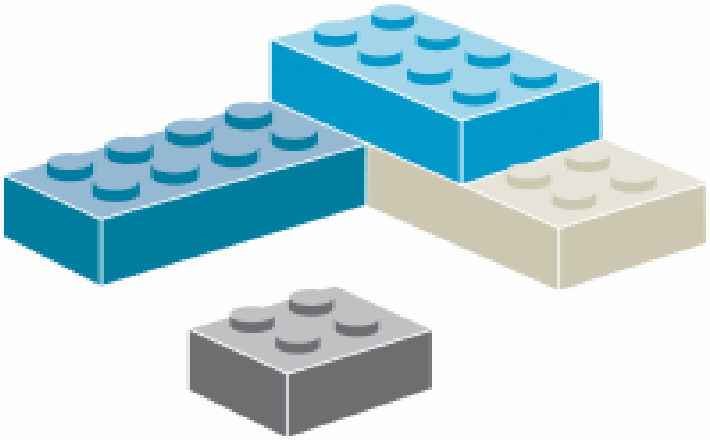
UNIVERSITY OF GOTHENBURG



Evaluating New MAC algorithms for Mobile Ad-Hoc Networks

Master of Science Thesis in the programme Computer Science

GONGXI ZHU

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2010

Evaluating New MAC algorithms for Mobile Ad-Hoc Networks

Gongxi Zhu

Examiner: Marina Papatriantafilou

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2010

Abstract

The problem of media access control (MAC) implies significant challenges when considering the relocation of mobile nodes. In the algorithmic arena of MAC solutions, it is common that when nodes are considered to be non-stationary, designers tend to assume that some nodes temporarily do not change their location. Further, they assume that these stationary nodes coordinate the communications among mobile nodes. Thus, the relation between the performance of MAC algorithms and the different settings by which the location of the mobile nodes is modeled requires further inquiry.

We study this relationship and suggest an extension of a recently proposed abstract model, which models the relocation of nodes via a small set of parameters. These parameters refer to the relocation rate, α, and the similarity ratio, β. Namely, we assume that the rate, α, by which the relocation occurs is constant. Moreover, a ratio of at least β neighbors of any mobile node that relocates from its current neighborhood to a new one, are going to be in its new neighborhood.

We numerically evaluate the throughput and convergence period of a recently proposed MAC algorithm. We show that the algorithm's throughput depends on α and ß, unlike algorithms such as slotted ALOHA that their throughput is independent of α and ß. Moreover, we identify critical threshold, β_c, of the similarity ratio above which the throughput of slotted ALOHA is always lower than the one of the studied algorithm. We use our results to estimate the throughput of the studied MAC algorithm in vehicular ad-hoc networks.

Remark: This master project has resulted in two technical reports and a scientific publication. This report is adapted from the paper "Analyzing Protocols for Media Access Control in Large-Scale Mobile Ad Hoc Networks" published in Self-Organizing Wireless Sensor and Communication Networks (SOMSED 2009). - 978-3-941492-10-3; s. 77

Acknowledgements

This project is not possible to be finished without the help and support of many wonderful individuals. I would like to thank everyone who has helped me along the way. Particularly: Dr. Elad Michael Schiller for providing me an opportunity to conduct my master's research with him and for his guidance and support over the course of my thesis; Dr. Marina Papatriantafilou for some useful suggestions; Andreas Larsson for helping me with various technical details; Ning He, my project partner for giving me so much help and support.

Contents

List of Figures

1. Introduction

We study a MAC algorithm for mobile ad hoc networks (MANETs). The studied algorithm is the one in Leone et al. [1] that is based on vertex coloring. In that work the authors suggest a model for relocation analysis in which mobile nodes randomly change their location according to a constant *relocation rate*, α. We present extensions of the relocation analysis of [1]. The extension considers the similarity ratio, i.e., a bound, β, on the minimal ratio of neighbors that a mobile node maintains when relocating to a new neighborhood.

In our study, we place emphasis on stabilization concepts, which are imperative in MANETs. We present an experimental study of the throughput and convergence period of the algorithm in [1] on the extended model. The results show dependency between the algorithm's throughput and the parameters α and β, unlike algorithms such as slotted ALOHA [2] that their throughput is independent of α and β. Moreover, the results allow us to identify a critical threshold, β_c, of the similarity ratio, above which the throughput of slotted ALOHA [2] is always lower than the one of the studied algorithm.

Leone et al.'s [1] model focuses on the location of mobile nodes rather than the "movements of the mobile users", cf. a survey on mobility model [3]. It is an abstract model that can also represent existing mobility models. For example, population protocols [4] can be represented by considering relocation rate of 1 and the similarity ratio (close to) 0. Moreover, using the extension of similarity rate described here, random walks can be represented by considering relocation rate of close to 0 and the similarity ratio close to 1. Thus, in the context of MAC algorithms, the models presented in [1] and in this dissertation have a clear advantage, because the studied algorithm can be analyzed for a wider set of scenarios. Moreover, the expressiveness of the studied model for relocation analysis allows the algorithm designers to follow

an analytical approach, as in [1], and an experimental approach, which is taken in this dissertation.

The studied model is expressive and allows us to estimate the algorithm's throughput in new settings. We demonstrate the expressiveness of the relocation analysis by considering vehicular ad-hoc networks (VANETs). We show typical examples of road settings in which the studied algorithm has throughput that is higher than the one of slotted ALOHA [2].

2. Communication Model

The system consists of a set of communicating entities, which we call (mobile) nodes. Denote the set of nodes by P (processors) and every node $p_i \in P$ with a unique index, i, that p_i can access.

We assume that the MAC protocol is invoked periodically by synchronized common pulses. The term (broadcasting) timeslot refers to the period between two consecutive common pulses, t_x and t_{x+1}, such that $t_{x+1} = (t_x \bmod T) + 1$, where T is a predefined constant named the Frame Size, i.e., the number of timeslots in a TDMA frame (or broadcasting round).

We consider a standard radio interference unit that allows sensing the carrier and reading the energy level of the communication channel. Sometimes, we simplify the description of our algorithms and relocation models by considering concepts from graph theory. Nevertheless, the simulations consider a standard physical layer model.

At any instance of time, the directly communicating ability of any pair of nodes is defined by the set, Ni (subset of P), of neighbors that node p_i in P can communicate with directly. Wireless transmissions are subject to collisions and we consider the potential of nodes to interfere with others' communications. We say that nodes A (subset of P) broadcast simultaneously if the nodes in A broadcast within the same timeslot. We denote by $M_i = \{p_k \in N_j : p_j \in N_i \cup \{ p_i \}\}\backslash\{ p_i \}$ the set of nodes that may interfere with pi's communications when any nonempty subset of them, $A \subseteq M_i : A \neq \emptyset$, transmit simultaneously with p_i. We call M_i the interference neighborhood of node $p_i \in P$, $|M_i|$ the interference degree of node p_i is in P.

3. Modeling the location of mobile nodes

Let us look into scenarios in which each mobile node randomly moves in the Euclidian plane and in which two mobile nodes can directly communicate (or interfere with each other's communications) if their distance is less than a threshold χ. This scenario can be modeled by a sequence of evolving graphs; at time instant t, the communication graph, $G(t)=(V,E(t))$, includes the set of mobile nodes, V, and the set of edges, $E(t)$, which represents pairs of processors that can directly communicate at time t.

Let us consider two consecutive communication graphs, $G(t)$, $G(t+1)$, of the evolving graph. In this short run, it can be expected that many of the mobile nodes have *similar neighborhoods* in $G(t)$ and $G(t+1)$, say, when the threshold χ is large. In the long run, this similarity may disappear because there are *(independent) random relocations* of the mobile nodes due to their random motion, e.g., $G(t)$ and $G(t+x)$ are independent when x goes to infinity. These properties of neighborhood similarity and (independent) random relocation motivate the studied system settings.

