Proactive SLA Negotiation for Service Based Systems:Initial Implementation and Evaluation Experience

Khaled Mahbub
Department of Computing
City University London
United Kingdom
K.Mahbub@soi.city.ac.uk

George Spanoudakis
Department of Computing
City University London
United Kingdom
G.Spanoudakis@soi.city.ac.uk

Abstract—**This paper describes a framework that we have developed to integrate proactive SLA negotiation with dynamic service discovery to provide cohesive runtime support for both these activities. The proactive negotiation of SLAs as part of service discovery is necessary for reducing the extent of interruptions during the operation of a service based system when the need for replacing services in it arises. The developed framework discovers alternative candidate constituent services for a service client applications, and negotiates/agrees but does not activate SLAs with these services until the need for using a service becomes necessary. A prototype tool has been implemented to realize the framework. This prototype is discussed in the paper along with the results of the initial evaluation of the framework.**

Keywords-Service discovery, Service level agreements; Proactive SLA negotiation; service monitoring

I. INTRODUCTION

A trustworthy use of software services often requires as a prerequisite the existence of a service level agreement (SLA) between the provider and the consumer of a service. SLAs define quality of service (QoS) and functional properties, which should be guaranteed during the provision of a software service, as well as the penalties that should be applied in cases where the properties are not fulfilled [7][10][11]. An SLA is set through a negotiation between the provider and the consumer of a service [4][12]. SLA negotiation can be particularly complex depending on the requirements and affordances of the two parties. Furthermore, it may need to be carried out at runtime, if a constituent service of a service based client application (SCA) becomes unavailable whilst SCA is in operation, or it fails to perform according to its established SLA. In such cases, SCA should be able to discover alternative replacement services for the failed service, and negotiate SLAs with them at runtime.

To minimize the runtime interruption of the SCA in such circumstances, the discovery of back up replacement services for SCA constituent services should be performed proactively before any of these constituent services becomes unavailable or fails to perform according to its established SLAs [15]. This is important since service discovery and SLA negotiation are time consuming processes that would delay significantly the responsiveness of SCA if they were executed every time that it becomes necessary to identify and use a new service at runtime.

Existing work on service level agreements has focused on SLA specification [13][14], negotiation [5] and monitoring [9]. The need for runtime SLA negotiation or re-negotiation has been acknowledged in [2][3][5][10]. Existing approaches, however, are reactive supporting corrective actions only after SLA violations and, thus, they cannot ensure uninterrupted runtime SCA operations when services fail.

To address the above shortcoming, we have developed a proactive runtime SLA negotiation tool, and integrated it with a tool supporting proactive runtime service discovery, which has been previously described in [15]. The integrated tools constitute a framework called PROSDIN (PROactive Service DIscovery and Negotiation). In PROSDIN, SLA negotiation has been developed as an integrated part of the service discovery process enabling the execution of both activities in a coordinated manner. More specifically, proactive SLA negotiation is performed immediately after the execution of service discovery queries to ensure that adequate SLAs are provisionally agreed for given periods of time with the providers of the discovered services if possible. Also when a pre-agreed SLA expires and it is proactively re-negotiated.

The initial design of our approach to proactive SLA negotiation has been discussed in [24]. The contributions of this paper with respect to [24] are that: (a) it describes the implemented version of RROSDIN, which realises a new version of the SLA negotiation process and the use of a rule-based approach to negotiation using the Jess rule engine [21], and (b) it presents the results of an initial experimental evaluation of the framework.

The rest of this paper is structured as follows. In Sect. II, we discuss the architecture of the service discovery and SLA negotiation framework. In Sect. III, we describe the negotiation process. In Section IV, we provide an overview of the language for specifying SLAs. In Sect. V, we discuss the representation of negotiation rules and the realisation of the negotiation process by the Jess rule engine. In Sect. VI, we describe the results ofan initial evaluation the framework. Finally, in Sect. VII and VIII, we review related work and provide concluding remarks and directions for future work, respectively.

II. Framework Overview

The architecture of PROSDIN is shown in Fig. 1. More specifically, PROSDIN consists of a (runtime) *service discovery tool*, a *service listener*, and a *negotiation broker*. It also interacts with external *service registries* and is available itself to external service client applications as a service.

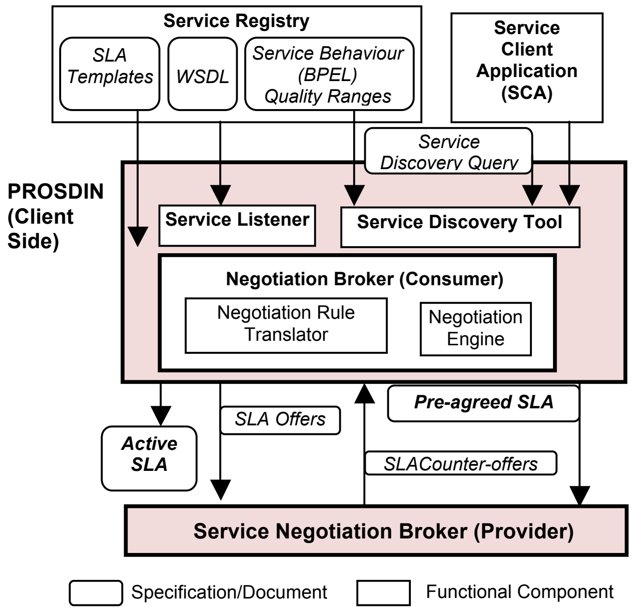

Figure 1. PROSDIN architecture

The *service discovery tool* in PROSDIN is used to identify candidate services that could potentially be used by the service client application (SCA). Service discovery is based on queries that express conditions about the interface, behaviour, contextual and quality characteristics of services. To use PROSDIN, each of the constituent services S_c of SCA that could be replaced at runtime should be associated with a discovery query specifying the conditions for discovering services that could potentially replace it. These queries should be specified by the developers of SCAs during the development of SCA, and passed to PROSDIN by SCA at runtime in order to be executed when service failures occur and enable discovery.

