Journal of Transportation Technologies, 2015, 5, 102-112
Published Online April 2015 in SciRes. http://www.scirp.org/journal/jtts
http://dx.doi.org/10.4236/jtts.2015.52010

Implementation and Evaluation of Local Dynamic Map in Safety Driving Systems

Hideki Shimada[1], Akihiro Yamaguchi[2], Hiroaki Takada[2], Kenya Sato[1]

[1]Mobility Research Center, Doshisha University, Kyotanabe, Japan
[2]Center for Embedded Computing Systems, Nagoya University, Nagoya, Japan
Email: hideki-s@is.naist.jp, ksato@mail.doshisha.ac.jp, yamagut@nces.is.nagoya-u.ac.jp, hiro@ertl.jp

Received 3 February 2015; accepted 27 March 2015; published 31 March 2015

Abstract

Cooperative safety driving systems using vehicle-to-vehicle and vehicle-to infrastructure communication are developed. Sensor data of vehicles and infrastructures are communicated in the cooperative safety driving system. LDM (Local Dynamic Map) is standardized by ETSI (European Telecommunications Standards Institute) to manage the vehicle sensor data and the map data. Implementations of LDM are reported on documents of ETSI, but there are no numerical results. The implementations of LDM are deployed the database management system. We think that the response time of the database becomes higher as the number of vehicles grows. In this paper, we have implemented and evaluated the LDM with the collision detection application.

Keywords

Local Dynamic Map, ITS, Cooperative Safety Driving Systems

1. Introduction

In addition to autonomous Intelligent Transport Systems (ITS) that use only the sensor information of an ego motor vehicle, recent studies have also been examining cooperative ITS systems that exchange sensor information through the use of vehicle-to-vehicle (V2V) communication and infrastructure-to-vehicle (I2V) communication. In Japan, the Advanced Safety Vehicle (ASV) [1] system promoted by the Ministry of Land, Infrastructure, Transport and Tourism (MLIT), Driving Safety Support System (DSSS) [2] overseen by the National Police Agency (NPA), and other systems are progressing toward actual implementation. Meanwhile, in Europe, the standardization of cooperative systems is progressing at standardization bodies such as the European Committee for Standardization (CEN) and European Telecommunications Standards Institute (ETSI). The adoption and spread of such cooperative ITS systems will generate a need for managing not only the sensor information of an ego motor vehicle but also the sensor information of other motor vehicles obtained through some means of communication.

As an Automotive Data Integration Project, we have been studying a mechanism for achieving vehicle control by integrated management of multiple types of in-vehicle sensor information using a database [3].

Against the above background, this paper focuses on Local Dynamic Map (LDM) [4] technology that is now being standardized in Europe. LDM achieves integrated management of map information and vehicle information. It provides a mechanism for dividing up data into layers according to its characteristics and for managing data used in cooperative ITS systems. A key effort related to LDM is the SAFESPOT Integrated Project that designs cooperative ITS systems to improve road safety [5]. The SAFESPOT development period ended in 2010. Portions of the project report have been released and research papers by project members have been presented [6]-[9]. These papers, however, while reporting on simulation-based evaluation experiments and field experiments at test sites, have not described any performance evaluations, so it is still not understood whether LDM specifications are applicable to actual operations.

In light of the above, we have performed an LDM implementation on computer based on the LDM specifications released by SAFESPOT. In this paper, we evaluate and analyze LDM performance while varying certain parameters such as the number of vehicle data items registered in LDM and the computer environment itself. The paper is organized as follows. Section 2 outlines the LDM concept, section 3 describes the LDM that we implemented, section 4 describes the evaluation experiment that we performed for this implementation and analyzes experimental results, and section 5 concludes the paper.

2. LDM (Local Dynamic Map)

LDM, which is now being standardized in Europe, is an aggregation of data for use by cooperative ITS. It adopts a four-layer model as shown in **Figure 1** [10]. The first or bottom layer consists of static data such as road data, the second layer consists of relatively static data such as signals not included in map data, the third layer consists of relatively dynamic data such as congestion and other traffic conditions, and the fourth or top layer consists of dynamic data such as automotive sensor information. In terms of a concrete data model, data tables have been defined in the SAFESPOT project for each of these four layers as shown in **Figure 2**.

Examples of LDM implementations have been reported including PG-LDM by Bosch and Tele Atlas and NAVTEQ-LDM by NAVTEQ [10]. The PG-LDM implementation adopts PostgreSQL as its database engine and provides for PostGIS stored procedures and spatial operations. The NAVTEQ-LDM implementation, meanwhile, adopts SQLite as its database engine.