To model the evolution of the communication graphs, Leone et al. [1] assume that between every two consecutive communication graphs, $G(t)$, $G(t+1)$, a fraction of the mobile nodes, α, relocates from their neighborhood, where $0 \leq \alpha \leq 1$ is the relocation rate. We extend the model of Leone et al. [1]. Let us consider a mobile node, p_i, that relocates from its current neighborhood, M_i, to a new one, M'_i. We assume that a ratio of β neighbors of p_i in its current neighborhood is going to be in its new neighborhood, i.e., $\beta \leq |M_i \cap M'_i|/|M'_i|$, where β *in* $[0,1]$ is named the *similarity ratio*.

The relocating nodes and their new neighborhoods are chosen randomly. This leads to a mixed property of *short-term* (independent) random relocation and *long-term* neighborhood similarity. The mix is defined by the relocation rate, α, and the

similarity ratio, β. The relocation rate can be viewed as the ratio of non-stationary nodes over (temporarily) stationary ones. The similarity ratio can be related to an upper bound on the mobile nodes' speed. In other words, the slower mobile nodes move the higher the similarly ratio gets.

On one hand, when the value of α is unbounded and the value of β is small, the property of *short-term* (independent) random relocation can be more dominating than the property of *long-term* neighborhood similarity. For example, Leone et al. [1] show that in this case, efficient MAC algorithms should employ a strategy similar to the one of slotted ALOHA [2], which ignores the history of broadcasts among neighbors. On the other hand, when the dominating property is neighborhood similarity, i.e., α is bounded and β is far from 0, the algorithm can allow each mobile node to effectively learn the history of the neighbors' broadcasts, say, using the algorithm in Leone et al. [1] that is based on vertex coloring.

4. The Studied Algorithm

We consider settings in which the relocation rate α and the similarity ratio β are bounded. Leone et al. [1] explain how in these settings, mobile nodes are able to learn some information about the success of the neighbors' broadcasts. The algorithm divides the radio time into timeslots. Moreover, it is based on vertex coloring; nodes avoid broadcasting in the timeslots in which their neighbors successfully broadcast. Given a broadcasting round, we define the throughput as the number of mobile nodes that successfully transmit at least once in that round divided by the number of mobile nodes in the entire system.

Keeping track of broadcast history is complicated in the non-stationary settings of MANETs, because of node relocations and transmission collations. The studied algorithm presents a randomized solution that respects the recent history of the neighbors' broadcasts. This information is inaccurate. However, when the relocation rate is not too high and the similarity ratio is not too low, the timeslots can be allocated by the studied algorithm.

The algorithm in Leone et al. [1] uses a randomized construction that lets every node inform its interference neighborhood on its broadcasting timeslot and allows the neighbors to record this timeslot as an occupied/unused one. The construction is based on a randomized competition among neighboring nodes that attempt to broadcast within the same timeslot. When there is a single competing node, that node is guaranteed to win. Namely, the node succeeds in informing its interference neighborhood on its broadcasting timeslot and letting the interference neighborhood mark its broadcasting timeslot as an occupied one. In the case where there are $x>1$ competing nodes, there might be more than one winner. However, most of the competitors are expected to lose. The nodes that have lost mark their broadcasting timeslot as unused (when there is more than one winner). Thus, on the next broadcast,

it is expected that only a few of the losing nodes will compete for the same timeslot again; they are expected to re-choose their timeslots. Why this is guaranteed to converge is showed in [1].

5. Experimental Analysis using Simulations

5.1 platform and tools

TinyOS is designed for wireless sensor networks. It is written in nesC programming language and is a component-based operating system [10].

TinyOS uses a component-based architecture. These components are abstractions of hardware. By interfaces, different components are connected together in TinyOS. TinyOS implements an even-driven programming model. In this project, we use TinyOs 2.1.x.

TOSSIM is an embedded simulator in TinyOS, which is used for simulating entire TinyOS applications. It offers a simulation environment, which is easier to be controlled and monitored.

TOSSIM is based on event driven system. When it runs, it pulls events of the event queue and executes them. There are two programming interfaces to TOSSIM: python and C++. In our implementation, we use python.

In order to simulate, network topology is needed, such as the size and shape of the network. The topology will be reflected in terms of gain value which represents the transmit signal strength between any pair of nodes. The default values for TOSSIM's radio model are based on the CC2420 radio, used in the micaZ, telos family.

5.2 The Simulations

Recall that the studied model extends the one presented in [1]; the relocation analysis in [1] considers the relocation rate, α, and we consider the similarity ratio, β, in addition. The studied algorithm is a randomized one that requires a convergence period.

We show that after a convergence period, the throughput is within a bounded range and its expected value is asymptotically constant. Moreover, the value of the throughput is a function that monotonically decreases as α increases and increases as β increases. In addition, we explain the relation between the throughput asymptotical value and the convergence period.

We identify a critical threshold, β_c=50%, of the similarity ratio, above which the throughput of slotted ALOHA [2] is always lower than the one of the studied algorithm. We use the algorithm of slotted ALOHA [2] as a benchmark because its throughput is independent of α and ß (unlike the studied algorithm).

Our experiments were conducted using TOSSIM [6]. All experiments had 400 nodes. At any time, the location of a node is uniquely associated with an entry in a 20 by 20 grid. The selected communication and interference ranges are such that there are at most five nodes within the communication range and at most 20 nodes with the interference range. The size of the interference neighborhoods defines the number of timeslots in every broadcasting round. We assume zero propagation delay (due to the short transmission range). The transmit-to-receive turnaround time is 176 μs (i.e., TOSSIM's default value when configured to act as the standard TinyOS 2.0 with the CC2420 stack.)

The number of nodes that take a *relocation step* is defined by the relocation rate. A relocation step is carried out by selecting two random nodes and swapping their

locations. The maximal distance between two nodes that we swap is bounded, so that a desirable similarity ratio is achieved.

In the starting configuration, we let every node choose its broadcasting timeslot, say, uniformly at random or uniquely to its interference neighborhood. In the following configurations, the nodes select their broadcasting timeslots according to the studied algorithm. Moreover, relocation steps change the locations of mobile nodes. We note that these relocation steps change the broadcasting timeslots of the neighboring nodes. In order to study the throughput, we conduct experiments that differ in their starting configurations and the values of α and β (i.e., the timeslots can be chosen uniformly at random or uniquely to their interference neighborhood).

6. Eventual throughput

The aim of this experiment is to understand the relationship between the throughput and the parameters α and β. We present the results of an experiment that demonstrate that the value of the throughput is a function that monotonically decreases as α increases and increases as β increases.

We conduct an experiment in which every mobile node in the starting configuration has a timeslot that is unique to its neighborhood. Then, during the system run, the mobile nodes have to rechoose their timeslots due to the random relocation. In the experiment, the relocation rate, α, increases from 0% to 100% in steps of 1% every 10 broadcasting rounds. We consider five different similarity ratios: 6%, 12%, 25%, 50% and 62%. (For each similarity ratio, we instantiate different experiment, i.e., there are five experiments and the value of the similarity ratio, β, is fixed throughout each instance of the experiment.)

The results of the experiment are presented in Fig. 1 and show that the throughput is almost 100% when $\alpha=0$. Moreover, the throughput monotonically decreases as the relocation rate increases. However, only for the similarity ratios that are above β_c=50% the throughput of the studied algorithm is always higher than the throughput of the slotted ALOHA [2] algorithm.