Following the subscription of such queries for a constituent service S_c of SCA, PROSDIN executes them proactively and in parallel with the execution of SCA, and uses the services that match with then in an external registry to maintain an up-to-date set of candidate replacement services for S_c (referred to as *replacement service* set RS in the following). Furthermore, after the initial creation of the RS set for a service, the query associated with it is also executed when the description of some service in RS has been changed or a new service that could be a candidate for inclusion in RS has emerged in the external service registry. Notifications of such changes are generated by the *service listener* of PROSDIN, which polls the external service registry periodically to identify service changes relevant to a query and the services in RS.

The *negotiation broker* in PROSDIN manages the negotiation process on behalf of service client applications. More specifically, it provides access to different negotiation engines that may be plugged into the framework by translating negotiation rules expressed in the common language of the framework into the different negotiation specifications accepted by these engines (see Sect. V) and realizes the interface for interacting with brokers carrying out the negotiation process on behalf of services (aka *service negotiation brokers*). The latter may be the same as the broker used by PROSDIN or other brokers that realize the same SLA negotiation interaction interface with it. The interface of the negotiation broker provides operations for: (i) initializing the broker, (ii) starting the negotiation process, and (iii) notifying a broker that an SLA offer (or counter-offer) that has been generated/rejected/accepted by the other party in the negotiation process.

The negotiation process is carried out according to a two-phase protocol that may result in a pre-agreed but not activated SLA or fail. Pre-agreed SLAs have an expiry period within which they can become active, if the service client application decides to activate them. The SLA negotiation process is described in Sect. III.

III. Service Discovery/SLA Negotiation Process

The activity of service discovery and SLA negotiation realized in PROSDIN is shown in Fig. 2.

According to the UML activity diagram in the figure, the process starts with the submission of a service discovery query by an SCA. The initial execution of the query (see *Execute Query* in Fig. 2) is followed by the build of the set RS. RS includes the best N candidate services, ranked in ascending order of their distance to the query. RS is updated by executing the service discovery query when the framework is informed by the service listener that a new service has become available in the service registry or the description of an existing service has been modified (see the *New/Amended Service Description* signal in Fig. 2). Hence, the process considers new and updated services.

After RS is initially built or updated, the framework selects the first service in it that does not have a negotiated SLA, and starts a proactive negotiation of an SLA with it (see *Select Service in RS for Negotiation* and *Negotiate SLA* activities in Fig. 2, respectively). In this phase, the *QoS* characteristics of the candidate service are negotiated in order to achieve the best possible SLA given the boundary constraints of the two parties.

If the negotiation with a service S fails, S is removed from *RS* and discovery is re-triggered to find another service to replace it. If negotiation succeeds, a provisional SLA is established and the candidate service in *RS* is updated to flag the existence of a pre-agreed SLA with it. Subsequently, the process continues by attempting to negotiate SLAs with all the services in *RS*, which do not have a pre-agreed SLA, until all of them have pre-negotiated SLAs or it is known (through unsuccessful earlier negotiation attempts) that an SLA cannot be established with them.

The negotiated SLAs of services in *RS* do not come into force immediately. For each pre-agreed SLA, the negotiation process establishes a time period over which the pre-agreed SLA can be automatically brought into force without further negotiation. This happens when a service with a pre-agreed SLA in RS is selected for binding to SCA. If the validity period of a pre-agreed SLA expires without the candidate service being bound to SCA, the SLA between the service and SCA will be re-negotiated.

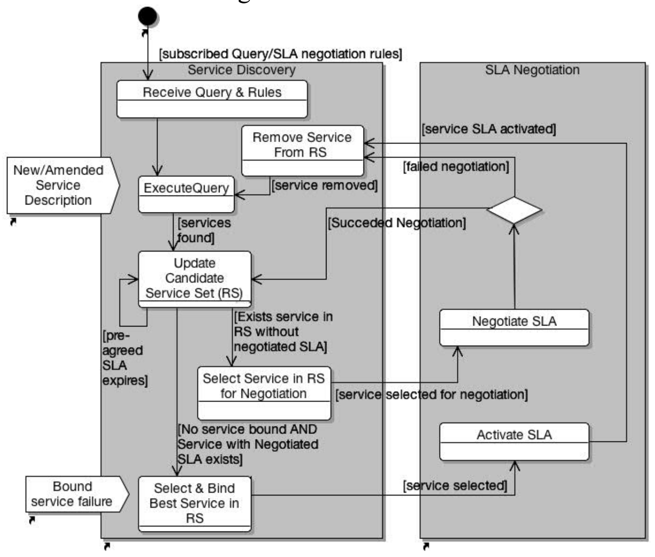
[subscribed Query/SLA negotiation rules]
Service Discovery
SLA Negotiation
Receive Query & Rules
Remove Service From RS
[service SLA activated]
[failed negotiation]
New/Amended Service Description
ExecuteQuery
[service removed]
[services found]
[Succeded Negotiation]
Update Candidate Service Set (RS)
[Exists service in RS without negotiated SLA]
Negotiate SLA
[pre-agreed SLA expires]
Select Service in RS for Negotiation
[service selected for negotiation]
[No service bound AND Service with Negotiated SLA exists]
Activate SLA
Bound service failure
Select & Bind Best Service in RS
[service selected]

Figure 2. Service discovery and SLA negotiation process

Following the selection of a service S in *RS* for binding to SCA at runtime, its SLA is automatically activated (see *Activate SLA* in Fig. 2), the service is removed from RS (see *Remove Service from RS*) and the discovery query is re-executed to identify if there is a new service that could be included in the RS set.

