Failure-Proof Spatio-temporal Composition of Sensor Cloud Services

Azadeh Ghari Neiat, Athman Bouguettaya, Timos Sellis, and Hai Dong

School of Computer Science and Information Technology, RMIT, Australia
{azadeh.gharineiat,athman.bouguettaya,timos.sellis,
hai.dong}@rmit.edu.au

Abstract. We propose a new failure-proof composition model for Sensor-Cloud services based on dynamic features such as spatio-temporal aspects. To evaluate Sensor-Cloud services, a novel spatio-temporal quality model is introduced. We present a new failure-proof composition algorithm based on D* Lite to handle QoS changes of Sensor-Cloud services at run-time. Analytical and simulation results are presented to show the performance of the proposed approach.

Keywords: Spatio-temporal Sensor-Cloud service, spatio-temporal composition, Sensor-Cloud service composition, spatio-temporal QoS, service re-composition.

1 Introduction

The large amount of real-time sensor data streaming from Wireless Sensor Networks (WSNs) is a challenging issue because of storage capacity, processing power and data management constraints [1]. Cloud computing is a promising technology to support the storage and processing of the ever increasing amount of data [2]. The integration of WSNs with the cloud (i.e., Sensor-Cloud) [3] provides unique capabilities and opportunities, particularly for the use of data service-centric applications. Sensor-Cloud is a potential key enabler for large-scale data sharing and cooperation among different users and applications.

A main challenge in Sensor-Cloud is the efficient and real-time delivery of sensor data to end users. The preferred technology to enable delivery is services [4], i.e., sensor data made available as a service (i.e. Sensor-Cloud service) to different clients over a Sensor-Cloud infrastructure. The service paradigm is a powerful abstraction hiding data-specific information focusing on how data is to be used. In this regard, sensor data on the cloud is abstracted as Sensor-Cloud services easily accessible irrespective of the distribution of sensor data sources. In this paper, we propose a service-oriented Sensor-Cloud architecture that provides an integrated view of the sensor data shared on the cloud and delivered as services.

The *"position"* and *"time"* of sensed data are of paramount importance reflecting the spatio-temporal characteristics. Spatio-temporal features are fundamental to the functional aspect of the Sensor-Cloud. In this regard, we focus on spatio-temporal aspects as key parameters to query the Sensor-Cloud.

Composition provides a means to aggregate Sensor-Cloud services. In a highly dynamic environment such as those found in sensed environments, the non-functional

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 368–377, 2014.

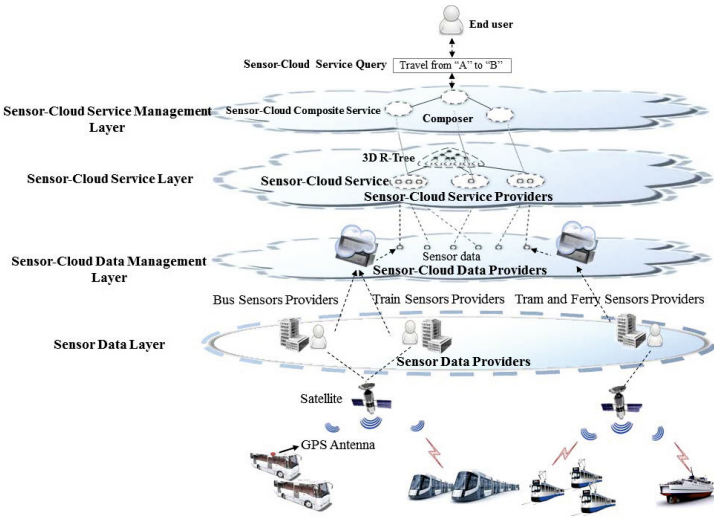

Fig. 1. The Public Transport Motivation Scenario

properties (QoS) of Sensor-Cloud services may fluctuate [5]. For example, a participant service may no longer be available or its QoS constraint has been fluctuated at runtime. As a result, the service may no longer provide the required QoS and fail. Therefore, the initial composition plan may become non-optimal and needs to be replanned to deal with the changing conditions of such environments.