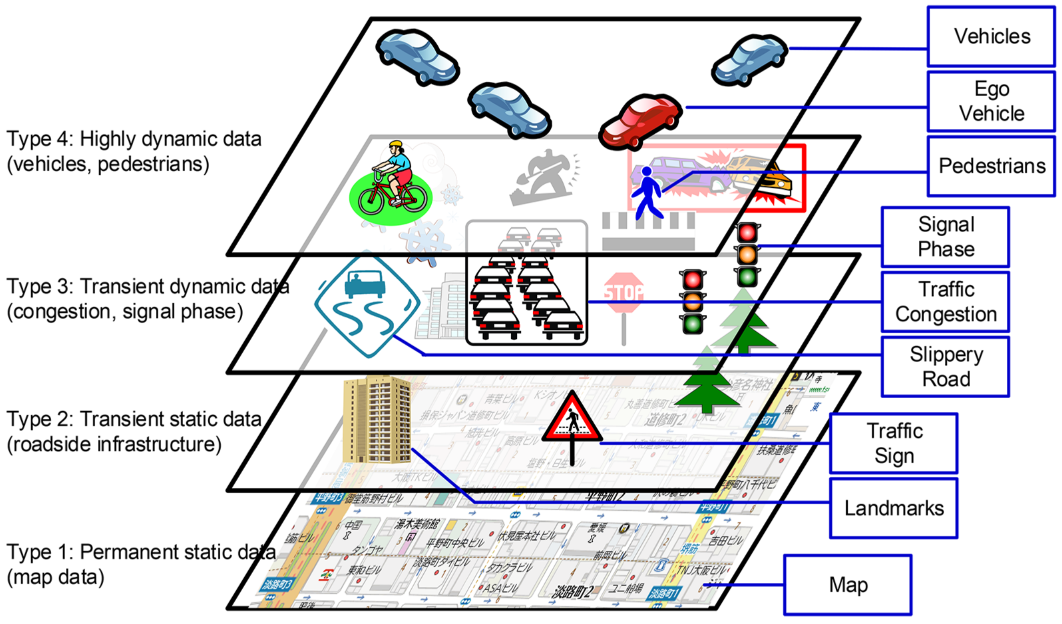

Type 4: Highly dynamic data
(vehicles, pedestrians)
Type 3: Transient dynamic data
(congestion, signal phase)
Type 2: Transient static data
(roadside infrastructure)
Type 1: Permanent static data
(map data)
Vehicles
Ego Vehicle
Pedestrians
Signal Phase
Traffic Congestion
Slippery Road
Traffic Sign
Landmarks
Map

Figure 1. The four layers of the LDM.

Staticfeatures
- arealandmarks
- crossing
- crossingforreferencetracks
- crossingsignalgroup
- detectionarea
- detectionareaforcrossing
- junction
- linelandmarks
- pointlandmarks
- referencetrack
- roadelement
- roadintersection
- sensor
- sensorfordetectionarea
- signalgroup
- signalgroupforreferencetrack
- trafficsign
- trafficsigninformation
- egorsu
- rsu

Movingobjects
- egomotorvehicle
- motorvehicle
- uo
- trailer

Relationships
- alongroadelement
- conceptualalongroadelement
- trajectory

Others
- gatewaycommunication
- messagebox

Conceptualobjects
- accidenthotspot
- dynamicblackspot
- dynamicreferencetrackattributes
- dynamicroadelementattribute
- dynamicsensorattributes
- dynamicsensorstatus
- dynamictrafficsign
- dynamictrafficsigninformation
- environmentalevent
- fcdevent
- meteodetection
- roadconditionevent
- roadconditionmeasurement
- signalgroupstate
- trafficevent
- trafficobject
- dynamicreferencetrack

Figure 2. LDM data model.

3. LDM Implementation

3.1. Overview

Based on the SAFESPOT specifications described in section 2, we set out to implement LDM on a general-purpose computer. We used PostgreSQL as a database management system (DBMS), PostGIS as a library for spatial operations extending PostgreSQL, and PL/pgSQL to implement stored procedures within PostgreSQL, all on Linux (Fedora10). We also used C++ to create an application programming interface (API) to access LDM from an application. In this section, we present the LDM elements needed by the safety driving system that we implemented and describe the structure of those elements.

3.2. Schema Definition

We created a schema of LDM tables as needed for achieving a safety driving system in conformance with current SAFESPOT specifications. In particular, we designed this schema with separate tables for managing static data such as road and map data and dynamic data such as vehicle data. Specifically, for map data on the first layer of LDM, we implemented a roadelement table for managing road data and a junction table and roadintersection table for managing intersection data. We used these tables of map data to manage road data such as intersections and associated location information. Next, for vehicle data on the fourth layer, we implemented an egomotorvehicle table for managing information of the ego motor vehicle and a motorvehicle table for managing information of other motor vehicles. We used these tables of vehicle information to manage vehicle IDs, vehicle sensor data, and associated location information. We also implemented the along road element table, which is one example of a relationship table defined by SAFESPOT for spanning and interconnecting different layers. Although road data on the first layer and vehicle data on the fourth layer are managed as independent layers, this relationship table relates map-data IDs to vehicle-data IDs enabling powerful searching to be performed. For example, the IDs of vehicles driving on a certain road can be obtained from the ID of that road and a road ID can be obtained from the ID of a vehicle driving on that road.

3.3. API

The LDM mechanism is essentially a database to which an application sends queries to obtain data. For this reason, an API is defined so that an application developer can work with the data stored in LDM. The LDM API is divided into Level 1 API and Level 2 API for performing basic operations and specialized processing, respectively.

In more detail, Level 1 API defines database operations like "select" and "update" also defined in SQL and includes spatial operations and transaction processing. Level 2 API extends Level 1 API and defines an API for special queries submitted by an application. For example, searching for vehicles on a certain road can be accomplished by calling Level 1 API several times, but Level 2 API enables such a search to be defined in the form of LDM:: L2API.getVehiclesOnRoadElement thereby simplifying the writing of an application.

Defining the LDM API in this way enables an application developer to implement an application without having to worry about the actual database software used to implement the LDM.

3.4. Data Used in Implementation

LDM groups data into layers according to the characteristics of that data. In this implementation, we used map data (first layer) and vehicle data (fourth layer) as described below to implement a safety driving system.

