A Discrete Event Simulation Model for
“Efficient Selection of Relay Vehicles for
Broadcasting on Vehicular Ad-hoc NETworks”

Wei-Hsiang Hung and Shun-Ren Yang

I. SIMULATION DESIGN

This chapter describes discrete event-driven simulation model for DIB and EDIB protocols on VANET. We define six types of events listed as follows:

- The MOVE event represents that vehicles cross a grid.
- The T_RTB event implies that vehicles disseminate RTB message before warning message transmissions occur.
- The T_CTB event represents that rebel vehicles reply the corresponding CTB message.
- The T_DATA event represents that senders starts to transmit data.
- The T_ACK event represents that relay vehicles received the warning message and transmit the ACK message to the sender.

The following variables are used in the simulation model:

- \textit{vehs} stores the status of all vehicles, i.e., vehicle ID, current grid, moving speed, residence time of current grid and whether it is on data transmission.
- \textit{grids} describes vehicle distribution, for example, grid \textit{n} has vehicles \textit{A}, \textit{B} and \textit{C}.
- \textit{tsc} stores the state of current transmission.
- \textit{bwUsage} describes the usage of network bandwidth. In Fig. 1 example, the transmission range of each vehicle is 1 grid. When vehicle 3 transmit the data to vehicle 4, \textit{bwUsage} will record the grid 2,3 and 4 in the \texttt{sender_usage} part for vehicle 3, and the grid 1,2 and 3 in the \texttt{receiver_usage} part for vehicle 4.
Fig. 1. Bandwidth usage example

- $AR$ stores the available range of emergency message.

Fig. 2 shows the software architecture of our simulator. We divide this architecture into HandOff (H.O) Chain module, Transmission (TSC) Chain module and Data module in the software architecture. H.O Chain module mainly handles the MOVE event whenever any vehicle crosses a grid. Then, if the vehicle is on the transmission as MOVE event occurs, it will to check whether the distance between this vehicle and the sender is larger than the transmission range. If it is true, we set its transmission event to be failure. TSC Chain module handles each related transmission event (T_RTB, T_CTB, T_DATA and T_ACK), and it will query or update the Data module when any event occurs. Besides, if the transmission event is failure, TSC Chain module does not do anything for this event, and clear related information about this event. Data module is responsible for recording $vehs$, $grids$ and $bwUsage$, i.e., $vehs$ is recorded in Vechilce distribution entity, $grids$ is used in Collision record table entity and $bwUsage$ is recorded in Bandwidth usage entity. Besides, it will provide the required information of others Chain module.

A timestamp is maintained in each event to indicate when the event occurs. Events are inserted into an event list and are deleted/processed from the list in a non-decreasing timestamp order.
In the process of MOVE event, we generate the speed of each vehicle by discrete uniform distribution, and utilize this speed to calculate the residence time of one grid for this vehicle. Therefore, we can calculate the timestamp of next MOVE event. Fig. 3 illustrates the flowchart of our simulation model.

Step 1-4 initializes the variables \( \text{vehs}, \text{grids}, \text{tsc} \) and \( \text{bwUsage} \). In Step 5, a vehicle wants to transmit emergency message and generates T_RTB event for this transmission request. Then, determines when this event occurs and push this event into the event queue. Besides, we generate the next MOVE event for each vehicle, and push theirs into the event queue. Step 6 retrieves the event \( e \) from the top of the event list. Step 7 checks the type of event \( e \), and dispatches this event to suitable event handler of TSC Chain module. Step 8 checks whether there are other vehicles around its transmit black burst through by querying \( \text{bwUsage} \) variable. If it is true, it represents there exists the farther vehicle and this vehicle will give up to transmit T_CTB.

