A Reactive Method for Real Time Dynamic Vehicle Routing Problem

Kenny Qili Zhu and Kar-Loon Ong
Dept. Computer Science, National University of Singapore, Singapore 119260
{kzhu, ongkarlo}@comp.nus.edu.sg

June 15, 2000

Abstract

Traditional Vehicle Routing Problem with Time Windows (VRPTW), which has been well studied, involves some cost matrix, time and capacity constraints. The objective is to service all customers with minimal number of vehicles and at lowest cost. The solution is however static. In practice, situations may arise when vehicles on the pre-routed assignment encounter unexpected obstruction and cannot visit their designated customers on time, therefore the routing has to be dynamically computed. We classify this type of problems as *Real Time Dynamic Vehicle Routing Problem* (RTDVRP). This paper describes the RTDVRP, its solution: a concurrent, agent-based, *Reactive Vehicle Routing System* (RVRS) and the implementation of RVRS, which combines a concurrent architecture partially customized from Open Constraint Programming (OCP) framework and an *incremental local optimization* (ILO) heuristic.

1 Introduction

VRPTW is a well-known *NP*-hard problem which is encountered very frequently by the logistic department that makes decision about the distribution of goods and services. The objective of the problem is to find routes for the vehicles to service all the customers at a minimal cost (in terms of travel distance, etc.) without violating the capacity and travel time constraints of the vehicles and the time window constraints set by the customers. Almost all VRPTW methods proposed are devoted to a static problem as the problem parameters and constraints have to be known in advance, and any change in these data requires a complete re-calculation. Until now little research has been focused on a dynamic VRPTW, where problem size and parameters change after the vehicles are already commissioned. Brown and Graves [2] make the early attempt at real-time vehicle dispatching problem in 1981. Following that, Li, Rousseau and Gendreau [9] described a method in solving real-time bus dispatching problem. Recently, Gendreau, Guertin, *et al.* [5] published a parallel tabu search for real-time VRP. Their research, however, concentrated on the solving algorithms and failed to acknowledge the real time concurrency and reactivity issues of the problem.

In our research, we attempt to describe a Real Time Dynamic Vehicle Routing Problem (RTDVRP) which is reactive in nature and benefits from a level of concurrency control. In this model, we view vehicles as agents that communicate with each other and with the central control by a common solver store. The medium of communication is reactors, simple atomic programs that involve a condition, an action and a time-out. The relationship between vehicles can be represented by constraints as one vehicle's breakdown or slowdown can affect other vehicles' designated routes. In this sense, the VRPTW solver is also a constraint solver that governs the interactions between various vehicle agents. To facilitate the solving of RTDVRP, we introduce a reactive vehicle routing (RVR) infrastructure, which is derived from the *Open Constraint Programming* (OCP) [7] concept. The main component of the infrastructure is a registry that handles concurrent submission, serialization, delay and wake-up of the reactors. It is a highly structured architecture for concurrent constraint programming. The VRPTW solver uses an incremental local optimization (ILO) algorithm, and a persistent memory store. It solves VRPTW on a dynamic basis without solving the entire problem again, hence it is quick and effective.

In this paper, we first introduce the concept of RTDVRP, followed by the RVR functional framework. Then we present an implementation of RVR infrastructure and a simulation based on the RVR System. Finally we show the the experimental benchmark results.

2 Description of RTDVRP

The VRPTW is given by a set of identical vehicles, a special node called the depot, a set of customers to be visited, a directed network connecting the depot and the customers [14]. Let us assume there are M customers and K vehicles. Each arc in the network corresponds to a connection between two nodes. A route is defined as starting from the depot, going through a number of customers and ending at the depot. The number of routes in the traffic network is equal to the number of vehicles used, that is, K. Therefore, exactly K directed arcs leave the depot and K arcs return to the depot. A cost d_{ij} and a travel time t_{ij} are associated with each arc of the network. Every customer in the network must be visited only once by one of the vehicles. Since each vehicle has a limited capacity C_k (identical for all vehicles in the question), and each customer has a varying demand m_i, C_k must be greater than or equal to the summation of all demands on the route traveled by that vehicle k. On the other hand, any customer must be serviced within a pre-defined time interval, limited by an earliest service time and latest service time. Vehicles arriving later than the latest arrival time are penalized while those arriving earlier than the earliest arrival time incur waiting. Vehicles are also supposed to complete their individual routes within a total route time which is essentially the time window of the depot. Various heuristics have been developed in handling this problem. However in real life situation, traffic conditions change dynamically, causing initial best assignment not to apply anymore. Worse, time window constraints may be violated. Our system tries to attack this situation with a reactive system that dynamically restructures the routing assignment based on current traffic conditions.