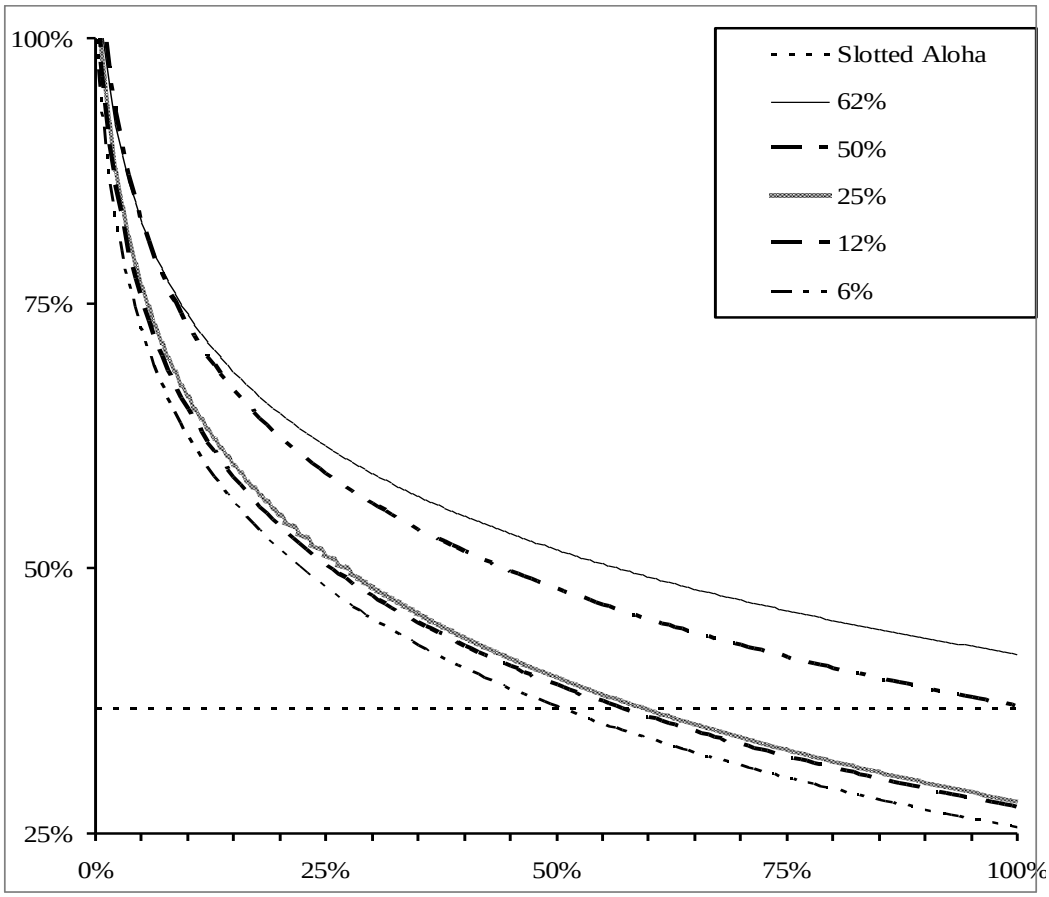


Fig. 1 Throughput of the studied algorithm and slotted aloha [2]. The throughput values (y axis) are depicted as a function of the relocation rate α (x axis). Throughput of slotted Aloha [2] is depicted by the dashed horizontal line. The other lines depict the logarithmic trend line of the throughput for the similarity ratios: 6%, 12%, 25%, 50% and 62%.

7. Convergence period

The studied algorithm is a randomized one that requires a convergence period. During the convergence period, the throughput changes and might eventually converge to a narrower range. We study the relationship between the convergence period and the parameters α and β by examining different starting configurations for which we assume that the broadcasting timeslot are selected uniformly at random. The results of the examinations are presented in Fig. 2. The results show that required convergence period decreases as the relocation rate, α, increases.

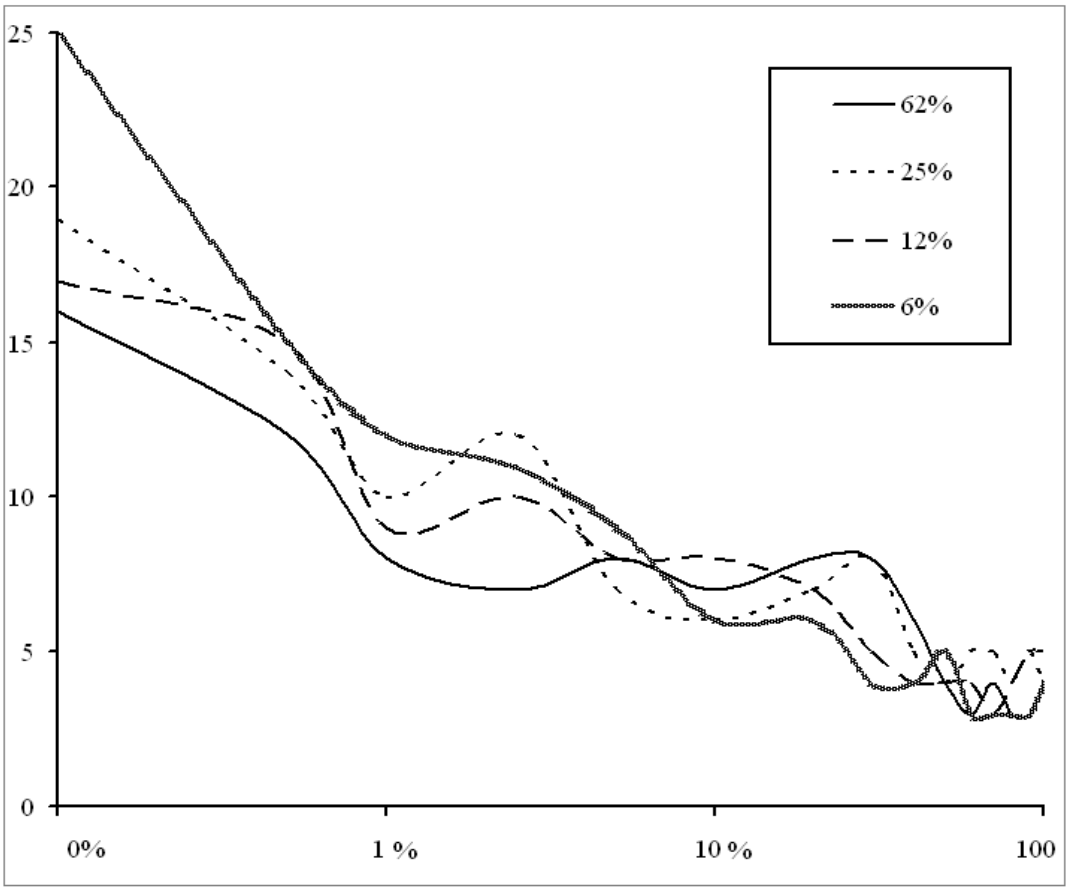


Fig. 2 The convergence period of the studied algorithm. The number of broadcasting rounds needed before the mobile nodes has access to at least 97% of the eventual throughput values (y axis). These values are depicted as a function of the relocation rate α in logarithmic scale (x axis).

One way to explain these results is presented in Fig. 3. That figure considers different values of relocation rate and a single value of similarity ratio, β=6%. The chart shows the for these relocation rates, the throughput the first broadcasting round is about 50%. This is because the starting timeslot values follow a uniform random distribution. In the later broadcasting rounds, it seems that the required convergence period is directly influenced by the time it takes to reach the (eventual) throughput value from the starting configuration.