3.4.1. Map Data

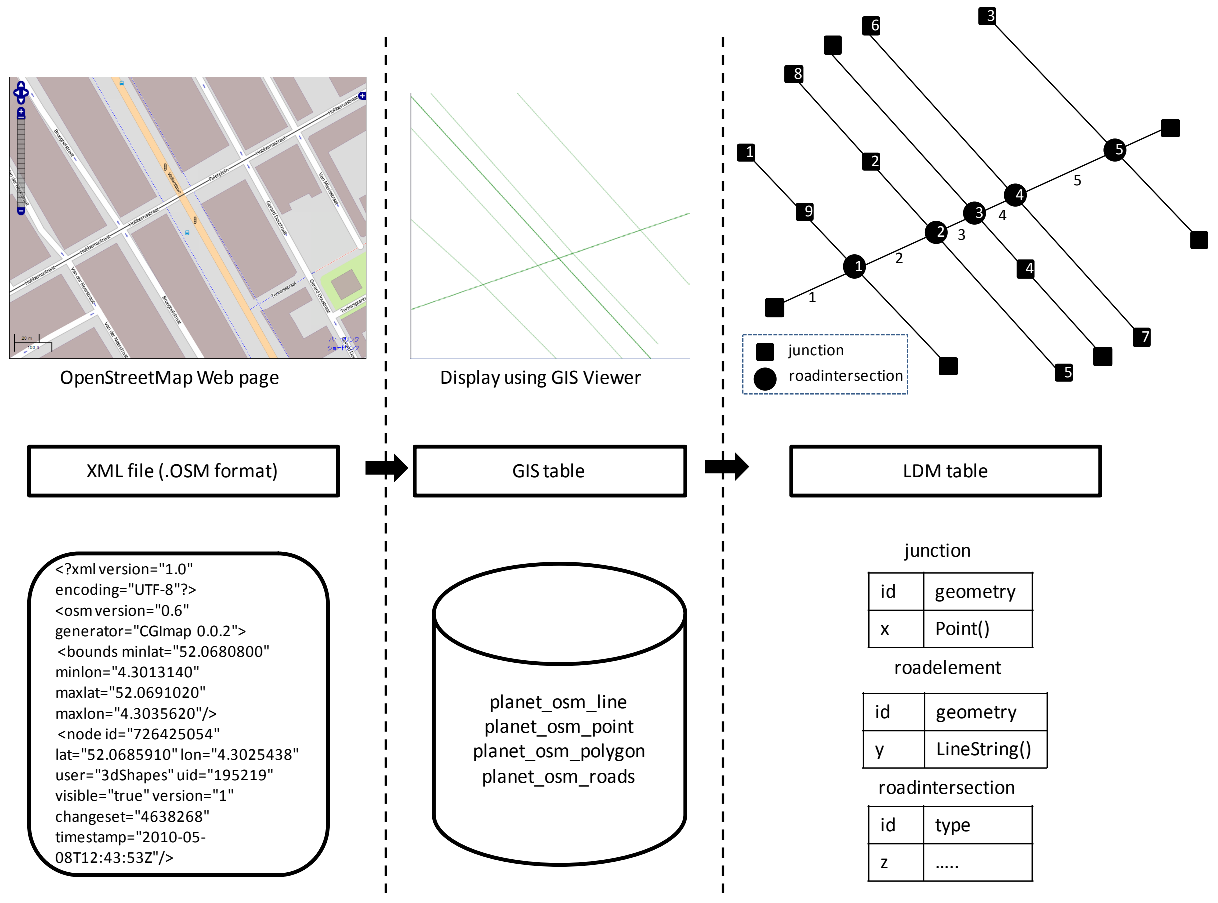
We used OpenStreetMap [11] as our source of map data. The road information in OpenStreetMap can be used and edited as an XML file in OSM format. If the contents of this XML file are converted to PostgreSQL, roads and intersections can be managed as LineString and Point data, respectively, as defined in PostGIS. Point data is a data type consisting of latitude and longitude in the form of point (x, y) while LineString data is a data type that connects Point data in the form of LineString (Point a, Point b,...). However, registering XML data in PostgreSQL does not in itself assign IDs, so we do this by extracting target areas and assigning IDs to intersections and roads.

The process flow for registering map data is shown in **Figure 3** and summarized below.

1) Select area to be used from the OpenStreetMap Web page.

2) Download file in .osm format after selecting the target area.

3) Register the downloaded .osm file in the PostgreSQL database using the osm2pgsql [12] program. This



```
<?xml version="1.0"
encoding="UTF-8"?>
<osm version="0.6"
generator="CGImap 0.0.2">
 <bounds minlat="52.0680800"
minlon="4.3013140"
maxlat="52.0691020"
maxlon="4.3035620"/>
 <node id="726425054"
lat="52.0685910" lon="4.3025438"
user="3dShapes" uid="195219"
visible="true" version="1"
changeset="4638268"
timestamp="2010-05-
08T12:43:53Z"/>
```

id	geometry
x	Point()

id	geometry
y	LineString()

id	type
z	.....

Figure 3. Registration process of map data.

action will create planet_osm_line, planet_osm_point, planet_osm_polygon, and planet_osm_roads tables under an arbitrary database name.

4) Extract LineString end points from LineString data contained in planet_osm_line as Point-type data and store those data in the LDM junction table.

5) Extract adjacent Point-type data from LineString data contained in planet_osm_line in the manner of LineString (Point a, Point b)… and store such LineString data in the LDM roadelement table.

6) Search for intersections of LineString data contained in planet_osm_line and store points where such data cross in the roadintersection table.

3.4.2. Vehicle Data

In LDM, vehicle-related tables (egomotorvehicle, motorvehicle) contain entries for managing various types of sensor data such as location information, velocity, and acceleration. Using the PreScan [13] simulation platform, we created vehicle sensor models and driving scenarios. In PreScan, it is possible to apply the control logic of Matlab/Simulink [14], so we were able to perform simulations using a vehicle model even closer than usual to an actual vehicle. The process flow for creating vehicle data is summarized below.

1) Read map data of OpenStreetMap into PreScan.

2) Arrange vehicles on the roads indicated on the map and set vehicle-movement scenarios.

3) Create a Simulink model and set automotive sensor blocks.

4) Execute the simulation and output vehicle location together with sensor data as a.csv file.

4. LDM Evaluation

4.1. Overview

Based on the LDM design described in the previous section, we implemented a collision detection application as a safety driving system and evaluated and analyzed the LDM that we implemented using a database. Specifically, we performed an experiment while varying the computer environment and simulation parameters and investigated whether the response time of the database could be applied to a safety driving system.

4.2. Evaluation Environment

The computer hardware that we used in the experiment is summarized in **Table 1**. Computer A, a high-performance computer compared to computer B, ran Fedora10 on VMWare, while computer B ran a similar environment as a host. We used PostgreSQL, PostGIS, and PL/pgSQL on both computers to implement LDM.

