιο χαμηλή τιμή πχ. 10 δευτερόλεπτα εδώ, θα δούμε την δραματική πτώση της διακύμανσης γύρω στην μέση τιμή του 0.7

5 Παραμετρικη Μελετη Διακυμανσης



Στο πάνω γράφημα, κάθε γραμμή παριστάνει τον μέσο όρο, καθώς περνάει ο χρόνος, της διακύμανσης μιας συγκεκριμένης εκτέλεσης μιας παραμετροποιημένης προσωμοίωσης με διαφορετικές ακτίνες συνδεσιμότητας rc. Βλέπουμε ότι με ακτίνα 0.9 σε ένα δίκτυο με διαστάσεις 1 επί 1, έχουμε ένα πλήρως συνδεδεμένο γράφο και όποτε μπορούμε να δούμε την θεωρητική μέγιστη απόδοση του αλγορίθμου μας για τον συγχρονισμό με περίοδο 50 δευτερολέπτων, φυσικά όπως θα περιμέναμε όλες οι άλλες εκτελέσεις είναι υποδεέστερες αυτής.
Συνολικά τρέξαμε 7 διαφορετικές προσωμοιώσεις για την παραγωγή αυτού του γραφήματος, με μόνη διαφορά μεταξύ των την ακτίνα συνδεσιμότητας, και έπειτα για κάθε διαφορετική εκτέλεση υπολογίσαμε τον μέσο όρο σε κάθε στιγμή της χρονικής διάρκειάς της. Τελικά βάλαμε όλα αυτά τα γραφήματα μέσων όρων διακύμανσης, σε ένα γράφημα ώστε να τα συγκρίνουμε.
Όλα τα γραφήματα παράχθηκαν για χρήση στην αναφορά χρησιμοποιώντας την εσωτερική μηχανή ανάλυσης στατιστικών στοιχείων προσωμοιώσεων του omnet++.

1 Appendix: Omnet++ NED Topology Code

```
network Network
{
    parameters:
        int init;
        int numNodes;
        double connectivity_radius;

    types:
        channel Channel extends ned.DelayChannel { delay = 100ms; }

    submodules:
        NODE[numNodes]: NODE;
        Observer_of_Variance: Observer;

    connections:
        for i = 0..numNodes - 1, for j = i + 1..numNodes - 1
            NODE[i].port++ <--> Channel <--> NODE[j].port++  // continue below
                    if sqrt((NODE[i].xCor - NODE[j].xCor)^2 +
                    (NODE[i].yCor - NODE[j].yCor)^2) <= connectivity_radius;
}

simple NODE
{
    parameters:
        bool sink;
        double xCor;
        double yCor;
        int numNodes;
    gates:
        inout port[];
}

simple Observer
{
    gates:
        input in directIn;
        // This will cause OMNeT++ not to complain that the gate
        // is not connected in the network or compound module
        // where the module is used.
}
```

2 Appendix: Omnet++ .ini Parameter File

```
[General]
network = Network
seed-set = ${runnumber}
**.init = ${starter = 0}
**.numNodes = 100 #tested working at 15 nodes, 0.3 rc, 568 seed
**.connectivity_radius = ${0.95, 1}
**.xCor = uniform(0, 1)
**.yCor = uniform(0, 1)
**.sink = (index == ${starter}) ? true : false # if index = 0 (starter), then initiator = true, else false.
```

3 Appendix: Observer.cc

```
#include <omnetpp.h>

using namespace omnetpp;

#include "variance_m.h"

class Observer : public cSimpleModule
{
    private:
        cOutVector varianceCountVector;
    protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
        variance* observerMsg;
        double all_clocks[100];
        int num_clocks;     // number of clocks we gathered
        double mean;        // average of all clocks per cycle
        double mean_sum;
        double numerator_sum; // (x-mean)^2 sum..
        double f_variance;
        int numberofnodes;
};

Define_Module(Observer);

void Observer::initialize()
{
    for(int i = 0; i < 100; ++i){
        all_clocks[i] = 0;
    }
    mean = 0;
    num_clocks = 0; WATCH(num_clocks);
    numerator_sum = 0;
    mean_sum = 0;
```

```
    f_variance = 0;
    numberofnodes = int(getParentModule()->par("numNodes"));
}

void Observer::handleMessage(cMessage *msg)
{
    if(msg->getKind() == 44){
        variance * observerMsg = check_and_cast<variance *>(msg);
        all_clocks[num_clocks] = observerMsg->getMyclock();
        num_clocks += 1;
    }
    if(num_clocks == numberofnodes){
        for(int z = 0; z < numberofnodes; ++z)
            mean_sum += all_clocks[z];
        EV << "mean_sum: " << mean_sum;
        mean = double(mean_sum / numberofnodes);
        EV << "numberofnodes: " << numberofnodes;
        EV << "mean: " << mean;
        for(int x = 0; x < numberofnodes; ++x){
         // summing all the expanded numerator terms of the variance equation
            numerator_sum += pow((all_clocks[x] - mean), 2);
        }
        f_variance = numerator_sum / numberofnodes;
        EV << "Variance: " << f_variance;
        varianceCountVector.record(f_variance); // statistics recording
        // cleanup
        f_variance = 0;
        mean_sum = 0;
        numerator_sum = 0;
        num_clocks = 0;
        mean = 0;
        for(int i = 0; i < 100; ++i){
            all_clocks[i] = 0;
        }
    }
    delete msg;
}
```