Fig. 2. Simulation software architecture
(1) Initialize vehs array
(2) Initialize grids array
(3) Initialize tsc record
(4) Initialize bwUsage array
(5) Generate a T_RTB event and the MOVE events for all item in vehs
(6) Remove the next event e from the event list
(8) Whether hears any noise by bwUsage
(9) generate the T_ACK event.
(10) e.veh.isTsc==true
(11) e.veh.position > AR?
(12) generate the T_RTB event.
(13) Whether out-of-range of tsc
(14) Find all candidates within the transmission range of e.sender
(15) Set these candidates into bwUsage
(16) Calculate the length of BB signal and generate the T_CTB events for each candidate.
(17) Set tsc.sender and tsc.receiver
(18) Clear all information from tsc and bwUsage
(19) Generate T_RTB event.
(20) Set e.veh.position and assign to veh
(21) Set e.veh.speed
Generate next MOVE event of e.veh.
(22) T_DATA event.
(23) Clear e.veh from bwUsage
(24) e.time = now + Time_RTB
(25) e.time = now + Time_CTB
(26) e.time = now + Time_DATA
(27) T_ACK
(28) T_DATA
(29) T_ACK

Fig. 3. Flowchart of Simulation Model

Conversely, we generate the timestamp of the T_Data event by calculating the consumed time of CTB message transmission, and push this T_Data event into the event queue. Step 9 represents that the sender start to transmit the data to the relay vehicle, and Time_Data is the consumed time of data transmission. We assign the timestamp of a new T_ACK event for the receiver by calculating the consumed time of data transmission and push this T_ACK event into the event queue. Step 10 implies that a vehicle crosses one grind, i.e., MOVE event occurs. We checks whether the vehicle is on transmission. If it is true, we have to check whether the transmission will be failed by out-of-range transmission, and the detailed process we have described above. Step 11 checks the position of the sender is greater than the available range. If it is true, it means that it does not need to relay, that is, the finish of this data dissemination. If it is false, this sender will find out each candidate within its transmission range and calculate the timestamp of...
T_CTB event for each candidate according to the length of black-burst calculation. Then, push these T_CTB events into the event queue. Step 12 generates a new T_RTB event for the next data transmission, and \textit{rndTime} is an interval time for next data dissemination. Step 13 checks whether the vehicle is out of transmission range. Step 14 finds out the relay vehicle from all candidates during candidate competition phase. Step 15 sets the \textit{bwUsage} for this transmission. Step 16 calculates the length of the black burst signal for the relay vehicle, and set its timestamp of T_CTB event occurrence (\textit{Time_{RTB}} is the RTB message transmission time). Step 17 sets the current \textit{tsc} for this transmission. Step 18, 23 clears all information from \textit{bwUsage}. Step 19 generates a new T_RTB event for the sender. Step 20 changes the position of the vehicle. Step 21 re-determines the speed of the vehicle, and re-determines the residence time (\textit{rRsiTime}) by this new speed. Step 22 generates a new T_DATA event for the sender.

II. SIMULATION CODES

In this chapter, we show an engine code of our simulator. We refer this code to be EventDispatcher. It manage life cycle of all events, and determine which event will be handle according to its timestamp. Besides, \texttt{Simulate()} function is the entry point of this program, and it utilize a while-loop to repeat all event occurrence until the count the emergency message reaches 100000. The detail is shown below:

\begin{verbatim}
1 #include "EventDispatcher.h"
2
3 #include "Config.h"
4 #include "random.h"
5 #include "VDL.h"
6 #include "BUL.h"
7 #include "NetworkUtils.h"
8 #include "TRTBEvent.h"
9 #include "ReRTBTable.h"
10 #include "TCTBEvent.h"
\end{verbatim}

June 20, 2008
```cpp
#include "AnalyticLog.h"

#include <sstream>

using namespace std;

EventDispatcher* EventDispatcher::_Instance = 0;

EventDispatcher::EventDispatcher()
{
    this->_eventQueue = priority_queue<IEvent*, vector<IEvent*>,
                        allocator<IEvent*>>, EventComparison>();

    // calculate each grid length
    double lengthPerGrid = (Config::ROAD_LENGTH*1.0/Config::MAXIMUMROADGRID*1.0);

    // calculate the speed upper bound
    double speedForMPerSecondU = ((Config::MEAN_VEHICLE_SPEED_U
                                *1000.0)/3600.0);

    // calculate the speed lower bound
    double speedForMPerSecondL = ((Config::MEAN_VEHICLE_SPEED_L
                                *1000.0)/3600.0);

    // find the lower bound of uniform distribution
    double meanGridResiTimeU = lengthPerGrid/speedForMPerSecondL;