As with conventional VRPTW, the initial best solution is obtained with static data. Each vehicle is given an assignment and departs from the same depot. During the trip, they follow default routes unless a new assignment is given from the center. All vehicles keep the center updated whenever they reach a customer place. When a vehicle encounters traffic jam, it feeds back the traffic condition to the control center. The center will check whether the vehicle is able to achieve its target even with the traffic jam. If true, it will ignore the report while updating the traffic condition to the cost matrix. Otherwise, re-routing will be carried out to provide an alternative route for this vehicle. The re-routing process involves other vehicles, too. Thus all vehicles should be updated with the new solution. To reduce disturbance to other vehicles, restructuring does not involve current destination of unjammed vehicles. In the case where restructuring with current vehicles is not possible, additional vehicle will be issued and departs at the time immediately after the new assignment is obtained.

The Model of RTDVRP

The problem is modeled with Z notation.

M = number of customers

K = number of vehicles

Triptime = maximum travel times allocated to each vehicle

C_k = maximum load of vehicle $k = C$

d_{ij} = distance from node i to node j

t_{ij} = travel time from node i to node j

m_i = demand of customer i

TimeReady_i = earliest time customer i can be served

TimeDue_i = latest time customer i can be served

Service_i = service time at customer i

Next_i = successor of node i

TimeReach_i = reaching time at customer i

Wait_i = waiting time at customer i

Load_i = load of vehicle when it reaches i including load of customer i

$\text{Customers} = \{x : N | 1 \leq x \leq M\}$

$\text{Depot} = \{x : N | M + 1 \leq x \leq M + K\}$

$t_{ij} = \{ i, j \in \text{Customers} \cup \text{Depot} \,|\forall i, j \bullet \exists \tau \in N_1 \bullet t_{ij} = \tau * d_{ij}\}$

$\text{Time} = \{x : N | 0 \leq x \leq \text{Triptime}\}$

$\text{Next} = \{(i, x) : i \in \text{Customers};$

$\quad x \in \text{Customers} \cup \text{Depot} \mid \forall i, j \in \text{Customers} \bullet (i \neq j) \bigwedge$

$(\text{Next}(i) \neq \text{Next}(j)) \ \bigwedge \ (\text{Next}(i) = j \Rightarrow \text{Next}(j) \neq i) \bigwedge (\text{Next}(i) \neq i)\}$

$\text{Next} = \{(i, x) : i \in \text{Depot}; x \in \text{Customers} \mid \forall i, j \in \text{Depot} \bullet (i \neq j) \Lambda(\text{Next(i)} \neq \text{Next(j)})\}$

$\text{TimeReach} = \{(j, x) : j \in \text{Customers}; x \in \text{Time} \mid (t_{j(N+1)} \leq x \leq \text{TimeDue}_j) \bigwedge$

$(x \leq \text{Triptime} \text{ -} t_{j(N+1)} + \text{Service}_j \ ) \bigwedge (\forall i, j \in \text{Customers} \bullet \text{Next}(i) = j \Rightarrow$

$\text{TimeReach}(j) = \text{TimeReach}(i) + t_{ij} + \text{Service}_i + \text{Wait}(i)\}$

$\text{Wait} = \{(i, x) : i \in \text{Customers}; x \in \text{Time} \mid (0 \leq x \leq \text{TimeReady}_j \text{ - } t_{i(N+1)}) \bigwedge$

$(\text{TimeReach(i)} \leq \text{TimeReady}_i \Rightarrow x = \text{TimeReady}_i \text{ -TimeReach}(i)) \bigwedge$

$(\text{TimeReach(i)} \geq \text{TimeReady}_i \Rightarrow x = 0)\}$

$\text{Wait} = \{(i, x) : i \in \text{Depot}; x \in \text{Time} \mid x = 0\}$

$\text{Load} = \{(j, x) : j \in \text{Customers}; x \in N \mid (0 \leq x \leq C) \bigwedge$

$(\forall i \in \text{Customers} \bigcup \text{Depot}; \forall j \in \text{Customer} \bullet \text{Next}(i) = j \Rightarrow$

$\text{Load}(j) = \text{Load}(i) + m_i\}$

There are two differences of this model from conventional VRPTW's:

1. Travel time between nodes

   For static problem, traveling time between two nodes is equal to the distance between them. However, in RTDVRP, traveling time between nodes changes dynamically due to the change of traffic condition. The severity of the traffic jam is represented by a variable τ which is the ratio of travel time over travel distance, as shown in the model. When there is no traffic jam, the variable τ is equal to one.

2. Departure time of vehicles

   For static problems, all vehicles depart at time zero. Contrast to that, in RTDVRP, there are some dynamically appointed vehicles that have differing departure times.