4 Appendix: NODE.cc

```
#include <omnetpp.h>
using namespace omnetpp;

//generated header files from .msg files
#include "echo_m.h"
#include "sync_m.h"
#include "variance_m.h"

class NODE : public cSimpleModule
{
    public:
        NODE();
        virtual ~NODE();

    protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);

    private:
        cModule *Observer;
        double clock;          // current node clock
        double clock_sum;      // clock sum variable for echo
        cMessage* timerMsg;    // the self clock timer for each node to add [0,1] to its clock
        bool isSink;
        int numNeighbors;
        bool reached;          // echo: has node been reached by the explorer wave
        int numResponses;      // how many of the neighbors have responded
        int predecessor;       // the gate id of the parent who sent the explorer message to this node
        int baseId;
        cMessage* exploreMsg; // explore wave message
        echo* echoMsg;         // echo type, backwards echo message
        variance* observerMsg;
        int numberofechoes;    // node variable, number of nodes it got echoes back from
        sync* syncMsg;         // sync type, sync clocks message with the average clock value from the sink..
        cMessage* sinkTimer;   // timer (default=7secs) for each clock sync cycle
        bool synced;           // accept only one sync per sync cycle
        int syncCount;         // how many sync requests we got from neighbors (used to reset synced status t
        // ayylmao
        //double holdClocks[100]; // holder array in each node for the values of their children's nodes
                                  // every time it receives an echo, it adds the value to the holdClocks[numbe
};
Define_Module(NODE);


void NODE::initialize()
{
    isSink = par("sink").boolValue();
    numNeighbors = gateSize("port"); // Reads the size of the "port" gate vector which is the number of neig
```

```
    clock_sum = 0;
    numberofechoes = 0; // local echo number (there's also an echo number that gets transferred from child t
    baseId = gateBaseId("port$o");
    // clocks
    clock = 0;
    WATCH(clock); // able to examine the variable under simulation
    timerMsg = new cMessage("clock_timer");
    scheduleAt(simTime() + 0, timerMsg); // first clock adding, run instantly

    synced = false;
    syncCount = 0;

    reached = false;
    // sink
    if(isSink){
        sinkTimer = new cMessage("timer for sync");
        scheduleAt(50, sinkTimer);
    }
    numResponses = 0; // all nodes start with 0, even the sink (since he didn't get the explorer from anyone

    // VARIANCE STUFF >>>>



    //DEBUG WATCHES
    WATCH(numResponses);
    WATCH(reached);
    WATCH(baseId); WATCH(numNeighbors); WATCH(predecessor); WATCH(clock_sum); WATCH(numberofechoes); WATCH(sy

}

void NODE::handleMessage(cMessage *msg)
{
    // SINK explorer sync period timer T = 7s
    if( msg == sinkTimer && isSink){
        reached = true;
        exploreMsg = new cMessage("explorer wave", 14);
        for(int i = 0; i < numNeighbors; ++i){
            send(exploreMsg->dup(), "port$o", i); // in order to send the same message to multiple nodes, yo
        }
        delete exploreMsg;
        scheduleAt(simTime() + 50, sinkTimer);
    }

    // SELF CLOCK MESSAGE
    if ( msg == timerMsg ){
        // every 1 second, the self timerMsg arrives, and adds a random value to the clock from 0 to 1.
        clock += uniform(0, 1);
        EV << "my clock is" << clock << "\n";
```

```
    // <<= VARIANCE ==> sending clocks to observer
    variance* observerMsg = new variance("clocks", 44);
    observerMsg->setMyclock(clock);

    cModule *Observer = getParentModule()->getSubmodule("Observer_of_Variance"); //getting pointer to ob
    sendDirect(observerMsg, Observer, "in"); // sending directly to the unconnected input gate of Observe
    // <<= end variance section ==>

    scheduleAt(simTime() + 1, timerMsg);
}

// ECHO "EXPLORER" WAVE (no need for sink checks anywhere since sink will never get an explorer wave bac
if ( msg->getKind() == 14 ){
    // if its first time being reached..
    if (!reached){
        numResponses += 1;
        reached = true;
        EV << "i am reached..";
        predecessor = msg->getArrivalGate()->getIndex(); // since it wasn't reached before, the one who

        for( int i = 0; i < numNeighbors ; i++){
            cGate * gatef = gate(baseId + i);

            /*
            // debug logging to console
            EV << gatef->getFullName() << ": ";
            EV << "id=" << gatef->getId() << ", ";
            if (!gatef->isVector()){
                EV << "scalar gate, ";}
            else{
                EV << "gate " << gatef->getIndex() << " in vector " << gatef->getName() << " of size " <<
                EV << "type:" << cGate::getTypeName(gatef->getType()) << "\n";}
            */
            if(gatef->getIndex() != predecessor){
                send(msg->dup(), gatef); // propagate explorer wave to every neighbor except parent
            }
        }