IV. Specification of SLAs

The operation of PROSDIN is driven by specifications of: (a) discovery queries, (b) SLAs and SLA templates, and (c) SLA negotiation rules. In this section we give an overview of the languages for specifying (b) and in Sect. V we overview the language for specifying (c) (the language that is used to specify service discovery queries is beyond the scope of this paper and can be found in [15]).

A. Specification of Service Level Agreements

In PROSDIN, SLA templates, offers, agreed and activated SLAs are specified using an XML schema whose high level structure is shown in Fig. 3. According to this schema, an SLA is specified by an *SLA contract* element, containing one or more *SLA terms*. An SLA term specifies one or more guaranteed quality constraints (i.e., constraints over values of *QoS* attributes). It also refers to the actor(s) who have proposed it in the negotiation process (see *Actor* element in *SLATermsType*). An actor may take different roles in the negotiation process (e.g., service requester or service provider) and have a negotiation strategy, i.e., a set of rules governing the negotiation process and the communication (e.g. multiphase, multi issue negotiation) with other negotiating parties.

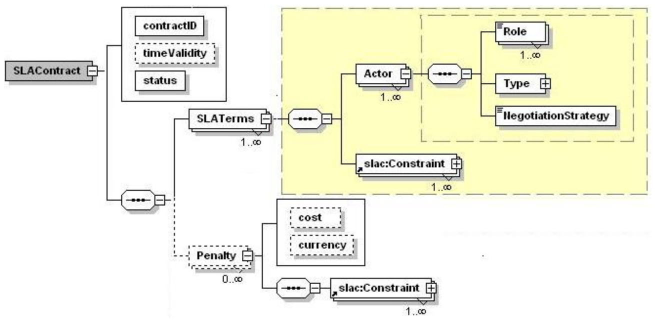

Figure 3. High Level Schema for SLA Specification

An SLA contract may also describe the penalties that will apply in case that any of the parties who have agreed the contract (contractors) fail to fulfill the SLA terms (see the sub-element *Penalty*). Furthermore, SLA contracts have: (i) a *contractID* attribute, (ii) an attribute, called *status*, signifying the status of the contract (i.e., under negotiation, pre-agreed or active), and (iii) a *time validity* attribute signifying the period for which the contract is valid.

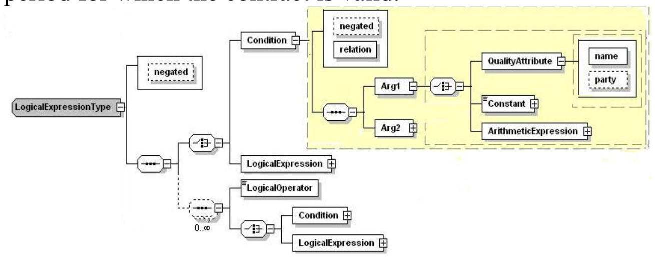

Figure 4. SLA Specification – Constraint element

```
<sla:SLAContractxmlns:sla="http://scube.eu/.."
xmlns:slac="http://scube.eu/schema/Constraint"
contractID="SLA-No-2"timeValidity="1Y"
status="PRE_AGREED">
 <sla:SLATerms>
  <sla:Actor>
   <sla:Role>PROVIDER</sla:Role>
   <sla:Type><sla:Companyname="XYZ"… /></sla:Type>
   <sla:NegotiationStrategy>
               MULTI-PHASE_MULTI-ISSUE
   </sla:NegotiationStrategy>
  </sla:Actor>
  <sla:Actor>.. ..</sla:Actor>
  <slac:Constraint>
   <slac:LogicalExpression>
    <slac:Conditionrelation="GREATER-THAN">
      <slac:Arg1><slac:QualityAttribute
             name="AVAILABILITY"/></slac:Arg1>
      <slac:Arg2><slac:Constanttype="NUMERICAL"
            unit="PC">80</slac:Constant></slac:Arg2>
    </slac:Condition>
   </slac:LogicalExpression>
   <slac:LogicalOperator>AND</slac:LogicalOperator>
   <slac:LogicalExpression>
    <slac:Conditionrelation="LESS-THAN">
      <slac:Arg1><slac:QualityAttribute
         name="RESPONSE_TIME"/></slac:Arg1>
      <slac:Arg2><slac:Constanttype="NUMERICAL"
         unit="MS">9</slac:Constant></slac:Arg2>
    </slac:Condition>
   </slac:LogicalExpression>
  </slac:Constraint>
 </sla:SLATerms>
 <sla:Penalty>...</sla:Penalty>
</sla:SLAContract>
```

Figure 5. Example SLA

Fig. 4 shows the part of the SLA schema that is used to specify constraints for SLA terms. A constraint is defined as an atomic logical expression or a conjunction/disjunction of two or more logical expressions. Atomic logical expressions are conditions over quality attributes of services. These conditions are defined as a relation between two arguments(e.g., *equalTo*, *lessThan, greaterThan*) and can be negated. The arguments of a relation can be a quality attribute of a service, constant, or an arithmetic expression over quality attributes and constants.

