Optimally Distributing Interactions Between Composed Semantic Web Services

Ion Constantinescu, Walter Binder, and Boi Faltings

Ecole Polytechnique Fédérale de Lausanne (EPFL),
Artificial Intelligence Laboratory,
CH-1015 Lausanne, Switzerland
`firstname.lastname@epfl.ch`

Abstract. When information services are organized to provide some composed functionality, their interactions can be formally represented as workflows. Traditionally, workflows are executed by centralized engines that invoke the necessary services and collect results. If services are clustered (e.g., based on geographic criteria), locally routing intermediary results between services in the same cluster can be more efficient.

This paper has several contributions: First, it presents a framework allowing the execution of workflow parts to be mediated by special *execution sites*. Second, we describe a trigger-based mechanism allowing partial results to be routed between execution sites. Finally, we present an approach for optimally computing the distribution of workflow parts to execution sites accordingly to an integrated cost model for workflow execution. The model is created by merging cost-models provided by individual execution sites trough a Contract Net task brokering protocol. The models consider cost measures for service activation, parameter transfer, and service execution.[1]

Keywords: Service composition, service execution, distributed computing, invocation triggers, workflows.

1   Introduction

A considerable amount of recent research work has focused on the automated composition of agent and web services based on a semantic description of user requirements and service capabilities [7, 3, 4, 9, 2].

Interactions between individual services can be organized according to constraints (e.g., data dependencies) so that they provide some required functionality that none of the individual services could offer alone. The resulting interactions can be represented as a workflow which specifies in which order the individual services have to be invoked and how data has to be passed between these
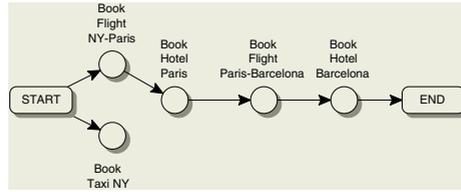

Fig. 1. Travel planning workflow

services. Some workflow specifications may include conditionals [2] or loops [3]. In this paper we consider workflows that do not have conditionals or loops. Also we do not consider transactions and semantic compensation of the effects of service invocations.

Usually, a workflow is executed in a centralized way, either by the client who needs the service or by a server that acts as workflow execution engine. While this approach gives complete control over the workflow execution to a single entity (which may monitor the progress), it may lead to inefficient communication, as all intermediary results are transmitted to the central workflow execution site.

When services are clustered together, transmission of the intermediary results to the client may significantly degrade the performance of the overall workflow execution. Consider the following travel scenario: On behalf of its user living in New York, a Personal Agent (PA) has to book travel arrangements for a trip to Paris and to Barcelona. For this purpose, a workflow as in Fig. 1 has been created (possibly by a semantic web services composition algorithm as in [2]) that books a plane ticket from NY to Paris, arranges a Taxi drive from the home of the user to the airport in NY, books a hotel in Paris, books a plane to Barcelona, and finally reserves a hotel in Barcelona. If we consider that the geographical distribution of the services to be invoked is reflected by the computational cost of the execution of the workflow (e.g., remote interactions between NY and Paris and Barcelona might be slow) the classical solution of centrally executing the workflow might not be the best.

In this paper we propose an alternate solution adapted to this kind of scenarios (clustered services). In our approach, the invocation of services listed in the workflow is not direct but rather mediated by an extra layer of execution sites situated closely to the services to be invoked. These sites are able to provide cost measures regarding the invocation of services and the transfer of partial results between execution sites. For managing the routing of partial results, execution sites use the concept of service invocation triggers, short *triggers* [1]. Each invocation of a service has an associated *trigger* configured on a given execution site. A trigger is able to collect the required input data before it invokes the service, i.e., triggers are also data buffers. Partial results can be sent to one or more triggers on the same execution site or on other execution sites, meaning that triggers also support multicast. According to the cost measures proposed by the execution sites, a workflow is optimally decomposed in several parts. Each

part is assigned to an execution site and triggers are configured for handling the required data-transfers.

Once the trigger for the first services in the workflow have received all input data, the associated services are invoked and the outputs are forwarded to the triggers of subsequent services. Consequently, the workflow is executed in a fully distributed way, the actual service invocations are done with small overhead due to the "closeness" of execution sites, and the data is transmitted directly from the producer sites to all consumer sites.

In the next section we present the formalism that we use for specifying different kinds of workflows and we list some of the assumptions that we use for the rest of the paper. Then in Section 3 we explain our approach to mediated and distributed workflow execution and we introduce the concept of service invocation triggers. In Section 4 we present an overview of the operation of our system, giving the main steps required for computing the optimal distribution of an workflow. Section 5 presents more details regarding how the integrated cost model is computed and in Section 6 we show how the optimal assignment for the workflow distribution is generated from the integrated cost-model. In section 7 we present more details regarding the API for creating and manipulating triggers on execution sites. In section 8 we consider the handling of failures during the execution of a composed service. In section 9 we compare our approach with some related work. Finally, the last section concludes this paper.

