RC22800 (W0305-082) May 16, 2003
Computer Science

IBM Research Report

Compatibility Analysis of WSLA Service Level Objectives

Weilai Yang*, Heiko Ludwig, Asit Dan
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Georgia Institute of Technology

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Compatibility Analysis of WSLA Service Level Objectives

Weilai Yang[‡], Heiko Ludwig[?], Asit Dan[?]
[?]IBM T. J. Watson Research Center
[‡]Georgia Institute of Technology

Abstract

In On Demand Service environments, both a client and a service provider need to make sure of their common agreements on the quality of service (QoS), i.e., performance, reliability and availability, prior to accessing a managed service. This is due to the fact that varied and customized requests of individual clients may not always match the QoS supported by an autonomous service. To mitigate such a potential incompatibility, an automated analysis is necessary to make sure that specific QoS requirements by a client can be satisfied by the QoS supported by a service. In different scenarios, such an analysis can be performed either by the client, or the provider or both. Note that even if both the client and service use a common syntactic expression language, semantic differences in supported/requested metrics and service level expressions make this a very challenging problem. This paper presents an algorithm for automated compatibility analysis assuming QoS is expressed using a rich and flexible specification language such as the Web Service Level Agreement (WSLA).

1. Introduction

On Demand services are understood to be IT and business services that customers buy from service providers when the services are needed and for the specific amount of time the service is needed. This dynamic process of setting up and disposing of service relationships between customers and providers requires efficient means for providers to advertise their services, for potential customers to search and find available services, and to negotiate a service contract upon which the relationship will be based. An important criterion of differentiation of a service, over and above what the service is about, are its quality properties in terms of performance (e.g., response time), availability (downtime), and others. Contracts on the quality properties of a service are often called Service Level Agreements (SLAs).

The process of acquiring an on demand service at a required quality of service involves multiple steps. A simple model of setting up a service relationship is illustrated in the next figure.

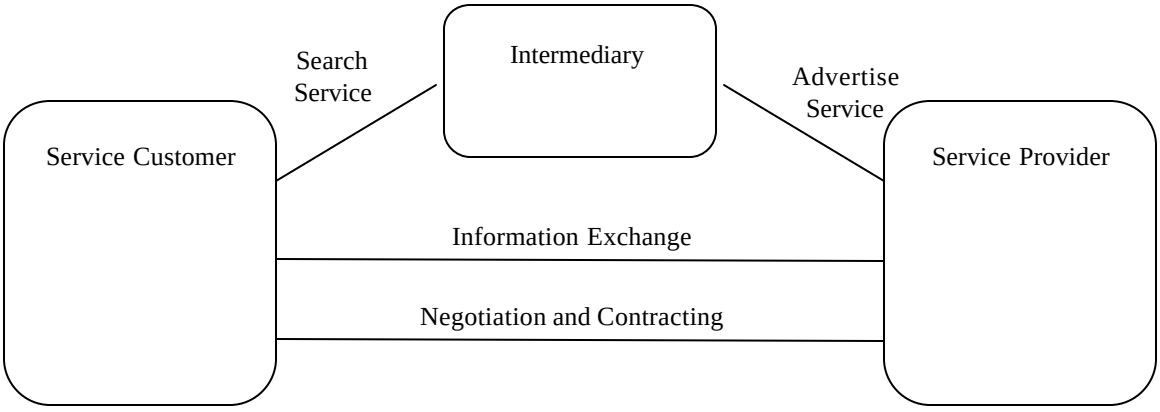

Figure 1: Relationship establishment in an open environment.

For a service customer and a service provider to get in contact, they need a common point of contact, typically a

directory service such as UDDI or a well known electronic marketplace in a particular industry such as Covisint in the automotive environment. A service provider organization can advertise its services for they can be found by customers searching using this intermediary. Further information exchange can take place on a bilateral basis between customer and provider such that both parties can base their decision-making on case-specific information that is not available at an intermediary. Finally, customer and provider can negotiate the details and sign the contract.

In the course of the interaction, the compatibility of the client's request and the provider's offer are compared at multiple steps. The intermediary compares the customer's search with the provider's advertisement. Furthermore, both provider and customer match properties and requirements having received additional, case-by-case information. Depending on the details of the advertisement submitted to an intermediary, the match-making task may predominantly take place in the intermediary or within customer and provider. In a common situation, a service description of a provider is matched against a request of customer, while, in a more general case, both providers and customers may describe properties and have requirements on the service and their business partners [2].

There are a number of options to describe service properties in general. In a Web services environment, WS-Policy provides a general mechanism for expressing service properties, i.e., supported as well as constraints in accessing a service. QoS properties in particular can be expressed using various options, e.g., Web Service Level Agreement (WSLA) language [3], [4] and the language proposed by Sahai et al. [6], which can also be used for representing SLAs. In an open marketplace, quality of service guarantees of resources will be described using different metrics, depending on what appears relevant to the resource owner and of the particular implementation of this resource. For example, for a particular application on a resource, one resource owner may be able to give a guaranteed response time to a request of 2 seconds, while another resource owner may guarantee that 95% of all requests can be responded to in 2 seconds. Hence, a similar QoS guarantee is expressed using slightly different metrics, i.e. the number of seconds and a percentile.

