A Thread-Based Distributed Traffic Micro-Simulation

Pedro G. Gonnet (pedro@vis.ethz.ch)

June 27, 2001

Abstract

This Term Paper presents a distributed Traffic Micro-Simulation. The design goals were speed, realism, distributeability and simplicity. The speed goal was clearly achieved. As for the others, further work/testing is necessary.

Contents

Chapter 1

Introduction

A bit of blabla about why we do traffic simulation and what it is good for

1.1 Design Concepts

The main goals set for the `vsim` Traffic Simulation were

- **Speed** The Simulation should run faster than existing programs.
- **Realism** The Simulation should be accurate enough to simulate even small traffic situations reliably.
- **Distributeability** The Simulation should be efficiently distributeable to take advantage of computer clusters.
- **Simplicity** The Simulation should be simple enough for anybody with basic computer science knowledge to extend.

To achieve this, a number of concepts were used, the most important four being:

- **Queue-Based Traffic Model** Provides a more realistic traffic model, where for each time-step the acceleration and not the displacement is calculated.
- **Multi-threaded Computer-Node Implementation** Takes advantage of multi-processor machines and amortizes time spent waiting on communication.
- **Explicit, Asynchronous Inter-Node Communication** Avoids overhead inherent to MPI and other libraries.
- **Multi-Constraint Graph Partitioning** Gives better graph partitions at the expense of more explicit node and street load data.

These four concepts and others will be explained in more detail further on.

Chapter 2

Traffic Model

As mentioned earlier, realism was one of the primary design goals. For the `vsim` Traffic Simulation, a classic particle simulation approach was taken where every car is aware of it's position and speed and its acceleration is calculated and applied at every time-step.

Although this approach is more realistic, it is definitely more computationally intensive than cell-based models. Furthermore, this model also requires more network detail – i.e. definition of connectivity and priorities at nodes – which is not always available.

2.1 Cars

As mentioned earlier, the cars in `vsim` are aware of their position (depth into a street) and velocity. They have a route to follow, and a given start time. For added realism, each car also has a bias factor that influences it's driving behavior – i.e. it's tendency to speed and cut-off other vehicles.

Car acceleration is governed by a single procedure which, given an obstacle and it's velocity, the car will calculate – knowing its own velocity – its acceleration for the next time-step.

Each car also decides on its own where and when to switch lanes, given information on if it has to or can switch lanes and the distances to and velocities of the cars ahead and behind on the neighboring lanes.

2.2 Streets

In `vsim`, streets are a collection of linked lists of cars, one for each lane. The streets are uni-directional and have therefore distinct from- and to-nodes.

In each time-step, all lanes are traversed in parallel from the from- to the to-end. Each car calculates its acceleration taking the car ahead of it as a moving obstacle. For the last car in each lane, the intersection is queried for distance and velocity on the destination street.

When a car leaves a street, it is passed over to the to-node which relays it to the next street in its route.

2.3 Intersections

Intersections in `vsim` contain information about incoming and outgoing streets and about the way their lanes are interconnected, which connection has priority over which and which connections are activated in which time-steps (traffic lights).

In each time-step, a node calls on all incoming streets to move their cars. When it is queried by a street for a car to pass over, it first checks if there is a connection open for the car in question (lane and route defendant). If not, it will return an obstacle with distance and velocity 0. Otherwise, it queries the destination street (which can also be on a different computer) and returns the distance to and velocity of the first car there.

Chapter 3

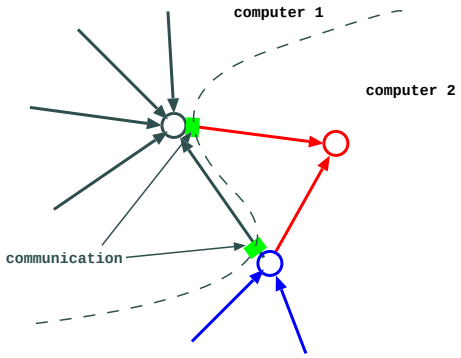
Distributed Traffic Model

explain what changes have to be made and how these can affect the efficiency/reliability of the simulation.

3.1 Partitioned the Network

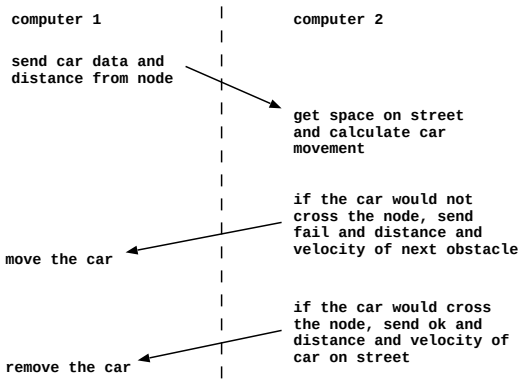
The network is partitioned by separating nodes and outgoing streets. A node with all its incoming streets can be seen as the smallest unit of computation.

Communication is reduced to querying another computer only when a car passes over a node into a foreign street.



3.2 Vehicle Passing

The process of passing a car from one computer to the other consists of only two messages – a request and a response. For a node being queried, this looks as follows:



After sending a request, the node can remember the distance and velocity of the next obstacle returned by the other computer an so avoid multiple queries to the same street within the same time-step.

3.3 Coordination

The distributed simulation is controlled in a master/slave fashion. One machine (master) controls the others who actualy run the simulation (slaves). The master can run on the same machine as a slave.

Once the simulation has been started in slave mode, it waits for a connection from a master. Once this connection is established, it recieves information on where its input files resides an reads them. Once this has been done, it waits for a "tick" message, simulates one timestep and answers with either "ok" or "fail".

When running the simulation, the master sends out a "tick" to all slaves and waits until he has a response from all machines before continuing. A slave does not need to have received a "tick" before processing requests from other slaves.

Between timesteps the master can also request data, such as car positions and velocities or street travel times.

Chapter 4

Implementation

This chapter explains how the above mentioned traffic model was implemented for the vsim traffic simulation.