In this experiment, we used one computer at a time having an architecture made up of LDM, API, Longitudinal Collision Risk Warning (LCRW) application, update program, and vehicle data as shown in **Figure 4**. To perform this experiment on one computer, we took no communication with other motor vehicles into account resulting in a network environment by V2V communication with no delay or packet loss. The map data registered in LDM was for a portion of a city (approximately 300 m square) in the Netherlands having a grid street plan as shown by the map in the upper-left portion of **Figure 3** accessible by a URL[1]. We registered this

Table 1. Evaluation environment.

	Computer A		Computer B
	VM (host)	VM (guest)	
CPU	Core i7-2600 3.4 GHz 8-core	Core i7-2600 3.4 GHz 4-core	Pentium D 2.8 GHz 1-core
Memory	8.00 GB	2.00 GB	1.00 GB
HDD	1 TB SATA 6 Gb/s, 7200 rpm	20 GB SCSI	150 GB SATA
OS	Windows 7 (64 bit)	Fedora 10 (32 bit)	Fedora 10 (32 bit)

[1]http://www.openstreetmap.org/?lat=52.068491&lon=4.302388&zoom=18&layers=M

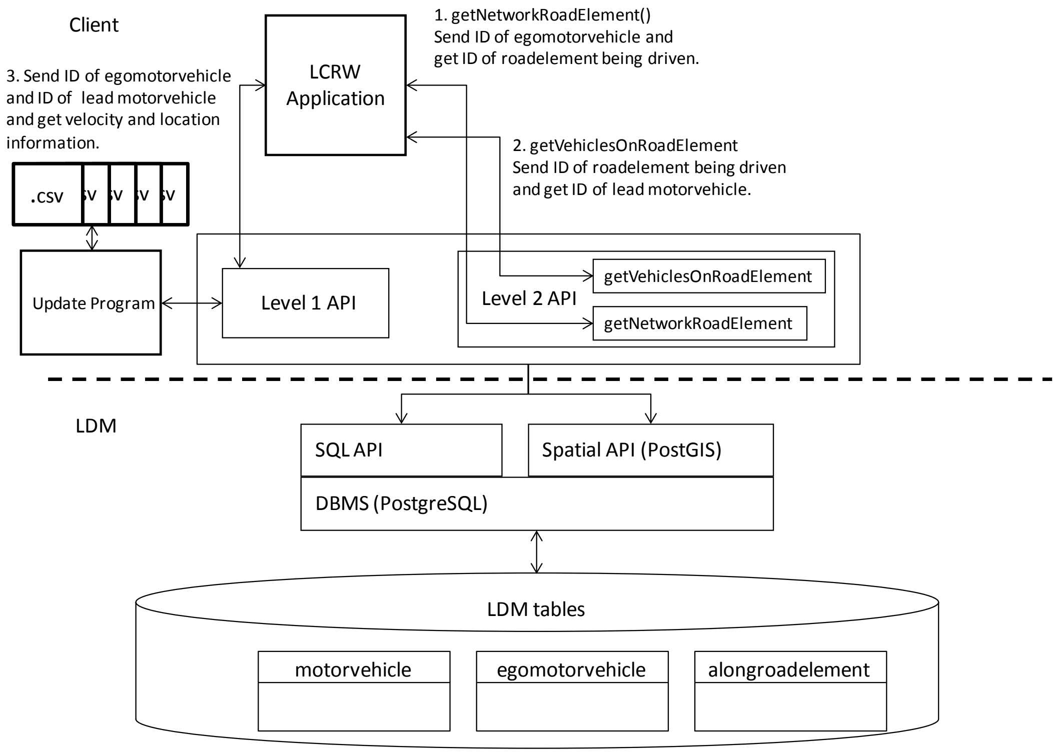

Client
LCRW Application
1. getNetworkRoadElement() Send ID of egomotorvehicle and get ID of roadelement being driven.
3. Send ID of egomotorvehicle and ID of lead motorvehicle and get velocity and location information.
2. getVehiclesOnRoadElement Send ID of roadelement being driven and get ID of lead motorvehicle.
.csv
Update Program
Level 1 API
Level 2 API
getVehiclesOnRoadElement
getNetworkRoadElement
LDM
SQL API
Spatial API (PostGIS)
DBMS (PostgreSQL)
LDM tables
motorvehicle
egomotorvehicle
alongroadelement

Figure 4. Architecture of experimental computer.

OpenStreetMap data in LDM using the procedure described in section 3.4.1. In addition, we created a vehicle-movement model using this map data and PreScan and a vehicle model using Simulink, and we collected the location of vehicle movement obtained from GPS together with velocity and other sensor data. This collected vehicle data, which is stored in CSV format, is read by the update program and registered in the LDM vehicle-data tables at a frequency based on the sensing period (50 ms). In the experiment, one vehicle is registered in the egomotorvehicle table and the remaining vehicles in the motorvehicle table.

4.3. Collision Detection Application

We use LCRW as a collision detection application in this evaluation experiment. Written in C++, this application calls Level 1 API and Level 2 API to obtain information within LDM with the aim of checking whether the ego motor vehicle and lead vehicle are about to collide. The LCRW application flowchart is shown in **Figure 5**. First, the application gets the ID of the road that the ego motor vehicle is currently driving on using L2API. getNetworkRoadElement() of Level 2 API. It then gets the IDs of the vehicles driving on the same road using L2API. getVehiclesOnRoadElement() of Level 2 API and determines the ID of the vehicle driving immediately ahead of the ego motor vehicle. Next, the application calculates the distance between the ego motor vehicle and lead vehicle using location information and queries LDM for the velocities of each vehicle. It then applies this information to a stopping distance algorithm (SDA) [15] and outputs a warning message if necessary. This checking for a lead vehicle and associated operations processing are performed in 100 msor 500 ms cycles.