The WSLA language, on which the discussion in this paper will be based, facilitates the use of customer metrics enabling resource owners to describe QoS guarantees in two steps:

1. Metrics subjected to guarantees can be custom metrics described as functions of other metrics, either custom or commonly understood metrics provided by a resource's instrumentation, e.g., counters and gauges.
2. Based on these custom metrics, service level objectives (SLOs) are defined as logic expressions over metrics.

While custom metrics are beneficial in exactly describing the properties of a service, it makes the match-making process of requirements and properties more difficult. In the case discussed above, let's assume that a managing element requires a response time of less than 2 seconds for 90 percent of all requests. From our understanding of the subject matter, both the hard guarantee and the percentage guarantee satisfy the requirements of the managing element. However, simple match-making algorithms are not applicable in such a scenario.

In this paper, we introduce a match-making algorithm that facilitates the comparison of SLOs that are based on related but differently expressed custom metrics. We proceed as follows: In the next section, we introduce the expressions of the WSLA language for the definition of metrics and SLOs. Subsequently, we describe the match-making algorithm. We illustrate the algorithm in section 4 and introduce some optimization approaches in section 5. We conclude the article by a summary and discussion.

2. Structure of SLO and Metrics expressions

2.1. SLAs and SLOs

We assume that both properties of a resource and resource requirements are expressed in the WSLA language. To check whether a resource fulfills a managing element's requirements, these WSLA language expressions are matched by the managing elements. The managing element can iterate through the resources of resource owners known to it.

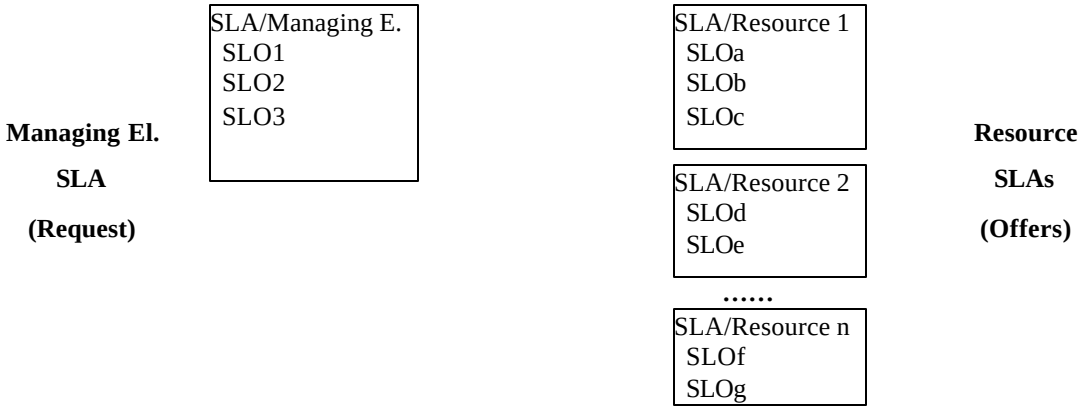

Figure 2: SLAs and SLOs of the managing element and resources

The most important element of a WSLA SLA is its set of SLOs, describing particular guarantees related to performance characteristics of a resource. The managing element defines performance requirements related to a resource as a set of SLOs. One SLO could define the average response time of a resource at a particular throughput level; another could address the required availability of this resource. Likewise, resource owners describe the performance properties of their resources as a set of SLOs that have the same structure than the SLOs in the managing element's SLA.

For a resource to meet the requirements of the managing element's request, all SLOs of the request have to be satisfied. The resource owner may give additional performance guarantees of a resource in additional SLOs, which are ignored by the managing element assembling resources.

WLSA documents can contain additional information that is not relevant in the context of self-management as discussed here.

2.2. WSLA language expressions

The WSLA language is a general-purpose means to express performance characteristics of services [5]. This includes a reference to the service whose characteristics are described, a detailed definition of the metrics that characterize the performance, and a definition of the service level objective. In addition, it contains a definition of the parties being involved in the SLA and a specification of their respective roles, which is generally important in a cross-organizational scenario but not necessarily in the context of self-managing systems. Furthermore, a WSLA can contain action guarantees, which are promises to perform an action if a particular situation occurs. Action guarantees are not subject of this discussion.