```
/find the upper bound of uniform distribution

double meanGridResiTimeL = lengthPerGrid / speedForMPerSecondU;

// generate the residence random variable generator
_RnResiTime = Uniform(meanGridResiTimeL, meanGridResiTimeU);

_NowTimeStamp = 0.0;

EventDispatcher::~EventDispatcher()
{
    _Instance = NULL;
    while (!_eventQueue.empty()) {
        IEvent* event = _eventQueue.top();
        _eventQueue.pop();
        delete event;
    }
}

EventDispatcher* EventDispatcher::GetInstance() {
    if (_Instance == NULL) {
        _Instance = new EventDispatcher();
    }
    return _Instance;
}

void EventDispatcher::Simulate() {
    int tmp = -1;

// Event dispatcher loop
while (AnalyticLog::TRANSMISSION_COUNT < 100000) {

    // retrieve an event from the queue.
    IEvent* event = this->_eventQueue.top();
    this->_NowTimeStamp = event->GetTimeStamp();
    this->_eventQueue.pop();
    IEvent* nextEvent = this->_eventQueue.top();

    // push the events with the same timeslot into the tmp queue
    list<IEvent*> entList;
    entList.push_back(event);
    while (event->GetTimeSlot() == nextEvent->GetTimeSlot()) {
        entList.push_back(nextEvent);
        this->_eventQueue.pop();

        event = nextEvent;

        if (this->_eventQueue.empty())
            break;

        nextEvent = this->_eventQueue.top();
    }

    // check the size of tmp queue is greater than 1.
    if (entList.size() > 1) {

list <IEvent*> TEntList;

list <IEvent*>::iterator it;

for (it = entList.begin(); it != entList.end(); it++) {
    if (((*it)->EventType == EVENT_NORMAL_T) {
        TEntList.push_back(*it);
    }
}

// check whether the hidden node problem exists
if (TEntList.size() > 1) {
    // check whether CTB competition occurs
    if (this->IsLegalMultipleCTB(&TEntList)) {
        // check whether CTB collision occurs
        if (this->IsFarestCTBCollision(TEntList)) {
            this->GenerateReTRTB(&TEntList);
        }
    } else {
        // Clear all transmission information
        list <IEvent*>::iterator it;
        for (it = TEntList.begin(); it != TEntList.end(); it++) {
            Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
            tmpTsc->ptSender->IsOnTSC = false;
        }
    }
}
BUL::GetInstance() -> RemoveTransaction(tmpTsc);
}

}  }

}  
else  {
  // error handler
  Config::PAUSE();
}

// it is nothing event
for (it = entList.begin(); it != entList.end(); it++) {
  delete (*it);
}

  continue;
}

// the combination of events
for (it = entList.begin(); it != entList.end(); it++) {
  this->HandleNonMultipleCTBEvent(*it);
}

else  {
  this->HandleNonMultipleCTBEvent(event);
}

VDL::GetInstance() -> PrintData();

void EventDispatcher::HandleNonMultipleCTBEvent ( IEvent* pEvent ) {
    Transaction* tmpTsc = ( Transaction* ) pEvent->ptEventUnit;
    if ( pEvent->EventType==EVENT_NORMAL_T && tmpTsc->TscState == TscState_CTB ) {
        if ( this->IsFarestCTBCollision ( pEvent ) )
            pEvent->ProcessEvent ( );
        else {
            tmpTsc->ptSender->IsOnTSC = false;
            BUL::GetInstance ()->RemoveTransaction ( tmpTsc );
        }
    } else {
        pEvent->ProcessEvent ( );
    }
    delete pEvent;
}

bool EventDispatcher::IsFarestCTBCollision ( list < IEvent* > pEntList ) {
    // check whether it is longest black-burst
    TCTBEvent* tmpEvent = ( TCTBEvent* ) ( * ( pEntList.begin() ) );
    Transaction* tmpTsc = ( Transaction* ) tmpEvent->ptEventUnit;
    }
int farestSeg = ReRTBTable::GetInstance() \rightarrow GetParamByID(tmpTsc \rightarrow InitTscID) \rightarrow FarestSegmentNo;

int tmpSeg = tmpEvent\rightarrow GetReceiverSegmentNo();

if (farestSeg > tmpSeg) {
    list <IEvent*>::iterator it;

    for (it = pEntList.begin(); it != pEntList.end(); it++) {
        Transaction* tmpTsc = (Transaction*)(*it)\rightarrow ptEventUnit;
        Logger::Get(LOG_INFO, false) << ": [ " << tmpTsc\rightarrow ptSender\rightarrow VehicleID << "] \n";
    }