This paper focuses on providing an efficient failure-proof spatio-temporal composition model for Sensor-Cloud services. In particular, new spatio-temporal QoS attributes to evaluate Sensor-Cloud services based on spatio-temporal properties of the services are proposed. We propose a failure-proof spatio-temporal combinatorial search algorithm to deal with the affecting component Sensor-Cloud services based on D* Lite algorithm [6] which is an incremental version of A* algorithm. D* Lite algorithm is efficient at repairing the plan when the new information about the environment is received [10]. Our proposed approach continually improves its initial composition plan and find the best composition plan from a given source-point to a given destination point while QoS constraints change.

The rest of the paper is structured as follows: Section 2 presents the proposed spatio-temporal model for Sensor-Cloud services. Section 3 illustrates the spatio-temporal QoS model. Section 4 elaborates the details of the proposed failure-proof composition approach. Section 5 evaluates the approach and shows the experiment results. Section 6 concludes the paper and highlights some future work.

Motivating Scenario

We use a typical scenario from public transport as our motivating scenario. Suppose Sarah is planning to travel from 'A' to 'B'. She wants to get information about the travel

services (i.e., buses, trams, trains and ferries) in the city to plan her journey. Different users may have different requirements and preferences regarding QoS. For example, Sarah may specify her requirements as maximum walk 300 meters and waiting time 10 minutes at any connecting stop. In this scenario, we assume that each bus (tram / train / ferry) has a set of deployed sensors (see Fig. 1). We also assume that there are several bus sensor providers (i.e., *sensor data providers*) who supply sensor data collected from different buses. Assuming that each *sensor data provider* owns a subset of a set of sensors on each bus. In addition, there are several *Sensor-Cloud data providers* who supply Infrastructure as a Service (IaaS), i.e., CPU services, storage services, and network services to sensor data providers. *Sensor-Cloud service providers* make services available that may query multiple heterogeneous *sensor data providers*. We assume that each *Sensor-Cloud service provider* offers one or more Sensor-Cloud services to help commuters devise the " best " journey plan. Different *Sensor-Cloud service providers* may query the same *sensor data providers*. The quality of services that they provide may also be different.

In our scenario, Sarah uses the Sensor-Cloud services to plan her journey. It is quite possible that a single service cannot satisfy Sarah's requirements. In such cases, Sensor-Cloud services may need to be composed to provide the best travel plan. *The composer* acts on behalf of the end users to compose Sensor-Cloud services from different *Sensor-Cloud service providers*.

2 Spatio-temporal Model for Sensor-Cloud Service

To model and access spatio-temporal Sensor-Cloud services, we consider spatio-temporal dependency constraints between different Sensor-Cloud service operations. We propose a new formal spatio-temporal model for a Sensor-Cloud service and Sensor-Cloud service composition.

2.1 Spatio-temporal Model for Atomic Sensor-Cloud Service

We introduce the notion of Sensor-Cloud service based on spatio-temporal aspects. We discuss the key concepts to model a Sensor-Cloud service in terms of spatio-temporal features. The Sensor-Cloud service model is formally defined as follows:

- *Definition 1:* Sensor sen. A sensor sen_i is a tuple of $< id, (loc_i, ts_i)>$ where
  - id is a unique sensor ID,
  - (loc_i, ts_i) shows the latest recorded location of sensor sen_i and timestamp ts_i is the latest time in which sensor data related to Sensor-Cloud service is collected from sensor sen_i.
- *Definition 2:* Sensor-Cloud Service S. A Sensor-Cloud service S_i is a tuple of $< id, IS_i, FS_i, d_i, F_i, Q_i, SEN_i >$ where
  - id is a unique service ID,
  - IS_i (Initial State) is a tuple $< P_s, t_s >$, where
    - P_s is a GPS start-point of S_i ,
    - t_s is a start-time of S_i.