SLOs contain a number of elements, as outlined in the example in Figure 3:

```
<ServiceLevelObjective name="slo1">
 <Obliged>Resource1</Obliged>
 <Validity>
  <Start>2002-11-30T14:00:00.000-05:00</Start>
  <End>2002-12-31T14:00:00.000-05:00</End>
 </Validity>
 <Expression>
  <Implies>
   <Expression>
    <Predicate xsi:type="Less">
     <SLAParameter>TransactionRate</SLAParameter>
     <Value>10000</Value>
    </Predicate>
   </Expression>
   <Expression>
    <Predicate xsi:type="Less">
     <SLAParameter>AverageResponseTime</SLAParameter>
```

```
     <Value>0.5</Value>
    </Predicate>
   </Expression>
  </Implies>
 </Expression>
 <EvaluationEvent>NewValue</EvaluationEvent>
</ServiceLevelObjective>
```

Figure 3: Example Service Level Objective.

Besides a unique name, it contains the name of the resource addressed, a validity period, a logic expression describing the guaranteed state of the resource, and a specification when this SLO should be checked. In the example resource 1 is guaranteed from end of November until end of December to have the following properties: The SLAParameter (metric) average response time is less than 0.5 (e.g., seconds), given that the metric transaction rate is less than 10000 (e.g., transactions per minute). What the metrics mean is represented in the metric specification part of a WSLA document to which SLOs refer by using names of metrics in SLO expression definitions.

The **metric definition** part of WSLA describes how to measure or compute values that can be subjected to SLOs. Metric definitions may contain a measurement directive that describes how a value is measured from a system's instrumentation or how it is gained by probing a system from outside. Measurement directives are types of measurement instructions that are parameterized for an individual case, e.g., by giving an address to probe response time or availability. Furthermore, measurement directives contain a specification to what a measurement relates, e.g., the operation the response time of which is measured.

A specific type of measurement directive is used in our example, Gauge. Its attributes are the name of the gauge and the operation to which it belongs, optimize in the example case.

```
<Metric name="TXCount" type="integer">
 <Source> resource1</Source>
 <MeasurementDirective xsi:type="Gauge" resultType="integer">
 <Gauge>TXCount</Gauge>
 <OperatonName>optimize</OperationName>
 </MeasurementDirective>
</Metric>
```

Figure 4: Measurement directive example.

Metrics having measurement directives are called **resource metrics**.

Alternatively, metrics definitions can contain a function that describes how its value is computed from other metrics. These metrics are called **composite metrics**. Figure 6 shows an example composite metric containing a function.

```
<Metric name=" AverageResponseTime" type="double" unit="seconds">
 <Source>resource1</Source>
 <Function xsi:type="Divide" resultType="double">
  <Operand>
   <Metric>TimeSpentTS</Metric>
  </Operand>
  <Operand>
   <Metric>TXCountTS</Metric>
  </Operand>
 </Function>
</Metric>
```

Figure 5: Composite metric example.

This example describes the metric average response. The metric is of type double and its unit is seconds. In its function definition, the function Divide is applied to two operands: The time spent processing the optimize operation is divided by the number of times it has been invoked. Operands of functions can be metrics, scalars and other functions.

Besides its function or measurement directive, all metrics have a name, a type and a unit. WSLA language expressions can be computed and evaluated using the WSLA compliance monitor [2].

The syntax of WSLA SLO expressions has a hierarchical structure as outline in Figure 6.

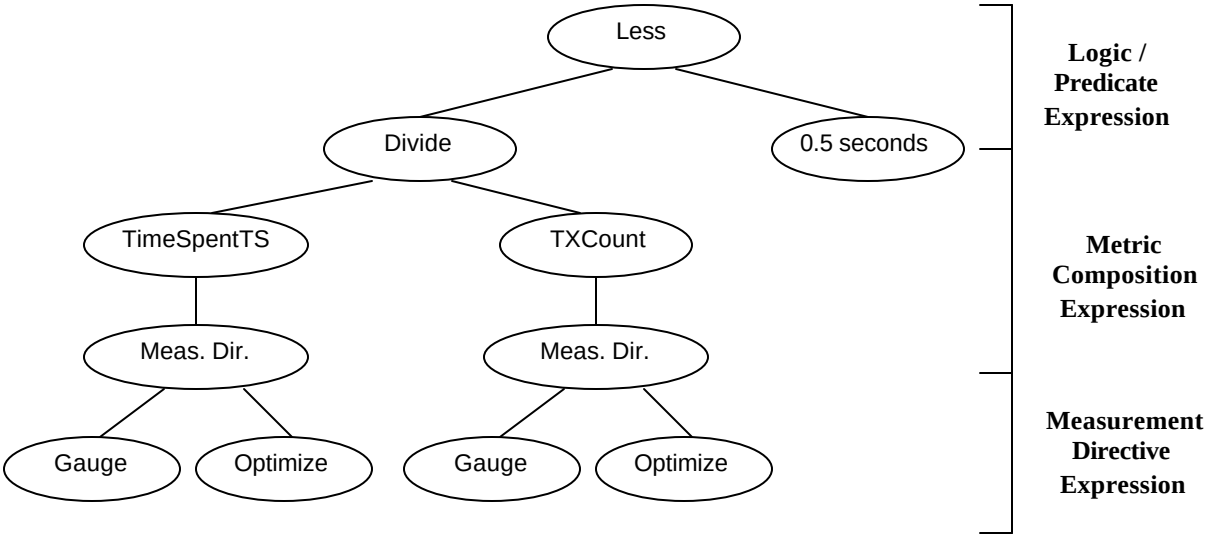

Figure 6: Visualization of a WSLA SLO syntax tree.