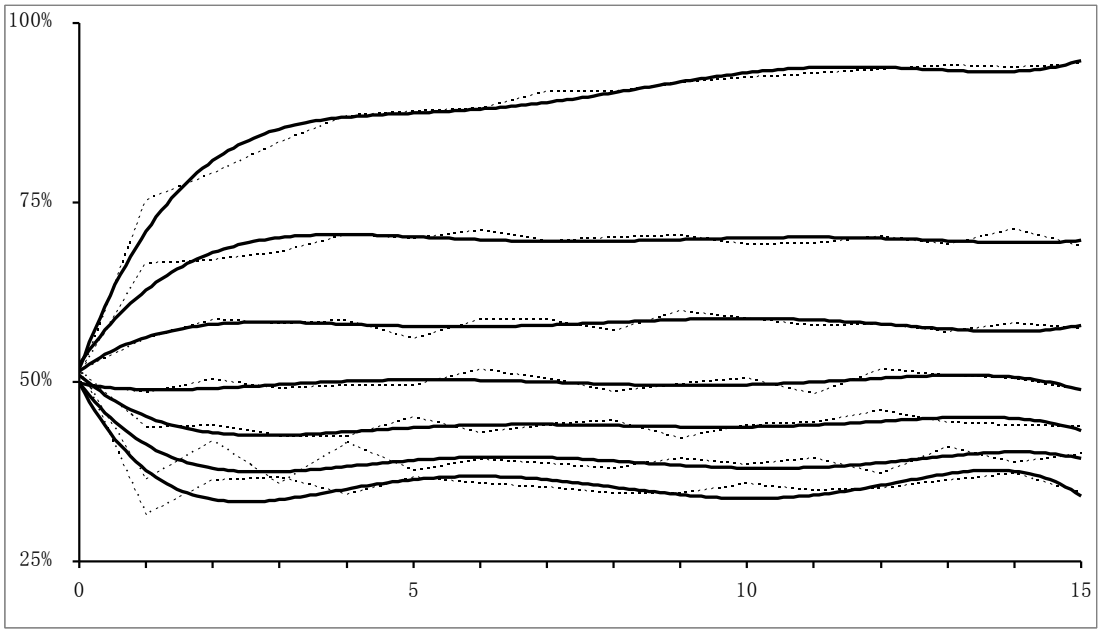


Fig. 3 The throughput of the studied algorithm (y axis) is depicted by the dotted lines (and their polynomial trends in solid lines) as a function of the number of broadcasting rounds from the starting configuration (x axis). The chart considers similarity ratio of 6%. The chart's lines from top to bottom respectively represent relocation rates from 0% to 60% in steps of 10%.

8. Conclusions

This work is a numerical study of the relationship between a fundamental protocol for MANETs and the settings that model the location of mobile nodes. We present an extension to an abstract model of Leone et al. [1] for relocation analysis. The abstractions in the model enable a detailed numerical analysis of the MAC algorithm in [1]. Namely, we study the eventual throughput as well as the convergence period.

This dissertation explains how to estimate the algorithm's throughput in typical settings of VANETs using results from the abstract model of relocation analysis. The studied examples suggest the usefulness of the studied algorithm in VANETs.

We expect that our relocation analysis may be used for the design and understanding of other protocols in the context of MANETs.

References

[1] P. Leone, M. Papatriantafilou and E. M. Schiller, "Relocation Analysis of Stabilizing MAC Algorithms for Large-Scale Mobile Ad Hoc Networks," in: ALGOSENSORS, 2009. Also appears as a technical report 2008:23, Department of Computer Science, and Engineering, Chalmers University of Technology (Sweden), Sep. 2008.

[2] N. Abramson. "Development of the ALOHANET." IEEE Information Theory, 31(2):119–123, 1985.

[3] T. Camp, J. Boleng, and V. Davies, "A survey of mobility models for ad hoc network research," Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications 2 (2002), no. 5, 483-502.

[4] I. Chatzigiannakis, O. Michail and P. Spirakis, "Recent Advances in Population Protocols," in: 34th International Symposium on Mathematical Foundations of Computer Science (MFCS), pages 56-76, Springer-Verlag Berlin Heidelberg, Novy Smokovec, High Tatras, Slovak Republic, 2009.

[5] C. Avin, M. Kouck´y, and Z. Lotker. "How to explore a fast-changing world (cover time of a simple random walk on evolving graphs)." In ICALP (1), LNCS 5125:121–132. Springer, 2008.

[6] P. Levis, N. Lee, M. Welsh, and D. Culler. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications." ACM Conference on Embedded Networked Sensor Systems, 2003.

APPENDIX A Architecture and models

A.1 Architecture

As shown in Fig. 4, we defined several modules and interfaces, which could be wired together with TOSSIM (through default interfaces in TOSSIM like Packet, Receive, etc.).