        delete msg; // since we are sending duplicates of msg, we should delete our own copy at the end.
    }
    else{
        numResponses += 1;
        delete msg; // added recently, to combat non-destroyed explorer wave msg at end of simulation
        // if it was already reached..
        // we are getting an explorer message from a "non-child" node.. we just increment echoes from ne
    }
    if ( numResponses == numNeighbors ){
        // in this case, we are done with our children echoes, and got the final message (a late explore
```

```
        // OR we are a "leaf" node with only 1 neighbor.. our predecessor (parent)
        EV << " echoes == neighbors\n";
        numResponses = 0; // resetting for next run
        reached = false;

        echo * echoMsg = new echo("echo", 20);
        echoMsg->setNumberofnodes(0); //since we are a leaf (not 1 but 0, since we are counting our chil
        echoMsg->setEchoClock(clock_sum + clock); // clock sum from children + current self clock
        send(echoMsg, "port$o" , predecessor); //msg, gatename, gateindex
    }
}


// receiving ECHO message (msg_id=20) from child node
// add its clock variable to our clock_sum, and delete the message
if (msg->getKind() == 20){
    echo *echom = check_and_cast<echo *>(msg); // casting msg to echo??

    EV << "received echo \n";
    numResponses += 1;      // increments both from explorer messages and from echoes.
    numberofechoes = numberofechoes + 1 + echom->getNumberofnodes(); // for each child that sends an ech

    clock_sum += echom->getEchoClock(); // adding child's clock to the sum

    if ( numResponses == numNeighbors ){
        reached = false;
        numResponses = 0;
        //delete msg;  //we do not need the children's echo messages since we created a new one to send.

        if (!isSink){
            // done: cleaning up node for next run, and sending clock echo to parent
            echo * echoMsg = new echo("echo", 20);
            echoMsg->setEchoClock(clock_sum + clock); // clock sum from children + current self clock
            echoMsg->setNumberofnodes(numberofechoes);
            send(echoMsg, "port$o" , predecessor); //msg, gatename, gateindex
            }
        else{
            // SYNC START
            // we have the "full" echo back at SINK, and we are finished with gathering the clock values
            // check for connected network
            bubble("Clocks read, starting sync");
            if(numberofechoes == int(getParentModule()->par("numNodes")) -1 ) // nodes-1 because we did
                bubble("CONNECTED NETWORK!");
            else
                bubble("DISCONNECTED NETWORK!");

            // we have to now resync every clock in the networks nodes
            for( int i = 0; i < numNeighbors ; i++){
                sync * syncMsg = new sync("clock sync message", 99); // the sync clocks message
```

```
                syncMsg->setSyncClock(clock_sum / numberofechoes);   // SyncClock is the average clock o
                cGate * gatef = gate(baseId + i);
                send(syncMsg, gatef);
            }
            clock = (clock + (clock_sum/numberofechoes)) / 2; // === !!! NEW: updating the clock of sink
        }
        clock_sum = 0;
        numberofechoes = 0; //reset for next run
    }
    delete msg; // added recently.. seems to eliminate non-destroyed echo msg at end of simulation
}


// CLOCK AVERAGE SYNC MESSAGE (msg_id=99)
if(msg->getKind() == 99){
    // we got the sync command, lets sync to all neighbors except our predecessor..
    if(isSink){
        delete msg;
        return;//sync does not send out sync messages since he already did
    }
    if(!synced){
        synced = true;
        syncCount += 1;
        sync *syncm = check_and_cast<sync *>(msg); // casting msg to sync??
        clock = (clock + syncm->getSyncClock()) / 2; //current clock value + sync value (new) /2

        for( int i = 0; i < numNeighbors ; i++){
            cGate * gatef = gate(baseId + i);
            send(syncm->dup(), gatef);
            /*if(gatef->getIndex() != predecessor){
                send(syncm->dup(), gatef); // push clock sync command to every neighbor except parent
                }*/
        }
        if(syncCount == numNeighbors){
            // case of 1 neighbor only
            synced = false;
            syncCount = 0;
        }
        delete msg;
    }
    else{
        // if has already synced
        syncCount += 1;
        if(syncCount == numNeighbors){ // +1 because we will get one from our parent too
            synced = false;
            syncCount = 0;
        }
        delete msg;//recently added to combat non-destroyed sync msg at end of simulation run, works!
```

```
        }
    }
}


NODE::NODE(){
    // here if initialization never takes place, for destructor below to not crash and burn
    sinkTimer = nullptr;
    timerMsg = nullptr;
    exploreMsg = nullptr;
    echoMsg = nullptr;
    syncMsg = nullptr;
}

NODE::~NODE(){
    // destructor of self messages, every other message is destroyed during runtime
    cancelAndDelete(sinkTimer);
    cancelAndDelete(timerMsg);
}
```