Fig. 5 shows an example of a pre-agreed SLA for service X between a company XYZ that provides X and the service consumer C. The SLA sets a conjunction of two conditions. The first of these conditions states that the availability of the service should be greater than 80%, and the second condition states that the response time of the service should be less than 9 milliseconds.

V. NEGOTIATION RULES AND BROKER

The SLA negotiation process in PROSDIN is executed by the negotiation broker according to negotiation rules. These rules are specified using the XML schema that is partly shown in Fig. 6. This schema allows the expression of negotiation rules as condition-action rules of the form: *IF (condition) THEN (action) ELSE (action).*

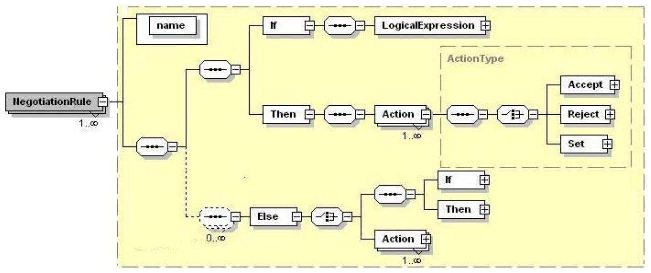

Figure 6. High Level Schema for Negotiation Rule Specification

The conditions in the negotiation rules are either atomic conditions or logical combinations of atomic conditions over QoS attributes of services having the same structure as the SLA term conditions discussed in Sect. IV. Rule actions can be of three types: (i) *accept* actions that are used to accept the value of one or more QoS attributes in a given SLA offer, (ii) *reject* actions that are used to reject the value of one or more QoS attributes in a given SLA offer, and(iii) *set* actions that are used to propose a new value or range of values for one or more QoS attributes as part of an SLA offer.

An example of a negotiation rule is shown in Fig.7. The rule is used by the negotiation broker of a service provider and states that if the consumer of a service has made an offer (or counter-offer) where the response time of the service must be less than 10 milliseconds (ms) and the price to be paid per service use is 0.5 pounds, the offered values will be accepted.

The negotiation rules expressed in the common XML language of PROSDIN are translated into the negotiation specification of the particular negotiation engine plugged into the broker. The negotiation engine used in the current implementation of the framework is a rule-driven negotiation engine that we have developed based on Jess [17].

A rule in Jess has the form

```
(defrule rule-name
  (logical-operator (cond-1 …cond-n))
 ⇒action)
```

where *cond-i* is defined as an atomic condition of the form *(<fact-pattern-i><cond-i>)* or a complex logical condition over such atomic conditions. The fact patterns in a rule define logical conditions over facts known to the Jess engine. Jess uses a form of the algorithm *Rete* [6] to match rules against facts and when a match is found the actions specified in a rule are taken. Such actions can assert or modify the values of facts.

Negotiation Rule

```
<tnsr:NegotiationRule name="rule1">
 <tnsr:If>
  <tnsr:LogicalExpression>
   <slac:Condition relation="LESS-THAN">
    <slac:Arg1><slac:QualityAttribute
     name="RESPONSE_TIME" party="CONSUMER"/>
    </slac:Arg1>
    <slac:Arg2><slac:Constant
     type="NUMERICAL" unit="ms"> 10
         </slac:Constant></slac:Arg2>
  </slac:Condition>
  <slac:LogicalOperator>AND
  </slac:LogicalOperator>
  <slac:Condition relation="EQUAL-TO">
    <slac:Arg1><slac:QualityAttribute
      name="PRICE" party="CONSUMER"
     unit="GBP"/></slac:Arg1>
    <slac:Arg2><slac:Constant
      type="NUMERICAL">0.5</slac:Constant>
    </slac:Arg2>
   </slac:Condition>
  </tnsr:LogicalExpression>
 </tnsr:If>
<tnsr:Then>
<tnsr:Action>
  <tnsr:Accept>
   <tnsr:QualityAttribute name="PRICE"
       party="CONSUMER" />
    <tnsr:QualityAttribute
     name="RESPONSE_TIME" party="CONSUMER"/>
   </tnsr:Accept>
  </tnsr:Action>
 </tnsr:Then>
</tnsr:NegotiationRule>
```

Figure 7. Example negotiation rule.

The translator inside the negotiation broker translates the negotiation rules into Jess rule. The basic transformations used are show in Figure 8. The first two rows in Figure 8 show the transformation of quality attribute and constants into Jess. The third row shows transformation of a condition in negotiation rule into Jess. The fourth row shows the transformation of a PROSDIN negotiation rule action into Jess. It should be noted that, in our implementation, we assume that Jess uses the same working memory for different phases of the negotiation process. Thus, we use the same identifier (i.e. 0) for the working memory in Jess representation (see the Jess expression `(fact-slot-value 0 Jess-Rep(Arg2))` in the example patterns).