The program was entirely written in the C Programming Language.

4.1 Modules

Figure 4.1 shows the main modules in `vsim` and how they interact. Solid arrows denote interaction in the form of function calls, dashed arrows through network communication.

The modules `runner`, `handler` and `msgr` appear more than once, since various instances of those objects can be active at the same time in different threads.

The modules `bcaster`, `car`, `route` and `model` are missing from the diagram.

4.1.1 Traffic Modules

The modules that actual simulate traffic are `node`, `street`, `car`, `route` and `model`. As shown in Figure 4.2, the `node`s, `street`s and `route`s are stored in

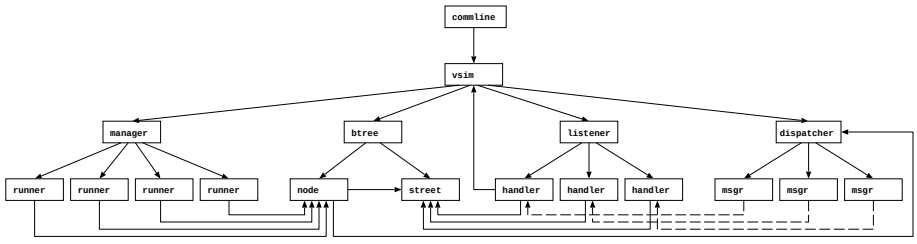


Figure 4.1: Main modules and their interactions. Solid arrows denote communication in the form of function calls, dashed arrows through network communication.

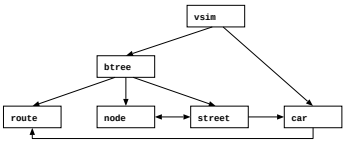


Figure 4.2: Modules involved in simulating traffic. The arrows denote referencing between modules.

a `btree` in the main module `vsim`. The `cars` are stored directly in `vsim` as a linked listed, sorted by their departure times.

The `nodes`, `streets`, `cars` and `routes` can be read by the functions `node_read`, `street_read`, `car_read` and `route_read` respectively. These functions take a file descriptor (`FILE *`) as an argument and return a pointer to a structure of the respective type or `NULL` if an error occurs.

These functions are called through the main module `vsim` through the functions `vsim_readnodes`, `vsim_readstreets`, `vsim_readroutes` and `vsim_readcars` respectively. All four procedures take the name of the file to be read (`char *`) as an argument and return the number of objects read (`int`) or a negative value if an error occurs.

The input functions in `vsim` must be called in the order given above for the objects to be linked together correctly.

Each time-step is initiated by either the command-line (module `commline`) or, in the case of a single slave in a distributed simulation, from the a network (module `handler`) by calling `vsim_step`. `vsim_step` first checks if any cars need to be inserted, then calls `node_step` on all nodes (how this is node in detail will be explained further in this text). `node_step` then calls `street_step` on each incoming `street`.

In the event of a car driving towards or over a `node`, `street_step` will call `node_pass` on its to-`node` with the car in question as an argument.

4.1.2 Communication Modules

As mentioned earlier, communication is done over TCP/IP using an explicit protocol. When not running in stand-alone mode, the main module (`vsim`)

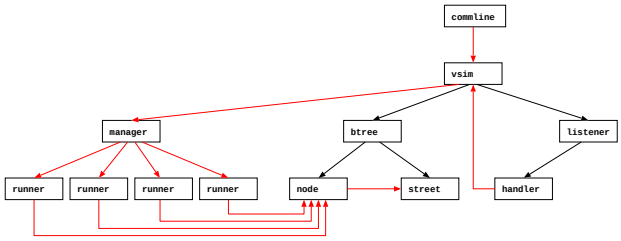


Figure 4.3: Control flow for each time-step (in red).

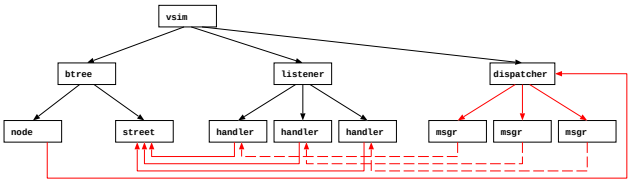


Figure 4.4: Communication for distributed simulations. Solid arrows represent function calls in other modules. Dashed arrows represent communication over the network.

initializes a `listener` and a `dispatcher` as shown in Figure 4.4.

The `listener`, once initialized, runs in its own thread and waits for connections on the port specified by the variable `vsim_port` in the module `vsim`. Once a connection is established, a new `handler` is created with the connection socket as an argument. The `handler` then reads messages from the given socket and processes them until program termination. If no messages are available, the `handler` waits.

The `dispatcher` handles requests for communication with a given IP-address (usual from a `node`) with the function `dispatcher_get`, returning a `msgr` to handle message passing. If no `msgr` exists for the given IP-address, a new one is created with `msgr_create`. Once the client is done communicating, the `msgr` is returned to the `dispatcher` with the function `dispatcher_put` without closing the network connection. The `msgr` can then be reused by any other thread requesting communication with the same IP-address.

4.1.3 Control Modules

As could be seen in Figure 4.2, `vsim` uses the modules `manager` and `runner`. When running in stand-alone or slave mode, the main module initializes a `manager` which, in turn, will create a given number of `runners` (the exact number is given by the constant `manager_nrrunners`), each with their own thread, and gives each one a linked list of pointers to the `nodes`.

After being initialized, the `runners` wait at a barrier (`manager_wait`). At every time-step, the `manager` opens the barrier and waits. The `runners` then call `node_step` on each of the nodes in their list and reenter the barrier. Once all `runners` have finished, the `manager` is signaled and returns.

To guaranty an even load distribution between the `runners`, the `nodes` must be redistributed according to the time it takes to process them. The time spent on each `node` is measured every `runner_clockstep` seconds (since this implies a system call for every node, it is not done every time-step) and the `nodes` are redistributed every `manager_rschedrate` seconds.