    Logger::Get(LOG_INFO, false) << endl;

    return false;
}

return true;

bool EventDispatcher::IsFarestCTBCollision(IEvent* pEnt) {
    list <IEvent*> tmpList;
    tmpList.push_back(pEnt);
    return IsFarestCTBCollision(tmpList);
}
bool EventDispatcher::IsLegalMultipleCTB(list <IEvent*>* pEntList)
{
    list <IEvent*>*::iterator it;

    for (it = pEntList->begin(); it != pEntList->end(); it++) {
        Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
        if (tmpTsc->TscState != TscState_CTB) {
            return false;
        }
    }

    int tmpInitTscID = -1;
    for (it = pEntList->begin(); it != pEntList->end(); it++) {
        Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
        if (tmpInitTscID < 0) {
            tmpInitTscID = tmpTsc->InitTscID;
        } else {
            if (tmpInitTscID != tmpTsc->InitTscID)
            {
                return false;
            }
            tmpInitTscID = tmpTsc->InitTscID;
        }
    }

    return true;
}
void EventDispatcher::GenerateReTRTB(list<IEvent*>* pEntList) {
    list<IEvent*>::iterator it;

    Transaction* tmpTsc;
    Vehicle* originSender;
    int originTscID, originAvailableRange;

    for (it = pEntList->begin(); it != pEntList->end(); it++) {
        tmpTsc = (Transaction*)(*it)->ptEventUnit;

        originSender = tmpTsc->ptReceiver;
        originTscID = tmpTsc->InitTscID;
        originAvailableRange = tmpTsc->AvailableRange;
        tmpTsc->ptSender->IsOnTSC = false;
        BUL::GetInstance()->RemoveTransaction(tmpTsc);
    }

    RTBParameter* param = ReRTBTable::GetInstance()->GetParamByID(originTscID);

    // generate new RTB event
    Transaction* tsc = BUL::GetInstance()->GetNewTransaction(TscState_RTB, originSender, NULL);
    tsc->IsTscValid = true;
    tsc->ptPreTsc = NULL;
    tsc->ResidualServiceTime = -1;
    tsc->InitTscID = originTscID;
}
tsc->AvailableRange = originAvailableRange;

int newStartingPos = param->FarestSegmentNo * param->
    NowSegmentWidth + param->NextStartingPoint;

ReRTBTable::GetInstance() -> SetStartingPosByID(tsc->InitTscID,
    newStartingPos);

TRTBEvent* event = new TRTBEvent(tsc, this->GetNowTime() +
    NetworkUtils::GetTransmissionTime(newStartingPos, Config::
    CTB_SIZE), EVENT_NORMAL_T);

ReRTBTable::GetInstance() -> ResetFarestSegByID(tsc->InitTscID);

PushEvent(event);
}

void EventDispatcher::GenerateNewRTBEvent() {
    Vehicle* sender = NULL;
    sender = VDL::GetInstance() -> GetLatestVeh();
    Transaction* tsc = BUL::GetInstance() -> GetNewTransaction(
        TscState_RTB, sender, NULL);
    tsc->IsTscValid = true;
    tsc->ptPreTsc = NULL;
    tsc->ResidualServiceTime = -1;
}
EventDispatcher::GetInstance() -> PushEvent(new TRTBEvent(tsc, 
    EventDispatcher::GetInstance() -> GetNowTime() + NetworkUtils:: 
    GetNewBackoffTime() , EVENT_NORMAL_T, tsc -> TscID ));

} 

void EventDispatcher::PushEvent( IEvent* pEvent ) { 
    this -> _eventQueue . push ( pEvent );

}