3 A Concurrent Reactive Framework for Solving RTDVRP

In the following sections, we present a framework that is capable of concurrent traffic updates and reactive routing. The Reactive Vehicle Routing (RVR) model constitutes an concurrent registry, an incremental local optimization solver, a number of vehicle agents and reactors launched by the agents. Figure 1 shows the infrastructure and the interaction in the framework. The arrows in

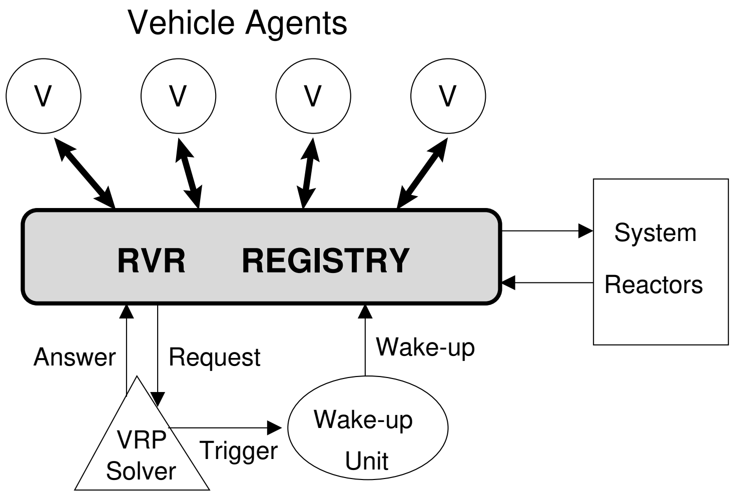

Figure 1: The infrastructure of RVR

the figure denote the movement of reactors. The registry provides the interface and concurrency control to the agents, which launch reactors that may hibernate in the registry for a long time and only wake up when some event takes place. In the meantime, the registry serializes the requests from the agents and active reactors and sends them to the VRPTW solver sequentially for process. The VRPTW solver can process one request at a time, which can be solving for a particular VRP problem, updating the location of the vehicles, or reporting route condition, etc. The solver is also a data store that keeps the most up-to-date solution, vehicle current locations, the cost matrix and various constraints. Reactors interact with each other through the solver/store. A trigger mechanism facilitates the wake-up of hibernating reactors when some events occur to the database. The architecture of RVR is partly customized from Open Constraint Programming (OCP) proposed by Jaffar, *et al.* In what follows, we provide detailed account for the theory and functionality of the components in RVR.

3.1 Vehicle Agents and Reactors

In the context of RTDVRP, vehicles can be viewed as distributed real time *agents*. Distributed agents are independent intelligent entities that are each loaded with some small programs to accomplish certain tasks. These small programs are referred to as "*reactors*" because they usually react to events and perform some job. According to Wooldridge and Jennings [13], a distributed intelligent agent exhibits the following characteristics:

Autonomy: Agents operate without direct human intervention and have control over their own actions;

Social ability: Agents can communicate with each other using a uniform, mutually understandable language or mechanism;

Reactivity: Agents understand changes of environment and react to the changes;

Pro-activeness: Agents are capable of exhibiting goal-directed behavior by taking the initiative.

The vehicle agents are independent of each other once dispatched and they are self-controlled in the sense that they only follow the original routing plan given by the central server and new destinations given by their own reactor programs in view of traffic situation changes. The vehicles communicate through a common memory in the solver which contains the distance matrix, time/capacity constraints and the current routing solution for all vehicles. One vehicle's reporting of traffic condition could affect some or all other vehicles' next moves. Every vehicle is reactive because they issue a number of reactors from simple short-lived update of location or cost to more sophisticated residual query of next new moves as a result of some cost update from any vehicle agent. Vehicles, when equipped with smart reactor programs, could actually deal with complex situation such as weather, geographic location, etc. Therefore they have the potential to take pro-active initiatives. All these characteristics conclude that the vehicle agents are instances of intelligent distributed agents.

We can express the reactor invoked by an agent informally as

$$REACT\{\delta(\Delta) \rightarrow \delta'(\Delta)\} \tag{1}$$

where Δ denotes the database store, and $\delta(\Delta)$, $\delta'(\Delta)$ are updates to the data store, in that order. Formula (1) indicates that if and only if $\delta(\Delta)$ is true, both updates will be executed. In its simplest form, a reactor can be $REACT\{\delta'(\Delta)\}$, which means an update of the store Δ. In the above cases, the updates of the store are performed *atomically*. We can stretch the concept of reactor a little further by introducing queries. A *query* is a question about the state of the store and can be shown simply as: $QUERY\{\Delta\}$. A query only involves a question and an answer about the database and there is no update or change made to the store Δ. In fact query can be viewed as a special case of reactor, because formula (1) can be degraded to a query if $\delta'(\Delta)$ is ϕ (empty) and $\delta(\Delta)$ is written as $\zeta(\Delta, \theta)$ which asks Δ about some condition θ. In the OCP language, we can write the reactor as:

$$repeat\,(\delta_1 \Rightarrow \delta_2\ watching\,\epsilon) \tag{2}$$

Here we can treat δ_1 as condition, δ_2 as action and ϵ the timeout. The timeout determines how long the reactor will remain in the system. If ϵ is true, the reactor is timed out and no longer valid. Some reactors keep looping while others are run only once. In RVR model, we will have the following reactors:

new_soln: ONCE (*true*$\Rightarrow$ load new VRPTW problem and solve it, WATCHING *true*)

new_customer: ONCE (*true* $\Rightarrow$ add new customer and solve the new problem, WATCHING *true*)

update_tmwin: ONCE(*true*$\Rightarrow$ tell the store new time window of a customer, WATCHING *true*)

new_vehicles: LOOP (new solution from more vehicles $\Rightarrow$ dispatch new vehicles, WATCHING all customers visited)

next_move: LOOP (new solution $\Rightarrow$ ask for next moves, WATCHING all customers visited)

update_location: ONCE (true $\Rightarrow$ tell my present customer, WATCHING end of route)

update_cost: ONCE (true $\Rightarrow$ tell my new cost of running from customer i to j, WATCHING end of route)