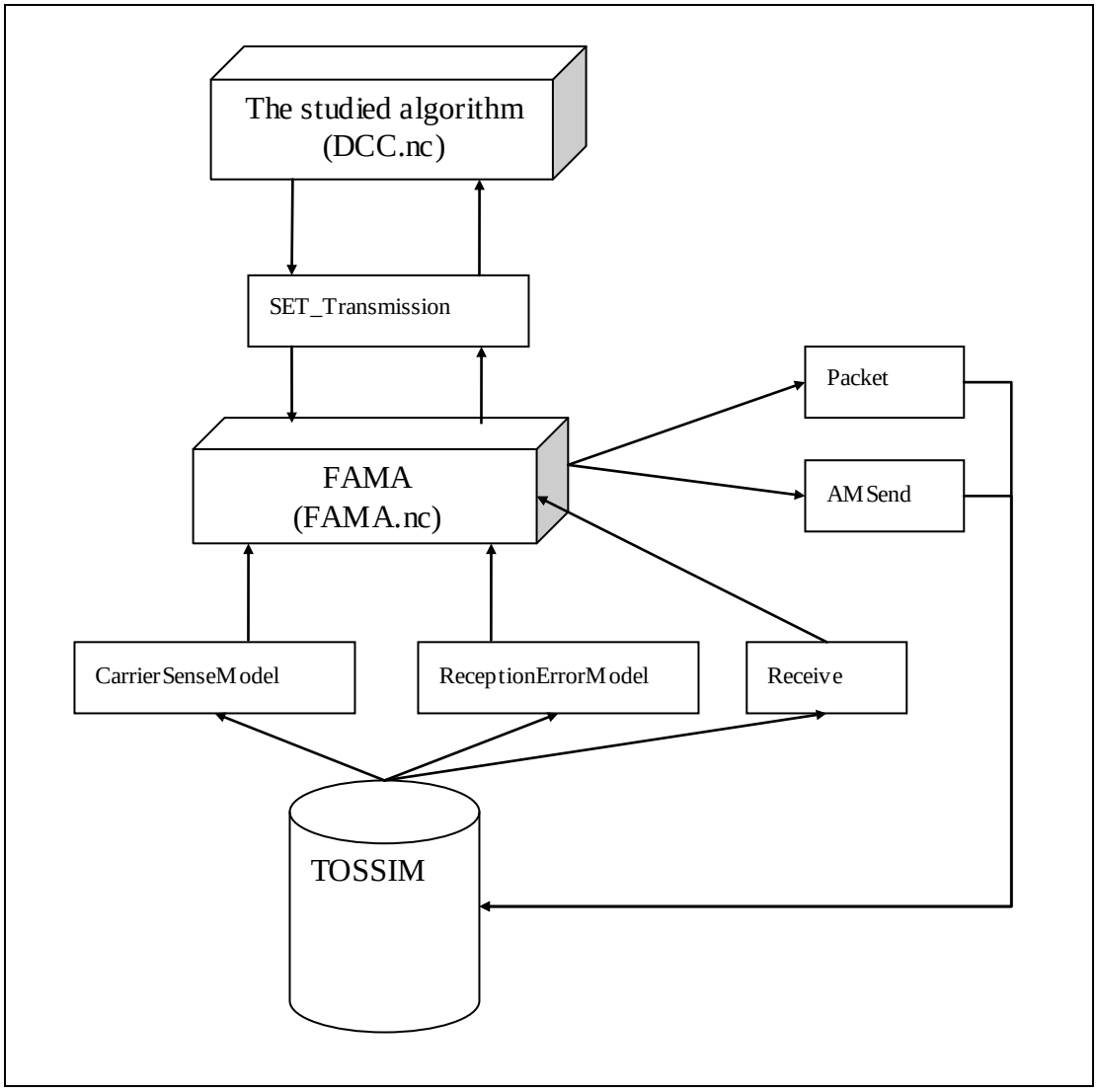


Figure 4: Structure of our program

A.2 Modules

Modules	Description
DCC.nc	Implementation of non-oblivious algorithm of Leone et al. [1]. Responsible for controlling the whole program, sending packets to FAMA level, and process all the received packets.
FAMA.nc	Implementation of FAMA algorithm. Responsible for RTS/CTS exchange procedure before transmitting the packets passed from DCC, signal SET_carrier_sense() and SET_ reception_error () to DCC.

A.3 Interfaces

Interfaces	Description
SET_Transmission.nc	Responsible for interaction between DCC.nc and FAMA.nc, including methods: ● SET_carrier_sense() is used to signal the event SET_carrier_sense() of Leone et al. [1]. ● SET_reception_error() is used to signal the event SET_ reception_error () of Leone et al. [1]. ● SET_broadcast() is used to transmit packets to the other processors. ● SET_receive() is used to pass received packets to DCC.nc.

CarrierSenseModel.nc	Responsible for signaling the event carrier_sense() from interference model of TOSSIM to packet-level radio component; And signaling the event carrier_sense() from packet-level radio component of TOSSIM to FAMA.
ReceptionErrorModel.nc	Responsible for signaling the event reception_error() from interference model of TOSSIM to packet-level radio component; And signaling the event reception_error () from packet-level radio component of TOSSIM to FAMA.

APPENDIX B Codes

B.1 CarrierSenseModel.nc

```
interface CarrierSenseModel{
  /**
   * signal a carrier_sense event to upper-layer
   * @author gongxi zhu
  **/
  event void senseCarrier();
}
```

B.2 ReceptionErrorModel.nc

```
interface ReceptionErrorModel{
  /**
   * signal a reception_error event to upper-layer
   * @author gongxi zhu
  **/
  event void reception_error();
}
```

B.3 SET_Transmission.nc

```
interface SET_Transmission{
  /**
   * inform other components that program is booting
   * @author gongxi zhu
  **/
  command void start();

  /**
   * signal a SET_carrier_sense event to upper-layer
   * @author gongxi zhu
  **/
  event void SET_carrier_sense();

  /**
   * signal a SET_reception_error event to upper-layer
   * @author gongxi zhu
  **/
  event void SET_reception_error();

  /**
   * signal a SET_receive event to upper-layer, a message is received
   * @author gongxi zhu
   * @param msg received message
   * @param payload data payload of received message
   * @param len length of payload
   * @return received message
  **/
  event message_t* SET_receive(message_t* msg, void* payload, uint8_t len);

  /**
   * send a message
   * @author gongxi zhu
   * @param msg message
   * @param len length of payload
   * @param timeslot to whom I should send my RTS
   * @param competingRnd how many competition rnds I used
  **/
  command void SET_broadcast(message_t* msg, uint8_t len, int timeslot, int competingRnd);

  /**
```

```
   * inform other components the beginning of a timeslot
   * @author gongxi zhu
   * @param myslot the broadcasting timeslot I am using
   * @param round current broadcasting round
   * @param timeslot current broadcasting timeslot
  **/
  command void setTimeslot(int myslot, int16_t round, int16_t timeslot);

  /**
   * indicates a message has been broadcasted successfully
   * @author gongxi zhu
  **/
  event void successBroadcast();
}
```

B.4 HilTimerMicroC.nc

```
#include <Timer.h>

module HilTimerMicroC {
 provides interface Init;
 provides interface Timer<TMicro> as TimerMicro[uint8_t num];
}
implementation {

 enum {
  TIMER_COUNT = uniqueCount(UQ_TIMER_MICRO)
 };

 typedef struct tossim_timer {
  uint32_t t0;
  uint32_t dt;
  bool isPeriodic;
  bool isActive;
  sim_event_t* evt;
 } tossim_timer_t;

 tossim_timer_t timers[TIMER_COUNT];
 sim_time_t initTime;

 void initializeEvent(sim_event_t* evt, uint8_t timerID);

 sim_time_t clockToSim(sim_time_t clockVal) {
  return (clockVal * sim_ticks_per_sec()) / (1024 * 1024);
 }

 sim_time_t simToClock(sim_time_t sim) {
  return (sim * (1024 * 1024)) / sim_ticks_per_sec();
 }

 command error_t Init.init() {
  memset(timers, 0, sizeof(timers));
  initTime = sim_time();
  return SUCCESS;
 }

 command void TimerMicro.startPeriodic[uint8_t id]( uint32_t dt ) {
```

```
  call TimerMicro.startPeriodicAt[id](call TimerMicro.getNow[id](), dt);
}
command void TimerMicro.startOneShot[uint8_t id]( uint32_t dt ) {
  //dbg("Test_FAMA", "Timer %d starts!\n", id);
  call TimerMicro.startOneShotAt[id](call TimerMicro.getNow[id](), dt);
}

command void TimerMicro.stop[uint8_t id]() {
  //dbg("Test_FAMA", "Timer %d stops!\n", id);
  timers[id].isActive = 0;
  if (timers[0].evt != NULL) {
    timers[0].evt->cancelled = 1;
    timers[0].evt->cleanup = sim_queue_cleanup_total;
    timers[0].evt = NULL;
  }
}

// extended interface
command bool TimerMicro.isRunning[uint8_t id]() {return timers[id].isActive;}
command bool TimerMicro.isOneShot[uint8_t id]() {return !timers[id].isActive;}

command void TimerMicro.startPeriodicAt[uint8_t id]( uint32_t t0, uint32_t dt ) {
  call TimerMicro.startOneShotAt[id](t0, dt);
  timers[id].isPeriodic = 1;
}
command void TimerMicro.startOneShotAt[uint8_t id]( uint32_t t0, uint32_t dt ) {
  uint32_t currentTime = call TimerMicro.getNow[id]();
  sim_time_t fireTime = sim_time();

  call TimerMicro.stop[id]();

  timers[id].evt = sim_queue_allocate_event();
  initializeEvent(timers[id].evt, id);

  fireTime += clockToSim(dt);

  // Be careful about signing and casts, etc.
  if (currentTime > t0) {
    fireTime -= clockToSim(currentTime - t0);
  }
  else {
    fireTime += clockToSim(t0 - currentTime);
  }
```

```
  timers[id].evt->time = fireTime;
  timers[id].isPeriodic = 0;
  timers[id].isActive = 1;
  timers[id].t0 = t0;
  timers[id].dt = dt;

  sim_queue_insert(timers[id].evt);
}

command uint32_t TimerMicro.getNow[uint8_t id]() {
  sim_time_t nowTime = sim_time();
  nowTime -= initTime;
  nowTime = simToClock(nowTime);
  return nowTime & 0xffffffff;
}

command uint32_t TimerMicro.gett0[uint8_t id]() {
  return timers[id].t0;
}
command uint32_t TimerMicro.getdt[uint8_t id]() {
  return timers[id].dt;
}

void tossim_timer_handle(sim_event_t* evt) {
  uint8_t* datum = (uint8_t*)evt->data;
  uint8_t id = *datum;
  //dbg("Test_FAMA", "Timer %d fires!\n", id);
  if(timers[id].isActive)
    signal TimerMicro.fired[id]();

  // We should only re-enqueue the event if it is a follow-up firing
  // of the same timer.  If the timer is stopped, it's a one shot,
  // or someone has started a new timer, don't re-enqueue it.
  if (timers[id].isActive &&
 timers[id].isPeriodic &&
 timers[id].evt == evt) {
    evt->time = evt->time += clockToSim(timers[id].dt);
    sim_queue_insert(evt);
  }
  // If we aren't enqueueing it, and nobody has done something that
  // would cause the event to have been garbage collected, then do
  // so.
  else if (timers[id].evt == evt) {
    call TimerMicro.stop[id]();
```

```
    }
  }

  void initializeEvent(sim_event_t* evt, uint8_t timerID) {
    uint8_t* data = (uint8_t*)malloc(sizeof(uint8_t));
    *data = timerID;

    evt->handle = tossim_timer_handle;
    evt->cleanup = sim_queue_cleanup_none;
    evt->data = data;
  }

  default event void TimerMicro.fired[uint8_t id]() {}
}
```