Ignoring some aspects such as validity periods and periods, which can be dealt with easily, the syntax tree has three layers. On top is the logic expression, here just the predicate less with its two operands. The operands can be either constants or they are metrics. This layer of the tree corresponds to SLO expressions. The next layer is the metric composition layer. It is defined using one or multiple composite metric definitions, a division of two metrics in this case. Finally, the leaf structures are measurement directive expressions that both refer to a gauge relating to an optimize function.

2.3. Heterogeneous definition of SLAs

Matching SLOs of the managing element and resources can be performed by comparing their respective syntax trees. Using the expressive WSLA language, however, SLOs can be described in many different ways although the same semantic content is meant. While, in general, this issue is difficult to address, e.g., if the same terms are called differently, some problems can be dealt with if different ways to describe a constraint embody that one constraint is weaker than another. For example, a resource may offer a response time of 2 seconds if the number of requests is less than 100 per minute, giving an absolute guarantee. The managing element may require that more than 95% of all requests are served in less than 2 seconds. While the structure of both SLO, phrase in WSLA, is entirely different, it is apparent that the resource fulfills the requirements of the managing element.

We call functions that "weaken" requirements metrics **weakening functions**. Let m be a metric, p and q be predicates, f be a function, and ps_1 and ps_2 be parameter sets.

The function f is weakening if a predicate q exists for which holds: p(m, ps1) ? q(f(p(m, ps_1)), ps_2).

In the case given above, m is response time and p is less. The function f is the percentile function applied to p(m, …). The predicate q is less, stating the "less than 95%".

The match-making algorithm proposed in the next section deals with heterogeneous SLO definitions based on weakening functions.

3. Match-Making Algorithm

3.1. Overview and Approach

The match-making algorithm compares an SLA of a managing element, also referred to as **customer**, with an SLA offered by a resource owner for a resource, also referred to as **service provider**. In its main loop, the algorithm iterates over all SLOs of the customer's (managing element's) SLA. For each customer SLO, the algorithm iterates over the resource (service provider) SLOs and matches the customer's SLO against the provider's. If a matching provider SLO is found, the algorithm proceeds to the next customer SLO. If no customer SLO matches. The algorithm stops and returns that this resource SLA does not match. If a match is found for all customer SLOs, the provider SLO matches.

The novel approach lies in the matching of individual SLOs. We must determine whether the guarantee given in the provider SLO is "stronger" than the one requested by the customer, i.e. the customer SLO is always true if the provider SLO is true. The approach that is taken by the algorithm is to compare the syntax trees of the SLOs. We need to address two important issues:

1. How do we capture the semantics of predicates? We need to understand that an expression "response time < 2" is better than "response time < 5". For this purpose, we assume that the algorithm is aware of the semantics of the common comparators such as <, <= and the like. For additional predicates, we can introduce a separate evaluation function better($p1(ps_1)$, $p2(ps_2)$) that evaluates for a domain-specific predicate, whether an expression is better than another.
2. How do we address structurally different phrasing on the basis of weakening functions as introduced above? The approach that we take in our algorithm is to use specific knowledge about functions, such as the function "percentage less than threshold", to restructure the customer request syntax tree in a way that it has the same structure than the stronger, more demanding syntax tree of the provider SLO. Then regular comparison is resumed.

The SLO match-making algorithm presented in the next section addresses these issues.

3.2. SLO Match-Making Algorithm

The inputs into the algorithm are the customer SLO and the resource SLO. The two SLO trees are each examined to decide if they are compatible.

The algorithm assumes that the inputs are proper SLO trees. This guarantees that:

- The root node is a predicate operator (<, >, etc.)
- The left child of the root is not a constant.
- For each function type (average, percentage less than threshold …) there is a tree reconstruction function defined.

3.2.1 Algorithm Description

Suppose there is one Customer's SLO and one resource owner's SLA and we are trying to figure out the compatibility between them. As shown in Figure 7, we walk simultaneously from the root of the two trees downwards to compare the node one by one.

1) If both the first nodes are implication, we will get the left trees and start from 1) again.
2) If the first nodes are the same, go to 5). Otherwise, if the left children are the same, it is not compatible. Go to 7)
3) If the left children are different, refer to Table 1, which specifies the weakening relationships among all related functions. If the resource owner's function is weaker, the SLO trees are incompatible and the algorithm returns.
4) Otherwise, if the customer's function is weakening, the syntax tree of the customer SLO is reconstructed from the sub-tree representing the function p (as in the definition of weakening) for further compatibility testing. In order to make the sub-tree into a proper SLO tree, we have to reformat the function into a new

predicate. The procedure for the reformatting depends on which function it is, and is listed in a table. Now we go back to Step 2 using the new tree as the customer SLO.

