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:

- $vehs$ 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.
- $grids$ describes vehicle distribution, for example, grid n has vehicles A, B and C.
- tsc stores the state of current transmission.
- $bwUsage$ 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, $bwUsage$ will record the grid 2,3 and 4 in the $sender_usage$ part for vehicle 3, and the grid 1,2 and 3 in the $receiver_usage$ part for vehicle 4.

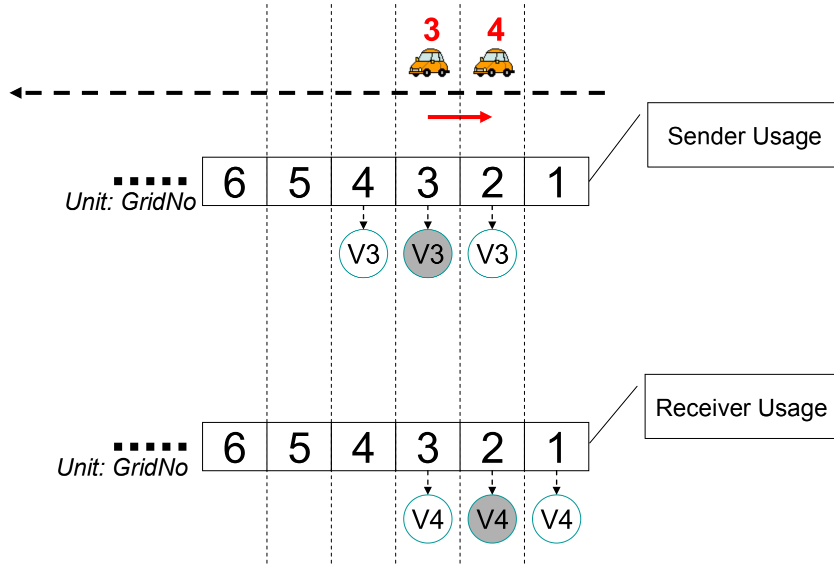

Fig. 1. Bandwidth usage example

- 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.

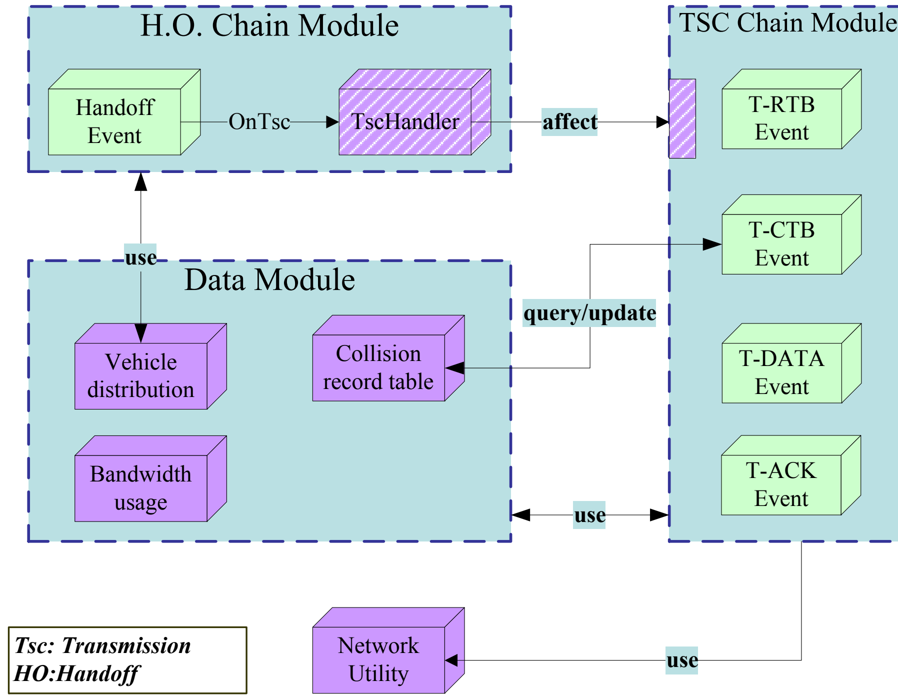

Fig. 2. Simulation software architecture

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 $vehs$, $grids$, tsc and $bwUsage$. 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 $bwUsage$ variable. If it is true, it represents there exists the farther vehicle and this vehicle will give up to transmit T_CTB.

