Dynamic Timetable Information in Smart Cities

Kalliopi Giannakopoulou[1,2], Sotiris Nikoletseas[1,2], Andreas Paraskevopoulos[2] and Christos Zaroliagis[1,2]
[1] Computer Technology Institute and Press "Diophantus"
[2] Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece
{gianakok,nikole,paraskevop,zaro}@ceid.upatras.gr

***Abstract*—We provide a cloud-based journey planner for public transport built upon an efficient core routing engine that updates efficiently timetable information in case of delays. We describe our mobile application along with a service that allows users to assess the suggested journeys offered by the application, built on top of an IoT/FIRE+ infrastructure. Our journey planner contributes to the establishment of a live community of travelers, equipped with an arsenal of inter-operable personalized renewable mobility services, for modern mobility in smart cities.**

I. Introduction

Mobile and web applications (aka *journey planners*) that compute *best journeys* in public transportation systems are abundant. Given as input a timetable associated with a public transportation system, the *journey planning* problem asks for efficiently answering queries of the form: "What is the best journey from some station A to some other station B, provided that I wish to depart at time t?".

Depending on the considered metrics and modeling assumptions, the journey planning problem can be specialized into various optimization problems. In the *earliest arrival-time problem* (EAP), we are interested in finding the best (or optimal) journey that minimizes the traveling time required to complete it. In the *minimum number of transfers problem* (MNTP), we are interested in computing a best journey that minimizes the number of times a passenger has to move from one vehicle to another, during the journey. Sometimes, these two optimization criteria are considered in combination. We refer to the recent survey in [1] for a comprehensive overview on journey planning. A typical approach to deal with the various journey planning optimization problems is to create, in a preprocessing phase, a structure that represents the timetable information, which subsequently allows for fast answering of queries. There is vast literature on this approach; see e.g., [1] and the references therein.

The journey planning problem is quite challenging (despite its simple formulation), much more than its route planning counterpart in road networks. Schedule-based transportation systems exhibit an inherent time-dependent component that requires more complex modeling assumptions in order to obtain meaningful results, especially when transfer times from one vehicle to another has to be taken into account [8].

One additional challenge is to accommodate delays of public transport vehicles that often occur. The key issue is how to efficiently update the underlying timetable information system so that best journey (typically earliest arrival) queries are still answered fast and optimally with respect to the updated timetable. Given the fact that a journey planner is usually in heavy demand (e.g., the journey planner of the German railways in peak hours receives more than 400 queries per second), and that delays occur frequently, in order to support such a computationally demanding service efficiently one needs a systems architecture of sufficient computational capacity. In this respect, the elasticity of a cloud architecture allows for the adaptation of the reserved computing resources to the actual demands for processing a vast number of queries and for updating the timetable information.

A very recent effort [2] introduces the so-called *dynamic timetable model* (DTM) that constitutes a (graph-based) space-efficient model to represent timetable information. DTM also possesses an efficient procedure for answering earlier arrival queries as well as the currently fastest procedure for updating the timetable information when delays occur. Preliminary experimentation [2] on real-world data sets exhibited an excellent practical behaviour of DTM.

The aforementioned challenges for real-time response and efficient digestion of delays in a timetable information system, necessitate the adoption of more sophisticated architectures. The heart of such a sophisticated journey planning service would have to lie on a cloud architecture, which would be able to guarantee data persistence, interoperability with other sources (that report vehicle delays), real-time elasticity of computing resources, and also transparent accessibility of the journey planner by the travelers. In such an environment, the queries are sent to a routing engine residing at the cloud infrastructure, which in turn sends back the answers taking into account the updated timetable information.

In this work, we describe a cloud-based journey planner whose core routing engine is based on DTM. We provide the algorithmic details of this efficient journey planner that digests extremely fast timetable updates in order to respond in real-time to arbitrary journey planning queries. We also present our mobile client application along with a service that allows users to assess the suggested journeys offered by the application. The user assessment service has been built on top of an IoT/FIRE+ infrastructure [7]. Our journey planner contributes to the establishment of a live community of travelers, equipped with an arsenal of inter-operable personalized renewable mobility services, for modern mobility in smart cities in the context of [6].