4.4. Update Program

Map data consists of static data and can therefore be registered in LDM beforehand—it does not change during the experiment. Vehicle data, on the other hand, is dynamic and must be updated during the experiment. We use an update program for this purpose. This program, like the collision detection application, is written in C++. It uses Level 1 API to update the tables in LDM related to vehicle data, that is, the egomotorvehicle and motorvehicle tables. As for vehicle data to be updated, the number of CSV files prepared using PreScan is equivalent to the number of vehicles used in the experiment. The updating of these vehicle data is accompanied by the updating

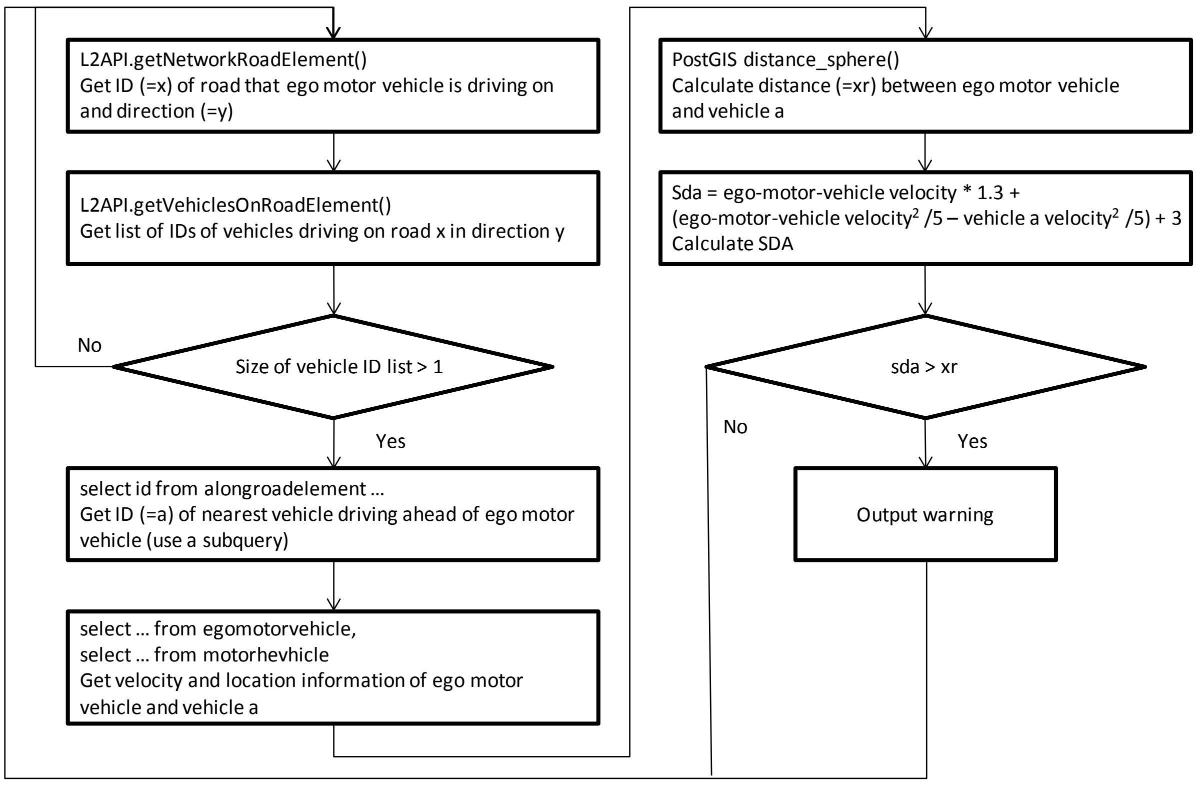

L2API.getNetworkRoadElement()
Get ID (=x) of road that ego motor vehicle is driving on and direction (=y)
L2API.getVehiclesOnRoadElement()
Get list of IDs of vehicles driving on road x in direction y
Size of vehicle ID list > 1
No
Yes
select id from alongroadelement ...
Get ID (=a) of nearest vehicle driving ahead of ego motor vehicle (use a subquery)
select ... from egomotorvehicle,
select ... from motorhevhicle
Get velocity and location information of ego motor vehicle and vehicle a
PostGIS distance_sphere()
Calculate distance (=xr) between ego motor vehicle and vehicle a
Sda = ego-motor-vehicle velocity * 1.3 + (ego-motor-vehicle velocity² /5 – vehicle a velocity² /5) + 3
Calculate SDA
sda > xr
No
Yes
Output warning

Figure 5. Flowchart of LCRW application.

of the alongroadelement relationship table in LDM by a stored procedure in PostgreSQL. Here, the updating of the vehicle data tables by the update program is used as a trigger to call this procedure and update the alongroadelement table. This updating process is described using the PL/pgSQL procedural language that is used to achieve PostgreSQL stored procedures.

4.5. Evaluation Experiment

As shown in **Figure 6**, we performed this evaluation experiment by running the collision detection application and update program simultaneously as separate processes that submit queries to LDM.

Given that vehicle data is obtained at 50 ms intervals, the update program performs update operations with respect to LDM in 50 ms cycles. The collision detection application, meanwhile, sends queries to LDM to check for a lead vehicle at intervals of 100 ms or 500 ms. In addition, the number of vehicles to be updated is variable at 5, 10, and 20 vehicles. In the experiment, we measured the response time for outputting a warning message by the collision detection application as an evaluation value. Response time here can be expressed as follows:

$$\begin{aligned}\text{response time} &= \text{warning-message output time} - \text{sensing time}\\ &= \text{communication delay time} + \text{API processing time} + \text{operations processing time}\end{aligned}$$

In this experiment, communication delay is taken to be zero, so response time can be expressed as the sum of Level 2 API processing time and operations processing time with respect to the target vehicles as shown in **Figure 5**.