2   Formalism and Assumptions

We model our system using a formalism similar to OWL-S (`http://www.swsi.org`) or WSMO (`http://www.wsmo.org`) that describes services and service workflows. We differentiate between services and workflow descriptions that include only functional aspects, *grounded* service and workflow descriptions that specify also pragmatic aspects relevant to the usage of the service, and cost-annotated workflows that specify also cost measures for different operations in the life-cycle of a web service.

Services are described functionally by a set of input (required) and a set of output (provided) parameters. Each input (resp. output) parameter has an associated name that is unique within the set of input (resp. output) parameters. In this paper we do not consider the type of parameters, as we assume that whenever a service receives an input for a particular parameter, the actual type of the passed value corresponds to the formal type of that service parameter. We assume that services are invoked by some sort of remote procedure call (RPC), such as SOAP RPC [8]. The input values for the service parameters are provided in a request message, the output values are returned in a response message. Asynchronous (one-way) calls can be easily mapped to RPC, which is actually the case for SOAP over HTTP.

A workflow is defined by a set of service invocation steps and a set of data dependencies between required parameters and available parameters. The workflow itself can be seen as a composed service with a number of input and output

parameters. We assume that workflows are consistent regarding any semantic and syntactic constraints that appear in the functional service descriptions. This could be straightforward when the workflows are generated by automatic procedures like the type-compatible service composition algorithm described in [2].

A service grounding makes reference to a functional service description and specifies in addition the pragmatic aspects needed for the actual usage of the service, like transport endpoint details.

A grounded workflow makes reference to a non-grounded workflow and to the respective set of service invocation steps and set of data dependencies. In addition, a grounded workflow specifies a set of groundings and a set of bindings which associates a grounding from the set of groundings to each step in the service invocation set.

A cost-annotated workflow extends a normal workflow by introducing a set of groundings and defining execution cost-measures as functions between the groundings and the workflow steps and data-dependencies. A cost-annotated-grounded workflow specifies also the binding of different groundings to different steps, thus allowing for an exact value for the cost of the workflow execution to be computed.

Formally we define a workflow W as a directed acyclic graph $W = < S, T >$, where S, the set of nodes, contains steps that refer to functional service profiles and T, the set of edges, contains data dependencies specified as tuples of the form $< s_1, s_2 >$ where $s_1, s_2 \in S$. Any workflow graph contains two special nodes s_{start} and s_{end} to model the parameters provided as input to the workflow, respectively required as output results. s_{start} has no data-dependencies and there are no services having data-dependencies on s_{end}.

We defined a grounded workflow GW as a tuple containing the same sets S and T as the workflow W above but extended with a set of service groundings G and a set of bindings B: $GW = < S, T, G, B >$. The set G contains groundings for the functional services referred from steps in S. The set of bindings B contains tuples of the form $< s, g >$ where $s \in S$ and $g \in G$.

We define a cost-annotated workflow CW as a tuple containing all the sets as the grounded workflow GW excepting the set of bindings B but including three cost functions C_G, C_S and C_T: $CW = < S, T, G, C_G, C_S, C_T >$. The three functions correspond to operation costs for each of the three main stages in the life-cycle of a service:

– Service activation $C_G(g) : G \rightarrow \mathcal{R}$. Any service in the frame of a workflow must be activated once, prior to its first invocation. This is independent on how many times a service is used in the workflow. This could correspond to the installation of some infrastructure-required components like local client stubs or third-party applications. The C_G function defines the cost of activating the service specified by the grounding g as a real value. The configuration costs of the triggers presented below can be included in cost returned by the function.

– Parameter transfer $C_T(t, g_1, g_2) : T \times G \times G \rightarrow \mathcal{R}$. In order to be able to execute a service, its parameters must be transferred from the execution site cor-

responding to the service prior in the workflow graph (possibly s_{start}). The C_T function defines the cost of transferring the parameters specified in the data-dependency t between the execution sites specified in the groundings g_1 and g_2 corresponding to the two services involved in the data-dependency.

– Service execution $C_S(s, g) : S \times G \rightarrow \mathcal{R}$. After the service has been activated and its input parameters have been transferred to the current execution site, the service can be executed. The C_S function defines the execution cost of the workflow step s when the service in the step uses the grounding g. Please note that this cost could include the cost for transfering the parameters between the execution site specified in the grounding g and the actual service.