Negotiation Rule Element	Jess Representation
`<slac:Arg> <slac:QualityAttribute name="A" party="P"/></slac:Arg>`	A-P
`<slac:Arg> <slac:Constanttype="NUMERICAL">C </slac:Constant></slac:Arg>`	C
`<slac:Condition relation="REL"> <slac:Arg1>...</slac:Arg1> <slac:Arg2>...</slac:Arg2> </slac:Condition>`	{Jess-Rep(Arg1) REL Jess-Rep(Arg2)}
`<tnsr:Action> <tnsr:Set> <tnsr:LogicalExpression> <slac:Condition relation="EQUAL-TO"> <slac:Arg1>...</slac:Arg1> <slac:Arg2>...</slac:Arg2> </slac:Condition> </tnsr:LogicalExpression> </tnsr:Set> </tnsr:Action>`	(modify 0 (Jess-Rep(Arg1) (fact-slot-value 0 Jess-Rep(Arg2))))

Figure 8. Negotiation to Jess rule Transformation Patterns

Based on the transformations listed in Fig. 8, the Jess rule generated for *rule1* is:

(defrule rule1
(SLA {RESPONSE_TIME-CONSUMER < 10}{PRICE-CONSUMER = 0.5}) =>
(modify 0
(PRICE-PROVIDER
(fact-slot-value 0 PRICE-CONSUMER)))
(modify 0 (RESPONSE_TIME-PROVIDER
(fact-slot-value 0 RESPONSE_TIME-CONSUMER))))

VI. IMPLEMENTATION &EVALUATION

All the major components of PROSDIN (i.e., the negotiation broker, service discovery tool, and service listeners) have been implemented in Java and are available as a web service. This service can be deployed by service client applications programmed in a way that can notify service discovery queries and SLA negotiation rules to PROSDIN, and receive endpoints of discovered services with negotiated SLAs from it. The external service registry used in the current implementation is a faceted registry as the one developed by the SECSE project [22]. This registry has been implemented using eXist [18] database and is accessed by PROSDIN through Java remote method invocation (RMI).

To evaluate the implementation of PROSDIN, we performed a series of experiments. The purpose of these experiments was to: (a) measure the overhead of SLA negotiation (whether reactive or proactive) on the execution time of the runtime service discovery process, and (b) assess the effectiveness of proactive SLA negotiation over reactive SLA negotiation during runtime service discovery process.

A. Experimental Setup

In the experiments, we have used an SCA, called *Route-Planner*, as a case study. Implemented by a BPEL service orchestration process, this system allows a user to find an optimal route from his/her current location to another location by using a *Global Positioning Service* (*GPS*), and displays electronic maps of the area where the user is located and the identified route between two points. The latter functionality is supported by the use of an electronic map service (*eMapS*). The service discovery query used in the experiments was specified in order to identify candidate replacement services for the GPS service of *Route-Planner*. The query expressed structural discovery criteria and a soft quality constraint. The structural criteria referred to the required (WSDL) interface for possible alternative services that could be used in the place of the GPS service should this service fail at runtime, and the quality constraint expressed a condition about service availability.

In the experiments, we also used an SLA template with four QoS terms for negotiation. These QoS terms were related to the service *price*, *availability* and *response time,* and the *mean number of service requests* per hour. For negotiation, we specified a set of 15 service consumer negotiation rules (CNR set), and 20 different sets of provider negotiation rules (PNR sets). Each of the PNR sets contained between 5 and 20 negotiation rules. During negotiation with each of the candidate services identified by the discovery process, the negotiation broker of the service provider side picked up randomly one of the PNR sets and carried out the negotiation based on it. In this way, we simulated the different behaviour that different service providers who participate in the negotiation process might have. The specifications of the SLA template, CNR and PNR sets and discovery query used in the experiments cannot be listed here due to space restrictions but can be found in [19].

To assess whether the number of considered services affects the performance of the service discovery and SLA negotiation processes, we performed the experiments with three different service sets (registries). These sets contained 100, 300 and 500 services, respectively and were populated by geographic location related services taken from the SEEKDA [20] and the SECSE service registry [22]. Each service that was used in the experiments had a WSDL (i.e., a structural) description and a quality of service description. These descriptions were used during the service discovery process.

All experiments were carried out using a Pentium 2.33 GHz with 3.23 GB RAM machine.

B. Results

In the experiments we measured the time needed to:

(a) Build the initial RS set (see Sect. II);
(b) Maintain the RS set at runtime due to the *arrival of new services (type_1* events*)* or *change in the description of an existing service* in the registry *(type_2* events*)*; and
(c) Select a service for replacing a service S in the service based application due to *unavailability of S (type_3* events*)*.

The times required for (a), (b) and (c) were measured, for executions of the service discovery process without SLA negotiation (*SD Only*), with proactive SLA negotiation (*SD with Proactive SLA*), and with reactive SLA negotiation (*SD with Reactive SLA*). Table I presents the formulas used to measure execution times in cases (a), (b) and (c).