II. Preliminaries

In this section, we provide the necessary definitions and notation that will be used throughout the paper adopted from [2]. A *timetable* $\mathscr{T}$ is defined by a triple $\mathscr{T} = (\mathscr{Z}, \mathscr{B}, \mathscr{C})$, where $\mathscr{Z}$ is a set of *vehicles*, $\mathscr{B}$ is a set of *stations* (or *stops*), and $\mathscr{C}$ is a set of *elementary connections* whose elements are 5-tuples of the form $c = (Z, S_d, S_a, t_d, t_a)$. Such a tuple is interpreted as vehicle $Z \in \mathscr{Z}$ leaves station $S_d \in \mathscr{B}$ at time t_d, and the immediately next stop of vehicle Z is station $S_a \in \mathscr{B}$ at time t_a. If x denotes a tuple's field, then the notation $x(c)$ specifies the value of x in the elementary connection c (e.g., $t_d(c)$ denotes the departure time in c). The departure and arrival times $t_d(c)$ and $t_a(c)$ of an elementary connection c within a day are integers in the interval $\{0, 1, \ldots, 1439\}$ representing time in minutes after midnight.

Given two time instances t_1, t_2 ($t_2 \geq t_1$), let $\Delta(t_1, t_2) = t_2 - t_1 (\bmod\ 1440)$. The *length* of an elementary connection c, denoted by $\Delta(c)$, is the time that passes between the departure and the arrival times of c assuming that c lasts for less than 24 hours, i.e., $\Delta(c) = \Delta(t_d(c), t_a(c))$.

Given an elementary connection c_1 arriving at station S and an elementary connection c_2 departing from the same station S, if $Z(c_1) \neq Z(c_2)$, it follows that it is possible to transfer from $Z(c_1)$ to $Z(c_2)$ only if the time between the arrival and the departure at station S is larger than or equal to a given *minimum transfer time*, denoted by $transfer(S)$. We assume that $transfer(S)$ is always smaller than 1440, for each $S \in \mathscr{B}$. An *itinerary* (a.k.a. a *journey*) in a timetable $\mathscr{T}$ is a sequence of elementary connections $P = (c_1, c_2, \ldots, c_k)$ such that, for each $i = 2, 3, \ldots, k$, $S_a(c_{i-1}) = S_d(c_i)$, and either $\Delta(t_a(c_{i-1}), t_d(c_i)) \geq 0$, if $Z(c_{i-1}) = Z(c_i)$; or $\Delta(t_a(c_{i-1}), t_d(c_i)) \geq transfer(S_a(c_{i-1}))$, otherwise.

We say that the itinerary starts from station $S_d(c_1)$ at time $t_d(c_1)$ and arrives at station $S_a(c_k)$ at time $t_a(c_k)$. The *length* $\Delta(P)$ of an itinerary P is given by the sum of the lengths of its elementary connections plus the associated transfer times, i.e., $\Delta(P) = \sum_{i=1}^{k} \Delta(c_i) + \Delta(t_a(c_i), t_d(c_i + 1))$.

A *timetable query* is defined by a triple (S, T, t_S) where $S \in \mathscr{B}$ is a departure station, $T \in \mathscr{B}$ is an arrival station and t_S is a minimum departure time. There are two natural criteria used to answer a query that lead to the following optimization problems [8].

- The *Earliest Arrival Problem (EAP)* is the problem of finding an itinerary from S to T which starts at a time after t_S and arrives at T as early as possible.
- The *Minimum Number of Transfers Problem (MNTP)* is the problem of finding an itinerary from S to T which starts at a time after t_S and has as few transfers from a vehicle to another one as possible.

Given a timetable $\mathscr{T}$, a delay occurring on a connection c is modelled as an increase of d minutes on the arrival time, $t'_a(c) = t_a(c) + d (\bmod\ 1440)$. The timetable is then updated according to some specific policy which depends on the network infrastructure. The obtained timetable is called *disposition timetable* $\mathscr{T}'$ and it differs from $\mathscr{T}$ for the arrival and departure times of the vehicles that depend on $Z(c)$ in $\mathscr{T}$. We assume that the policy adopted is that no vehicle waits for a delayed one. Therefore, when a delay occurs on a connection c, the only time references which are updated are those regarding the departure times of $Z(c)$. Moreover, we assume that the policy does not take into account any possible slack times and hence the time references are updated by adding $d (\bmod\ 1440)$.