The first four reactors are system reactors, because they are usually invoked by the RVR system, or the central vehicle controller. In particular the first three have impact on the integrity of the solver database, hence trigger other reactors to be active. The latter three are issued by vehicle agents. The most important one is *next_move*, as it guides the route for a vehicle dynamically. *Update_cost* is the only vehicle reactor that may change the solution in the database. It represents the situation of a traffic jam or vehicle breakdown, when the pre-designated route cannot be completed. In such instances, the solver will have to re-route the remaining problem in the store and give new solution. Together these seven reactors forms the foundation for solving RTDVRP problems. In this paper, all but *new_ customer* and *update_ tmwin* are implemented and tested. The implementation of these reactors will be discussed later.

3.2 RVR Registry

In a large VRPTW problem, up to dozens of vehicles are involved, each of which visiting several or even tens of customers on its route. The vehicles can be scattered across a very vast area, encountering different traffic, geographic and weather conditions. In such a complex system, it is common for two vehicles to breakdown at the same time, or five vehicles reporting the arrival at a destination simultaneously. Therefore concurrency is a apparent feature of RVR system. The underlying VRPTW solver, however, is sequential, taking one request at a time and answer the request before proceeding to the next. In other words, we need some kind of concurrency control to coordinate the external requests. On the other hand, we also need some mechanism to manage the repeating reactors internally. A proper architecture for the reactor circulation and trigger techniques are useful here. Figure 2 shows a schematic of the RVR registry.

The RVR registry, derived from that of a conceptual OCP model, solves the above problems by taking a modular approach, specifically, a *Receptionist*, a *Delay* module, *Run* module and a *Wakeup* module. The registry functions as follows.

The Receptionist maintains two internal queues, one for incoming requests and one for outgoing replies. Incoming reactors are temporarily stored in the incoming queue and are waiting to be processed. When it is time for a reactor to go, it is ejected from the reception queue and appended to the *conditional-run*, or *cond_ run* queue in the Run module. Reactors in the cond-run queue are then fed to the solver one by one, but only have their condition δ_1 tested. If δ_1 fails, they go straight to the Delay module and start to sleep. If the condition δ_1 succeeds, in general δ_2 will succeed too, so we submit the entire reactor to the *actual-run* queue. When its turn arrives, an actual-run procedure unpacks the request from the reactor and translate it from a well-defined RVR language to a form that is understandable to the VRPTW solver, which executes the request and gives an answer. If the reactor is a repeating one, it will be queued in

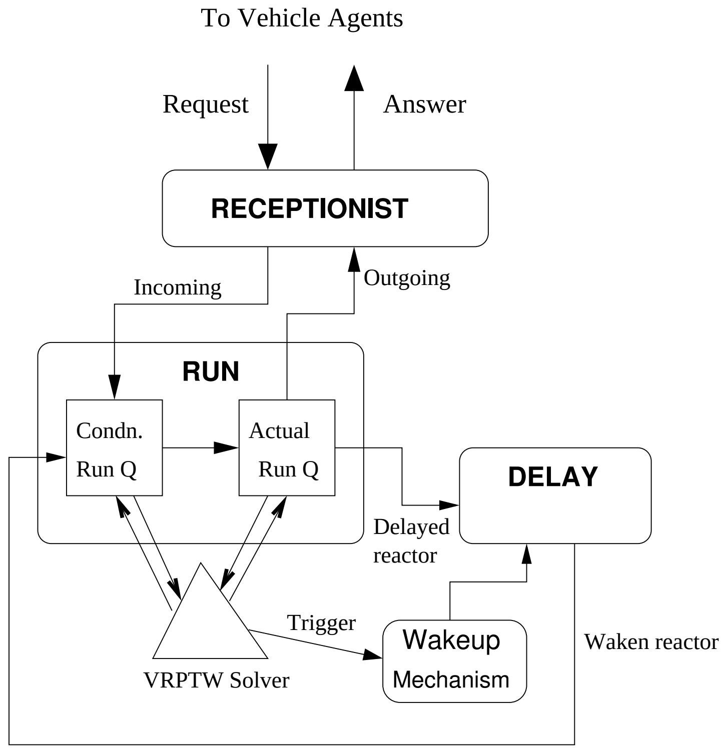

Figure 2: A schematic of RVR registry (with solver and wake-up)

the Delay module and put to sleep. Otherwise the reactor is killed. In any case the answer is returned to the output queue of the Receptionist and stays there until it is time to be sent back to the agent which issued the reactor earlier. At this point, after the solver determines if the database has been changed (either there is a new solutions or the cost matrix has been updated), it sends a triggering signal to the Wake-up unit. The Wake-up mechanism makes the decision of waking up the relevant dormant reactors in the Delay module and put them to the cond-run queue again.

