Journal of Technical Science & Technologies; ISSN: 2298-0032; e-ISSN: 2346-8270; Volume 6, Issue 1, 2017

Application of Autonomic Component Ensembles Methods and Cloud Computing to MDVRPWTM Problem

Irakli RODONAIA*
Artioma MERABIANI**
Vakhtang RODONAIA***

Abstract

This study will examine an application of autonomic component ensembles methods and cloud computing to MDVRPWTM Problem to optimize vehicle routing in a non-stationary stochastic network. The goal of this article is to find a systematic approach to implement it to the transportation systems integrated with real-time information technology. Our adaptive algorithm based on the jDEECo concept will provide benefits to logistics and supply chains in Georgia. With studies based on a road network in Georgia, it is aimed to reduce vehicle usage while satisfying or improving service levels for just-in-time delivery.

Keywords: Jdeeco, Vehicle Routing Problem, Traffic Congestion Reduction, Optimization.

Introduction

In the previous paper (Rodonaia & Merabiani, 2016), the first stage of the solution of the MDVRPTW (Multi Depots Vehicle Routing Planning with Time Windows) problem was described and solved by implementing Adaptive Large Neighborhood Search (ALNS) framework to the initial solution. For further works in this area, additional constrains and limitations should be added to the solution of the approach such as traffic congestions and overloaded bottleneck segments on the road. Due to that kind of constrains and limitations as congestions, big delays and unnecessary expenditures occur, if not regulated and preplanned. So that algorithm of solving MDVRPWTM should be real-time adaptable and implementable for stochastic environment of real traffic flow. For such kind of algorithms, modification of the ALNS algorithm has been developed that considers probabilities of road segments (links) to be congested and plans the route to minimize cost of the travel. This modification is implemented in Jsprit framework and to provide on-line adaptability we used concept of autonomic components (AC) and autonomic component ensembles (ACE) (Prangishvili et al.,2014). ACs are dynamically organized into ACEs. AC members of an ACE are connected by the interdependency relations defined through predicates (used to specify the targets of communication actions.

The functional description of the AC and ACE is shown in Fig.1.

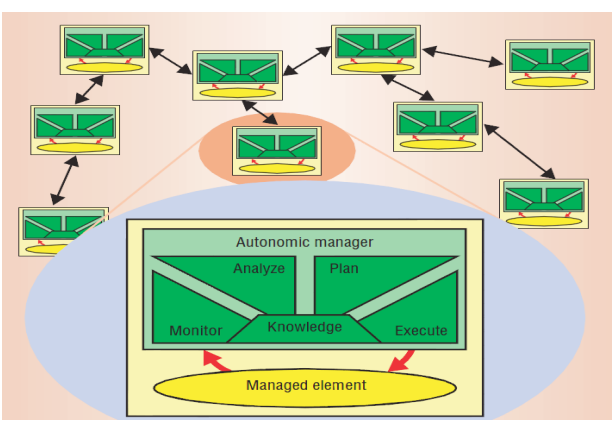

Figure 1. *Functional description of a component*

The latest studies in computation and mathematics, have shown that many large scale distributed systems are quite influential on our environment. To handle such large-scale problems and issues, engineers have to develop very

* Prof. Dr., Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia.
E-mail: irakli.rodonaia@ibsu.edu.ge
**MSc., Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia.
E-mail: merabianiartioma@ibsu.edu.ge
*** Assoc. Prof. Dr., Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia.
E-mail: vrodonaia@ibsu.edu.ge

complex systems, that can work dynamically and autonomic. In the presented work, VRP has been studied and to improve the algorithm new component was added, in particular, Ensemble- Based Component Systems – EBCS, which include fully autonomic components with recursive execution by means of the dynamic component ensembles that are controlling data exchange.

The process part of a component is split into an autonomic manager controlling execution of a managed element (Prangishvili et al., 2014). A component in DEECo comprises knowledge, exposed via a set of interfaces and processes. Knowledge reflects the state and available functionality of the component. It is organized as a hierarchical data structure, which maps knowledge identifiers to values. Specifically, values may be either (potentially structured) data or executable functions. The autonomic manager continuously checks the condition of the component, as well as the execution context and identifies relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. It also plans adaptations in order to meet the new functional or non-functional requirements, executes them and monitors that its goals are achieved, possibly without any interruption. A managed element can be seen as an empty “executor" which retrieves from the knowledge repository the process implementing a required functionality and bounds it to a process variable, the retrieved process for execution and waits until it terminates.