III. The Algorithmic Core Engine

In this section we present the algorithmic core engine of our journey planner for solving EAP and MNTP, by following the exposition in [2]. It is based on the dynamic timetable model (DTM) and its query and update algorithms. We also provide the engineering details that speed-up our algorithmic engine.

A. DTM along with its Query and Update Mechanisms

Given a timetable $\mathscr{T} = (\mathscr{Z}, \mathscr{B}, \mathscr{C})$, we define a directed graph $G = (V, E)$ called *dynamic timetable graph* and an associated weight function $w : E \to \mathbb{N}$ as follows:

- For each station S in $\mathscr{B}$, a node s_S, called *switch node* of S, is added to V;
- For each elementary connection $c = (Z, S_d, S_a, t_d, t_a) \in \mathscr{C}$ a node d_c, called *departure node* of c, is added to V and an arc (d_c, s_{S_a}), called *connection arc* of c, connecting d_c to the switch node s_{S_a} of S_a, is added to E;
- For each elementary connection $c = (Z, S_d, S_a, t_d, t_a) \in \mathscr{C}$ an arc (s_{S_d}, d_c), called *switch arc*, connecting the switch node s_{S_d} of the departure station S_d to the departure node d_c of c, is added to E;
- For each vehicle $Z \in \mathscr{Z}$ which travels through the itinerary $(c_1, c_2, \ldots, c_k)$, an arc, called *vehicle arc*, connecting the departure node d_{c_i} of c_i with the departure node $d_{c_{i+1}}$ of c_{i+1} is added to E, for each $i = 1, 2, \ldots, k-1$;
- The weight of each connection arc (d_c, s_{S_a}) is set to $w(d_c, s_{S_a}) = \Delta(t_a(c), t_d(c))$;
- The weight of each vehicle arc $(d_{c_i}, d_{c_{i+1}})$ is set to $w(d_{c_i}, d_{c_{i+1}}) = \Delta(t_d(c_i), t_d(c_{i+1}))$;
- The weight of each switch arc is initially set to ∞.

For each switch node s_S, we store the station S it is associated with while, for each departure node d_c, we maintain both the departure time reference $t_d(c)$ and the vehicle $Z(c)$ of connection c which d_c is associated with. Figure 1 shows an example of a dynamic timetable graph.

1) The Query Algorithm: The query algorithm (`DTM-Q`) for solving EAP works as follows (the algorithm can be easily adapted to solve MNTP queries, as we explain later). Given a *dynamic timetable graph* G, an EAP query (S, T, t_S) can be answered by executing a modified Dijkstra's algorithm on G, starting from the switch node s_S of S. Before discussing the details of the algorithm, we first describe the additional data structures used by `DTM-Q` with respect to the classic Dijkstra's algorithm. In particular, for each switch node s_A of a station A, algorithm `DTM-Q` stores: (i) A vector of boolean flags D_A, whose size is given by the number of stations A' such that

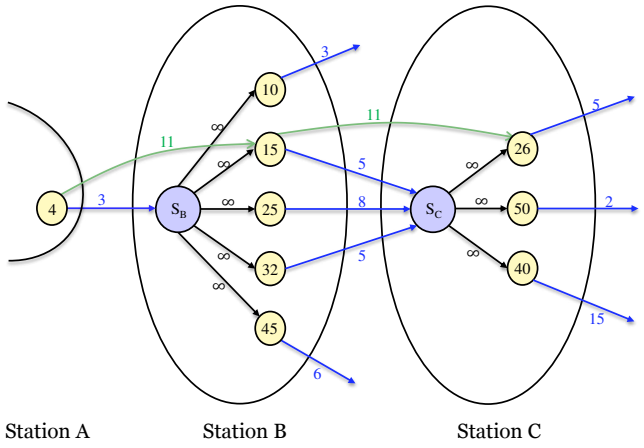

Fig. 1. An example of a dynamic timetable graph. Switch nodes are drawn in blue while departure nodes, ordered by arrival time, are drawn in yellow. Inside each departure node the departure time of the corresponding elementary connection is reported. Connection arcs are drawn in blue, switch arcs are drawn in black, while vehicle arcs are drawn in green.

there exists an elementary connection departing from A and arriving at A'. We denote the element of D_A associated to A' as $D_A[A']$. Initially, all flags of D_A are set to false. (ii) Along with the distance from s_S to s_A in the priority queue (PQ) used by Dijkstra's algorithm, the associated connection c' (is stored) such that the arc $(d_{c'}, s_A)$ is the one through which the particular distance has been obtained during a relaxation step. For non-switch nodes such an associated connection is assumed to be equal to a default value NIL.