In essence, the main characteristics of the RVR registry are summarized as follows: First, the registry provides the concurrency control at the reception front end to accommodate simultaneous agent requests. Second, it serves as a parser that translates the agent language to solver language. Third, it coordinates the requests and queue them up according to certain priority policy, and then feeds the request one by one to the solver. Fourth, it maintains a delayed queue that contains all the requests that cannot be completed or those that are looping. And finally, the registry is equipped with a trigger mechanism to wake up appropriate reactors for re-evaluation.

3.3 VRPTW Solver

The solver adopts an incremental local optimization scheme similar to Caseau's [3] for routing. The solution is formed incrementally, while local optimization is done along the process of constructing full solution. Earlier optimization tries to bring the search to a better space all the time rather than trying hard to improve a poor first solution. Currently, most techniques for solving vehicle routing problem are based on local search, where there are two distinct phases. First phase uses simple insertion heuristics to form an initial feasible solution. Second phase is an iterative local search trying to improve the initial solution. Various local search techniques have been attempted, including tabu search [4] and genetic algorithm [12]. Recently, there is increasing interest in applying constraint programming techniques in vehicle routing problems. The advantages are twofold, firstly, the constraint programming has an attractive feature that allows additional constraints. Secondly, the propagation helps in pruning search space. Paul Shaw *et al.* [1] adopted constraint propagation for validity checking and to reduce search space. Paul Shaw further endorsed a constraint propagation technique [11] by relaxing some related nodes of the solution and applying branch and bound search to these nodes for optimal solutions.

In the beginning, all customers are put into a stack. Then one customer is popped out from the stack. This customer will be first destination of the first vehicle. When the second customer is popped out, the heuristic checks whether it can be located before or after first customer. If both positions are not possible, it allocates a new vehicle. Each customer is served following the order in the stack. Each time the heuristic looks for best location for the customer, with fewer vehicles and less travel distance. After allocation of one customer, a series of optimization is performed involving operations on routes aimed at reducing the number of vehicles and travel distances. The solver ensures that all changes during optimization do not violate the constraints. Algorithm 1 summarizes the routing process.

Routes operations in optimization are:

1. exchange within a route and between routes

2. 3-opt within a route and between routes

3. swap within a route and between routes

Algorithm 1 Routing process

```
stack customer;
while customer not empty
begin
    cust = customer.pop_back()
    if(not insertion(cust, existing routes)) // no solution
      newroute(cust)
    else
      optimize(routes)
end
```

The optimization on a route is initiated whenever the route is modified. Route operations further modify other routes, putting each route into a queue waiting for their turn to be optimized. The optimization ends when the queue becomes empty. This optimization is greedy, as only downhill move is allowed, which potentially causes the search to be trapped in local minima. The heuristic handles it by inserting the customers with different order based on one criteria. The criteria considers the priority of customer to be served. Put it other way, the customer which is "harder" should be served earlier compared to others. Two factors are taken into consideration in determining the "hardness" or priority:

1. The distance of the node from previous node in the priority queue (the nearer the higher priority: tendency to form a shorter route) and
2. The node's due time (the earlier the higher priority: tendency to be able to be served in time).

The relative importance of these two factors is compromised with a weighting factor w. Thus the priority p formula is:

$$p(customer_j) = w \times d_{ij} + (1 - w) \times TimeDue_j$$

where i is the customer before j in the priority list (higher priority). Note that the lower the value of p, the higher priority is the customer. This weighting factor need to be tuned to suit different problems. It has sizeable affect in the quality of the solution.

In a word, the quality of result heavily relies on the route operations (local moves) and customers priority. The optimization performs an exhaustive search looking for all opportunity in reducing the number of vehicles and travel distance. This approach tends to bring the search into local minima. Customers reordering, to a great extent, bring the search out of poor local minima but does not help in all cases.

4 RVRS: Implementation and Simulation

In the following sections, we introduce the Reactive Vehicle Routing System (RVRS) we have implemented for solving RTDVRP's and a simulation we did with the help of RVRS as well as its computation results.

4.1 RVRS System Implementation

The receptionist process of RVRS listens to incoming requests on a designated port through Internet socket connection. After an initialization conversation with the vehicle agent, the agent handed over the reactor in a character string to the receptionist. The string appears like macros in C, with "#" separating the condition, action and time out expressions. And the condition, action and time out come in the form of a Prolog/CLP(R) like statement with a special RTDVRP predicate, for example:

```
#true#update_location(vehicle_no, customer_no)#end_of_route(vehicle_no).
```

Every reactor object is given a unique identification code so that it can be tracked in the system and queried about if necessary. Noticeably, agent is given a quick acknowledgment by the receptionist immediately after the reactor is received and appended to the input queue. The real answer from the solver is attached to the reactor object and circulated back the reception's output queue. It waits there until the original agent polls for that reactor's id.