B.5 DCC.nc

```
module DCC
{
  uses interface Boot;
  uses interface Packet;
  uses interface SET_Transmission;
  uses interface SplitControl as AMControl;
  uses interface Timer<TMicro> as Timer_Pulse;
  uses interface Timer<TMicro> as Timer2;
}

implementation
{
  message_t packet;
  int16_t counter;
  int16_t NumSlot = FRAME_SIZE;

  /**
   * index of broadcasting timeslot
  **/
  int16_t current_slot;

  /**
   * index of broadcasting round
  **/
  int16_t current_round;

  /**
   * am I still competing for this broadcasting timeslot
  **/
  bool competing;

  /**
   * the broadcasting timeslot I am using
  **/
  int16_t s = -1;

  /**
   * exposure time
  **/
  int16_t e = EXPOSURE_TIME;
```

```
/**
 * index of current competition round
**/
int k;

/**
 * to whom I should send my RTS
**/
int ds_timeslot;

/**
 * is this "ds_timeslot" available: if not, choose a new "ds_timeslot"
**/
bool avail_ds;

bool isUnique;
bool unused[FRAME_SIZE];
int terminated = 0;

FAMAMsgA_t* sopkt_a;
FAMAMsgB_t* sopkt_b;
FAMAMsgC_t* sopkt_c;
FAMAMsgD_t* sopkt_d;
FAMAMsgE_t* sopkt_e;
FAMAMsgF_t* sopkt_f;

/**
 * initilization
**/
void init(){
  int i;
  current_slot = FRAME_SIZE;
  current_round = -1;
  //MaxRound = 5;
  avail_ds = FALSE;
  counter = 0;
  for(i = 0; i < NumSlot; i ++){
    unused[i] = TRUE; // should each timeslot be unused?
  }
  call SET_Transmission.start();
}

event void Boot.booted() {
```

```
  call AMControl.start();
}

event void AMControl.startDone(error_t err) {
  if (err == SUCCESS) {
    dbg("Boot", "Node %d boots!\n", TOS_NODE_ID);
    init();
    call Timer_Pulse.startPeriodic(TIMER_PERIOD_MICRO);
  }
  else {
    call AMControl.start();
  }
}

event void AMControl.stopDone(error_t err){
}

bool test(){
  return isUnique;
}

uint16_t get_random_all(){
  double randomseed;
  double dslot;
  randomseed = fabs(RandomUniform());
  if(randomseed >= 1 || randomseed < 0){
    randomseed = 0;
  }
  dslot = floor(randomseed * NumSlot);
  return (uint16_t)dslot;
}

int16_t get_random_unused(){
  double randomseed;
  int i = 0;
  int j = 0;
  int unused_num = 0;
  uint16_t dslot;
  uint16_t dslot_index;

  while(i < NumSlot){
    if(unused[i] == TRUE){
      unused_num ++;
    }
```

```
    i ++;
  }

  if(unused_num == 1){
    i = 0;
    while(i < NumSlot){
      if(unused[i] == TRUE){
        dslot = i;
        break;
      }
      i ++;
    }
  }else if (unused_num == 0){
    //dslot = get_random_all();
    dslot = -1;
  }else{
    randomseed = fabs(RandomUniform());
    if(randomseed > 1){
      randomseed = 1;
    }else if(randomseed < 0){
      randomseed = 0;
    }
    dslot_index = (uint16_t)(floor(randomseed * unused_num));
    if(dslot_index == unused_num){
      dslot_index = dslot_index - 1;
    }
    i = 0;
    while(i <= dslot_index){
      while(j < NumSlot){
        if(unused[j] == TRUE){
          j ++;
          break;
        }
        j ++;
      }
      i ++;
    }
    dslot = -- j;
  }
  return dslot;
}

int16_t get_random_used(){
  double randomseed;
```

```
int i = 0;
int j = 0;
int unused_num = 0;
uint16_t dslot;
uint16_t dslot_index;

while(i < NumSlot){
  if(unused[i] == FALSE){
    unused_num ++;
  }
  i ++;
}

if(unused_num == 1){
  i = 0;
  while(i < NumSlot){
    if(unused[i] == FALSE){
      dslot = i;
      break;
    }
    i ++;
  }
}else if (unused_num == 0){
  dslot = get_random_all();
}else{
  randomseed = fabs(RandomUniform());
  if(randomseed > 1){
    randomseed = 1;
  }else if(randomseed < 0){
    randomseed = 0;
  }
  dslot_index = (uint16_t)(floor(randomseed * unused_num));
  if(dslot_index == unused_num){
    dslot_index = dslot_index - 1;
  }
  i = 0;
  while(i <= dslot_index){
    while(j < NumSlot){
      if(unused[j] == FALSE){
        j ++;
        break;
      }
      j ++;
    }
```

```
      i ++;
    }
    dslot = -- j;
    if(dslot == s){
      //dslot --;
      i = 0;
      while(i < NumSlot){
        if(unused[i] == FALSE && dslot != s){
          dslot = i;
          break;
        }
        i ++;
      }
    }
  }
  if(unused_num == 0 || unused_num == 1){
    while(dslot == s){
      dslot = get_random_all();
    }
  }
  return dslot;
}

bool prob(double possibility){
  double randomseed = fabs(RandomUniform());
  if(possibility >= 1 || possibility <= 0){
    return possibility;
  }else{
    if(randomseed < possibility){
      return 1;
    }else{
      return 0;
    }
  }
}

/**
 * start competition, send different packets according to how many competition rnds are used
**/
void send(sim_event_t* evt){
  int i = 0;
  int unused_num = 0;
  int comprnd = 0;
  if(!avail_ds){
```

```
  while(i < NumSlot){
    if(unused[i] == TRUE){
      unused_num ++;
    }
    i ++;
  }
  dbg("Test_DCC", "Node %d has %d unused slots!\n", TOS_NODE_ID, unused_num);
  ds_timeslot = get_random_used();
}
avail_ds = FALSE;
k = 1;
competing = TRUE;
isUnique = TRUE;
if(k <= MaxRound && competing){
  if( prob(pow(2.0, k - MaxRound))){
    competing = FALSE;
    if(k == 5){
      sopkt_a = (FAMAMsgA_t*)(call Packet.getPayload(&packet, NULL));
      if (sopkt_a == NULL) {
        return;
      }
      sopkt_a->nodeid = TOS_NODE_ID;
      sopkt_a->counter = counter;
      sopkt_a->pkt_len = sizeof(FAMAMsgA_t);
      counter ++;
      call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgA_t), ds_timeslot, k);
    }else if(k == 4){
      sopkt_b = (FAMAMsgB_t*)(call Packet.getPayload(&packet, NULL));
      if (sopkt_b == NULL) {
        return;
      }
      sopkt_b->nodeid = TOS_NODE_ID;
      sopkt_b->counter = counter;
      sopkt_b->pkt_len = sizeof(FAMAMsgB_t);