We define a cost-annotated-grounded workflow as a combination between the grounded and cost-annotated workflows above as the tuple: $CGW = < S, T, G, B, C_G, C_S, C_T >$. The cost of this kind of workflow can be uniquely computed as the sum of activation costs for components included in at least one binding plus the sum of the data-transfer costs and execution costs taking into the consideration service groundings as specified by the bindings in B. This can be expressed formally by the formula:

$$cost(CGW) = \sum_{g \in G, \, \exists <s,g> \in B} C_G(g)+$$

$$\sum_{<s_1,s_2> \in T, \, \exists <s_1,g_1> \in B, \, \exists <s_2,g_2> \in B} C_T(<s_1, s_2>, g_1, g_2)+$$

$$\sum_{s \in S, \, \exists <s,g> \in B} C_S(s, g).$$

3   Service Invocation Triggers

In this section we give an overview of service invocation triggers, a concept that we use to model the management of execution sites. A service invocation trigger, in short trigger, is configured on an execution site and corresponds to one invocation of a service. Basically, the trigger together with the underlying execution site can be considered a specialized proxy for a single service invocation. A trigger plays four different roles:

1. It collects the input values for a service invocation.
2. It acts as a message buffer, as each input value may be transmitted by a distinct sender at a different time.
3. It triggers the service invocation when all required input values are available (synchronization).
4. It defines the routing of service output values. Each output value may be routed to multiple different triggers. That is, a trigger is capable of multi-casting.

With the aid of triggers it is possible to distribute the knowledge concerning the data dependencies of the services within a workflow. The main difference

between the workflow itself and the corresponding triggers is not in the information which is more or less the same, but in the fact that the triggers are distributed on different execution sites. Each trigger defines the service that it will have to invoke. The trigger waits until all required input values are available before it fires (i.e., triggers the service invocation). Moreover, each trigger encapsulates workflow-specific knowledge where the results of the service invocation are needed. As the trigger (enacted by the underlying execution site) behaves as a proxy for the service, it handles the results of the service invocation and forwards them to other triggers or services according to its routing information.

In order to illustrate the advantages of using triggers let's compare how the workflow presented in the introduction would be executed, first in the classic case of a centralized workflow engine and secondly in an environment where triggers and execution sites are used.

In the first case (Fig. 2 (a)) the centralized execution of the workflow would require a number of direct request-reply interactions with the services involved: booking a flight and a taxi in NY, booking a hotel in Paris and a flight to Barcelona, booking a hotel in Barcelona. This will result in 5 request-reply interactions amounting to 10 messages, all possibly carried over slow data-links.

In the second case (Fig. 2 (b)) the results for booking the flight in NY could be directly fed to the hotel booking service in Paris; this service could send its time constraints directly to the service booking the hotel in Barcelona which could locally forward the booking result for booking the Paris-Barcelona flight; in turn the flight booking in Paris could directly send the arrival time to the hotel booking service in Barcelona. Finally all booking results could be returned as an
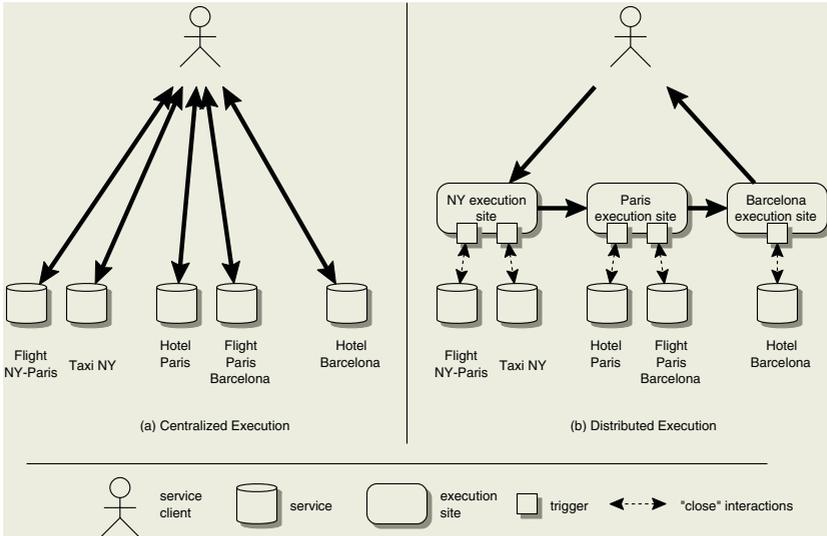

Fig. 2. Centralized and distributed execution of workflows using triggers and execution sites

acknowledgment to the service client. In the right-hand side of the diagram we have figured with a thinner dashed line the interactions between the execution sites and services. We assume that execution sites are very "close" to the services that they have to invoke and thus this invocation costs are much lower than the cost of the long-haul messages exchanged between the service client and the execution sites. In this case only 4 long-haul messages (thick lines) are needed, much less than the 10 required in the previous case.