To avoid the hassle of redistribution, one could make all `runners` get their `nodes` from a common linked list. This however would require us to control access to this list (concurrency), which causes an enormous loss of performance on multi-processor machines.

4.1.4 Utility Modules

The only utility module used in the simulation is the `btree` module. This is a generic, 16-way B-Tree with a 1024-entry cache of the last accessed members by hash-value.

4.1.5 Other Modules

The `commline` module acts as a front end for a user in stand-alone or server mode. after being initialized with `commline_init`, it reads commands from standard input an delegates them to the `vsim` module. A description of the commands can be found in Chapter 5.

The `bcaster` module is used by the simulation when running in server mode and broadcasts messages to a number of clients. Clients can be added with `bcaster_add`, yet only before connection is established with `bcaster_open`.

`model`

4.2 Functions and Data Structures

The following sections describe how the above mentioned modules were implemented and what types, variables and functions are available.

4.2.1 `bcaster`

The `bcaster`'s job is to maintain a number of connections to simulation clients and broadcast commands and requests (i.e. for travel time data) to them.

struct bcaster	
`struct conn conn[]`	the connections
`int nr_conns`	number of connections

struct conn	
`int socket`	the connection socket (for I/O)
`unsigned int ip`	IP-address of the machine connected

`int bcaster_init ( struct bcaster *b )`
Initialize the given `bcaster`. On error, a negative value is returned.

`int bcaster_add ( struct bcaster *b , unsigned int ip )`
Add the given IP-address to the list of clients. This may only be done before `bcaster_open` is called. On error, a negative value is returned.

`int bcaster_remove ( struct bcaster *b , unsigned int ip )`
Remove an IP-address from the list of clients. This may only be done before `bcaster_open` is called. On error, a negative value is returned.

`int bcaster_open ( struct bcaster *b )`
Open connections to all clients. On error, a negative value is returned.

`int bcaster_close ( struct bcaster *b )`
Close connections to all clients. On error, a negative value is returned.

`int bcaster_cast ( struct bcaster *b , char *msg , int msg_size , char *xpct , int xpct_size )`
Cast the given message to all clients and compare the response to the given expected message. On error, or if a different response is given, a negative value is returned.

`int bcaster_pos ( struct bcaster *b , FILE *out )`
Ask all clients for their car positions and velocities and dump them to the given file. On error, a negative value is returned.

`int bcaster_stimes ( struct bcaster *b , FILE *out )`
Ask all clients for the street travel times and dump them to the given file. On error, a negative value is returned.

4.2.2 `btree`

The `btree` is used to store `nodes`, `streets` and `routes`. It is generic in the sense that it can store any structure, as long as it starts with a unique integer value.

struct btree	
`struct btree_node *first`	first node in the tree
`struct btree_leaf *cache[]`	cache of last accesses

struct btree_node	
`int fill`	nr of subnodes/leafs
`int leaf`	am i a leaf?
`struct btree_leaf *cont[]`	leaves within node
`struct btree_node *nodes[]`	subnodes

struct btree_leaf	
`int key`	leaf key

`struct btree *btree_new ( )`
Create a new `btree`. On error, `NULL` is returned.

`struct btree_leaf *btree_find ( struct btree *t , int key )`
Search for the element with the given key. On error, or if nothing is found, `NULL` is returned.

`int btree_insert ( struct btree *t , struct btree_leaf *l )`
Insert an element into the tree. On error, a negative value is returned.

`int btree_map ( struct btree *t , int (*func)( struct btree_leaf * , void * ) , void *cont )`
Apply the given function to all the elements in the `btree` with the given argument. The applied function should return a negative value on error to interrupt execution. On error, a negative value is returned.

`int btree_count ( struct btree *b )`
Return the number of elements in this `btree`. On error, a negative value is returned.

4.2.3 car

The `car` data structure contains data relative to the car state (position and velocity), as well as route and timing data. The `next_step` field is there for speed purposes only, as it could be looked up in the route. The `end` field is (mis)used for timing the car on each street.

struct car	
`int id`	id of this car
`struct car *next`	next car for linked list
`float length`	length of this car
`float bias`	car bias (for driving behaviour)
`float pos`	current position in street
`float vel`	current velocity in m/s
`int start, end`	start and end time for route
`int step`	position in route
`int next_step`	id of next street in route
`struct route *route`	route used

`struct car *car_create ( int id , float length , float bias , int route_id , int start )`
Create a `car` from with the given values. If the first street on the car's route is not local, `NULL` is returned. On error, `NULL` is returned.

`struct car *car_read ( FILE *in )`
Read a `car` from the given file. If the first street on the car's route is not local, `NULL` is returned. On error, `NULL` is returned.

`struct car *car_rebuild ( int id , float length , float bias , int route_id , int start , int step )`
Rebuild a car with the given values. This function is called by `handler_pass`. On error, `NULL` is returned.

4.2.4 commline

The commline reads commands from standard input, parses them and passes them on to the main module (vsim).

struct commline	
FILE *in	commandline input
char *prompt	commandline prompt
int nr_tokens	number of tokens in last line
struct token tokens[]	tokens from last line

`int commline_init ( struct commline *c , FILE *in , char *prompt )`
Initialize the given commline with the given input file and prompt. On error, a negative value is returned.

`int commline_run ( struct commline *c )`
Read and execute commands until the user terminates. On error, a negative value is returned.

4.2.5 dispatcher

The dispatcher manages a list of open connections to different clients as a hashtable over their IP-addresses.

struct dispatcher	
struct msgr *msgrs[]	the msgrs
pthread_mutex_t mutex	mutex for synchronising access to the msgr list

`int dispatcher_init ( struct dispatcher *d )`
Initialize the given dispatcher. On error, a negative value is returned.

`struct msgr *dispatcher_get ( struct dispatcher *d , unsigned int ip )`
Get a msgr for the given IP-address. If none is available, a new one is created. On error, NULL is returned.

`int dispatcher_put ( struct dispatcher *d , struct msgr *m )`
Return a msgr to the dispatcher after use. On error, a negative value is returned.