Response-time results for computers A and B are shown in **Figure 7** and **Figure 8**, respectively. For each of these bar graphs, the horizontal axis represents variable parameters, that is, number of vehicles (5, 10, 20) and the checking interval (100 ms, 500 ms) of the collision detection application, while the vertical axis represents response time (ms). Here, as explained above, response time can be expressed as the sum of the processing time associated with two API calls in LCRW (getNetworkRoadElement(), getVehiclesOnRoadElement()) and the SDA operations processing time.

A significant difference can be seen in response time between computer A and computer B having different

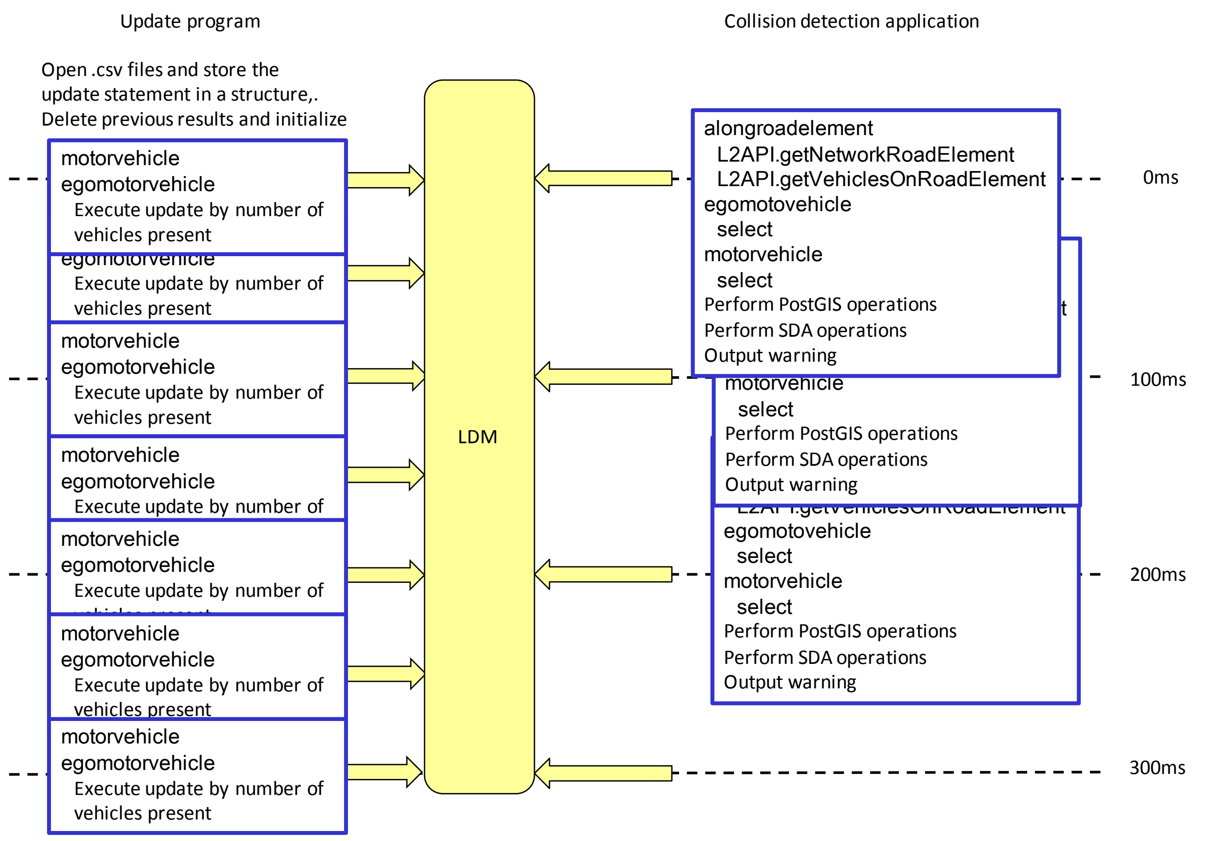

Update program
Collision detection application
Open .csv files and store the update statement in a structure,. Delete previous results and initialize
motorvehicle
egomotorvehicle
Execute update by number of vehicles present
LDM
alongroadelement
L2API.getNetworkRoadElement
L2API.getVehiclesOnRoadElement
egomotovehicle
select
motorvehicle
select
Perform PostGIS operations
Perform SDA operations
Output warning
0ms
100ms
200ms
300ms

Figure 6. Overview of experiment.