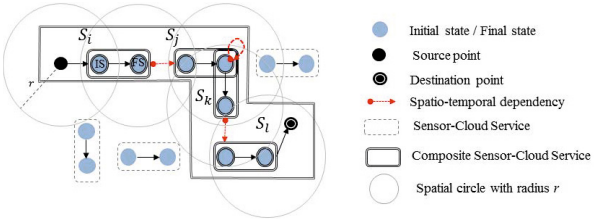

Fig. 2. Spatio-Temporal Composite Sensor-Cloud Service Model

- FS_i (Final State) is a tuple $< P_e, t_e >$, where
  - P_e is a GPS end-point of S_i,
  - t_e is an end-time of S_i.
- d_i represents the intra-dependency between two states from S_i meaning that FS_i is invoked after IS_i,
- F_i describes a set of functions offered by S_i,
- Q_i is a tuple $< q_1, q_2, \ldots, q_n >$, where each q_i denotes a QoS property of S_i,
- $SEN_i = \{sen_i | 1 \leqslant i \leqslant m\}$ represents a finite set of sensors sen_i collecting sensor data related to S_i.

For example, the function of a bus service S_{65} (F_{65}) can be defined as travelling from stop 4 at 5:10 pm (i.e., $IS_{65} =< stop\ 4\ ,\ 5:10pm >$) to stop 54 (i.e., $FS_{65} =< stop\ 54\ ,\ 5:22pm >$) *by bus*.

2.2 Spatio-temporal Model for Sensor-Cloud Service Composition

In some instances, an atomic Sensor-Cloud service may not fully meet user's requirements. In this case, a composition of services may be required. The main idea for composing Sensor-Cloud services is spatio-temporal dependencies among services. We represent a composite Sensor-Cloud service as a directed acyclic graph in which the nodes are service states (i.e., IS or FS) provided by component services and edges denote spatio-temporal dependencies among services (Fig. 2).

– *Definition 3:* Composite Sensor-Cloud Service CS. A composite Sensor-Cloud service CS is defined as a tuple $< SCS, r, t, D, \varsigma, \xi >$
  - $SCS = \{S_i | 1 \leqslant i \leqslant n\}$ represents a set of component services in which *n* is the total number of component services of CS,
  - r and t are user-defined spatial radius and time interval, respectively.
  - $D = \{< S_k, S_l > | 1 \leqslant k \leqslant n \wedge 1 \leqslant l \leqslant n \wedge k \neq l\}$ represents spatio-temporal neighbour dependencies between two component services S_k and S_l. A spatio-temporal neighbour dependency consists of two types of dependencies:
    - *spatial dependency*: two services S_k and S_l have spatial dependency if S_l. P_s is located inside the spatial circle centred at $S_k.P_e$ with a geographic radius *r*. For example, the bus stop 4 of bus service 65 has the spatial dependency with the tram stop 13 of tram service 8 supporting a walk of 300 meters (i.e., r = 300) between the bus stop and tram station.

  - ∗ *temporal dependency*: two services S_k and S_l have temporal dependency if S_l will be executed in a time window *t* of S_k, i.e., $S_k.t_e \leq S_l.t_s + t$. For example, the bus service 65 arrives in the bus stop 4 within 10 min (t = 10) before departing the tram service 8 from the tram stop 13.

  S_k and S_l have spatio-temporal neighbour dependency, if they are both spatially and temporally dependent.
- ς and ξ are a source-point and a destination-point, respectively.

In the remainder of the paper, the service and composite service are used to refer to a Sensor-Cloud service and composite Sensor-Cloud service, respectively.

3 Spatio-temporal Quality Model for Sensor-Cloud Service

Multiple Sensor-Cloud providers may offer similar services at varying quality levels. Given the diversity of service offerings, an important challenge for users is to discover the 'right' service satisfying their requirements. We introduce novel QoS attributes for services that focus on the spatio-temporal aspects. The proposed quality model is extensible. For the sake of clarity, we use a limited number of QoS.