4.2.6 error

The error module is used by all other modules and manages a stack of error messages which can be dumped or reset. In most modules, there is a macro error which maps to the function error_register with the appropriate message and __LINE__ and __FILE__ as parameters.

`int error_register ( int id , char *msg , int line , char *file )`
Add a new error message to the stack. The parameter `id` is returned.

`void error_dump ( FILE *out )`
Dump the error stack to the given file.

`void error_reset ( )`
Reset the error stack.

4.2.7 handler

A `handler`, once created, runs in its own thread reading and processing commands from the network. `handlers` are spawned by the `listener`.

struct handler	
`int socket`	I/O socket
`pthread_t thread`	thread in which this handler is running
`char buff[]`	input buffer
`int bytes`	nr of bytes in input buffer

`struct handler *handler_create ( int socket )`
Create a new `handler` for the given socket. The `handler` will spawn its own thread and start receiving and processing commands. On error, `NULL` is returned.

`int handler_kill ( struct handler *h )`
Tell the given `handler` to close its socket and terminate execution. On error, a negative value is returned.

4.2.8 listener

The `listener` runs in its own thread and waits for network connections on the port specified by `vsim_port`. Once a connection is received, it creates a new `handler`.

struct listener	
`struct sockaddr_in addr`	the address im listening on
`int sock`	the socket bound to this address for listening
`pthread_t thread`	the thread in which the listener is running

`int listener_init ( struct listener *l )`
Initialize the given `listener`. This will also start it. On error, a negative value is returned.

4.2.9 manager

The manager handles control of the runners. It's main task is synchronizing the runners and distributing the nodes over them evenly.

struct manager	
struct runner runners[]	the runners
struct node *nodes[]	node queues for the runners
int count	nr of runners that have reached the barrier
pthread_cond_t done_cond	condition to signal when all the runners are finnished
pthread_mutex_t barrier_mutex	mutex to access the barrier
pthread_cond_t barrier_cond	condition variable associated to the barrier mutex

`int manager_init ( struct manager *m )`
Initialize the given manager. On error, a negative value is returned.

`int manager_wait ( struct manager *m )`
Wait at the barrier. This function is called by the runners once they have finished processing their node-list. On error, a negative value is returned.

`int manager_nodes ( struct manager *m )`
Tell the manager to make a list of local nodes and distribute them over the runners. On error, a negative value is returned.

`int manager_step ( struct manager *m )`
Tell the manager to simulate one time-step by opening the barrier and waiting for all runners to return. On error, a negative value is returned.

4.2.10 model

The model module implements the driver behavior for each car. It has no data type, only three functions that govern acceleration and lane switching.

`float model_drive ( float speed , float dist , float obst_speed , float bias , float limit )`
Calculate the acceleration for a car driving at the given speed with an obstacle at dist meters with the given velocity. bias is the car-specific behavior value and limit is the speed limit on the current street.

`int model_fit ( float speed , float bias , float dist_back , float vel_back , float dist_front , float vel_front )`
Judges if the car will switch lanes or not, assuming it must. dist_back and vel_back refer to the first car behind on the lane to be switched to.

`int model_pass ( float speed , float bias , float dist_front , float vel_front , float limit )`
Decides if a car will switch lanes to pass the car with the given distance and velocity ahead of him, assuming his route allows it.

4.2.11 msgr

The `msgr` is in charge of communication with other machines in a distributed simulation.

struct msgr	
`struct msgr *next`	link pointer for dispatcher
`unsigned int ip`	IP-address this `msgr` is connected to
`int socket`	I/O socket through which this `msgr` is communicating

`struct msgr *msgr_create ( unsigned int ip )`
Create a new `msgr` and connect to the given IP-address. On error, `NULL` is returned.

`int msgr_pass ( struct msgr *m , int street , struct car *c , float *dist , float *vel )`
Attempt to pass the given car to the given street (in compressed form). The position and velocity to the next obstacle on the street are written into `*dist` and `*vel` respectively. On error, a negative value is returned.

4.2.12 node

The module `node` represents a local intersection in the network. When reading `nodes` in a distributed simulation, the `node` module will return either a `struct node` or a `struct node_foreign`, depending on the node being on the current machine or elsewhere.

struct node	
`int id`	id nr of this node
`unsigned int ip`	ip of the location of this node
`float x_coord, y_coord`	x and y coordinates
`struct street *in[]`	incomming streets
`struct street *out[]`	outgoing streets
`struct node *next`	next-pointer for linked list
`int nr_links`	nr of links in this node
`struct link links[]`	the links
`int nr_steps`	nr of steps in schedule (lights)
`node_masktype sched[]`	the schedule. this is a bitmask for the links active at each schedule step
`int curr_step`	the current step in the schedule
`int decay`	seconds remaining until schedule switch

struct node_foreign	
`int id`	id nr of this node
`unsigned int ip`	ip of the location of this node
`float x_coord, y_coord`	x and y coordinates
`struct street *in[]`	incomming streets
`struct street *out[]`	outgoing streets

struct link	
`int in, out`	incomming and outgoing street/lane in compressed form
`int wait[]`	street/lanes this link has to wait for (priority)
`int halt[]`	street/lanes this link has to halt on (stop)

The compressed for for street/lane values can be decompressed with the functions `street_getstreet` and `street_getlane` and compressed with `street_compact`.

`struct node *node_create ( int id , float x_coord , float y_coord , unsigned int ip )`
Create a new `node` with the given values. On error, `NULL` is returned.

`struct node *node_read ( FILE *in )`
Read a `node` from the given file. This function may also return a `node_foreign` if the node is not local. On error, `NULL` is returned.

`struct node *node_read_noip ( FILE *in )`
Read a `node` from the given file and assume that no IP-addresses have been specified. This is used in stand-alone mode. On error, `NULL` is returned.

`int node_step ( struct node *n )`
Simulate one time-step on this node. On error, a negative value is returned.