The ACs in an ACE may be implemented as virtual machines (VMs) in datacenters (ACE). Each AC is associated with the concrete vehicle and comprehensive information of the current location of the vehicle on the route, relevant data on its current state and etc. In the presented approach, the knowledge repository is used to store these data and exchange them with other ACs (Prangishvili et al., 2017). Occasionally, so called spatial-temporal event (that is, a vehicle arrives to a certain service point at a certain time) occurs. The equipment in the car (GPS receivers and GSM telephones (or some similar wireless communications technology) determines location using the GPS receiver and sends the coordinates and other relevant data to the Web server. The general infrastructure of our approach id shown in the Fig.2.

The base virtual machine VM0 ho`sts all main structural components of proposed system: JSpirit, MatSim, travel and congestion management database (TCMD), database of simulation results (SRD), web servers for connection with vehicles, GPS, and etc. Although VM0 is permanently used and maintained, it is convenient to represent it as a virtual machine because it will intensively interact and exchange data with other virtual machines, each of which represents autonomic component (AC). Autonomic components are associated with concrete vehicles and constitute an Autonomic Component Ensemble (ACE). The base VM0 executes the initial solution of MDVRPTW problem and generates the initial set of routes RI. The input parameters, such as time windows for each service points, are held at VM0 as well. After generating the initial set of routes, new virtual machines, enumerated from 1 to nr (where nr is the amount of routes in the initial set RI), are created. The recourses of the datacenter’s servers are dynamically allocated to virtual machines.

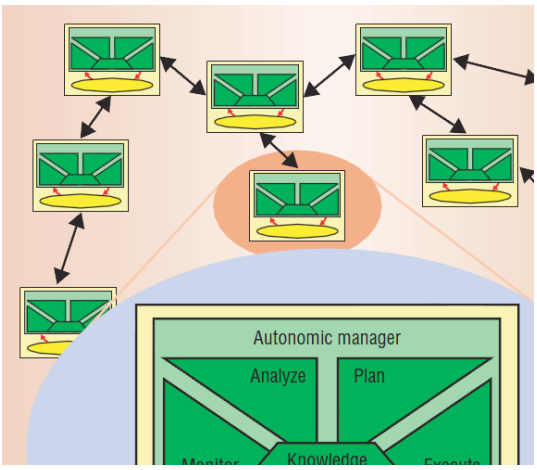

Figure 2. *General infrastructure*

DEECo is built on top of two first-class concepts: component and ensemble (Bures et al., 2014). A component is an independent and self-sustained unit of development, deployment and computation. An ensemble acts as a dynamic binding mechanism, which links a set of components together and manages their interaction. The fundamental idea in DEECo is that the how the components bind and communicate with one another is only through ensembles. The two main DEECo concepts are in detail elaborated below. An integral part of the component model is also the runtime framework providing the necessary management services for both components and ensembles.

A component in DEECo comprises knowledge, exposed via a set of interfaces and processes (Bures et al., 2014). Knowledge reflects the state and available functionality of the component (lines 6-19). It is organized as a hierarchical data structure, which maps knowledge identifiers to values. Specifically, values may be either potentially structured data or executable functions. In this context, the term belief refers to the part of a component’s knowledge that represents a copy of knowledge of another component, and is thus treated with a certain level of uncertainty as it might become obsolete or invalid.

A component’s knowledge (Bures et al., 2014) is exposed to the other components and environment via a set of interfaces. An interface thus represents a partial view on the component’s knowledge shown in Fig 3. Specifically, interfaces of a single component can overlap and multiple components can provide the same interface, thus allowing for polymorphism of components.

Component processes are essentially soft real-time tasks that manipulate the knowledge of the component. A process is characterized as a function associated with a list of input and output knowledge fields. Operation of the process is managed by the runtime framework and consists of atomically retrieving all input knowledge fields, computing the process function and atomically writing all output knowledge fields.