Given the EAP query (S, T, t_S), the DTM-Q algorithm works as follows. First, it starts by inserting the switch node s_S of the departure station S in PQ. The distance and connection associated with node s_S in PQ are set to 0 and NIL, respectively. Then, the visit proceeds by extracting nodes, one by one, from PQ in a Dijkstra-like fashion, while PQ is not empty. When a node is extracted, the behavior of the algorithm depends on the type of the extracted node.

Departure nodes are processed like in the classic Dijkstra's algorithm: (i) outgoing arcs are scanned and relaxed in order to discover shorter paths (if any); (ii) associated neighboring (either departure or switch) nodes are, according to the weights of the arcs, either inserted in PQ, if they were not present, or their distance is decreased, if possible.

Switch nodes are processed as follows. Let us consider the time when a switch node s_A, associated with a station $A \in \mathscr{B}$, is extracted from the priority queue. Let $dist(s_S, s_A)$ be the distance from s_S to s_A associated to s_A in the queue and let c' be the associated elementary connection through which $dist(s_S, s_A)$ has been obtained. The value of $dist(s_S, s_A)$ essentially corresponds to the minimum time required to reach station A from station S, departing at time t_S. Hence, the algorithm first computes the value $x = t_d(c') + w(d_{c'}, s_A) \pmod{1440}$, which represents the arrival time of connection c'. If c' is equal to NIL, then the algorithm assigns $x = t_S$. Then, for each outgoing switch arc (s_A, d_c) such that $S_d(c) = A$ and $t_d(c) \geq x$, the algorithm performs a so-called *enabling* phase, that is, it *enables* arc (s_A, d_c) if $D_S[S_a(c)] = \text{false}$ and $\Delta(x, t_d(c)) =$

$$t_d(c) - x \pmod{1440} \geq \begin{cases} 0, & \text{if } Z(c) = Z(c') \\ transfer(A), & \text{otherwise} \end{cases} \quad (1)$$

The enabling operation on arc (s_A, d_c) consists in: (i) setting $w(s_A, d_c) = \Delta(x, t_d(c))$; (ii) either inserting d_c in PQ, if it was not present, or in decreasing its distance, if possible.

This enabling phase clearly stops (and hence the Dijkstra's search is pruned) when the vector D_A contains only true values, which are set in the following case.

Let (s_A, d_c) be the switch arc with the smallest arrival time that is enabled for some station $S' = S_a(c)$. Then, the algorithm sets $D_A[S']$ to *true* when an arc $(s_A, d_{c'})$ such that $S_a(c') = S'$ and $\Delta(t_a(c), t_a(c')) \pmod{1440} > transfer(S')$ is scanned (the time instances $t_a(c)$ and $t_a(c')$ required to check such a condition can be computed by using x, $t_d(c)$, $t_d(c')$ and the associated arc weights). Since departure nodes are ordered by arrival time, this guarantees that the algorithm ignores any connection (in the form of departure nodes) leaving A towards S' after this step, as it cannot lead to a better solution to the considered EAP query.

In other words, if two switch arcs (s_A, d_{c_1}) and (s_A, d_{c_2}), corresponding to two elementary connections c_1 and c_2, lead to the same station B, fulfill Inequality (1), and have two arrival times that differ by a value greater than $transfer(B)$, then only the one with smallest arrival time is considered (i.e., enabled) while the other one is essentially discarded, since it will not lead to a better solution to reach B. This is obtained by suitably setting the weights and the values of D_A. In fact, if we assume that $x \leq \min\{t_d(c_1), t_d(c_2)\}$ and $t_a(c_1) < t_a(c_2) + transfer(B) \pmod{1440}$, i.e., that $t_a(c_2)$ is the smallest arrival time that fulfills the above condition, then the value of $D_A[B]$ is set to *true* when arc (s_A, d_{c_2}) is scanned, and weights are set to $w(s_A, d_{c_1}) = t_d(c_1) - x$ and $w(s_A, d_{c_2}) = \infty$. Hence, d_{c_2} will never be part of a solution to the associated EAP query. Note that ties can be broken arbitrarily. The overall DTM-Q search is stopped as soon as the switch node s_T, associated to the arrival station T, is extracted from the queue. The arrival time t_T, which is given by $dist(s_S, s_T) + t_S$, is accordingly returned.