`int node_pass ( struct node *n , struct car *c , struct street *s, int lane , int to )`
Try to pass the car `c` from the given street and lane into the street/lane `to` in compressed format. Returns `1` if the car could be passed, `0` otherwise (negative on error). The car's velocity and position are updated.

`int link_read ( FILE *in )`
Read a `link` from the given file. This will update the affected `nodes`. On error, a negative value is returned.

`int timing_read ( FILE *in )`
`int phase_read ( FILE *in )`
`int sign_read ( FILE *in )`
These functions read timing, phase and sign data for the `nodes`. This data is currently ignored by the simulation.

4.2.13 route

The `route` data structure is nothing more than a list of street ids. `route_hoptype` is defined either as an `int` or a `short int`. The id of last street in the route is `-1`.

struct route	
`int id`	id of this car
`int nr_steps`	nr of steps in this route
`route_hoptype step[]`	ids of the streets in this route

`struct route *route_create ( int id , int hops )`
Create a new `route` with the given number of hops. On error, `NULL` is returned.

`struct route *route_read ( FILE *in )`
Read a `route` from the given file. On error, `NULL` is returned.

4.2.14 runner

A `runner` runs in its own thread and calls `node_step` on every node in its node-list every time-step.

struct runner	
`struct manager *man`	the manager who controls this runner
`struct node *first`	first node in this `runner`'s node-list
`pthread_t thread`	the thread in which this `runner` is running

`int runner_init ( struct runner *c , struct manager *m )`
Initialize the given `runner` with the given `manager`. On error, a negative value is returned.

`int runner_run ( struct runner *c )`
Start the given `runner`. The `runner` will spawn its own thread and wait at the `manager`'s barrier until the next time-step. On error, a negative value is returned.

4.2.15 `street`

The `street` data type describes a unidirectional street. As with the `nodes`, we distinguish between `street` and `street_foreign`.

struct street	
`int id`	id of this street
`unsigned int ip`	ip address of the machine where this street resides
`struct node *to, *from`	nodes at both ends of the street
`struct lane *lanes[]`	the lanes in this street
`float length`	street length in meters
`float speed`	speed limit
`pthread_mutex_t mutex`	mutex to control access to this street
`int last_update`	time of last pass
`struct slot times[]`	slots for collecting statistical data (street travel times)

struct street_foreign	
`int id`	id of this street
`unsigned int ip`	ip address of the machine where this street resides
`struct node *to, *from`	nodes at both ends of the street
`int nr_lanes`	nr of lanes in street
`float dists[], vels[]`	distance and velocity of next obstacle on each lane

struct lane	
`struct car *first, last`	first and last cars in lane (`first` is the value that has to be protected by the street mutex)
`int exits[]`	ids of the streets on which this lane exits (for fast lookup)

struct slot	
`int count`	nr of cars that have passed through in this time-slot
`int acc`	accumulated time required by cars passing through in this time-slot

`struct street *street_read ( FILE *in )`
Read a `street` from the given file. This function can also return a `street_foreign` if the street is not local. On error, `NULL` is returned.

`int street_step ( struct street *s )`
Simulate one time-step on the given street. On error, a negative value is returned.

`int lane_push ( struct street *s , struct lane *l , struct car *c )`
Push the given car onto the given lane on the given street. This function is called by `node_pass` and by `handler_pass`. On error, a negative value is returned.

4.2.16 `vsim`

The `vsim` module is the main module in the simulation. It has no data types, but a number of variables and functions through which all operations and structures can be accessed.

`struct btree *vsim_nodes`
A `btree` containing all local and foreign `node`s in the simulation.

`struct btree *vsim_streets`
A `btree` containing all local and foreign `street`s in the simulation.

`struct btree *vsim_routes`
A `btree` containing all `route`s in the simulation.

`struct car *vsim_cars`
A linked list of cars waiting to enter the simulation.

`struct car *vsim_cars_out`
A linked list of cars that have already left the simulation.

`struct manager vsim_manager`
The simulation `manager`.

`struct listener vsim_listener`
The simulation `listener`.

`struct dispatcher vsim_dispatcher`
The simulation `dispatcher`.

`struct bcaster vsim_bcaster`
: The simulation `bcaster`.

`unsigned int vsim_ip`
: The IP-address of the current machine.

`int vsim_readnodes ( int with_ip , char *path )`
: Read the `nodes` from the file with the given name. `with_ip` specifies if the IP-addresses of the nodes should be ignored or not. On error, a negative value is returned.

`int vsim_readstreets ( char *path )`
: Read the `streets` from the file with the given name. On error, a negative value is returned.

`int vsim_readlinks ( char *path )`
: Read the links from the file with the given name. On error, a negative value is returned.

`int vsim_readsigns ( char *path )`
: Read the signs from the file with the given name. On error, a negative value is returned.

`int vsim_readroutes ( char *path )`
: Read the `routes` from the file with the given name. On error, a negative value is returned.

`int vsim_readcars ( char *path )`
: Read the `cars` from the file with the given name. On error, a negative value is returned.

`int vsim_step ( )`
: Run the simulation through one time-step. On error, a negative value is returned.

`int vsim_dump ( char *name )`
: Dump the car positions and velocities to a file with the given name. On error, a negative value is returned.

`int vsim_stimes ( char *name )`
: Dump the street travel times to a file with the given name. On error, a negative value is returned.

4.3 Communication

Communication between machines in a distributed simulation consists, in most cases, of a request with a single answer. The first byte in a message specifies the message type. The message headers are stored in the `handler` module.