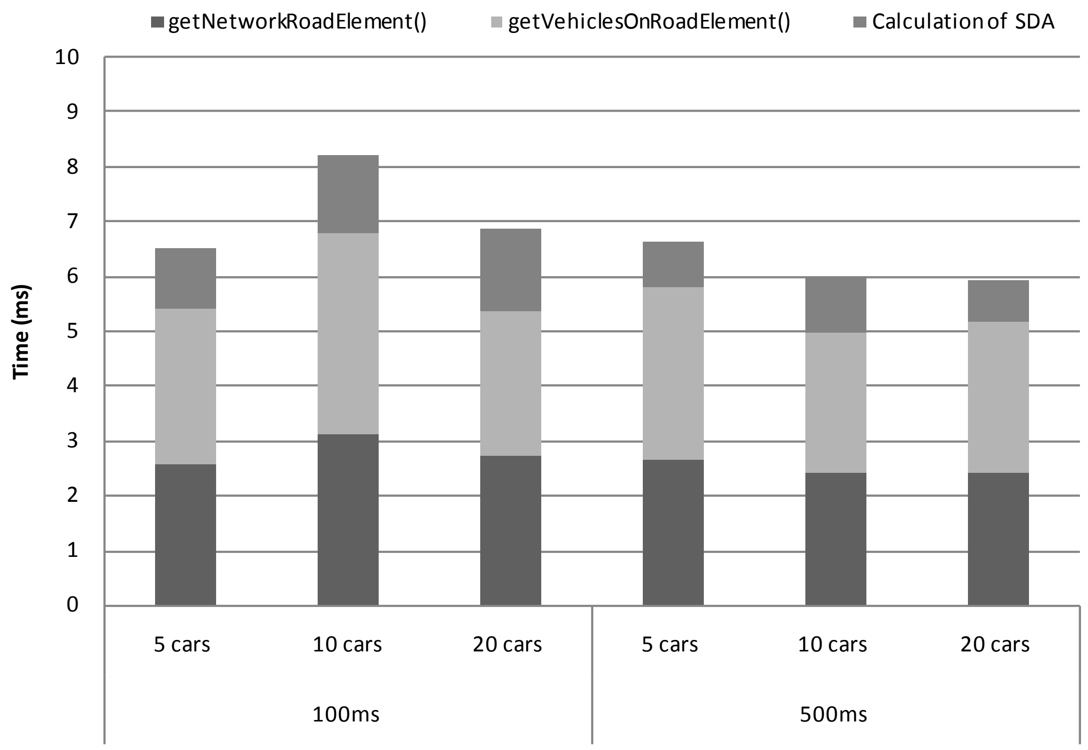


Figure 7. Response time (computer A).

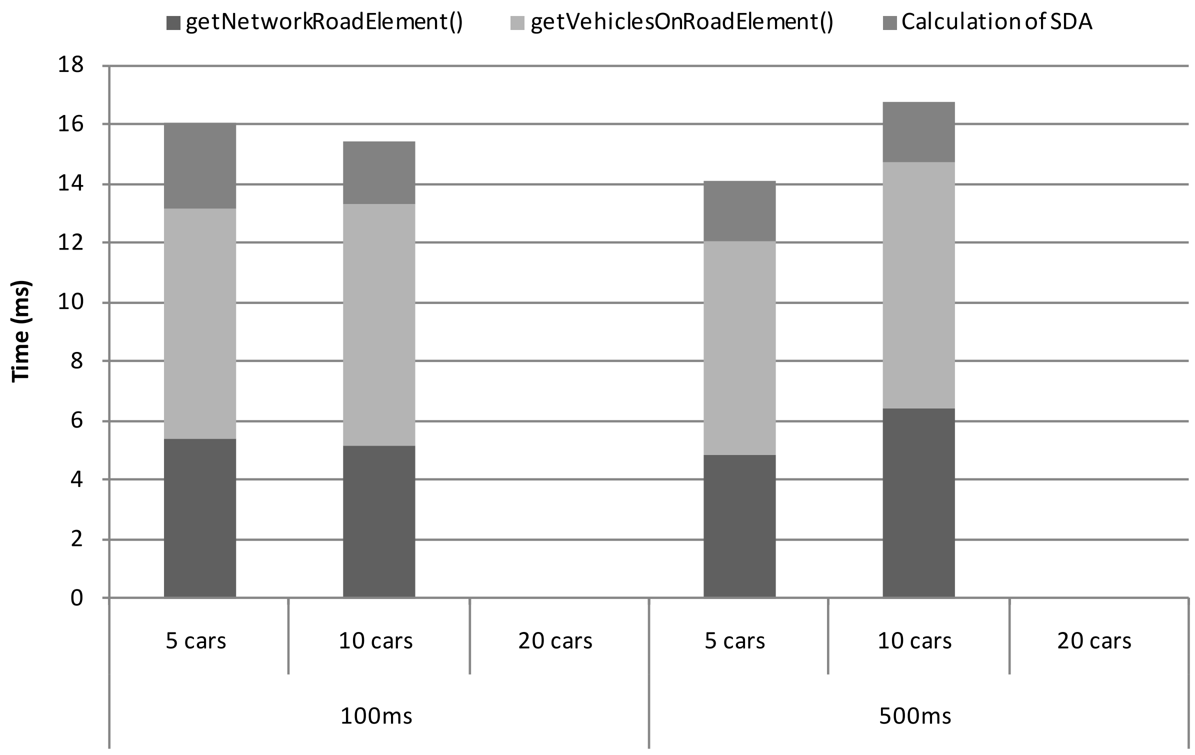


Figure 8. Response time (computer B).

levels of performance. In contrast to a response time of about 6 ms overall for high-performance computer A, computer B exhibits a response time of about 15 ms. Although the performance of individual computer elements like CPU, memory, and hard disks differs between these two computers, performing the same experiment when varying the number of CPU cores and memory size by VMW are settings on computer A revealed no changes in response time. This result indicates that hard-disk performance represents a major hardware effect. In addition, no significant differences in response time could be observed between checking intervals of 100 ms and 500 ms of the collision detection application. Furthermore, no major change in response time could be observed when increasing the number of vehicles managed by LDM as shown in **Figure 7** and **Figure 8**.

As shown in **Figure 8**, response-time calculations could not be performed for the case of 20 vehicles on computer B. The reason for this is related to the update processing time of vehicle data shown in **Table 2**, which lists the time needed by the update program to update vehicle data in LDM for the various parameters of this experiment. These results show that no major change in update processing time could be observed for different checking intervals of the collision detection application. They also show, however, that update processing time increased nearly in proportion to an increase in the number of vehicles. In addition, the time needed for update processing was significantly different between the two computer environments. For the case of 20 vehicles, update processing time was about 36 ms for computer A but about 71 ms for computer B.

In this experiment, the sensing interval for vehicle data was set to 50 ms so update processing of vehicle data was performed at intervals of 50 ms. However, update processing time on computer B for 20 vehicles became larger than this required update interval of 50 ms. In other words, updating could not keep up with the sensing interval, and as a result, no response time could be obtained for 20 vehicles on computer B as shown in **Figure 8**.

4.6. Summary of Evaluation Experiment

We evaluated the performance of an LDM implementation using a collision detection application. In this experiment, we used computers differing in performance as evaluation environments and analyzed LDM performance while varying the frequency of data registration with LDM and the number of data items. We found that the response time of the collision detection application did not significantly change with an increase in the

Table 2. Update processing time.