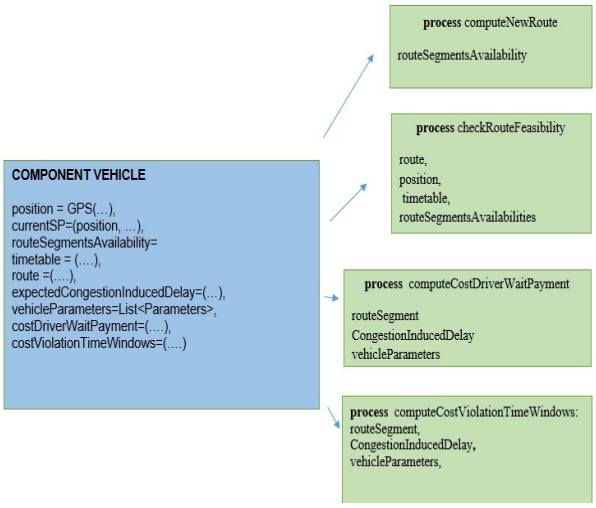

Figure 3. *Component vehicle*

Referring to the MDVRPTW running example, the components (each occurring in multiple instances) are the Vehicle and the Route Segments Congestion. A Vehicle maintains a belief over the availability of the relevant Route Segments Congestion (route Segments Availability).

It uses an Adaptive Large Neighborhood Search (ALNS) library to (re-) compute its route according to the availability belief and its timetable every time the availability belief or route feasibility changes. The Vehicle also checks if its route remains feasible, with respect to the corresponding route Segments Availabilities and its route's on Schedule property current position. A Route Segments Congestion just keeps track of its available route's segment availability and computes the expected Congestion Induced Delay time.

jDEECo Run-time Realization of MDVRPTW Problem

By building on Java annotations, the mapping of DEECo concepts relies on standard Java language primitives and does not require any language extensions or external tools (Prangishvili et al., 2017).

An example of a component definition has the form of a Java class:

```
1.  @DEECoComponent
2.  public class Vehicle extends
      ComponentKnowledge {
3.  public Position position;
4.  public ServicePoint currentSP
5.  public List< TimeWindowsForSPs > timetable;
6.  public Map<ID, segmentsStatus >
      routeSegmentsAvailability
7.  public Route  route;
8.  public Delay expectedCongestionInducedDelay;
9.  public List <vehicleParameters>
      vehicleParameters
10. public   Cost  costDriverWaitPayment,
11. public    Cost  costViolationTimeWindows
12. public Vehicle() {
13. // initialize the initial knowledge structure reflected
      by the class fields
14. }
15. @DEECoProcess
16. public static void  computeNewRoute(
17. @DEECoIn("routeSegmentsAvailability")
      @DEECoTriggered Map<…>
18. routeSegmentsAvailability
19. @DEECoIn("timetable")
      List< TimeWindowsForSPs > timetable,
20. @DEECoInOut("route") Route   route
21. ) {
22. // re--compute the vehicle's route  if it's infeasible
23. }
24. @DEECoProcess
25. @DEECoPeriodScheduling(2000))
26. public static void checkRouteFeasibility (
27. @DEECoIn("route") Route route,
28. @DEECoIn("timetablel") List< TimeWindowsFor
      SPs >   timetable,
29. @DEECoIn("position") Position position,
30. @DEECoOut("route.isFeasible")
      OutWrapper<Boolean> isRouteFeasible
31. ){
32. // determine feasibility of the route
33. }
34. ...
35. }
```

A component definition has the form of a Java class (see the above code). Such a class is marked by the @ DEECoComponent annotation and extends the Component Knowledge class. The initial knowledge structure of the component is captured by means of the public, non-static fields of the class (lines 3-11). At runtime, this initial knowledge structure is initialized either via static initializers or via the constructor of the class (lines 12-14). The component processes are defined as public static methods of the class, annotated with @DEECoProcess (e.g. lines 15-23).

The input and output knowledge of the process is represented by the methods' parameters.