Messages	
`0x0`	`error` reply
`0x1`	`ok` reply
`0x3 {char}* 0x0`	read nodes from the given path, `ok` or `error`
`0x4 {char}* 0x0`	read streets from the given path, `ok` or `error` as an answer
`0x5 {char}* 0x0`	read links from the given path, `ok` or `error` as an answer
`0x7 {char}* 0x0`	read cars from the given path, `ok` or `error` as an answer
`0x8 {char}* 0x0`	read routes from the given path, `ok` or `error` as an answer
`0x9 time[4]`	set the simulation time to the given value in seconds, `ok` or `fail` as an answer
`0xa`	simulate one timestep (tick), `ok` or `error` as an answer
`0xb street[4] car_id[4] length[4] bias[4] dist[4] vel[4] route_id[4] step[4] start[4]`	try to pass a car into the given street (compressed form with lane). the `dist`, `vel`, `length` and `bias` arguments are floats, the rest integers.
`(0x10 | 0xc) dist[4] vel[4]`	`pass_ok` or `fail` reply for a pass request. the `dist` and `vel` arguments are floats.
`0xd`	request street travel times
`(id[4] (acc[4] count[4])*)* 0x0 0x0 0x0 0x0`	reply for street travel times. one message consists of the street id and the accumulated time and car count for each time slot.
`0xf`	request for car positions and velocities
`(car_id[4] street_id[4] dist[4] vel[4])* 0x0 0x0 0x0 0x0`	reply for car positions and velocities request. one line is emitted for each car. the `dist` and `vel` parameters are floats.

4.4 Algorithms

Since most of the implementation is quite straight-forward, only a few interesting algorithms are presented here.

4.4.1 `street_step`

This is the function that simulates one time-step on a given street. Instead of processing one lane at a time and performing a linear search for lane neighbors, all the lanes are processed in parallel. Two arrays, `prev` and `next`, contain the

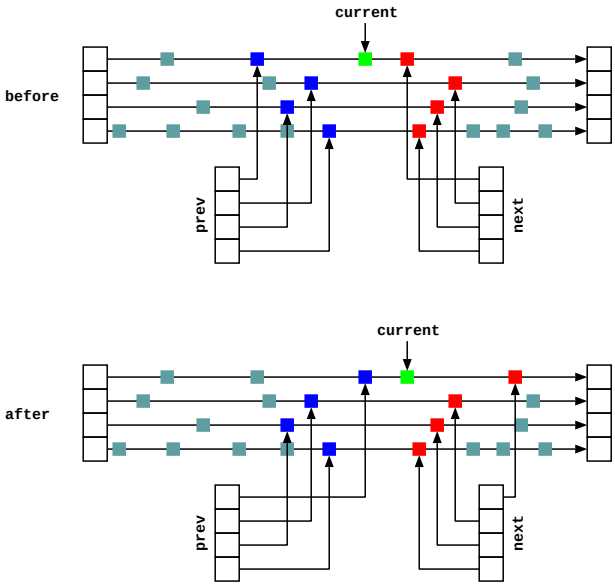


Figure 4.5: `prev` and `next` arrays for processing all lanes simultaneously

first and last cars in a sliding window over the street, making neighbor access a constant time operation (Figure 4.5, Algorithm 1).

4.4.2 `node_pass`

The `node_pass` algorithm (Algorithm 2) decides if a car can cross a node and if so, delegates the car to the appropriate street – local or foreign.

4.5 File Formats

The `vsim` Traffic Simulation uses, with some exceptions, the same file formats as the TRANSIMS Traffic Simulation. It is however possible to convert the differing formats from one program to the other with the help of some small scripts.

4.5.1 Nodes

The nodes-file is identical to that of TRANSIMS with one small detail: if the simulation is to run in a distributed way, then the IP-address of the machine where the node is to reside must be added to the "NOTES" column as an unsigned integer.

Algorithm 1 street_step

```
fill prev array with NULL
fill next array with first car in each lane

while next not empty do
  current = car with smallest pos in next
  pos = lane nr. of current
  next[pos] = current->next

  if current has a next street and
  this lane does not exit on that street then
    if switching to the right will get me where i want to go and
    i have enough room to switch lanes then
      prev[pos]->next = next[pos]
      current->next = next[pos - 1]
      prev[pos - 1]->next = current
    else if switching to the left will get me where i want to go and
    i have enough room to switch lanes then
      prev[pos]->next = next[pos]
      current->next = next[pos + 1]
      prev[pos + 1]->next = current
    end if
  else if i have a lane to my right where i fit then
    prev[pos]->next = next[pos]
    current->next = next[pos - 1]
    prev[pos - 1]->next = current
  end if

  if current must leave the simulation and
  current->pos > half the street length then
    prev[pos]->next = next[pos]
    vsim_out(current)
  else
    if current has a car in front then
      accel = model_drive(...)
      current->pos += current->vel + accel * 0.5
      current->vel += accel
      prev[pos] = current
    else
      if node_pass(...)  == 1 then
        prev[pos]->next = current->next
      else
        prev[pos] = current
      end if
    end if
  end if
end while
```

Algorithm 2 node_pass

```
Require: car c, street/lane from, street/lane to
  if no link from from to to or
  link has unfulfilled wait or halt condition then
    accel = model_drive(...) {act as if node were a dead end}
    return 0
  end if

  if to is local then
    accel = model_drive(...) {next obstacle is first car on to}
    c->pos += c->vel + accel * 0.5
    c->vel += accel

    {did the car cross over the node?}
    if c->pos > from->length then
      c->pos -= from->length
      lane_push(to,c)
      return 1
    else
      return 0
    end if
  else
    msgr = dispatcher_get(to->ip)
    dist = from->length - c->pos
    vel = c->vel

    {does the car leave us?}
    if msgr_pass(msgr,to,c,dist,vel) == 0 then
      accel = model_drive(...)  {use dist and vel as next obstacle}
      c->pos += c->vel + accel * 0.5
      c->vel += accel
      return 0
    else
      free c
      return 1
    end if
  end if
```

4.5.2 Cars and Routes

The cars- and routes-files used in the `vsim` Traffic Simulation depart from the TRANSIMS plans-format completely.