In this implementation the condition in the reactor is always *true* so we scrap the cond-run queue from the scheme and send reactors directly to the actual-run queue. The actual-run procedure is linked to the solver through a named pipe. The VRPTW solver is customized to be equipped with a simple parser that extracts out the parts in the reactor string. The solver executes the request and sends a reply back via the pipe. There are 4 modes of solution for the

	Rochat & Taillard '95		Caseau '99		**ILO with ordering**	
	distance	vehicle	distance	vehicle	distance	vehicle
R1 (12 probs)	1208.5	12.16	1243.53	12.67	1235	13.08
C1 (9 probs)	828.38	10	829.29	10	828.56	10
RC1(8 probs)	1377.39	11.87	1438.17	12.12	1415	12.88
R2(11 probs)	961.71	2.91	983.45	3.18	968	2.91
C2(8 probs)	589.85	3	599.81	3	590	3
RC2(8 probs)	1119.59	3.37	1295.58	3.37	1138	3.38

Table 1: Solutions for Solomon VRPTW problem sets

solver, each given a unique mode number: 0 for timeout, 1 for invalid condition, 2 for success but no change to solution and 3 for success and change to the solution. Note only mode 3 can trigger a wake-up of the delayed reactors. The solver writes the mode number to a special file which Run module reads to determine whether to wake up some of the dormant reactors in the Delay module. If it sees a 3 in the file, it sends a signal to the Delay module which pops out all the *next_move*, *new_vehicles* reactors in the delayed queue and *preempt* them in the actual-run queue ahead of other reactors already waiting there. The reason for the preemption is: some of the waken-up reactors were delayed because their conditions were not true earlier and they should be given higher priority than the first-timers in the run queue. If the reactor has been timed out or has completed successfully, a response is queued at the receptionist and the reactor gets removed from the system. In case of a false condition, the reactor is delayed, too.

The VRPTW Solver conceptually consists of several components:

daemon The daemon always stands by for requests. This is implemented by blocking read from the communication pipe. Once a request is received, the daemon decodes the content of the request. It then communicates with either the router or the store based on the type of request. After processing the request accordingly, it sends the feedback through another communication pipe.

router The router is responsible for providing the routing solution to the problem and performing re-routing when the traffic condition changes.

store The store keeps track of the vehicles and traffic status. Both daemon and router rely on it for data. Re-routing requires vehicles location and traffic condition. The daemon refers to the store for certain enquiries from vehicles such as "*next_move*" and "*new_vehicles*".

The router is the core of the solver, its effectiveness very much affects the overall system performance. The underlying heuristic is the ILO as mentioned in Section 3.3. Its robustness is demonstrated by the quick solving (maximum several seconds for a complete problem with a Pentium II 350 MHz 128M Ram on Linux 2.2.14) with promising results on the majority of Solomon problems.

Table 1 shows the result of the heuristic on the Solomon benchmark problem set. The performance is better on problem series 200 (C200, R200 and RC200) where vehicles have big capacity and there are fewer routes. The solutions are compared with those by one of the best known heuristics by Rochat and Taillard [10] and heuristic on which the idea of ILO is based by Caseau [3].

Re-routing is required when a vehicle encounters a traffic jam and is unable to complete the route without violating constraints. The re-routing uses same optimization technique in routing process, but only those unserved customers are re-routed. Furthermore, in order to avoid disturbance to other vehicles, current destination of these vehicles are not reallocated. There is a situation where solution can not be found even after re-routing. This happens when it is too late to issue new vehicles for certain customers and vehicles on the road are unable to serve it due to obstruction. Figure 3 shows how re-routing is performed.

4.2 A Discrete-time Simulator

This section describes a discrete real-time simulator we developed to evaluate the performance of RVRS. The simulator uses VRPTW problems from the Solomon 100-customer benchmark problem set. In all 56 problems, the 100 customers are distributed within a $[0, 100]^2$ square and the travel times are corresponding to the Eucleadian distances. The 56 problems are divided into 6 categories, namely C1, C2, R1, R2, RC1, RC2. The customers are clustered in the C type problems, uniformly distributed in the R problems, and mixed in the RC cases. Problems of type 1 have a short trip time for vehicles and the vehicles have smaller capacity, while the problems