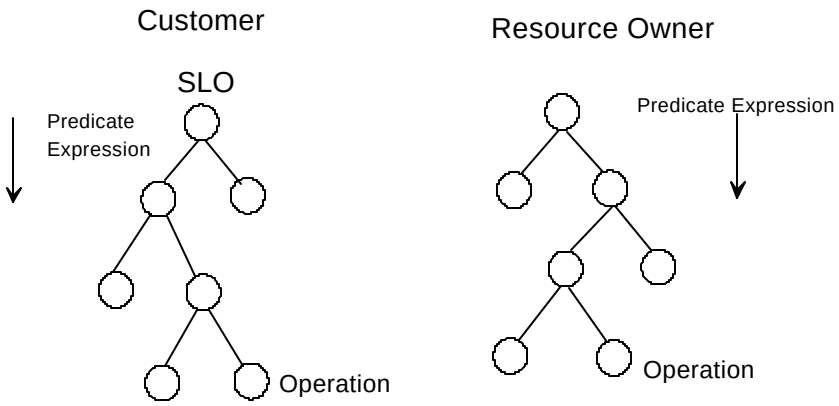

Figure 7: The comparison of SLO between customer and Resource Owner.

5) If the left children of the roots of the two trees do not match, go to step 4. Otherwise, do a look up Table 2 for the comparison required to decide which SLO is a stronger predicate expression. If there is no definition for it or the resource owner's SLO is weaker, they are not compatible.

6) Otherwise, keep traversing both SLO trees to check the measurement directive expressions. If they are the same, it's concluded that customer's and resource owner's SLO trees are compatible. Otherwise, they are not compatible.

7) If current trees are not original trees' children trees, the algorithm returns here.

8) Otherwise, if they are left child trees and they are compatible, get both the original roots' right child trees and start from 1); if they are left child trees and they are not compatible, return true as the final result; if they are right child trees and they are compatible, return compatible; if they are right child trees and they are not compatible, return no compatible.

Table 1 shows the relationships among functions. For example, when two different functions are detected from the tree, in order to decide which one is a weakening function, Table 1 provides this precedence information.

Functions	RT, ART, PLTT, PGTT
Relationships	RT>ART>(PLTT, PGTT)

Table 1: Relationships among functions.

Table 2 shows for each function, if it has different numbers to be compared, how to decide which one has the looser condition. For example, Resource owner has RT1 < 0.5s, a customer proposes RT2 < 1s. When we refer to table 2 second row, X1=0.5s and X2=1s, the comparison result is that RT1>RT2, which means resource owner can provide the service to the customer (it's compatible).

FunctionName	PredicateExpression	SLO1	SLO2	Comparison	Comparison Results
RT/ART	<	RT1/ART1 < X1	RT2/ART2 < X2	X1 < X2	RT1/ART1> RT2/ART2
Perc	>	Perc1 > Y1	Perc2 > Y2	Y1 > Y2	Perc1 > Perc2
Perc	<	Perc1 < Y1	Perc2 < Y2	Y1< Y2	Perc1 > Perc2

Table 2: Comparison principle.

3.2.2 Elaboration

At the root of each tree is a predicate operator (<, >, →). If the 2 operators are the same, and in addition, the left argument is the same, then we can compare the right arguments to decide which SLO statement is weaker. For example, in the case of "response time<2s" and "response time<5s", the common operator is <, and the left arguments are both response time. For the operator <, the SLO with the smaller right argument is weaker, thus "response time<2" is weaker. But for other operators (>, for example) it may be different. Thus we have a table (Table 2) mapping each predicate operator to another comparison that chooses the stronger SLO. Take a closer look at Table 2, predicator operator indexes the table, which returns a comparison given in the comparison column SLO1 and SLO2 look the way they do in their corresponding columns. If the comparison returns true, then the SLOs are compatible, otherwise they are not.

If the predicates operators are different but the left arguments the same, we stop the processing and deem the SLOs incompatible. In our current implementation, this is valid since we only have predicate operators > and <. For example, if we are given "response time>X" and "response time<Y", neither can be said to be a stronger claim than the other.

If the predicate operators are different and the left arguments are different functions, it is still possible that the customer SLO has a weakening function that matches the resource SLO. We check for this case by recursively matching the resource SLO with the branch of the customer SLO through reconstruction. We assume that for each reconstructable function, we know how to extract the parameters from the functions and construct a new SLO tree. The new tree will be used to compare with resource owner's SLO. The procedure runs recursively until it's declared there is compatibility or not.

Finally the algorithm has to go down to measurement directive expression level of the functions to make sure the operation types, etc. are the same so that the expressions are the same, before we can claim customer and resource owner's SLOs are compatible.

3.3. Example

To better understand the algorithm, we go through the example outlined in Figure 8.