The cars-file contains a list with three entries: the car id, its start time in seconds and the id of the route to take. The route-file contains a list of routes specified by id, number of steps and the steps themselves. The route steps are node ids.

cars-file		
car id	integer	id by which the car will be referenced
start time	integer	departure time in seconds
route id	integer	route this car will take

routes-file		
route id	integer	id by which the route will be referenced
nr steps	integer	nr of steps in this route
steps*	integer	ids of the nodes on this routed

4.5.3 Street Travel Times

The `vsim` Traffic Simulation can generate a file containing information on street useage. The file has one line per street and contains the number of cars and the sum of their travel times per time-slot. The number of time-slots per day is defined in the variable `street_slotsize`.

street travel times		
street id	integer	id of the street
[count,total time]*	integer	nr of cars and total time spent on this street per time-slot

Chapter 5

`vsim` – The Traffic Simulation

This chapter describes how to use the `vsim` Traffic Simulation in it's different modes and presents some results obtained with the implementation described here.

The `vsim` Traffic Simulation is started with the command `./vsim.out` and accepts the following command-line arguments:

command-line arguments	
`-d` *path*	use *path* as the default path
`-n` *file*	use *file* as the default node-file
`-s` *file*	use *file* as the default street-file
`-r` *file*	use *file* as the default route-file
`-c` *file*	use *file* as the default car-file
`-l` *file*	use *file* as the default link-file
`-i` *ip*	set the IP-address to *ip*
`-m` *mode*	set the mode in which to run. *mode* must be one of `stdalone`, `server` or `client`. `stdalone` is the default.

If you started in stand-alone or server mode, you will get a command prompt from which you can control the simulation. The commands are described in the following sections.

5.1 Running in Stand-Alone Mode

When running in stand-alone mode, the `-i` parameter will have no effect. Neither will any IP-addresses in the node-file.

stand-alone mode	
`time`	print the current simulation time
`time` *int*	set the current time to *int* seconds
`step`	simulate one timestep
`step` *int*	simulate *int* timesteps
`view`	open or update the viewer
`view` *int*	update the view every *int* seconds
`prefix`	print the prefix used for all files
`prefix "`*path*`"`	set the file prefix to *path*
`nodes`	load the nodes from the default file
`nodes "`*file*`"`	load the nodes from *file*
`streets`	load the streets from the default file
`streets "`*file*`"`	load the streets from *file*
`cars`	load the cars from the default file
`cars "`*file*`"`	load the cars from *file*
`routes`	load the routes from the default file
`routes "`*file*`"`	load the routes from *file*
`count`	print the nr of cars currently in the simulation
`dump`	dump the car positions and velocities to a file in a TRANSIMS-compatible format with an automaticaly generated name ("`dump/vsim.[time].dump`").
`dump "`*file*`"`	dump the car positions and velocities int *file*
`dump` *int*	automaticaly dump car positions and velocities all *int* seconds
`stimes "`*file*`"`	dump the street travel times into *file*
`quit`	don't know – the program keeps shutting down when I do this...

A typical simulation session will look as follows:

```
vsim - distributed traffic simulation
(c) 2001 by pedro gonnet (pedro@vis.ethz.ch)
vsim_stdalone: running in stdalone-mode...

[vsim_stdalone] prefix "files/"
commline: current directory set to files/

[vsim_stdalone] nodes
commline: read 10564 nodes from file files/nodes.data.

[vsim_stdalone] streets
commline: read 28622 streets from file files/streets.data.

[vsim_stdalone] routes
```

```
commline: read 49980 routes from file files/routes.data.

[vsim_stdalone] cars
commline: read 49980 cars from file files/cars.data.

[vsim_stdalone] time 21600
commline: current time set to 06:00:00 (21600s)

[vsim_stdalone] dump 600
commline: dumpstep set to 600.

[vsim_stdalone] step 3600
commline: 344 seconds elapsed (10.47 ratio)...

[vsim_stdalone] view

[vsim_stdalone] quit
```

The session tells the simulation where to look for its file (`prefix "files/"`), reads first the nodes, then the streets, routes and cars. This order must always be observed for the simulation to work. The time is then set to 06.00 (`time 21600`). The simulation is told to produce dumps every 600 seconds (`dump 600`) and is let run for one hour (`step 3600`). Finaly, a viewer is opened and the simulation is quit.

5.2 Running in Server Mode

To run the `vsim` Traffic Simulation in server mode, one must first create a node-file with the IP-addresses of all client machines. Furthermore, the `vsim` Traffic Simulation must be running in client mode on all client machines with the `-i` flag specifying the IP-address.

server mode	
`client "`*ip*`"`	adds a client with the IP-address *ip* in dot notation
`connect`	connects to all clients
`disconnect`	close all open connections to clients
`time`	print the current simulation time
`time` *int*	set the current time to *int* seconds
`step`	simulate one timestep
`step` *int*	simulate *int* timesteps
`prefix`	print the prefix used for all files
`prefix "`*path*`"`	set the file prefix to *path*
`nodes`	load the nodes from the default file
`nodes "`*file*`"`	load the nodes from *file*
`streets`	load the streets from the default file
`streets "`*file*`"`	load the streets from *file*
`cars`	load the cars from the default file
`cars "`*file*`"`	load the cars from *file*
`routes`	load the routes from the default file
`routes "`*file*`"`	load the routes from *file*
`count`	print the nr of cars currently in the simulation
`dump`	dump the car positions and velocities to a file in a TRANSIMS-compatible format with an automaticaly generated name ("`dump/vsim.[time].dump`").
`dump "`*file*`"`	dump the car positions and velocities int *file*
`dump` *int*	automaticaly dump car positions and velocities all *int* seconds
`stimes "`*file*`"`	dump the street travel times into *file*
`quit`	don't know – the program keeps shutting down when I do this...

A typical session would look like a normal stand-alone session, yet with a few `client` lines and a `connnect` at the beginning.

5.3 Running in Client Mode

Running the `vsim` Traffic Simulation in client mode is trivial, since there is no direct interaction with the user. The simulation must be started with the `-i` parameter specifying the IP-address through which the machine will be called.