3.1 Spatio-temporal Quality Model for Atomic Sensor-Cloud Service

We propose to use spatio-temporal quality criteria which is part of describing the non-functional aspects of services:

– *Service time (st):* Given an atomic service S, the service time q_{st}(S) measures the expected time in minutes between the start and destination points. The value of q_{st}(S) is computed as follows:

$$q_{st}(S) = S.t_e - S.t_s \tag{1}$$

– *Currency (cur):* Currency indicates the temporal accuracy of a service. Given an atomic service S, currency q_{cur}(S) is computed using the expression $(currenttime - timestamp(S))$. Since each service consists of a set of sensors $\{sen_1, ..., sen_n\}$, $timestamp(S)$ will be computed as follows:

$$timestamp(S) = Avg(ts_i) \tag{2}$$

– *Accuracy (acc):* Accuracy reflects how a service is *assured*. For example, a smaller value of accuracy shows the fewer sensors contribute to the results of the service. Given an atomic service S, the accuracy q_{acc}(S) is the number of operating sensors covering the specific spatial area related to S. The value of the q_{acc}(S) is computed as follows:

$$0 \leqslant \frac{N_{sen}(S)}{T_c} \leqslant 1 \tag{3}$$

where $N_{sen}(S)$ is the expected number of operating sensors in S and T_c is the total number of sensors covering the spatial area related to S. $N_{sen}(S)$ can be estimated based on the number of sen in *S*. We assume that T_c is known. It is also assumed that all sensors have the same functionalities and accuracy. For example, sensor data related to the bus service S_{65} is collected from 4 sensors (N_{sen} = 4) from 20 sensors (T_c = 20) deployed on a bus where is the spatial area related to S_{65}.

3.2 Spatio-temporal Quality Model for Composite Sensor-Cloud Service

The quality criteria defined above are in the context of atomic services. Aggregation functions are used to compute the QoS of the composite service. Table 1 presents these aggregation functions:

- *Service time:* The service time of a composite service is the sum of the service time of all its component services in addition to the transition time *trans* between two component services. The transition time is computed as follows:

$$trans = \sum_{j=1}^{n-1} (S_{(j+1)}.start\text{-}time - S_j.end\text{-}time) \tag{4}$$

  where S_j and S_{j+1} are two subsequent component services and $S_1.end\text{-}time$ is the start time of a query Q_t.
- *Currency:* The currency value of a composite service is the average of the currency of all the selected services.
- *Accuracy:* The accuracy value for a composite service is the product of the accuracy of all its component services.

Table 1. QoS Aggregation Functions

QoS attribute	Service Time	Currency	Accuracy
Aggregation Function	$\sum_{i=1}^{m} q_{st}(S_i) + trans$	$\frac{\sum_{i=1}^{m} q_{cur}(S_i)}{m}$	$\prod_{i=1}^{m} q_{acc}(S_i)$

4 Failure-Proof Spatio-temporal Composition Approach

When a service experiences significant quality fluctuation at runtime, an established composition plan may no longer be optimal. There are two situations in which a composition may become non-optimal. First, when QoS constraints of a component service violate or a component service may no longer be available at runtime and the composition may fail. Second, when a component service may provide better QoS and a more optimal composition plan may be provided. In such situations, all compositions that include the affecting service should adapt to the fluctuation.

We propose a heuristic algorithm called Spatio-Temporal A* (*STA**) algorithm [7] which is a variation of A* offering an optimal composition plan. *STA** differs on *neighbour* and *search cost* functions. In [7], we present a new spatio-temporal index data structure based on a 3D R-tree [8] to organize and access services. The nodes of the 3D R-tree represent actual services. We define a new *neighbour* function to find spatio-temporal neighbour services (i.e. candidate services) of a service, called *Spatio-TemporalSearch* algorithm, based on a 3D R-tree [7]. We also define the *search cost* function *f-score* as follows:

$$f\text{-}score[S] = g\text{-}score[S] + h\text{-}score[S] \tag{5}$$

where *g-score* calculates the QoS utility score [9] of selected services from the source-point ς to the current service and heuristic function *h-score* estimates the Euclidean distance between the end-point of candidate service S and the destination-point ξ.