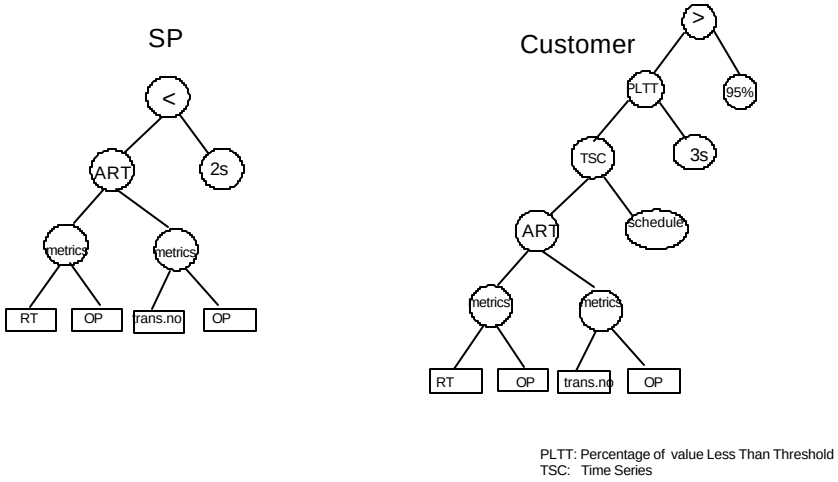

Figure 8: Example of tree comparison.

First we compare the predicate operators, which are different (< and >). Then we compare their left children. Since they are different (ART and PLTT), we look up Table 1 to find the weakening relationship between them. Table 1 tells us that ART has a stronger condition than PLTT, so we need to reconstruct a new SLO tree for the customer and test compatibility again. Reconstruction depends on the semantics of the weakening function, in this case PLTT. PLTT means percentage less than a threshold, so the predicate operator of the new tree should be "less than". We transfer the right child of PLTT, to the new SLO tree as the right child of "less than". In the original customer SLO, PLTT acts another function — ART, also expressed as a tree. Putting these pieces together, we reconstruct the SLO tree which now looks like:

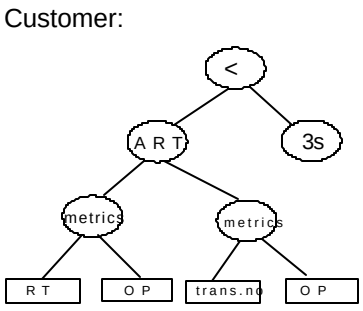

Figure 9: Customer's tree after reconstruction.

Now we restart the whole procedure again using the reconstructed customer SLO. We find that the predicate operators of the SLOs are the same, and that the left children are the same as well. We look up Table 2 for ART. The policy there shows that the smaller the right argument of ART, the stronger the condition is. In this case, the customer's argument is larger, thus the resource owner has a stronger condition than the customer. We still haven't completed the compatibility test because it might be possible that the two trees are built based on different operation types that are not compatible at all. Therefore, we need to keep traversing the tree till we get to the measurement directives. If those metrics are on the same operation type, then we claim the trees are compatible.

4. Optimization

The SLO match-making algorithm determines if a customer SLO and a resource owner SLO are compatible. The straightforward way to do a complete compatibility test between the two parties is to do an exhaustive search. When there are more than one SLO and/or SLA, walking through the whole tree for each combination of SLOs is tedious and does not scale. Our match-making algorithm traverses the tree from the root to leaves. If the two SLOs have different operation types, we can only find out after traversing down to the leave nodes. If somehow we can detect the difference between those leave nodes, no comparison needs to be done at beginning at all. We propose an algorithm to optimize this procedure.

The idea is to pre-select and eliminate those unqualified SLAs before we do the comparison in detail. We construct this in three levels. The first one is the highest-level outlook of the compatibility.

SLO Level Comparison:

Cust_SLA_Support(Cust_SLA, Owner_SLA sets)

Input: Customer's SLA, Owners' SLA sets

Output: The map of operation for resource owner

For each SLO

Find compatibility from resource owner's SLA sets

In doing a compatibility test, we have 2 aims. The first is that for each SLO in customer's SLA, we need to find at least one SLA from the resource owner to support it. The other is that the number of resource owners' SLAs that can support those SLOs service intermediary at run time. If one of the SLOs cannot be provided by any resource owner's SLA, we conclude this customer's SLA fails.

Next, we do the pre-selection for operation type. After this pre-selection done, we will have the compatible operation sets for customer(Cust) and resource owner (RO).

We say an operation can be supported if a SLO associated with this operation is compatible with RO's SLA. Before we do the comparison for SLOs, the first thing that we should check is the operation type compatibility. The reason is that no operation proposed by customer can be supported by any RO, we can drop this request immediately without going further. We can obtain the set of operations from the customer SLOs -- customer_setA and the set of the operations from the RO -- named RO_setB. The intersection of these two sets Cust_RO would be the operation set that customer requires and suppliers can support. Then we traverse all the SLOs from the customer. An SLO will be kept if at least one element in it appears in Cust_RO. Otherwise it will be simply dropped.