The parameters are marked with one of the annotations @DEECoIn, @DEECoOut or @DEECoInOut, in order to distinguish between input and output knowledge fields of the process (e.g. lines 17-20). Each annotation also includes an identifier of the knowledge field that the associated method parameter represents. When a non-structured knowledge field constitutes an in/out knowledge of a process, the asso-

ciated method parameter is for technical reasons (related to Java immutable types) passed inside an OutWrapper object (e.g. line 30). Periodic scheduling of a process is defined via the @DEECo Periodic Scheduling annotation of the process's method, which takes the period expressed in milliseconds in its parameter (line 25). Triggered scheduling is defined via @DEECo Triggered annotation of the method's parameter, change of which should trigger the execution of the process (lines 17-19).

Below the example of an ensemble definition Java (jDEECO) is given:

```
1.  @DEECoEnsemble
2.  @DEECoPeriodicScheduling(2000)
3.  public class UpdateRouteSegmentAvailability
    Information extends Ensemble {
4.
5.  @DEECoEnsembleMembership
6.  public static Boolean membership (
7.  @DEECoIn
    ("coordinator.routeSegmentsAvailability ")
    List< segmentsStatus>,
8.  @DEECoIn ("member.routeSegmentsAvailability ")
    SegmetStatus,
9.  @DEECoIn("member.
    expectedCongestionInducedDelay ") Delay
10. ) {
11. for (RouteSegment rs : segmentRoute)) {
12. if (isAvailable(rs.routeSegmentsAvailability)
    ==TRUE
13. return true;
14. }
15. return false;
16. }
17.
18.
19. @DEECoEnsembleKnowledgeExchange
20. @DEECoPeriodScheduling(2000))
21. public static void knowledgeExchange (
22. @DEECoOut("coordinator. routeSegments
    Availability ") Map <…> SegmentStatus,
23. @DEECoOut("coordinator.
    expectedCongestionInducedDelay ")  Delay,
24. @DEECoIn("member. routeSegments
    Availability]") Map <…> SegmentStatus,
25. @DEECoIn("member.
    expectedCongestionInducedDelay "") Delay,
26. )
27. }
```

The ensemble definition takes also the form of a Java class (Bures et al., 2014). In particular, the class is marked with the @DEECo Ensemble annotation and extends the Ensemble class (see the above example). Both the membership predicate and the knowledge exchange are defined as specifically-annotated static methods of this class. While the method representing the membership predicate is annotated by @DEECo Ensemble Membership (line 5), the methodrepresenting knowledge exchange is annotated by @DEECo Ensemble Knowledge Exchange (line 19).

The jDEECo runtime framework is primarily responsible for scheduling component processes, forming ensembles and performing knowledge exchange. It also allows for distribution of components (Bures et al., 2014).

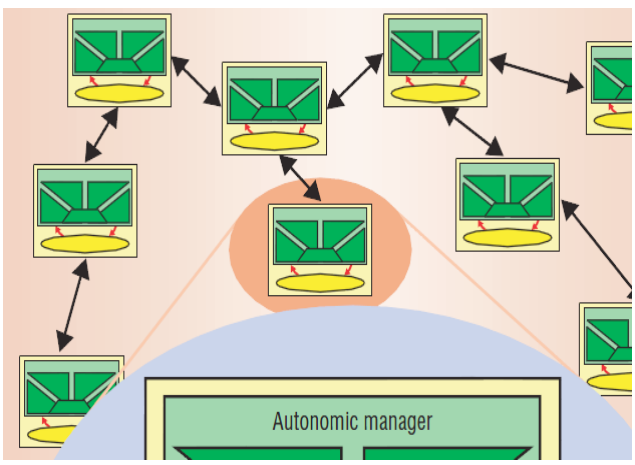

Figure 4. *jDEECo runtime framework architecture. (Bures et al., 2014)*

As illustrated in Fig.4, it is internally composed of the management part and the knowledge repository. The management part is further composed of two modules. One is responsible for scheduling and execution of component processes and knowledge exchange of ensembles. The other is responsible for managing access to the knowledge repository. Exploiting the fact that all modules of the runtime framework implementation are loosely coupled, we are able to introduce implementation variants for each of them. As a result, different variants can be selected in order to reflect specific requirements imposed to the platform (Bures et al., 2014).