Checking interval	100 ms			500 ms		
No. of vehicles	5 cars	10 cars	20 cars	5 cars	10 cars	20 cars
Update processing time (compter A)	8.9 ms	16.8 ms	36.4 ms	8.9 ms	16.9 ms	35.5 ms
Update processing time (computer B)	23.1 ms	42.2 ms	70.9 ms	18.6 ms	37.4 ms	72.1 ms

number of vehicles but that a situation in which the updating of vehicle data could not keep up with the sensing interval could occur depending on the computer environment. We considered that this was caused by a difference between the processing load of the SQL select statement within the collision detection application and that of the SQL update statement within the update program. In general, processing load of the select statement is less than that of the update statement, which would support this result. In addition, processing the update statement with respect to LDM means executing stored procedures implemented by PL/pgSQL and updating the along roadelement relationship table. This would also explain why processing of the update statement cannot keep up with the sensing interval as the number of vehicles increase.

The above problem did not appear when using the high-performance computer for the number of vehicles used in this experiment. We consider, however, that a similar problem would eventually occur here if the number of registered vehicles were to be further increased. The number of vehicles to be registered and the frequency of vehicle registration should therefore be set according to the computer environment being used.

5. Conclusions

Local Dynamic Map (LDM) has been attracting attention as a mechanism for managing map and vehicle data in ITS using vehicle-to-vehicle and infrastructure-to-vehicle communication. In this paper, we reported on a computer-based implementation of LDM using an RDBMS based on the LDM specifications released by the European SAFESPOT project for improving road safety. In particular, we registered map data for an actual environment and vehicle data in LDM and evaluated and analyzed this implementation of LDM on computer using a collision detection application. Based on the results of the evaluation experiment, we found that operation processing in the collision detection application had no problems but that LDM internal processing experienced a high load as the number of vehicles increased, which did have an impact on the application. Although examples of LDM implementations had been reported in the past, experimental results had not been released in papers or elsewhere, so the numerical results presented in this paper should serve as a reference for future implementations of LDM.

The experiment described in this paper was performed in an ideal environment with no consideration of vehicle-to-vehicle communication and therefore no delay or packet loss. We consider, however, that a wireless network would be used in an actual environment and that data delay and/or loss would occur owing to a variety of factors such as moving nodes. We leave for future research an experiment that simulates a network environment close to such a real-world environment. We also plan to implement a filtering mechanism for reducing the volume of vehicle data updates.

Acknowledgements

This work was partly supported by Japan Society for the Promotion of Science(JSPS), Grant-in-Aid for Scientific Research (KAKENHI), and Strategic Information and Communications R&D Promotion Programme (SCOPE) by the Ministry of Internal Affairs and Communications Japan.

References

[1] Report on Advanced Safety Vehicle (ASV) Promotion and Planning. (In Japanese) (2011).
http://www.mlit.go.jp/jidosha/anzen/01asv/resourse/data/asv4pamphlet.pdf

[2] Kobayashi, M., Oota, T. and Kamata, K. (2010) Research and Developments for Practical Use of DSSS. *Journal of Society of Automotive Engineers of Japan*, **64**, 43-48. (In Japanese).

[3] Yamada, M., Kamada, H., Sato, K. Teshima, S. and Takada, H. (2010) Integrated Sensor Data Management Method

for Vehicle Control System. *IEICE Transactions on Information and Systems*, **J93-D**, 1189-1201.

[4] ETSI TR 102 863 (V1.1.1) (2011) Intelligent Transport Systems (ITS): Vehicular Communications; Basic Set of Applications; Local Dynamic Map (LDM) Rationale for and Guidance on Standardization.

[5] SAFESPOT Integrated Project (2012) http://www.safespot-eu.org/

[6] Zott, C., Yuen, S.Y., Brown, C.L., Bartels, C., Papp, Z. and Netten, B.D. (2008) Safespot Local Dynamic Maps - Context-Dependent View Generation of a Platform's State & Environment. *Proceedings of the* 15*th ITS World Congress*.

[7] Ibanez-Guzman, J., Lefevre, S., Mokkadem, A. and Rodhaim, S. (2010) Vehicle to Vehicle Communications Applied to Road Intersection Safety, Field Results. 2010 13*th International IEEE Conference on Intelligent Transportation Systems* (*ITSC*), Funchal, 19-22 September 2010, 192-197. http://dx.doi.org/10.1109/ITSC.2010.5625246

[8] Schendzielorz, T., Vreeswijk, J. and Mathias, P. (2009) Intelligent Cooperative Intersection Safety Implementation, Test and Evaluation. *Proceedings of the* 16*th ITS World Congress*.

[9] Netten, B. and Wedemeijer, H. (2010) Testing Cooperative Systems with the Mars Simulator. 2010 13*th International IEEE Conference on Intelligent Transportation Systems* (*ITSC*), Funchal, 19-22 Septembwe 2010, 186-191. http://dx.doi.org/10.1109/ITSC.2010.5624981

[10] SAFESPOT SP 7 SCORE - SAFESPOT Core Architecture, D7.3.1 Annex2 - LDM API and Usage Reference (2010). http://www.safespot-eu.org/documents/SF_D7.3.1_Annex2_LDM_API_and_Usage_Reference_v0.7.pdf

[11] OpenStreetMap (2012) http://www.openstreetmap.org/

[12] osm2pgsql (2012) http://wiki.openstreetmap.org/wiki/Osm2pgsql

[13] Tass: PreScan (2012) http://www.tass-safe.jp/prescan/index.html

[14] MathWorks: Simulink (2012) http://www.mathworks.co.jp/products/simulink/

[15] Burgett, A.L., Carter, A., Miller, R.J., Najm, W.G. and Smith, D.L. (1998) A Collision Warning Algorithm for Rear-End Collisions. National Highway Traffic Safety Administration.