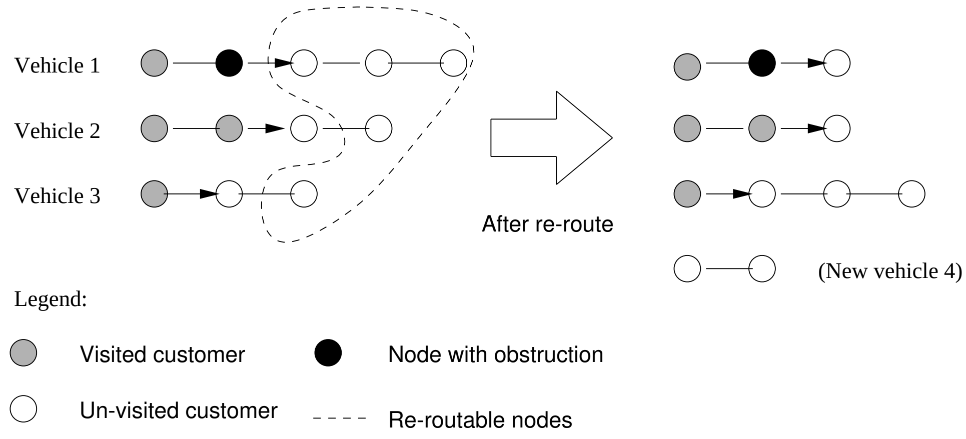

Figure 3: Re-routing process

of type 2 are the opposite. We scale the time horizon down to 1 second per time unit (distance unit) for the ease of the testing.

The main process first sends RVRS a *new_soln* reactor. This loads the initial problem into the solver and a new solution will be stored in the store. Meanwhile, vehicles program forks to a process that sends another system reactor *new_vehicles* to RVRS. This reactor watches out for situation where new vehicles need to be added to the scene.

Initially V vehicle processes were forked out according the optimization solution. They each invoke the *next_move* reactor that will guide them through the assignment. The vehicles then simulate setting off from the depot at the same time 0, and visiting customers on their given route. If new vehicles are needed, they are spawn out as new processes, that inherit the system time, so that the time windows and trip times can be correctly aligned with older vehicle processes. As a vehicle reaches a customer in its route, it reports its location by sending reactor *update_location(x)*, where x is the node number of the customer. At the same time, at every stop there is a fixed possibility φ $(0 \leq \varphi \leq 1)$ of a cost increase on the way to the next customer. And if there is a cost increase, a random number τ is used to as a multiplying factor to the cost, and the vehicle sends a *update_cost(x, f)*, where x is the current customer and f is the cost factor, to the registry. Every second, the vehicles poll the results from the reactors, especially *next_move*. The polling rate is reasonable because only the vehicle that encounters obstruction needs the result immediately to switch its next destination. More frequent polling adds to the system load of RVRS.

4.3 Simulation

In the simulations, we evaluate the performance of the RVRS by the quality of the solutions under the effect of traffic jams, and the robustness of the RVR architecture.

In VRPTW, all nodes are connected between each other with a distance. The traveling time from one node to another is equal to the distance. Hence they are used interchangeably. However in RTDVRP, the traveling time varies with traffic condition, and such equality does not apply. We introduce a notion of *network resistance*, defined as the difference between the total travel time and the total distance of the network. This parameter measures how badly the network is congested. Change in travel distance in light of the network resistance suggests robustness of the solver. We also introduce *elasticity*, an estimation of how stable the RVR system is. The parameters are defined formally:

Network resistance:

$$\Re = \sum_{i=0}^{n}\sum_{j=0}^{n} t_{ij} - \sum_{i=0}^{n}\sum_{j=0}^{n} d_{ij}$$

Total travel distance of vehicles:

$$D_r = \sum_{i=0}^{n}\sum_{j=0}^{n} d_{ij} \times x_{ijr}$$

where

$$x_{ijr} = \begin{cases} 0 & Next(i) \neq j \\ 1 & Next(i) = j \end{cases}$$

after r number of re-routings.

Elasticity :

$$\Omega = \Delta D / \Re$$

where $\Delta D = D_R - D_0$, and R is the number of re-routes done in the entire assignment. In essence, the smaller the Ω, the less affected is the overall solution under same network resistance.

The communication capability of the system can be demonstrated by the number of requests handled by the RVRS per second. This request rate, denoted by ω, gives an idea of how much concurrency and workload the RVRS can process. The requests include the submissions of new

	$\Re$	σ_R	ΔD	$\sigma_{\Delta D}$	Ω	Request rate (ω)
C101.txt	567.20	238.09	246.00	87.80	0.43	27 (req/s)
C201.txt	1289.20	557.12	195.60	105.62	0.15	11 (req/s)
R101.txt	1261.60	945.97	103.00	24.95	0.08	50 (req/s)
R201.txt	1947.60	1097.63	82.20	62.94	0.04	15 (req/s)
RC101.txt	244.00	123.93	42.00	57.34	0.17	40 (req/s)
RC201.txt	452.40	274.90	-120.20	75.15	-0.27	14 (req/s)

Table 2: Experiment results on several Solomon problems

reactors as well as polling for answers from all agents. Also, our experiment shows that the average turn around time for a reactor request, T_{ta}, is less than a second, where T_{ta} is defined as the time between an agent sends a reactor and it receives the answer by polling.