Algorithm DTM-Q can be adapted to answer a MNTP query (S, T, t_S), by slightly modifying it as follows. First, vector D is not needed. Second, when a switch node s_A is extracted from the priority queue with associated connection c', then all the switch arcs outgoing it that satisfy transfer time constraints (Inequality (1)) are enabled, and the weight of each switch arc (s_A, d_c) is set to 0, if $Z(c) = Z(c')$, and to 1 otherwise.

2) The Update Algorithm: We now turn to the update algorithm (DTM-U) that handles delays. Let us assume that we are given a timetable $\mathscr{T}$, a delay Δ occurring on a connection c of $\mathscr{T}$, and the corresponding *disposition timetable* $\mathscr{T}'$. If $\mathscr{T}$ is represented as a dynamic timetable graph G, then the DTM update algorithm computes the dynamic timetable graph G' corresponding to $\mathscr{T}'$ as follows. For the sake of clarity and

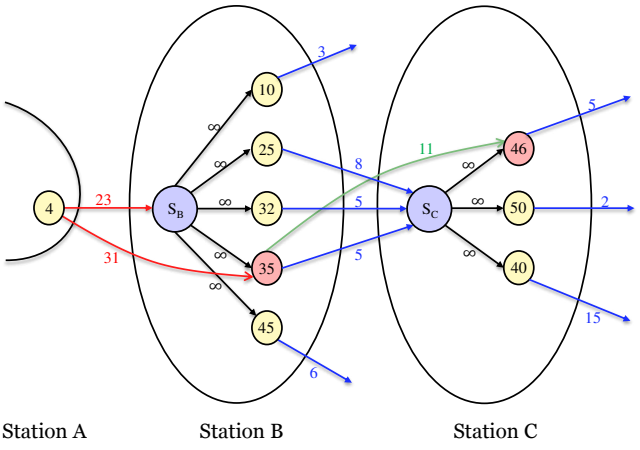

Fig. 2. Handling delays in the DTM model. A delay of 20 minutes induces two arc weight changes (red arcs) and the update of the time associated to the corresponding departure nodes (red nodes).

in order to better illustrate how DTM−U works, we make use of a running example based on the very same Station B of Figures 1 (before a delay) and 2 (after a delay). We assume that the vehicle's delay is $\Delta = 20$ mins.

First, the weight of both the connection arc $(d_c, s_{S'})$ and that of the vehicle arc $(d_c, d_{\hat{c}})$ is updated by adding the amount $\Delta(\bmod 1440)$, where $S' = S_a(c)$ and $d_{\hat{c}}$ is the departure node of S' such that $Z(c) = Z(\hat{c})$ (if any). An example of this behavior is shown in Figure 2, where the weight of connection arc connecting departure node 4 of Station A to switch node s_B of Station B, and the weight of the vehicle arc connecting the same departure node to the departure node 15 of Station B, are increased by $\Delta = 20$.

Second, for each connection c' that is *affected* by the delay occurring on c, i.e., such that the departure time in $\mathscr{T}$ differs from that in $\mathscr{T}'$, the time reference of the departure node $d_{c'}$ is updated by adding $\Delta(\bmod 1440)$. An example of this behavior is shown in Figure 2 where the departure nodes of Station B and C, originally having departure times 15 and 26 (see Figure 1), respectively, are updated to the delayed departure nodes of Station B and C, having departure times 35 and 46, respectively.

Note that the search for affected connections c' can be done by performing a graph visit of G, starting from the departure node d_c, and by selecting, during the visit, all connections c' such that $Z(c') = Z(c)$. Note also that, some further computation is required in the case G is used to answer EAP queries. In this case, the update of arc weights and time reference might break the ordering within the array representing the arcs. In order to restore this ordering, it is enough to compare the new values of arrival times of the few changed arcs and swap them, within the array, if needed. For instance, departure node 15 of Station B in Figure 1 is moved down by two positions in Figure 2 in order to restore the proper ordering with respect to departure time, while departure node 26 of Station C in Figure 1 keeps its position despite the increase.

B. Engineering the Core Algorithmic Services

To further boost performance, we have further engineered the DTM−Q algorithm along four axes, resulting in DTM−QH.