Now we have a smaller list of SLOs that's worth further testing. We start over from beginning of the list. For each operation that appears, if RO fails in providing this kind of service, this operation will be dropped from Cust_RO. The next time same operation appears, it doesn't have to go through test any more.

For example, assume Customer's SLOs are:

{(SLO_1: op1, op3),

(SLO_2: op3, op5),

(SLO_3:op4, op5),

(SLO_4: op1, op3, op6),

(SLO_5: op1, op2),

(SLO_6: op2, op4),

(SLO_7: op5, op8)}

RO's can provide: {op1, op3, op4, op6, op7}. The intersection of these two sets would be:

Cust_RO: {op1, op3, op4, op6},

Then, the customer's SLOs (after pruning) are: {(SLO_1: op1, op3), (SLO_2: op3, op5), (SLO_4: op1, op3, op6), (SLO_5: op1, op2), (SLO_6: op2, op4)}.

We start from SLO_1. First we check if op1 can be supported by RO. If so, move on to op3, otherwise, remove op1 from Cust_RO, which makes Cust_RO:{op3, op4}. When we get to SLO_5, we can drop it immediately since neither of the operation appears in Cust_RO. Whenever all operations in one SLO_i are met, SLO_i is marked as compatible.

After all these steps, the surviving customer SLOs will be the inputs to our compatibility algorithm.

5. Extensive Examples

Here is a complete example of the algorithm at work:

Our resource metrics are:

Periodical Time (PT) : the time interval between measurements.

Response Time (RT) : the time interval between the beginning and the end of an action.

Number of Transactions : how many transactions happen during a certain time.

Constant Time (CT) : a constant time interval.

As we said before, these resource metrics have to be combined with specified operations. These operation types are variables.

The SLA parameters that are derived from resource metrics are the following:

AverageResponseTime(ART):	The average response time of a time period ART = F(RT, no. of transaction)
TransactionRate(STR):	The number of transactions in a time interval submitted by customer. STR = G(CT, no. of transaction)
RealTransactionRate(RTR)	The number of real transactions in a given time interval. RTR=G(CT, no.of transaction)
Percentage of ART (PART):	The percentage of time in a time interval in which ART < threshold. PART = P(PT, ART)

We differentiate between Required TR(Req_TR), Submitted TR(STR) and Real TR(RTR). The reason is that the SP side wouldn't provide more service than the customer asks for. i.e. Req_TR>1000→RTR>1000. Meanwhile the customer should not expect more TR than what he submitted. That means that even if Req_TR=1000 at a certain time, if they only submitted 500, the SP will only provide 500.

The following example illustrates the issue of match-making provider offerings and customers' request:

Suppose a resource owner offers the following SLOs:

{(Req_TR>x→ RTR>x, (x∈int), data file transfer),

((Req_TR<1,000/sec) → (ART<1s), optimize)}

A managing element sends a request, asking for:

{(STR<y→ RTR<y ,(y∈int), data file transfer),

((Req_TR<500/sec) → (PLTT(1day, ART<1.2s)) > 95%, optimize)}

The managing element must decide whether this service provider can meet the customer's request. There are two

SLOs for each side(one on each line). The first is Req_TR>x→ RTR>x, which naturally matches with the customer SLO STR<y→ RTR<y. The next SLO involves TR, which both sides specified. We have defined that for SLO of the form TR<x, the one with the smaller x is the weaker condition. Therefore, TR<1000 is a stronger condition than TR<500. So the left part of the → are true. On the right side of the →, the SP has a bound on the value of ART, but customer is only interested in it being true for some percent of the time. Using the core algorithm specified in previous section, we construct a new SLO specification.

The new SLO represented by the reconstructed tree is: ((Req_TR<500/sec) → ART<1.2s, optimize). We now compare again and the customers function ART<1.2s is a weaker function. Finally, we check that both measurement directive expressions are "optimize". So these two SLOs are compatible.

6. Conclusion

In an on demand environment, the quality of service requested by a service customer must be matched with the supported quality of service offered by a service provider to select a suitable service provider. Due to the fact that varied and customized requests of individual clients may not always match the QoS supported by a service provider, we need support of a advanced algorithm that analyses the compatibility of the advertised service with a client's request. Such an analysis can be performed either by the client or the provider or both depending on the exact scenario. This paper presents an algorithm for automated compatibility analysis assuming QoS is expressed using the Web Service Level Agreement (WSLA). The analysis addresses the semantic differences in different types of QoS metrics and service levels.

The algorithm takes as input the service levels requested and supported by the service client and service, respectively. The service level objectives (SLOs) are specified as predicate expressions over service level parameters and their associated metrics. The WSLA XML documents can be parsed to create the inputs expressed as SLO tree. The algorithm is based on the matching of the syntax trees of the requested SLOs of a provider and the ones offered by resources. Given the different ways SLOs can be expressed, we introduce the notion of a weakening function that provides the basis of reconfiguring syntax trees to a homogeneous structure, which can be compared. Based on this match-making algorithm, service customres can determine suitable resources whose quality of service properties are described in an expressive SLA language such as the WSLA language.