In this section, we propose a novel failure-proof service composition approach based on spatio-temporal aspects of services to support real-time response to fluctuation of QoS attributes. We introduce a new heuristic algorithm based on D* Lite, called *STD*Lite*. D* Lite is a dynamic shortest path finding algorithm that has been extensively applied in mobile robot and autonomous vehicle navigation. D* Lite is capable of efficiently replanning paths in changing environment [10]. Whenever the QoS values of component services in the initial composition plan significantly change at runtime, *STD*Lite* recomputes a new optimal composition plan from its current position to the destination. Without loss of generality, we only consider temporal QoS fluctuations in service time q_{st}. In our approach, the existence of a temporal QoS change is ascertained by measuring the value of difference τ between the measured q_{st} of a service at runtime and its promised q_{st}. If τ is more than a defined threshold ϵ, a QoS change has occurred.

*STD*Lite* algorithm , like *STA**, maintains an estimate $g\text{-}score$ for each service S in the composition plan. Since *STD*Lite* searches backward from the destination-point to the source-point, $g\text{-}score$ estimates the QoS utility score of the optimal path from *S* to the destination. It also maintains a second kind of estimates called rhs value which is one step lookahead of $g\text{-}score$. Therefore, it is better informed than $g\text{-}score$ and computed as follows:

$$rhs(S) = \begin{cases} 0 & S.P_e = \xi \\ min_{S' \in SuccNeighboursList(S)}(trans(S', S) + g\text{-}score(S')) & S.P_e \neq \xi \end{cases} \quad (6)$$

where $trans(S', S)$ is the transition time between S' and S and $SuccNeighboursList$ is the set of *successor* neighbours of the service S. The rationale of using neighbours is that the optimal plan from S to the destination must pass through one of the neighbours of S. Therefore, if we can identify the optimal plans from any of the neighbours to the destination, we can compute the optimal plan for S. The successor neighbours of a service S are identified through *Spatio-TemporalSearch* algorithm in [7].

By comparing $g\text{-}score$ and rhs, the algorithm identifies all affecting, called inconsistent, component services. A service is called locally consistent iff its rhs value equals to its $g\text{-}score$ value, otherwise it is called locally inconsistent. A locally inconsistent service falls into two categories: *underconsistent* (if $g\text{-}score(S) < rhs(S)$) and *overconsistent* (if $g\text{-}score(S) > rhs(S)$). A service is called *underconsistent* if its QoS values degrades. In such a situation, the QoS values of affecting services should be updated and the composition plan should adapt to the violations. Moreover, a service is called *overconsistent* if its QoS values become better. An overconsistent service implies that a more optimal plan can be found from the current service. When a service is inconsistent, the algorithm updates all of it's neighbours and itself again. Updating services makes them consistent.

Algorithm 1 presents the details of *STD*Lite* algorithm. The algorithm generates an optimal initial composition plan like a *backward STA** search {Line 33-42}. If the QoS values of component services change after generating the initial composition plan, *STD*Lite* updates the inconsistent (i.e., affecting) component services and expands the services to recompute a new optimal composition plan {43-47}. All inconsistent

services are then inserted in a priority queue *CandidateQueue* to be updated and made consistent. *STD*Lite* avoids redundant updates through updating only the inconsistent services which are necessary to modify, while A* updates all of the plan. The priority of an inconsistent service in *CandidateQueue* is determined by *key value* as follows:

$$\begin{aligned} key(S) &= [k_1(S), k_2(S)] \\ &= [min(g\text{-}score(S), rhs(S)) + h\text{-}score(S_{start}, S), min(g\text{-}score(S), rhs(S))] \end{aligned} \tag{7}$$

The keys are compared in a lexicographical order. The priority of $key(S) < key(S')$, iff $k_1(S) < k_1(S')$ or $k_1(S) = k_1(S')$ and $k_2(S) < k_2(S')$. The heuristics in k_1 serves in the same way as $f\text{-}score$ in *STA**. The algorithm applies this heuristic to ensure that only the services either newly overconsistent or newly underconsistent that are relevant to repairing the current plan are processed. The inconsistent services are selected in the order of increasing priority which implies that the services which are closer to the S_{start} (i.e. less $h\text{-}score$ value) should be processed first. Note that as the algorithm tracks the execution of the composition plan, the start service S_{start} becomes the current running service of the plan. Therefore, when a QoS value fluctuates, a new optimal plan is computed from the original destination to the new start service (i.e. current service).