The role of the knowledge repository is to store the component's knowledge (e.g. CK1 – knowledge of component C1 – in Fig 4). Its responsibility is also to provide component processes and knowledge exchange of ensembles with access to this knowledge. In fact, we provide a local and a distributed implementation of the knowledge repository; the former is employed for simulation and verification of the code while the latter is used in case the runtime framework needs to run in a distributed setting (i.e., the distribution is achieved at the level of knowledge repository). Specifically, the distributed implementation of the knowledge repository

allows each component to run in a different Java virtual machine as illustrated in Fig.4.

The approach described above was implemented by using cloud computing service provider Google Cloud Platform. Cloud computing is an information technology concept that implies the provision of ubiquitous and convenient network access on demand to a common pool of configurable computing resources (for example, data networks, servers, storage devices, applications and services - as together and separately), which can be promptly provided and released with minimal operating costs or calls to the provider. Namely, IaaS (Infrastructure-as-a-Service) was used for creation and deployment Virtual Machines (VM), associated with the vehicles (totally 17 VMs) and the VM, associated with the base Virtual Machine (VM0) (Rodonaia & Merabiani et al., 2016). The VM0 hosts all main structural components of proposed system: JSpirit, MatSim, ALNS, travel and congestion management database (TCMD), database of simulation results (SRD), web servers for connection with vehicles, GPS, etc. VMs, associated with vehicles, run local reduced copies of ALNS algorithm, and local copy of TCMD and SRD databases (Rodonaia & Merabiani, 2016). Payments Pay-as-you-go for consumed resources of the Google Cloud Platform datacenter are on average 60% less for many compute workloads than other clouds. Implementation of Autonomic Components Ensembles (ACE) on Google Cloud Platform (and, in general, on other cloud providers platforms) shifts most of the costs from capital expenditures (or buying and installing servers, storage, networking, and related infrastructure) to an operating expenses model, where customers pay only for usage of these types of resources.

Conclusion

In our paper, we described an adaptive approach of the algorithm to solve MDVRPTW problem. The algorithm is aimed to account for realistic real-world situations, such as presence of various congestion types. The congestions are the most important critical constrains of the MDVRPTW problem. Since the realistic estimation of congestion duration is rather difficult and non-standard problem, we use the MatSim large-scale agent-based simulation tool which allows users to compose and run complex simulation models that are extremely close to the real-world situations. Also cloud technologies are used for decreasing total cost of calculations of the adaptive algorithm. In our case, this approach allows to calculate various input parameters, obtain process simulation outputs and a great lot of parameters and distributions functions of simulated processes. Another feature of our approach is implementation of the autonomic components ensembles model to VRP. We consider each vehicle to be associated with the corresponding autonomic component AC to share online information with other vehicles. This allows a vehicle to notify other vehicles about expected and actual congestion. The approach described above was implemented by using cloud computing service provider Google Cloud Platform. Besides, ACs can reroute vehicles to find the satisfying alternative routes that aid vehicles to not violate time windows requirements and, at the same time, avoid the congested roads. It is necessary to point out that the algorithm of adaptation is able to reschedule and find alternative routed for several vehicles in parallel. The jDEECo concept increases the performance of proposed approach.

References

Rodonaia, I., & Merabiani, A. (2016). Real-world applications of the vehicle routing problem in Georgia. *Journal Technical Science & Technologies (JTST),* is. (2) 41 -44 November.

Prangishvili, A., Shonia, O., Rodonaia, I., & Mousa, M. (2014). Formal verification in autonomic-component ensembles, WSEAS / NAUN International Conferences, Salerno, Italy.

Prangishvili, A., Shonia, O., Rodonaia, I., & Merabiani. A. (2017). Adaptive Real-World Algorithm of Solving MDVRPTW (Multi Depots Vehicle Routing Planning with Time Windows) Problem. *International Journal of Transportation Systems,* 2, 1-6.

Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznik, J., Kit, M., & Plasil, F. (2014). DEECo – an Ensemble-Based Component System Technical report No. D3S-TR-2013-02, Version 1.0,.