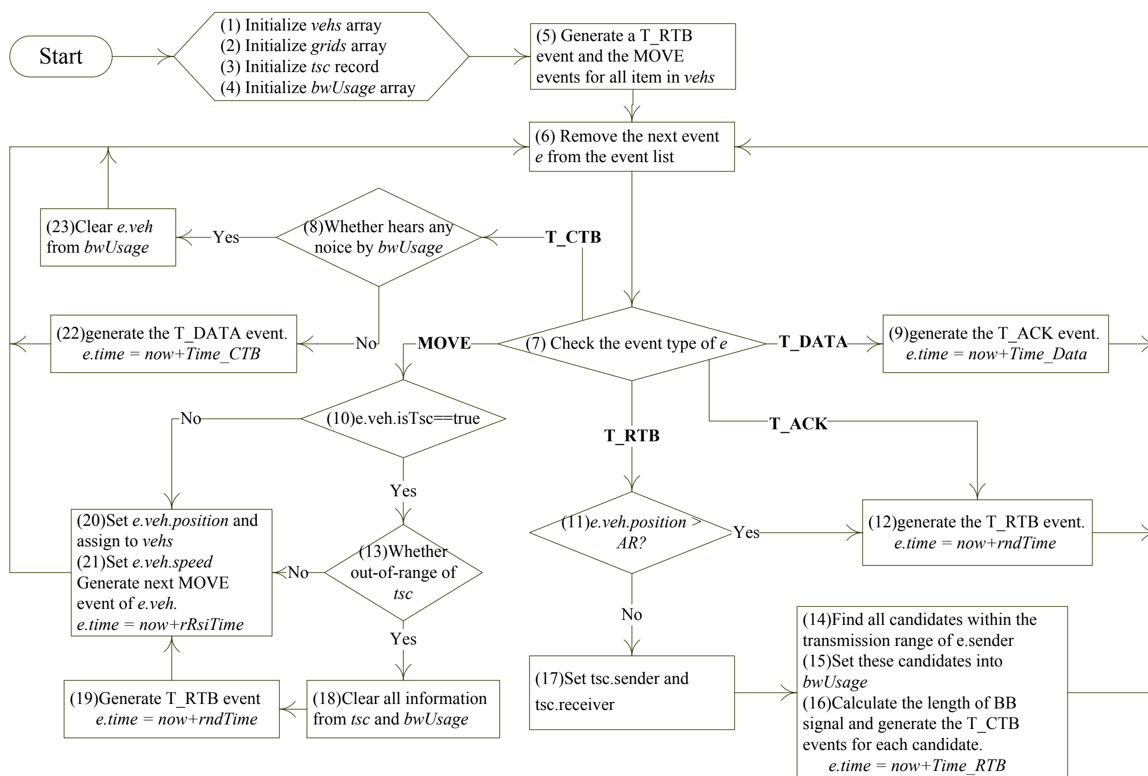

Fig. 3. Flowchart of Simulation Model

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 $rndTime$ 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 $bwUsage$ 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 ($Time_RTB$ is the RTB message transmission time). Step 17 sets the current tsc for this transmission. Step 18, 23 clears all information from $bwUsage$. 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 ($rRsiTime$) 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, 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:

```
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"
```

```
11 #include "AnalyticLog.h"
12
13 #include <sstream>
14
15 using namespace std;
16
17 EventDispatcher* EventDispatcher::_Instance = 0;
18
19 EventDispatcher::EventDispatcher()
20 {
21   this->_eventQueue = priority_queue<IEvent*, vector<IEvent*,
        allocator<IEvent*> >, EventComparison >();
22
23   //calculate each grid length
24   double lengthPerGrid = (Config::ROAD_LENGTH*1.0/Config::
       MAXIMUM_ROADGRID*1.0);
25
26   //calculate the speed upper bound
27   double speedForMPerSecondU = ((Config::MEAN_VEHICLE_SPEED_U
        *1000.0)/3600.0);
28
29   //calculate the speed lower bound
30   double speedForMPerSecondL = ((Config::MEAN_VEHICLE_SPEED_L
        *1000.0)/3600.0);
31
32   //find the lower bound of uniform distribution
33   double meanGridResiTimeU = lengthPerGrid/speedForMPerSecondL;
34
```

```
35    //find the upper bound of uniform distribution
36    double meanGridResiTimeL = lengthPerGrid/speedForMPerSecondU;
37
38    //generate the residence random variable generator
39    _RnResiTime = Uniform(meanGridResiTimeL, meanGridResiTimeU);
40
41    _NowTimeStamp = 0.0;
42  }
43
44  EventDispatcher::~EventDispatcher()
45  {
46    _Instance = NULL;
47    while (!_eventQueue.empty()) {
48      IEvent* event = _eventQueue.top();
49      _eventQueue.pop();
50      delete event;
51    }
52  }
53
54  EventDispatcher* EventDispatcher::GetInstance() {
55    if (_Instance == NULL) {
56      _Instance = new EventDispatcher();
57    }
58    return _Instance;
59  }
60
61  void EventDispatcher::Simulate() {
62    int tmp = -1;
```

```
63
64     //Event dispatcher loop
65     while (AnalyticLog::TRANSMISSION_COUNT < 100000) {
66
67       //retrieve an event from the queue.
68       IEvent* event = this->_eventQueue.top();
69       this->_NowTimeStamp = event->GetTimeStamp();
70       this->_eventQueue.pop();
71       IEvent* nextEvent = this->_eventQueue.top();
72
73       //push the events with the same timeslot into the tmp queue
74       list<IEvent*> entList;
75       entList.push_back(event);
76       while (event->GetTimeSlot() == nextEvent->GetTimeSlot()) {
77         entList.push_back(nextEvent);
78         this->_eventQueue.pop();
79
80         event = nextEvent;
81
82         if (this->_eventQueue.empty())
83           break;
84
85         nextEvent = this->_eventQueue.top();
86       }
87
88       //check the size of tmp queue is greater than 1.
89       if (entList.size() > 1) {
90
```

```
91      list<IEvent*> TEntList;
92
93      list<IEvent*>::iterator it;