The algorithm finally recompute a new optimal plan by calling *ComputePlan()* function $\{48\}$. *ComputePlan()* expands the local inconsistent services on *CandidateQueue* and updates $g\text{-}score$ and rhs values and add them to or remove them from *CandidateQueue* with their corresponding keys by calling *UpdateService()* function $\{4\text{-}15\}$.

When *ComputePlan()* expands an overconsistent service, it sets $g\text{-}score$ value of the service equals to its rhs value to make it locally consistent $\{20\}$. Since rhs values of predecessor neighbours of a service are computed based on the $g\text{-}score$ value of the service, any changes of its $g\text{-}score$ value can effect the local consistency of its predecessor neighbours. As a result, predecessor neighbours $\{19\}$ of an inconsistent service should be updated $\{21\text{-}23\}$.

When *ComputePlan()* expands an underconsistent service, it sets $g\text{-}score$ value of the service to infinity to make it either overconsistent or consistent $\{25\}$. The predecessor neighbour services of the service need also to be updated $\{26\text{-}28\}$. *ComputePlan()* expands the services until the key value of the next service to expand is not less than the key value of S_{start} and S_{start} is locally consistent $\{17\}$.

5 Experiments Results

We conduct a set of experiments to assess the effectiveness of the proposed approach over different QoS fluctuation ratio. We run our experiments on a 3.40 GHZ Intel Core i7 processor and 8 GB RAM under Windows 7. To the best of our knowledge, there is no spatio-temporal service test case to be used for experimental purposes. Therefore, we focus on evaluating the proposed approach using synthetic spatio-temporal services.

In our simulation, 1000 nodes are randomly distributed in a 30 *km* $\times$ 30 *km* region. The radius for neighbour search r as 0.5% of the specified region. All experiments are conducted 1000 times and the average results are computed. Each experiment starts from a different source and destination which are randomly generated. Two spatio-temporal QoS attributes of the syntactic service instances are randomly generated with

Algorithm 1. STD*Lite [basic version]

```
1: procedure CALCULATEKEY(S)
2:     return [min(g-score(S), rhs(S)) +h-score(S_start, S), min(g-score(S), rhs(S))]
3: end procedure
4: procedure UPDATESERVICE(S)
5:     if S.P_e ≠ ξ then
6:         SuccNeighboursList = Spatio-TemporalSearch(G, RT, S.p_e , S.t_e, r, t)
7:         rhs(S) = min_{S'∈SuccNeighboursList} (trans(S',S)+ g-score(S'))
8:     end if
9:     if S ∈ CandidateQueue then
10:        CandidateQueue.remove(S)
11:    end if
12:    if g-score(S) ≠ rhs(S) then
13:        CandidateQueue.insert(S,CalculateKey(S))
14:    end if
15: end procedure
16: procedure COMPUTEPLAN( )
17:    while min_{S∈CandidateQueue}(key(S))< key(S_start) or rhs(S_start) ≠ g-score(S_start) do
18:        CandidateQueue.remove(S with minimum key)
19:        PredNeighboursList = Spatio-TemporalSearch(G, RT, S.p_s, S.t_s-t, r, t)
20:        if g-score(S) > rhs(S) then g-score(S) = rhs(S)
21:            for all S' ∈ PredNeighboursList do
22:                UpdateService(S)
23:            end for
24:        else
25:            g-score(S) = ∞
26:            for all S' ∈ PredNeighboursList ∪ S do
27:                UpdateService(S)
28:            end for
29:        end if
30:    end while
31: end procedure
32: procedure MAIN( )
33:    CandidateQueue = ∅
34:    for all services S do
35:        g-score(S) = rhs(S) = ∞
36:    end for
37:    rhs(S_destination) = 0
38:    CandidateQueue.insert(S_destination, CalculateKey(S_destination))
39:    ComputePlan()
40:    if g-score(S_start) = ∞ then
41:        print "there is no plan"
42:    end if
43:    while S_start ≠ S_destination do
44:        Runtime monitoring to find the affecting services
45:        for all affecting services S do
46:            UpdateService(S)
47:        end for
48:        ComputePlan()
49:    end while
50: end procedure
```