      counter ++;
      call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgB_t), ds_timeslot, k);
    }else if(k == 3){
      sopkt_c = (FAMAMsgC_t*)(call Packet.getPayload(&packet, NULL));
      if (sopkt_c == NULL) {
        return;
      }
      sopkt_c->nodeid = TOS_NODE_ID;
      sopkt_c->counter = counter;
      sopkt_c->pkt_len = sizeof(FAMAMsgC_t);
```

```
        counter ++;
        call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgC_t), ds_timeslot, k);
      }else if(k == 2){
        sopkt_d = (FAMAMsgD_t*)(call Packet.getPayload(&packet, NULL));
        if (sopkt_d == NULL) {
          return;
        }
        sopkt_d->nodeid = TOS_NODE_ID;
        sopkt_d->counter = counter;
        sopkt_d->pkt_len = sizeof(FAMAMsgD_t);
        counter ++;
        call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgD_t), ds_timeslot, k);
      }else if(k == 1){
        sopkt_e = (FAMAMsgE_t*)(call Packet.getPayload(&packet, NULL));
        if (sopkt_e == NULL) {
          return;
        }
        sopkt_e->nodeid = TOS_NODE_ID;
        sopkt_e->counter = counter;
        sopkt_e->pkt_len = sizeof(FAMAMsgE_t);
        counter ++;
        call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgE_t), ds_timeslot, k);
      }
      return;
    }else{
      call Timer2.startOneShot(e);//wait(e);
    }
    k ++;
  }
}

event void Timer2.fired() {
  int comprnd = 0;
  if(k <= MaxRound && competing){
    if( prob(pow(2.0, k - MaxRound))){
      competing = FALSE;
      if(k == 5){
        sopkt_a = (FAMAMsgA_t*)(call Packet.getPayload(&packet, NULL));
        if (sopkt_a == NULL) {
          return;
        }
        sopkt_a->nodeid = TOS_NODE_ID;
        sopkt_a->counter = counter;
        sopkt_a->pkt_len = sizeof(FAMAMsgA_t);
```

```
  counter ++;
  call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgA_t), ds_timeslot, k);
}else if(k == 4){
  sopkt_b = (FAMAMsgB_t*)(call Packet.getPayload(&packet, NULL));
  if (sopkt_b == NULL) {
    return;
  }
  sopkt_b->nodeid = TOS_NODE_ID;
  sopkt_b->counter = counter;
  sopkt_b->pkt_len = sizeof(FAMAMsgB_t);
  counter ++;
  call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgB_t), ds_timeslot, k);
}else if(k == 3){
  sopkt_c = (FAMAMsgC_t*)(call Packet.getPayload(&packet, NULL));
  if (sopkt_c == NULL) {
    return;
  }
  sopkt_c->nodeid = TOS_NODE_ID;
  sopkt_c->counter = counter;
  sopkt_c->pkt_len = sizeof(FAMAMsgC_t);
  counter ++;
  call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgC_t), ds_timeslot, k);
}else if(k == 2){
  sopkt_d = (FAMAMsgD_t*)(call Packet.getPayload(&packet, NULL));
  if (sopkt_d == NULL) {
    return;
  }
  sopkt_d->nodeid = TOS_NODE_ID;
  sopkt_d->counter = counter;
  sopkt_d->pkt_len = sizeof(FAMAMsgD_t);
  counter ++;
  call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgD_t), ds_timeslot, k);
}else if(k == 1){
  sopkt_e = (FAMAMsgE_t*)(call Packet.getPayload(&packet, NULL));
  if (sopkt_e == NULL) {
    return;
  }
  sopkt_e->nodeid = TOS_NODE_ID;
  sopkt_e->counter = counter;
  sopkt_e->pkt_len = sizeof(FAMAMsgE_t);
  counter ++;
  call SET_Transmission.SET_broadcast( &packet, sizeof(FAMAMsgE_t), ds_timeslot, k);
}
return;
```

```
      }else{
        call Timer2.startOneShot(e);//wait(e);
      }
      k ++;
    }
  }

  /**
   * a message has been broadcasted successfully, I could successfully send my data packet after RTS-
CTS hand shake
   * @author gongxi zhu
  **/
  event void SET_Transmission.successBroadcast(){
    avail_ds = TRUE;
  }

  /**
   * sense the carrier, and read the energy on the channel
   * @author gongxi zhu
  **/
  event void SET_Transmission.SET_carrier_sense(){
    if(competing){
      isUnique = FALSE;
    }
    competing = FALSE;
    unused[current_slot] = FALSE;
  }

  /**
   * detect reception_error
   * @author gongxi zhu
  **/
  event void SET_Transmission.SET_reception_error(){
    unused[current_slot] = TRUE;
  }

  sim_event_t* allocate_send_event(sim_time_t endTime) {
     sim_event_t* evt = (sim_event_t*)malloc(sizeof(sim_event_t));
     evt->mote = sim_node();
     evt->time = endTime;
     evt->handle = send;
     evt->cleanup = sim_queue_cleanup_event;
     evt->cancelled = 0;
     evt->force = 1;
```

```
    return evt;
  }

  /**
   * start a timeslot
  **/
  event void Timer_Pulse.fired() {
    uint32_t time1;
    sim_event_t* evt;

    if(current_round > TestRnd){
      call Timer_Pulse.stop();
      call Timer2.stop();
      terminated = 1;
      return;
    }
    if(current_slot < FRAME_SIZE - 1){
      current_slot ++;
      if(TOS_NODE_ID == 0)
        dbg("Test_DCC", "=============Slot %d       ==============\n", current_slot);
    }else if (current_slot >= FRAME_SIZE - 1){
      current_slot = 0;
      current_round ++;
      if(TOS_NODE_ID == 0){
        dbg("Test_DCC", "=============Round %d Starts==============\n", current_round);
        dbg("Test_DCC", "=============Slot %d       ==============\n", current_slot);
      }
    }

    if(current_slot == 0){
      if(!test() || s == -1){
        s = get_random_unused();
        dbg("Test_DCC", "Node %d chooses slot %d in round %d!\n",TOS_NODE_ID, s,
current_round);
      }
      isUnique = FALSE;
    }
    unused[current_slot] = TRUE;
    call SET_Transmission.setTimeslot(s,current_round, current_slot);
    if(s != -1 && current_slot == s){
      evt = allocate_send_event(10 + sim_time());
      sim_queue_insert(evt);
    }
  }
```

```
  event message_t* SET_Transmission.SET_receive(message_t* msg, void* payload, uint8_t len){
    return msg;
  }
}
```