TABLE I. BASIC TIME MEASURES

Time	Definition/Calculation
t^n_{match}	This is the time needed to execute a service discovery query against *n* services from the registry. This is calculated as, $t^n_{match} = t^n_{reg} + t^n_{struct} + t^n_{non\text{-}context}$, where – t^n_{reg} is the time needed to retrieve *n* services from theregistry. – t^n_{struct} is the time needed to evaluate the structural constraints of a query against *n* services. – $t^n_{non\text{-}context}$ is the time needed to evaluate non contextual constraints of a query against *n* services.
$t^n_{SLA\text{-}Neg}$	This is the time needed to perform SLA negotiation with *n* services.
$t^n_{SLA\text{-}Act}$	This is the time needed to activate a pre-agreed SLA.
$t_{RS\text{-}Del}$	This is the time needed to delete a service from the candidate service set (RS) and pick up the best service from RS.
$t_{RS\text{-}Add}$	This is the time needed to add a new service to RS and/or sort the services within RS according to their total distance to the query used to build RS
$t_{rep\text{-}*}$	This is the time needed to select an alternative service for replacement due to *unavailability of a service*. This time is calculated as follows, – *SD Only* case: $t_{rep\text{-}sd} = t_{RS\text{-}Del}$ – *SD with Proactive SLA* case: $t_{rep\text{-}sd\text{-}pro\text{-}sla} = t_{RS\text{-}Del} + t_{SLA\text{-}Act}$ – *SD with Reactive SLA* case: $t_{rep\text{-}sd\text{-}rea\text{-}sla} = t_{RS\text{-}Del} + t^1_{SLA\text{-}Neg} + t_{SLA\text{-}Act}$
$t_{rsm\text{-}*}$	This is the time needed for runtime maintenance of the candidate service set (RS) due to arrival of a new service or *change in the specification of an existing service*. This time is calculated as follows, – *SD Only* case: $t_{rsm\text{-}sd} = t_{RS\text{-}Add} + t^1_{match}$ – *SD with Proactive SLA* case: $t_{rsm\text{-}sd\text{-}pro\text{-}sla} = t_{RS\text{-}Add} + t^1_{match} + t^1_{SLA\text{-}Neg}$ – *SD with Reactive SLA* case: $t_{rsm\text{-}sd\text{-}rea\text{-}sla} = t_{RS\text{-}Add} + t^1_{match}$
AG_{pro}	This is the average time gain per service replacement in case of *SD with Proactive SLA* over *SD with Reactive SLA*. This is calculated as,$AG_{pro} = avg(t_{rep\text{-}sd\text{-}rea\text{-}sla}) – avg(t_{rep\text{-}sd\text{-}pro\text{-}sla})$
RG_{pro}	This is the ratio of the service replacement time with reactive negotiation over the service replacement time with proactive SLA negotiation, i.e.: $RG_{pro}=avg(t_{rep\text{-}sd\text{-}rea\text{-}sla})/avg(t_{rep\text{-}sd\text{-}pro\text{-}sla})$
BE_{pro}	The ratio of the overhead of maintaining the replacement service set (RS) in case of *SD with Proactive SLA*over the gain in the time for service replacement with*SD with Proactive SLA* over *SD with Reactive SLA*, measured as: $BE_{pro}= (\sum t_{rsm\text{-}sd\text{-}pro\text{-}sla} – \sum t_{rsm\text{-}sd\text{-}rea\text{-}sla})/(\sum t_{rep\text{-}sd\text{-}pro\text{-}sla} – \sum t_{rep\text{-}sd\text{-}rea\text{-}sla})$

Table II presents the time needed to build the initial replacement services set RS. As expected, the total time required for building the RS set for a given service in the case of service discovery with proactive SLA negotiation is longer than the time required for building the same set in the cases of service discovery without SLA negotiation and service discovery with reactive SLA negotiation. This difference was observed across all the different sizes of service registries and occurred because when service discovery with proactive SLA negotiation is used, an SLA should be negotiated and (possibly) pre-agreed with each candidate service, whilst when the initial construction of the RS set is based on service discovery only or on service discovery with reactive negotiation, no SLA negotiation is required.

Overall, during the initial phase of building the RS set, the use of proactive negotiation has an overhead between 8% (500 services) and 7% (100 services) of the time required for pure service discovery. However, it should be noted that the initial phase for building RS is performed only once for each subscribed query, and in parallel to the execution of the service client application.

The time needed for maintaining the replacement service set (RS) and selecting a replacement service due to events of *type_1, type_2,* and *type_3* is shown in Table III. The time measures shown in the table are averages (and total sums where applicable) taken across five different executions of the discovery query for each of the three event types.

As shown in Table III, the time required to select a replacement service in case of service discovery with proactive SLA negotiation is slightly larger than the time required to identify a replacement service if service discovery without any SLA negotiation is used. This is because in the former case, the pre-agreed SLA needs to be activated before the replacement service is returned. It should be noted, however, that the main benefit shown in the table is that the time required to select and bind a replacement service at runtime in the case of service discovery with reactive SLA negotiation is significantly larger than the service selection and binding time in the case of service discovery with proactive SLA negotiation: 53.2 vs. 453 milliseconds for the registry with 100 services, 45.8 vs. 459.4 milliseconds for the registry with 300 services, and 50.2 vs. 443.8 milliseconds for the registry with 500 services.

Table IV shows the aggregate effectiveness of proactive SLA negotiation in absolute and relative terms by presenting the AG_{pro}, RG_{pro}, and BE_{pro} measures. More specifically, as shown in the table, the average gain in service replacement time of service discovery with proactive SLA negotiation over service discovery with reactive SLA negotiation is between 187 ms and 230 ms per service replacement (see the AG_{pro} column in the table).

TABLE II. TIME MEASURES FOR BUILDING CANDIDATE SERVICE SET (TIMES IN SECONDS)

	SD Only			**SD with Proactive SLA**			**SD with Reactive SLA**		
	100	***300***	***500***	***100***	***300***	***500***	***100***	***300***	***500***
t^n_{reg}	152.062	383.29	642.464	138.152	378.995	620.312	151.447	382.888	643.858
t^n_{struct}	8.39	24.326	43.561	8.046	24.058	40.78	8.17	24.712	43.687
$t^n_{non\text{-}context}$	0.172	0.422	0.657	0.172	0.422	0.657	0.172	0.421	0.672
$t^n_{SLA\text{-}Neg}$	–	–	–	11.029	32.495	53.936	–	–	–
Total	160.624	408.054	686.697	157.399	435.985	715.685	159.804	408.037	688.232