Table 2 illustrates the experiment results, which are the average values taken from 5 runs. It shows the elasticity of solver and maximal request rates on the registry on several Solomon benchmark problems. For problem sets C and R, $2 \leq \tau \leq 20$ and $\varphi = 0.1$, while the values are $2 \leq \tau \leq 10$ and $\varphi = 0.05$ for problem sets RC. The results shows that additional distance is always less than network resistance. Throughout the experiments for a same problem, although the network resistance varies, it is observed that the elasticity does not vary significantly. This exhibits the consistent performance of the solver, despite the randomness in traffic condition. Although the increase in the cost is likely to increase the travel distance, this is not always the case. In fact, there are instances where the total distance has been reduced after a series of re-routing, at the expense of more vehicles. We also compare the value of standard deviation σ_R and $\sigma_{\Delta D}$ for each problem. And since $\sigma_{\Delta D} \ll \sigma_R$, we conclude that our simulation results are stable with regards to random environment changes.

5 Conclusion

We described a real time dynamic vehicle routing problem (RTDVRP). The problem is reactive and dynamic so that re-routing the whole problem in real time with traditional local search VRPTW methods is neither useful nor practical. Instead, we proposed a Reactive Vehicle Routing (RVR) framework for solving this type of problem. The framework consists of a number of vehicle agents which launched various reactors into a concurrent constraint architecture. The architecture combines a registry from the OCP concept and a solver using incremental local optimization, a quick, smart VRPTW solver. Reactivity and concurrency are the two most prominent features of the model. They offer the possibility for the agent-based constraint programming to solve for the complex real-time problems. Our implementation, RVRS, and a discretized simulation proved this model promising and it opens up a lot of opportunities for future research and applications.

We foresee that system like RVRS can be applied to real life generic goods delivery where the traffic condition and routing metrics change dynamically. One important application could be on-call taxi routing. Taxi routing problems combine high degree of reactivity and concurrency due to the large number of taxis in a modern city and the importance of customer satisfaction. There is virtually no static factor: customer loads, time windows and locations are all random. In this case, the system must be able to handle changing problem size (new customer), nodes' positions as well as side constraints (time windows and demands, etc.), and for that RVRS is a suitable solution.

References

[1] B. D. Backer, V. Furnon, P. Kilby, P. Prosser and P. Shaw. "Solving vehicle routing problems using constraint programming and metaheuristics", *Journal of Heuristics*, 1-16, 1997.

[2] G. Brown and G. Graves. "Real-time dispatch of petroleum tank trucks," *Management Science*, vol. 27, pp. 19-32, 1981.

[3] Y. Caseau. "Heuristics for Large Constrained Vehicle Routing Problems," *Journal of Heuristics*, 5, 281-3-3, 1999.

[4] M. Gendreau, A. Hertz, and G. Laporte. "A tabu search Heuristic for the vehicle routing problem", *Management Science*, vol. 40, no. 10, 1276-1290, 1994.

[5] M. Gendreau, F. Guertin, J.-Y. Potvin, and E. Taillard. "Parallel tabu search for real-time vehicle routing and dispatching," *Transportation Science*, vol. 33, no. 4, pp. 381-390, 1999.

[6] J. Jaffar and R. Yap. "Open Constraint Programming," Invited Paper, *4th Intl. Conf on Principles and Practice of Constraint Programming (CP)*, Pisa, Oct. 1998.

[7] J. Jaffar, L. S. Wong and R. Yap. "Open Constraint Programming," draft, http://www.comp.nus.edu.sg/˜joxan/papers/ocp.ps.

[8] N. R. Jennings and M. J. Wooldridge. "Applications of Intelligent Agents". *Agent Technology: Foundations, Applications and Markets*, p.3-28, 1998.

[9] Y. Li, J. M. Rousseau, and M. Gendreau. "Real-time dispatching of public transit operations with and without bus location information," *Computer-Aided Transit Scheduling. Proceedings*, pp. 296-308, 1995.

[10] E. Rochat and E. Taillard. "Probabilistic Diversification and Intensification in Local Search for vehicle Routing," *Journal of Heuristic*, 1, 147-167, 1995.

[11] P. Shaw. "Using constraint programming and local search methods to solve vehicle routing problem". *Principles and Practice of Constraint Programming - CP98, Proceedings of the 4th International Conference, Lecture Notes in Computer Science 1520*, 417-431, 1998.

[12] S. R. Thangiah. "Vehicle Routing with Time Windows using Genetic Algorithms", *Application Handbook of Genetic Algorithms: New Frontiers*, Volume II. Lance Chambers (Ed.), CRC Press,253-277, 1995.

[13] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice, *Knowledge Engineering Review*, 10(2), 1995.

[14] K. Zhu, K. C. Tan and L. H. Lee. "Heuristics for Vehicle Routing Problem with Time Windows," *6th AI and Math*, 2000.