References

[1] D. Gantenbein, L. Deri: Categorizing Computing Assets According to Communication Patterns. In: Proceedings of the Conference on Networking 2002, Pisa, Italy, May 19-24, 2002.

[2] S. Field, C. Facciorusso, Y. Hoffner, A. Schade, M. Stolze: Design Criteria for a Virtual Market Place (ViMP). Proceedings of the European Conference on Digital Libraries (ECDL 1998), pp. 819-832, Springer-Verlag, Berlin, Heidelberg, 1998.

[3] A. Keller, H. Ludwig: Defining and Monitoring Service Level Agreements for dynamic e-Business. In: Proceedings of the 16th USENIX System Administration Conference (LISA'02), November 2002.

[4] Keller, A., Ludwig, H., The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services, accepted for publication in: *Journal of Network and Systems Management, Special Issue on "E-Business Management"*, Volume 11, Number 1, Plenum Publishing Corporation, March, 2003.

[5] H. Ludwig, A. Keller, A. Dan, R. King: A Service Level Agreement Language for Dynamic Electronic Services. Proceedings of WECWIS 2002, Newport Beach, CA, pp. 25 - 32, IEEE Computer Society, Los Alamitos, 2002.

[6] A. Sahai, V. Machiraju, M. Sayal, A. An Moorsel, F. Casati: Automated SLA Monitoring for Web Services. In: Proceedings of IEEE/IFIP DSOM 2002, Montreal, Canada, October 2002.

Appendix

Algorithm 1

```
CompatibilityTestAlgorithm:
        CCTA(Tree1, Tree2)
```

```
Input: Tree1- Customer; Tree2 – Resource Owner
Output: Result (compatible or not compatible)
10: Start from the top of the tree, which is predicate expressions (PE1, PE2)
     Get Pe1 left child LC1, PE2 left child LC2
    If (PE1="?  " && PE2="?  ")
          Tree 1 ?  PE2's left subtree;
          Tree 2 ?   PE1's left subtree;
          If (Comp(Tree1, Tree2))
                Tree 1 ?  PE1's right subtree;
                Tree 2 ?   PE2's right subtree;
                If (Comp(Tree1, Tree2)) return True;
                Else
                return False;
                Fi
          Else return True;
    Else if (((PE1 != "?  ") && (PE2 = "?  ")) || (PE1 = "?  " && PE2 !="?  "))
          Return False;
    Else
          Comp(Tree1, Tree2);

Comp(Tree1, Tree2)
          If (PE1 != PE2)
                If (LC1== LC2)
                      Return "not compatible"
                      Break;
                Else
                      If (Recons(LC2)) Comp(Tree1, Tree2)
                   Else return False;
                      Fi
     Else
          While (Tree has more elements)
                If (LC1==LC2)
                      Get Right Children, RC1, RC2;
                      Look up the table 2;
                      If ((SP>=Customer) & (not operation type comparison)
                            LC1?  LC1's left child
                            LC2?  LC2's left child
                      Else if ((SP>=Customer) & (LC1,LC2 are operation types))
                            Return "Compatible"; break;
                      Else
                            Return "not compatible"; break;
                      Endif
                Else
                      If (Recons(LC2)) Comp(Tree1, Tree2);
                      Else return false;
                      Endif
                Endif
          Endwhile

Recons(Node):
          Extraction(Node)
          If no entry
                Return false;
          Else
                PE2?  Predicate;
                LC2?  Function;
```

```
                    Return true;

        Extraction(Rec_node):
                Function Rec_node has been defined with the core parameter Func &
                        Predicate expression Pre
                Return (Pre, Func);
```

Algorithm 2

```
Optimization:
                Cust_RO_OP_test(Cust_SLO_i, RO_SLA set)
                Input: Customer's SLOs, Resource owner's SLA sets
                Output: The supportable operation groups for SLO_i
                        For (i=1 → n) //build the intersection of Cust's and RO's SLO
                          If (OP_j ∈ SLO_i) && (OP_j ∈ RO_SLA_k) && (OP_j ∈ Cust_RO)
                                Put OP_i → Cust_RO
                        Endfor
                        For (SLO_j, j=1 → m)
                                //to further prune operations appearing in cust_slo
                         For each operation in SLO_i
                                If (OP_k ∈ Cust_RO)
                                  Compare customer's OP_k condition and SP's
                                  If (SP's is weaker)
                                          Eliminate OP_k from Cust_RO set
                                          Flag = false; Break;
                                  Else if (Flag == false) break;
                                Else banner = true;
                                  Fi
                                 Fi
                        Endfor
                        If (banner) SLO_i = true;
                        Else SLO_i = false;
                        Endfor
```