Chapter 6

Performance Results

The `vsim` Traffic Simulation was tested with two different traffic scenarios:

- **Gotthard Scenario:** 50'000 cars trying to cross the swiss alps. The cars start between 06.00 and 07.00, get stuck at the Gotthard tunnel until about 06.00 the next day
- **Switzerland Scenario:** 6'000'000 cars travelling in, out and arround Switzerland. Traffic starts at 00.00 and accumulates until about 21.00 and disperses two days later. This scenario was designed as a stress test for large-scale microsimulations.

Both simulations use a traffic network consisting of 10'564 nodes and 28'622 streets. The network represents a rough roadmap of europe with more detail in Switzerland.

The distributed tests were run on the Institute of Computer Systems' CoPs Cluster [CoPs] consisting of 16 dual-Pentium III 1GHz machines with switched gigabit ethernet.

6.1 Gotthard Scenario

The simulation was run on only 8 machines in the CoPs cluster. This was not for performance reasons, but more due to the fact that the whole cluster could only be reserved over the weekend and not all test could be run on time.

Furthermore, the simulation was run using 10 `runners`. The exact number is rather arbitrary, but it produced better results than 5 or 20 threads.

The results are summed up as simulated seconds per wall-clock seconds over six hour timespans.

Gotthard Scenario	
06.00 – 12.00	155.40 s/s
12.00 – 18.00	209.71 s/s
18.00 – 24.00	191.15 s/s
00.00 – 06.00	245.45 s/s
geom. avg.	197.74 s/s

6.2 Switzerland Scenario

For this simulation, all 16 machines were used, each running 5 threads. As with the last scenario, the simulation time / real time ratio was measured for six hour time intervals.

Switzerland Scenario	
00.00 – 06.00	136.71 s/s
06.00 – 12.00	67.71 s/s
12.00 – 18.00	52.94 s/s
18.00 – 24.00	41.54 s/s
geom. avg.	67.17 s/s

6.3 Notes

6.3.1 Graph Partitioning

One of the greatest speedup factors was, without doubt, the partitioning of the traffic network. This was done with the METIS graph partitioning package [MET].

The input for the graph partitioning was generated from the street travel times data collected after one simulation run. For each time-slot, a node is

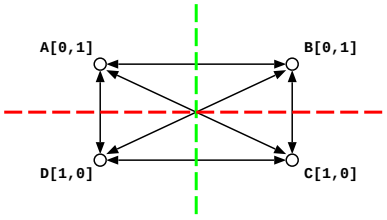


Figure 6.1: Multi-Constaint Partitioning: The values in square brackets for each node denote the load over two different time-slots. If the average load over both time-slots is considered, then both partitions (green and red) are equaly good. Yet if we consider both loads separately, the green partitioning is far better than the red.

assigned the sum of the accumulated travel times in seconds for each incomming street (i.e. the number of car-seconds it has to process). The streets are weighted by the number of cars that entered the street per time-slot (i.e. the number of transactions necessary if we were to cut that street).

For both simulations partitions were created using the data from eight time-slots as multiple constraints. This produced much better results than the partitioning used by TRANSIMS which divides the network according to one set of constraints. Figure 6.1 shows how and why this can go wrong.

6.3.2 Threads

It is rather difficult to quantify the number of threads needed for a given simulation. The overhead involved in task-switching has to be less than time spent on un-amortized communication.

For both scenarios, the best numbers of `runners` was calculated experimentaly.

6.3.3 Useability

Although the speed results of the `vsim` Traffic Simulation are rather impressive, the results produced are completely useless. Since no information on street priorities or traffic lights were available for the network, the realism of the traffic model quickly became a major drawback, creating pileups at almost every unsignalized crossing.

There are many ways this could be fixed, yet defining a better network would be preferrable.

Chapter 7

Outlook

7.1 What has been done

The best way to quantify what has been achieved is to compare the end results with the design goals mentioned at the beginning of this text.

7.1.1 Speed

The speed criteria is probably the easiest to quantify, since the results can be measured and compared. Since no reference numbers were available for either one of the scenarios tested, there is no direct comparison to previous work.

The only reference available is the theoretical limit of 100 simulated seconds per second with a similar cluster configuration postulated in [ParTS] for the parallel implementation of the TRANSIMS micro-simulation.

7.1.2 Realism

As mentioned earlier, the results of the `vsim` Traffic Simulation are completely unuseable. The question, wether this is a problem with the simulation itself or the network, would require running the simulation on a more complete network.

7.1.3 Distributeability

The `vsim` Traffic Simulation is easily distributeable and the distributed traffic model seems works. No real analysis was made of the useage of network resources or on the amount of time waiting on I/O, making it difficult to make a concrete statement.

7.1.4 Simplicity

The whole source of the `vsim` Traffic Simulation is less than 6'000 lines of c-code. The modularity of the system allows the user to modify parts of the

system without needing to be completely familiar with the rest. The separation of the `model` module should be especially usefull for those testing new driver behaviour models.

7.2 What has not yet been done

There are still a number of features that were originaly planed, yet not yet implemented into the `vsim` Traffic Simulation. Since the simulation was designed asuming their existence, it should be possible to add them without too many problems.

7.2.1 Signaled Intersections

Although the `node` module can read intersection data, it is not regarded in `node_pass`. One big problem here would be how to handle yellow phases. The easiest solution would be to add one function to the `model` module which decides if to pass or brake.

7.2.2 Wait and Halt Conditions

This information is read but not used by `node_pass`. Tere are comments in the source indicating where to insert eventual code. Unfortunately, the information in the TRANSIMS files only tells us where a stop or yield sign stands, and not which lanes or links it pertains to.

Bibliography

[CoPs] Research Group for Parallel and Distributed Systems – Cluster
http://www.cs.inf.ethz.ch/stricker/lab/

[MET] METIS: Serial Graph/Mesh Partitioning & Sparse Matrix Ordering
http://www-users.cs.umn.edu/~karypis/metis/metis/

[ParTS] Parallel implementation of the TRANSIMS micro-simulation.
Kai Nagel and Marcus Rickert