TABLE III. PERFORMANCE MEASURES FOR MAINTAINING CANDIDATE SERVCIE SET AND SERVICE REPLACEMENT (TIMES IN MILLI-SECONDS)

		SD Only			SD with Proactive SLA			SD with Reactive SLA		
		100	**300**	**500**	**100**	**300**	**500**	**100**	**300**	**500**
Replacement Service Set (RS) Maintenance	$avg(t_{rsm\text{-}*})$	690.5	698.3	673.5	837.2	881.2	892	637.4	668.5	677.6
	$\sum t_{rsm\text{-}*}$	6905	6983	6735	8372	8812	8920	6374	6685	6776
Selection of Replacement Service from RS	$avg(t_{rep\text{-}*})$	22	28	21.8	53.2	45.8	50.2	453	459.4	443.8
	$\sum t_{rep\text{-}*}$	110	140	109	266	229	251	2265	2297	2219

Also, in relative terms, the service replacement time with reactive SLA negotiation presents an 8.5 to 10-fold increase over the service replacement time when proactive SLA negotiation is applied (see the RG_{pro} column of the table). This relative increase is anything but negligible when considering that the need for service replacement arises whilst an SCA is in operation. Also, the cost of maintaining the replacement service set (RS) does not exceed the gain achieved by service discovery with proactive SLA negotiation (see the BE_{pro} column in Table IV).

Overall, albeit preliminary, our experiments have shown that proactive SLA negotiation can provide a substantial improvement of the time that will be required for dynamic replacement of services when agreed SLAs must be in place before using a service.

TABLE IV. EFFECTIVENESS OF PROACTIVE SLA NEGOTIATION

Service Registry	AG_{pro}	RG_{pro}	BE_{pro}
100	187	8.51	0.999
300	230.4	10.03	1.028
500	192.8	8.84	1.089

It should also be noted that although proactive service discovery and SLA negotiation are essential for achieving efficient service replacement at runtime, they also create the possibility of inefficient resource utilization. More specifically, the efficiency of resource utilization with proactive SLA discovery/negotiation over a period of time T can be measured by the formula:

$$U = \frac{T \times SRRR \times (t_{match} + t_{SLA\text{-}Neg})}{T \times SRUR \times (t_{match} + t_{SLA\text{-}Neg}) + t_{init\text{-}RS}}$$

In this formula, SRRR is the service request replacement rate; SRUR is the service registry update rate; t_{match} is the average time required to match a query with a service, $t_{SLA\text{-}Neg}$ is the average time required to negotiate an SLA with a service; and $t_{init\text{-}RS}$ is the time needed to build the initial copy of RS ($t_{init\text{-}RS}=R_{init}\times$ (t_{match} + $t_{SLA\text{-}Neg}$ where R_{init} is the number of services in the service registry at the time of the initial build of RS).

Hence, to have efficient resource utilization when deploying proactive service discovery and negotiation, it should be that SRRR $\geq$ SRUR + R_{init}/T or that SRRR $\geq$ SRUR since the factor R_{init}/T becomes arbitrarily close to zero as T increases. This means that the service replacement request rate must be higher or at least equal to the service registry update rate. Establishing the validity of this condition would require a long-term study. However, it is not unreasonable to expect that the condition SRRR $\geq$ SRUR holds in the long term.

VII. RELATED WORK

Proactive approaches to dynamic adaptation of service-based applications are increasingly appearing in the literature [16][23]. Most of the work in this area, however, focuses on mechanisms for forecasting operational problems that may require adaptation (see [16][23] for example) rather than focusing on proactive SLA negotiation. Also, existing work on SLA negotiation tends to focus on the mechanics of the negotiation process itself (e.g. [4][10]) rather than wider procedural issues as to when and under what conditions the negotiation process may be triggered.

An agent based framework for SLA management is presented in [9]. In this framework, an initiator agent from the service consumer's side and a responder agent from the service provider's side take part in the negotiation process. The responder agent advertises the service level capabilities and the initiator agent fetches these advertisements and initializes the SLA negotiation process. Different stages of SLA life cycle e.g. formation, enforcement and recovery is performed through the autonomous interactions among these agents. In the case of an SLA violation, the initiator agent may either claim compensation and renegotiate with the service provider or select a new service provider. The provision of compensation in case of violation of SLA is also the focus of [1]; an approach focusing on several aspects of compensations such as the legislation that is applicable in cases a conflict between the provider and the consumer of a service, and the impact of the penalty clauses on the choice of service level objectives.

Runtime SLA re-negotiation has been suggested in [2][3][4][7][5] to manage SLA violations. In [2] service level objectives are revised and renegotiated at runtime and deployed services are adjusted to dynamically agreed service level objectives. A similar approach allowing the change of service level objectives whilst keeping the existing SLA is described in [5]. In [3] a renegotiation protocol is described that allows the service consumer or service provider to initiate renegotiation while the existing SLA is still in force when this becomes necessary for service providers or consumers for different reasons (e.g., changes in the business requirements of a party).

Note, however, that all the above approaches are reactive, i.e., renegotiation starts only after an existing SLA is violated. Hence, they do not address the main problem that is the focus of our work, i.e., the development of a proactive SLA negotiation approach that can increase the chances of uninterrupted service provision when SLA negotiation is required at runtime. Furthermore, our framework integrates SLA negotiation with dynamic service discovery and it can, therefore, provide integrated runtime support for both these key activities, which is necessary for achieving runtime service based application with minimized interruptions.