94      for (it = entList.begin();it != entList.end();it++) {
95        if ((*it)->EventType == EVENT_NORMAL_T) {
96          TEntList.push_back(*it);
97        }
98      }
99
100     //check whether the hidden node problem exists
101     if (TEntList.size() > 1) {
102       ///check whether CTB competition occurs
103       if (this->IsLegalMultipleCTB(&TEntList)) {
104
105         //check whether CTB collision occurs
106         if (this->IsFarestCTBCollision(TEntList))
107         {
108           this->GenerateReTRTB(&TEntList);
109         }
110         else
111         {
112           //Clear all transmission information
113           list<IEvent*>::iterator it;
114           for (it =  TEntList.begin();it !=  TEntList.end();it
              ++) {
115             Transaction* tmpTsc = (Transaction*)(*it)->
                ptEventUnit;
116             tmpTsc->ptSender->IsOnTSC = false;
```

```
117                     BUL::GetInstance()->RemoveTransaction(tmpTsc);
118                   }
119                 }
120
121               } else {
122                 //error handler
123                 Config::PAUSE();
124               }
125
126               //it is nothing event
127               for (it = entList.begin(); it != entList.end(); it++) {
128                 delete (*it);
129               }
130
131               continue;
132             }
133
134             ///the combination of events
135             for (it = entList.begin(); it != entList.end(); it++) {
136               this->HandleNonMultipleCTBEvent(*it);
137             }
138           }
139           else {
140             this->HandleNonMultipleCTBEvent(event);
141           }
142           VDL::GetInstance()->PrintData();
143         }
144       }
```

```
145
146 void EventDispatcher::HandleNonMultipleCTBEvent(IEvent* pEvent) {
147   Transaction* tmpTsc = (Transaction*)pEvent->ptEventUnit;
148   if (pEvent->EventType==EVENT_NORMAL_T && tmpTsc->TscState ==
        TscState_CTB) {
149
150     if (this->IsFarestCTBCollision(pEvent))
151       pEvent->ProcessEvent();
152     else
153     {
154       tmpTsc->ptSender->IsOnTSC = false;
155       BUL::GetInstance()->RemoveTransaction(tmpTsc);
156     }
157
158   } else {
159     pEvent->ProcessEvent();
160   }
161
162   delete pEvent;
163 }
164
165 bool EventDispatcher::IsFarestCTBCollision(list<IEvent*> pEntList
      ) {
166   //check whether it is longest black-burst
167   TCTBEvent* tmpEvent = (TCTBEvent*)(*( pEntList.begin()));
168   Transaction* tmpTsc = (Transaction*)tmpEvent->ptEventUnit;
169
```

```
170     int farestSeg = ReRTBTable::GetInstance()->GetParamByID(tmpTsc
            ->InitTscID)->FarestSegmentNo;
171     int tmpSeg = tmpEvent->GetReceiverSegmentNo();
172     if (farestSeg > tmpSeg) {
173       list<IEvent*>::iterator it;
174
175       for (it = pEntList.begin(); it != pEntList.end(); it++) {
176         Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
177         Logger::Get(LOG_INFO, false) << "[" << tmpTsc->ptSender->
               VehicleID << "]␣";
178       }
179
180       Logger::Get(LOG_INFO, false) << endl;
181
182       return false;
183     }
184
185     return true;