B.6 FAMA.nc

```
module FAMAC
{
  uses interface Packet;
  uses interface AMSend;
  uses interface Receive;
  uses interface Timer<TMicro> as Timer;
  uses interface CarrierSenseModel as CarrierSense;
  uses interface ReceptionErrorModel as ReceptionError;

  provides interface SET_Transmission;
}
implementation
{
  bool locked = FALSE;
  message_t* pkt;
  message_t spkt;
  message_t rts;
  message_t cts;
  int state;
  uint8_t length;
  bool dflag = FALSE;
  int my_slot;
  int this_slot;
  int ds_slot;
  int16_t current_round = 0;
  int16_t current_slot = 0;
  bool canReceptError = FALSE;
  uint16_t Ta = 0;
  bool recording = FALSE;
  int competingRnd = -1;

  void passive();
  void rts_f(uint16_t);
  void transmitRTS();

  uint16_t getRandomBackoff(uint32_t time1){
    double randomseed;
    double factor;
    randomseed = fabs(RandomUniform());
    factor = randomseed * 10.0;
```

```
  time1 = (uint16_t)(floor(time1 * factor)) ;
  return time1;

}

/**
 * initialize the FAMA
**/
command void SET_Transmission.start(){
  state = START;
  call Timer.startOneShot(1);
}

/**
 * inform the beginning of a timeslot
**/
command void SET_Transmission.setTimeslot(int myslot, int16_t round, int16_t timeslot){
  current_round = round;
  current_slot = timeslot;
  my_slot = myslot;
  dflag = FALSE;
  recording = FALSE;
  Ta = 0;
  competingRnd = -1;
  call Timer.stop();
  passive();
}

/**
 * start FAMA
**/
command void SET_Transmission.SET_broadcast(message_t* msg, uint8_t len, int timeslot, int rnd){
  ds_slot = timeslot;
  pkt = msg;
  length = len;
  competingRnd = rnd;
  canReceptError = FALSE;
  if(state == PASSIVE){
    transmitRTS();
  }
}

/**
 * transmmit RTS
```

```
**/
void transmitRTS(){
  FAMARTS_t* myrts = (FAMARTS_t*)(call Packet.getPayload(&rts, NULL));
  myrts->receiver = ds_slot;
  myrts->sender = TOS_NODE_ID;
  if (call AMSend.send(AM_BROADCAST_ADDR, &rts, sizeof(FAMARTS_t)) == SUCCESS) {
  }
}

/**
 * turn to PASSIVE state
**/
void passive(){
  state = PASSIVE;
}

/**
 * turn to RTS state
**/
void rts_f(uint16_t time1){
  state = RTS;
  call Timer.startOneShot(time1);
}

void backoff(){
  state = BACKOFF;
  call Timer.startOneShot(getRandomBackoff(CTS_SIZE));
}

/**
 * transmmit packets
**/
void transmitData(){
  FAMAMsgA_t* sopkt;
  if (call AMSend.send(AM_BROADCAST_ADDR, pkt, length) == SUCCESS) {
    sopkt = (FAMAMsgA_t*)(call Packet.getPayload(pkt, NULL));
    signal SET_Transmission.successBroadcast();
  }
}

/**
 * turn to REMOTE state
**/
void remote(uint16_t time1, bool flag){
```

```
    state = REMOTE;
    dflag = flag;
    Ta = time1;
    call Timer.startOneShot(time1);
  }

  /**
   * turn to XMIT
  **/
  void xmit(){
    state = XMIT;
    call Timer.startOneShot(1);
  }

  event void Timer.fired(){
    switch(state){
      case START:
        passive();
        break;
      case RTS:
        passive();
        break;
      case CTS:

      case XMIT:
        transmitData();
        break;
      case REMOTE:
        passive();
        break;
      case BACKOFF:
        if(pkt != NULL)
          transmitRTS();
        break;
    }
  }

  event void AMSend.sendDone(message_t* msg, error_t err) {
    if (&rts == msg) {
      rts_f( TXRX_TURNAROUND + PROCESSING_TIME_CS + 2 * MAX_PROPA_DELAY +
CTS_SIZE );//+ 100
    }else if (&cts == msg) {
      remote(2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS + TXRX_TURNAROUND,
TRUE);
```

```
    }else if (pkt == msg) {
      pkt = NULL;
      passive();
      //locked = FALSE;
    }
  }

  /**
   * sense the carrier
   * @author gongxi zhu
  **/
  event void CarrierSense.senseCarrier(){
    if(state == START){
      call Timer.stop();
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
    }else if(state == PASSIVE){
      //dbg("Test_FAMA", "Node %d changes to state remote!\n", TOS_NODE_ID);
      canReceptError = TRUE;
      signal SET_Transmission.SET_carrier_sense();
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, FALSE);
    }else if(state == BACKOFF){
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, FALSE);
    }else if(state == RTS){
      call Timer.stop();
    }else if(state == REMOTE){
      call Timer.stop();
    }
  }

  /**
   * can not lock the signal, so signal reception_error
   * @author gongxi zhu
  **/
  event void ReceptionError.reception_error(){
    if(state == REMOTE){
      if(canReceptError){
        dbg("Test_FAMA", "Node %d reception error remote!\n", TOS_NODE_ID);
        canReceptError = FALSE;
        signal SET_Transmission.SET_reception_error();
      }
```

```
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
    }else if(state == PASSIVE){
      if(canReceptError){
        canReceptError = FALSE;
        signal SET_Transmission.SET_reception_error();
      }
    }else if(state == RTS){
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
    }
  }

  event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len){
    FAMARTS_t* ropkt;
    FAMACTS_t* copkt;
    FAMAMsgA_t* mopkt;
    message_t* message = NULL;
    if (len == sizeof(FAMARTS_t) && state == REMOTE) { //receive RTS
      call Timer.stop();
      ropkt = (FAMARTS_t*)payload;
      if(dflag){
        remote(Ta, TRUE);
        return msg;
      }
      canReceptError = FALSE;
      if(ropkt->receiver == my_slot){
        FAMACTS_t* mycts = (FAMACTS_t*)(call Packet.getPayload(&cts, NULL));
        mycts->receiver = ropkt->sender;
        mycts->sender = TOS_NODE_ID;
        mycts->slot = my_slot;

        recording = TRUE;
        if (call AMSend.send(AM_BROADCAST_ADDR, &cts, sizeof(FAMACTS_t)) == SUCCESS) {
        }
      }else{
        remote(CTS_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
      }
    }else if(len == sizeof(FAMACTS_t) && state == RTS){ //receive CTS
      copkt = (FAMACTS_t*)payload;
      if(copkt->receiver == TOS_NODE_ID){
        xmit();
      }
```

```
      else{
        remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
      }
    }else if(len == sizeof(FAMACTS_t) && state == REMOTE){ //receive CTS
      call Timer.stop();
      copkt = (FAMACTS_t*)payload;
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
      return msg;
    }else if((len == sizeof(FAMAMsgA_t)
          ||len == sizeof(FAMAMsgB_t)
          ||len == sizeof(FAMAMsgC_t)
          ||len == sizeof(FAMAMsgD_t)
          ||len == sizeof(FAMAMsgE_t)
          ||len == sizeof(FAMAMsgF_t)) && state == REMOTE){ //receive msg
      call Timer.stop();
      mopkt = (FAMAMsgA_t*)payload;
      if(recording){
        recording = FALSE;
      }else{
      }
      canReceptError = FALSE;
      message = signal SET_Transmission.SET_receive(msg, payload, len);
      remote(2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS + TXRX_TURNAROUND,
TRUE);
    }else if(len == sizeof(FAMACTS_t)){
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
    }else{
      remote(MAX_PACKET_SIZE + 2 * MAX_PROPA_DELAY + PROCESSING_TIME_CS +
TXRX_TURNAROUND, TRUE);
    }

    return msg;
  }
}
```