First, we apply to DTM−Q a modification of the general node blocking technique [3]. The goal is to prune the search by reducing the size of the Dijkstra's PQ. In particular, if nodes, belonging to a station that has already been reached (i.e., its switch node has already been settled by the visit), are touched (either visited for the first time or not), then they are immediately discarded and not inserted in PQ, thus resulting in reduced PQ overall size and number of PQ operations.

Second, we apply to DTM−Q a "runtime" modification of the omitting nodes technique proposed for reducing the size of time-expanded graph in [8]. In particular, we omit nodes with out-degree equal to one at runtime as follows: when the query routine touches a node x with out-degree equal to one, the relaxation depth is extended to two, node x is not inserted in PQ, and the target node of the single arc outgoing x is evaluated, to check whether it provides an improvement w.r.t. to the current shortest path search. This reduces the average PQ size and the number of PQ operations.

Third, we apply to DTM−Q a slightly different topological approach in order to reduce the unnecessary exploration and labeling on nodes and arcs during the query phase. In this variant, at each station, the switch arcs are grouped by adjacent arrival station and then for each group its arcs are sorted by arrival time. Therefore, when an adjacent arrival station S is reached, the whole group of the switch arcs heading to S can be skipped directly. Also in the procedure for handling delays this reduces the swap arc operations which are now limited only in the affected station's grouped arcs.

Fourth, we enhance DTM−Q with ALT [4], a simple and widely used goal-directed speed-up technique that pushes the shortest-path search faster towards the target node/station, and further boots query performance. The main idea of ALT is to assign *feasible potential* to the priority of each node in the queue. Given a subset of nodes $L \subseteq V$ called *landmarks*, the feasible potential of a node $u \in V$ towards a target t is defined as $\pi_t(u) = \max_{\ell \in L} \max\{d(u,\ell) - d(t,\ell); d(\ell,t) - d(\ell,u)\}$. By the triangle inequality, it follows that $\pi_t(u)$ is a lower bound to $d(u,t)$. It is easy to see that the tighter the lower bounds are, the more narrowed the search space is, and hence the faster the query algorithm performs. Choosing good landmarks that provide tight lower bounds is a fundamental part of the preprocessing phase of ALT. In our case, we select as landmarks the switch nodes, each of which represents the arrival node group of a station. Therefore the lower bound distance, $dist(s_A, s_B)$, between two switch nodes, s_A and s_B, denotes the minimum travel time among connections traveling from station A to station B. These lower bound distances can be computed during the preprocessing phase by running single-source queries from each switch node.

IV. The Journey Planner

The journey planner is a cloud-based application that has a server-side residing in the cloud and a client side. The

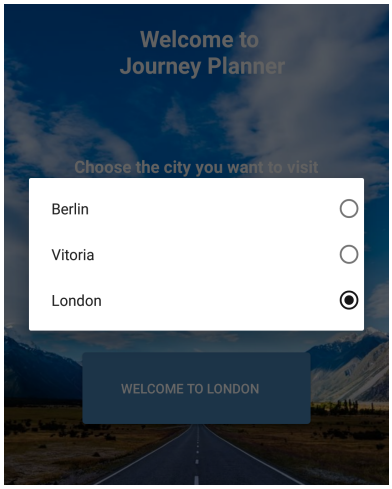

Fig. 3. City selection.

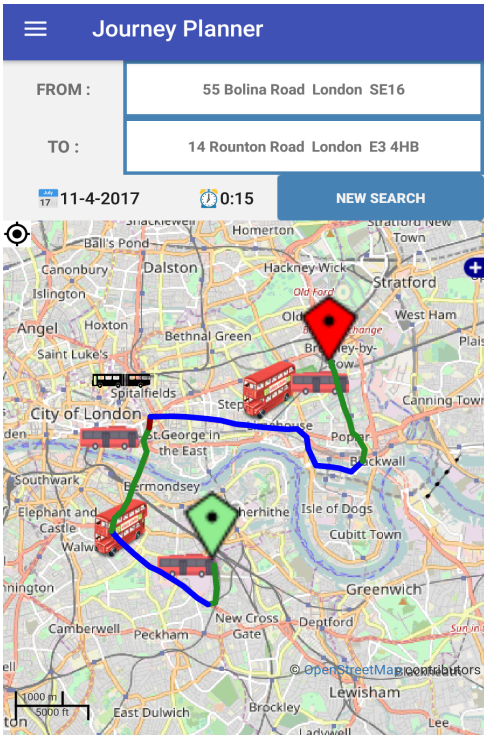

Fig. 4. Map and best journey query.