186   }
187
188   bool EventDispatcher::IsFarestCTBCollision(IEvent* pEnt) {
189
190
191     list<IEvent*> tmpList;
192     tmpList.push_back(pEnt);
193     return IsFarestCTBCollision(tmpList);
194   }
195
```

```
196 bool EventDispatcher::IsLegalMultipleCTB(list<IEvent*>* pEntList)
      {
197   list<IEvent*>::iterator it;
198
199   for (it = pEntList->begin();it != pEntList->end();it++) {
200     Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
201     if (tmpTsc->TscState != TscState_CTB) {
202       return false;
203     }
204   }
205
206   int tmpInitTscID = -1;
207   for (it = pEntList->begin();it != pEntList->end();it++) {
208     Transaction* tmpTsc = (Transaction*)(*it)->ptEventUnit;
209     if (tmpInitTscID < 0) {
210       tmpInitTscID = tmpTsc->InitTscID;
211     } else {
212       if (tmpInitTscID != tmpTsc->InitTscID)
213       {
214         return false;
215       }
216       tmpInitTscID = tmpTsc->InitTscID;
217     }
218   }
219
220   return true;
221
222 }
```

```
223
224 void EventDispatcher::GenerateReTRTB(list<IEvent*>* pEntList) {
225   list<IEvent*>::iterator it;
226
227   Transaction* tmpTsc;
228   Vehicle* originSender;
229   int originTscID, originAvaiableRange;
230   for (it = pEntList->begin(); it != pEntList->end(); it++) {
231     tmpTsc = (Transaction*)(*it)->ptEventUnit;
232
233     originSender = tmpTsc->ptReceiver;
234     originTscID = tmpTsc->InitTscID;
235     originAvaiableRange = tmpTsc->AvaiableRange;
236     tmpTsc->ptSender->IsOnTSC = false;
237     BUL::GetInstance()->RemoveTransaction(tmpTsc);
238   }
239
240
241   RTBParameter* param = ReRTBTable::GetInstance()->GetParamByID(
        originTscID);
242
243   ///generate new RTB event
244   Transaction* tsc = BUL::GetInstance()->GetNewTransaction(
        TscState_RTB, originSender, NULL);
245   tsc->IsTscValid = true;
246   tsc->ptPreTsc = NULL;
247   tsc->ResidualServiceTime = -1;
248   tsc->InitTscID = originTscID;
```

```
249    tsc->AvaiableRange = originAvaiableRange;
250
251    int newStartingPos =param->FarestSegmentNo*param->
           NowSegmentWidth + param->NextStartingPoint;
252
253    ReRTBTable::GetInstance()->SetStartingPosByID(tsc->InitTscID,
           newStartingPos);
254
255    TRTBEvent* event = new TRTBEvent(tsc,this->GetNowTime()+
           NetworkUtils::GetTransmissionTime(newStartingPos,Config::
           CTB_SIZE),EVENT_NORMAL_T);
256
257    ReRTBTable::GetInstance()->ResetFarestSegByID(tsc->InitTscID);
258
259    PushEvent(event);
260 }
261
262 void EventDispatcher::GenerateNewRTBEvent() {
263    Vehicle* sender = NULL;
264
265    sender = VDL::GetInstance()->GetLatestVeh();
266
267    Transaction* tsc = BUL::GetInstance()->GetNewTransaction(
           TscState_RTB, sender,NULL);
268    tsc->IsTscValid = true;
269    tsc->ptPreTsc = NULL;
270    tsc->ResidualServiceTime = -1;
271
```

```
272     EventDispatcher::GetInstance()->PushEvent(new TRTBEvent(tsc,
            EventDispatcher::GetInstance()->GetNowTime()+NetworkUtils::
            GetNewBackoffTime(),EVENT_NORMAL_T, tsc->TscID));
273 }
274
275 void EventDispatcher::PushEvent(IEvent* pEvent) {
276     this->_eventQueue.push(pEvent);
277 }
```