a uniform distribution from the following intervals: $q_{acc} \in [0, 1]$ and $q_{cur} \in [60, 1440]$. The q_{st} is assigned based on the distance between P_s and P_e considering a fixed speed. The remaining parameters are also randomly generated using a uniform distribution.

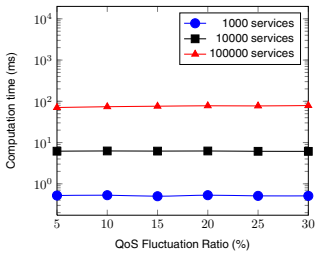

Fig. 3. Computation time vs. fluctuation ratio

We test the performance of *STD*Lite* in terms of computation time with the number of services varying from among 1000, 10000 and 100000. For each group of services, we also vary the QoS fluctuation ratio from 5 to 30 %. The QoS fluctuation ratio indicates that the ratio of the number of affecting services over the total number of services. For example, a fluctuation ratio of 10% denotes that the service time of 10% of the total number of services change at runtime. Fig. 3 shows *STD*Lite* performs very efficiently on a large number of services (i.e., less than 100 *ms* on 100000 services). The computation time increases along with the number of services, which is an expected result. It can be seen that the similar computation time is achieved regardless of the QoS fluctuation

ratio. The slight difference (i.e., less than 10 ms over 100000 services) shows the relative stability of our approach when QoS is highly violated.

6 Conclusion

This paper proposes a novel approach for failure-proof composition of Sensor-Cloud services in terms of spatio-temporal aspects. We introduce a new failure-proof spatio-temporal combinatorial search algorithm based on D* Lite to replan a composition plan in case of QoS changes. We conduct preliminary experiments to illustrate the performance of our approach. Future work includes implementing a prototype and test it with real-world applications with focusing on building Sensor-Clouds for public transport.

References

1. Hossain, M.A.: A survey on sensor-cloud: architecture, applications, and approaches. International Journal of Distributed Sensor Networks (2013)
2. Lee, K., Murray, D., Hughes, D., Joosen, W.: Extending sensor networks into the cloud using amazon web services. In: 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications (NESEA), pp. 1–7. IEEE Press (2010)
3. Rajesh, V., Gnanasekar, J., Ponmagal, R., Anbalagan, P.: Integration of wireless sensor network with cloud. In: 2010 International Conference on Recent Trends in Information, Telecommunication and Computing (ITC), pp. 321–323. IEEE Press (2010)
4. Carey, M.J., Onose, N., Petropoulos, M.: Data services. Communications of the ACM 55(6), 86–97 (2012)
5. Ben Mabrouk, N., Beauche, S., Kuznetsova, E., Georgantas, N., Issarny, V.: Qos-aware service composition in dynamic service oriented environments. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 123–142. Springer, Heidelberg (2009)
6. Koenig, S., Likhachev, M.: D* lite. In: AAAI/IAAI, pp. 476–483 (2002)
7. Ghari Neiat, A., Bouguettaya, A., Sellis, T., Ye, Z.: Spatio-temporal composition of sensor cloud services. In: 21th IEEE International Conference on Web Services (ICWS), pp. 241–248. IEEE Press (2014)
8. Theoderidis, Y., Vazirgiannis, M., Sellis, T.: Spatio-temporal indexing for large multimedia applications. In: Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, pp. 441–448. IEEE Press (1996)
9. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
10. Koenig, S., Likhachev, M.: Improved fast replanning for robot navigation in unknown terrain. In: IEEE International Conference on Robotics and Automation, pp. 968–975 (2002)