VIII. CONLUSIONS AND FUTURE WORK

In this paper, we have presented a framework that integrates service discovery with proactive SLA negotiation, called PROSDIN.

The identification of alternative services in PROSDIN is based on various characteristics of published services including structural, behavioural and QoS characteristics. PROSDIN also negotiates a service level agreement over QoS levels with each alternative service identified by the discovery process. The negotiation process is carried out according to a two-phase protocol and may result in a provisionally agreed but not activated SLA or negotiation failure. A provisional SLA has an expiry date by which it should either be activated or cease to exist.

The objective of proactive SLA negotiation in PROSDIN is to ensure that a service, which could be potentially used by a service client application, will have an agreed set of guaranteed provision terms if the need to deploy it arises at runtime. Hence, when this need arises it won't be necessary to engage in a lengthy negotiation process interrupting the operation of the service client application.

Our approach has been evaluated through an initial set of experiments showing that proactive SLA negotiation leads to significant reduction of the time required to perform service replacement at runtime if the existence of agreed SLAs is a prerequisite for service use.

PROSDIN opens a spectrum of possible lines for future investigation. These include support for proactive negotiation of hierarchical SLAs, i.e., SLAs of complex composite services deploying other composite services with their own sub-SLAs which will need to be negotiated separately and before coming to an higher level service level agreement. Other aspects for further investigation include the use of heuristics for tuning the triggering the proactive SLA negotiation process so as to reduce the number of cases where pre-agreed SLAs never get used, and the study of the performance of the framework when the negotiation rules used by different participants might change dynamically.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's 7th Framework Programme under Grant Agreement 215483 (S-Cube).

REFERENCES

[1] O. Rana, et al, "Managing Violations in Service Level Agreements", Work.on the Usage of Service Level Agreements in Grids, 2007

[2] G.Di Modica,O. Tomarhio and V.Lorenzo, "A framework for the management of dynamic SLAs in composite service scenarios", Service-Oriented Computing - ICSOC 2007 Workshops.

[3] M. Parkin, P. Hasselmeyer, B. Koller, and P. Wieder."An SLA Re-negotiation Protocol", 2nd Work. on Non Functional Properties and SLA in Service Oriented Computing at ECOWS 2008.

[4] O.Waeldrich,and W.Ziegler. "A WS-Agreement based Negotiation Protocol", Technical Report, Fraunhofer Institute SCAI, VIOLA - in DFN. 2006

[5] R. Sakellariou and V. Yarmolenko, "On the Flexibility of WS-Agreement for Job Submission", 3rd Int. Work. on Middleware for Grid Computing, 2005

[6] R. B. Doorenbos, "Production Matching for Large Learning Systems", January 31, 1995, CMU-CS-95-113

[7] P. Wieder, J. Seidel, O. Wäldrich, W. Ziegler, and R. Yahyapour, "Using SLA for Resource Management and Scheduling - A Survey", In Grid Middleware and Services Challenges and Solutions, Springer, 2008

[8] F. Raimondi, J. Skene, L. Chen, and W.Emmerich, "Efficient monitoring of web service SLAs", Research Notes (RN/07/01). UCL, London, UK. 2007

[9] Q. He, J. Yan, R. Kowalczyk, H. Jin, and Y. Yang, "Lifetime Service Level Agreement Management with Autonomous Agents for Services Provision", Information Sciences, Elsevier, 2009

[10] P. Hasselmeyer, et al. "Towards Autonomous Brokered SLA Negotiation". eChallenges 2006

[11] P. Karaenke and S.N Kirn. "Service Level Agreements: Evaluation from a Business Application Perspective", eChallenges 2007

[12] A.Pichot, P. Wieder, W. Ziegler, and O.Wäldrich, "Dynamic SLA-negotiation based on WS-Agreement", CoreGRIDTechnical Report, TR-0082, 2007

[13] V. Robu, D.J.A. Somefun, and J. A. La Poutre. "Modeling complex multi-issue negotiations using utility graphs", 4th Int. Conf. on Autonomous Agents & Multi Agent Systems, 2005

[14] K. Kritikos and B. Pernici, "Initial Concepts for Specifying End-to-End Quality Characteristics and Negotiating SLAs". S-Cube Project Deliverable CD-JRA-1.3.3, June 2009.

[15] A. Zisman, G. Spanoudakis, and J. Dooley. "A Framework for Dynamic Service Discovery", 23rd Int. IEEE/ACM Conf. on Automated Software Engineering, 2008.

[16] J.Hielscher, R.Kazhamiakin, A.Metzger,and M.Pistore, "A Framework for Proactive Self-Adaptation of Service-based Applications Based on Online Testing", ServiceWave 2008

[17] E. Friedman Hill, "Jess in Action: Rule Based Systems in Java", Manning Publications Co, ISBN 1-930110-89-8

[18] eXist. http://exist.sourceforge.net.

[19] Experiment Specifications, http://www.soi.city.ac.uk/~am697/sla/Case-Study-Specification.zip

[20] SEEKDA Web Services, http://seekda.com/en

[21] JESS: Java Expert System Shell. See http://herzberg.ca.sandia.gov/jess/.

[22] SECSE Project. http://secse.eng.it

[23] P.Leitner, A.Michlmayr,and S. Dustdar, "Monitoring, Prediction and Prevention of SLA Violations in Composite Services", IEEE Int. Conf. on Web Services 2010

[24] Mahbub K. and Spanoudakis G., "Proactive SLA Negotiation for Service Based Systems," 6th World Congress on Services, pp.519-526, 2010