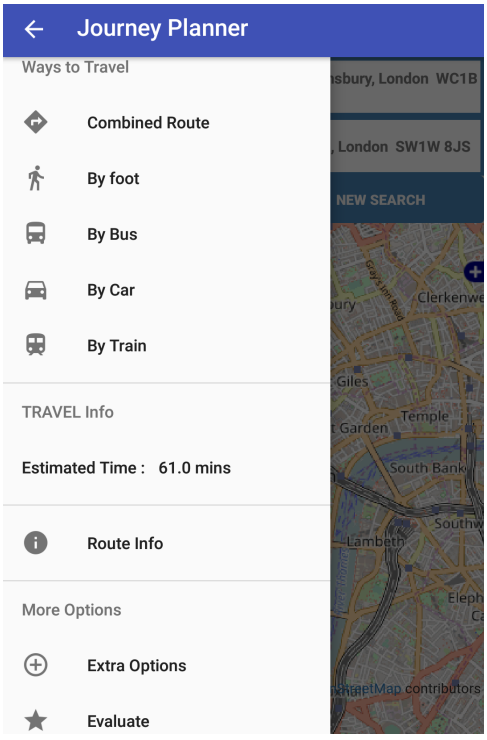

Fig. 5. Transport mode selection.

server-side is listening to user journey planning requests over TCP or HTTP/HTTPS communication modes. The input journey planning requests contain the user's selected origin and destination geographical points, departure time from the origin and transport modes. For each such request the service computes the requested best journey for the corresponding input query and sends it back to the client. The output of the cloud service is a sequence of points, along with info about the used transport modes and the departure/arrival times at each point of the journey, in JSON format. The service for the journey computation uses timetable data sets, in the General Transit Feed Specification (GTFS) format [11], containing stops, stations, various means of public transport (e.g., train, bus) connecting stations/stops, and departure/arrival times of the public transport means at each stations/stops. The timetable data are represented as a DTM graph.

Our mobile client application was developed for an Android environment. It implements the User Interface (UI) and the cloud-client communications from and to other services, including journey assessment.

The functionality of our mobile applications is shown in Figures 3, 4, and 5. Figure 3 shows the step of selecting a city in which a user may plan to travel. Figure 4 shows the result of an EAP query. The users can perform an EAP query by selecting an origin point, a destination point and a departure time from the origin. The computed optimal journey, which is sent by the cloud-service, is projected on the map. Figure 5 shows the transport mode selection option for the users on the EAP query.

We also conducted various studies for assessing the practical performance of our journey planner. First, it was used in a pilot application (as part of the cloud platform developed within [6]) in the city of Vitoria-Gasteiz (autumn 2016). The public transportation system of Vitoria-Gasteiz consisted of 355 stations and $129,050$ elementary connections, resulting in a DTM graph of 113K nodes and 335K arcs. Our journey planner achieved an average of 0.15ms for answering a query, and of 1.2μs for updating the timetable after a delay.

Second, we conducted an extensive experimental study with real-world timetables that were provided by [5], [9], [10].

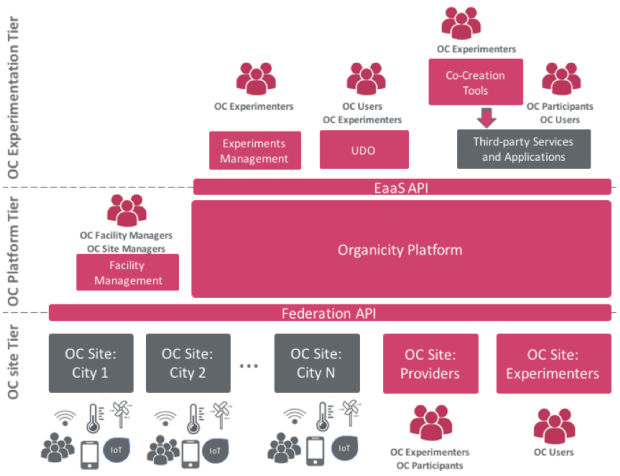

Fig. 6. Organicity platform high level architecture.

These data sets contained (among others) the metropolitan public transport networks of Athens, Berlin, London, and Madrid. The biggest in size was that of London, consisted of 20,843 stations and 14,064,967 connections, resulting in a DTM graph of 14,085,810 nodes and 41,837,355 arcs. In that instance, our journey planner achieved an average of 10.25ms for answering a query, and of 271.46μs for updating the timetable after a delay. More details can be found in [2].

To help the users assess the quality of the journeys suggested by our planner, we developed an additional *journey assessment service* (JAS) that has been built upon the core services offered by the Organicity IoT/FIRE+ platform [7]. In particular, Organicity (OC) provides an Experimentation-as-a-Service framework (EaaS API; cf. Figure 6) which supports, among others, the building of services for experimentation.

JAS builds upon the OC EaaS framework. It receives the journey assessments made by the mobile users. The communication is performed with HTTPS POST requests over three steps: (1) the client sends its authorization credentials; (2) the JAS sends an access token to the client to authorize him sending an assessment; and (3) finally the client sends the journey assessment. The journey assessment data is in JSON format, consisting of the user rate and his/her review about the returned journey. The option for enabling the journey's assessment (Figure 5, option "Evaluate") enables the assessment of the journey, shown in Figure 7, where the mobile users can submit their rate and review.

V. Conclusion

We described a cloud-based mobile journey planner, enforcing mobility in a public transport system, and which is able to adapt timetable updates, caused by vehicle delays, in real-time. Our next challenge is to extend the journey planner to accommodate answering journey requests that involve more than one criteria.

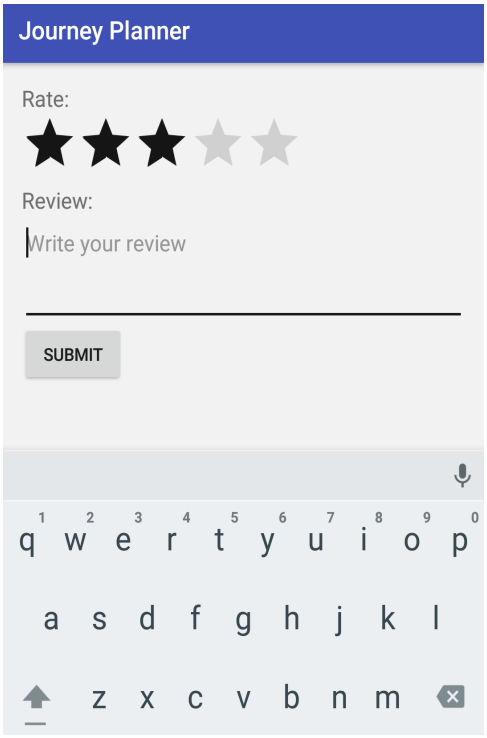

Fig. 7. Journey assessment.

Acknowledgment

We would like to thank D. Amaxilatis for assisting us with the Organicity platform. This work has been partially supported by the EU FP7/2007-2013 under grant agreements no. 609026 (project MOVESMART), no. 621133 (project HoPE), and by the DFG grant WA 654/23-1 within FOR 2083.

References

[1] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, *Lecture Notes in Computer Science* Vol. 9220 (Springer 2016), pp. 19–80.

[2] A. Cionini, G. D'Angelo, M. D'Emidio, D. Frigioni, K. Giannakopoulou, A. Paraskevopoulos, and C. D. Zaroliagis. Engineering graph-based models for dynamic timetable information systems. In *Algorithmic Approaches for Transportation Modelling, Optimization, and Systems* – ATMOS 2014, *OASICS* Vol. 42 (2014), pp. 46–61.

[3] D. Delling, T. Pajor, and D. Wagner. Engineering time-expanded graphs for faster timetable information. In *Robust and Online Large-Scale Optimization*, *Lecture Notes in Computer Science* Vol. 5868 (Springer, 2009), pp. 182–206.

[4] A. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, (SIAM, 2005), pp. 156–165.

[5] HaCon - Ingenieurgesellschaft mbH. http://www.hacon.de, 2008.

[6] MOVESMART project. http://www.movesmartfp7.eu/.

[7] ORGANICITY project. http://organicity.eu/.

[8] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[9] Transit Feeds. https://transitfeeds.com.

[10] Transport for London. https://tfl.gov.uk.

[11] General Transit Feed Specification. https://developers.google.com/transit/gtfs.