Using triggers, many different schemes of workflow execution can be implemented. The case that all triggers are to be hosted by the same execution site corresponds to a classic centralized workflow execution model. If each trigger is installed to an execution site "close" to the service that it will invoke, the workflow is executed in a fully distributed way, delivering intermediary results only to those places where they are needed. Our framework does not dictate any of these two extreme settings, allowing for any combination of triggers and execution sites.

Still different distributions of triggers to execution sites might result in workflows that will execute with different performance metrics. In the next section we will present the main contribution of this paper, a generic approach for computing the distribution of triggers over a set of execution sites such that the total workflow execution cost is minimized.

4   Computing Workflow Distributions for Optimal Execution

The system presented here addresses the issue of optimally invoking services in a workflow through a mediation layer, according to some cost measures which reflect the clustering of these services according to some criteria (e.g., geographic distribution). This corresponds to the following problem: Given a workflow, we have to establish how to distribute the triggers corresponding to the invocations of the services in the workflow on a set of available execution sites. The process for achieving that (Fig. 3) comprises the following steps:

1. Send an initial workflow $W = \langle S, T \rangle$ to all relevant execution sites as a Call For Proposals (CFP) (http://www.fipa.org/specs/fipa00029). The list of recipients is explicit – each recipient will know all other recipients. Please note that in the case of large numbers of available execution sites some filtering might be used when deciding which execution sites might be "relevant" but we do not address this issue in the current paper.
2. Each execution site returns a cost-annotated workflow CW that presents the execution site's perspective on how the services in the initial workflow could be grounded. The returned CW will also include the costs of activation, data-transfer and execution, again from the perspective of the current execution site.
3. The groundings and cost functions from cost-annotated workflows received as response to the CFP are merged together.
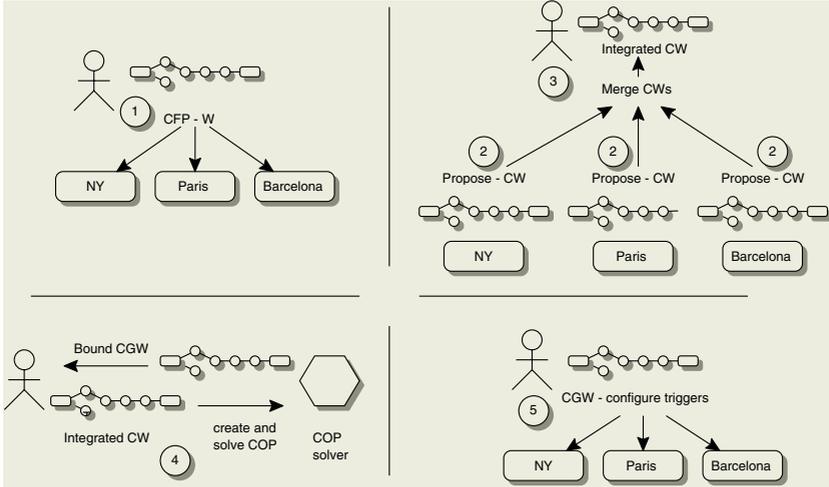
Fig. 3. Computing workflow distributions

4. From the new merged cost-annotated workflow, a Constraint Optimization Problem (COP) is created and solved. The solution of the COP represents a cost-annotated-grounded workflow CGW and specifies the optimal binding for the services such that the workflow cost is minimized. From the CGW we can determine what triggers have to be created on what execution site and how the results should be routed between execution sites.
5. Finally, the triggers are configured on the concerning execution sites accordingly to the CGW obtained as a solution to the COP and the distributed execution of the initial workflow can be started.

5   Generating Cost Measures for Workflow Execution

The first three steps in the process listed above correspond to the generation of a cost model regarding the possible invocation of the services listed in the workflow from any of the possible execution sites.

For this purpose, we use an approach based on a modified version of the FIPA Contract Net Protocol (http://www.fipa.org/specs/fipa00029). In the original version agents bid for solving a given task. Our case is slightly different in that the call includes several tasks and the agents/services have to know each other (in order to report peer-wise data-transfer costs). Hence, one shortcoming of our system is that it assumes cooperative services. We consider the case of competitive services as future work.

After receiving the *Call For Proposal (CFP)* with the initial workflow W, each execution site creates a cost-annotated workflow CW that specifies possible groundings for the services referred in the steps of the original workflow, groundings either locally available or exposed by other execution sites. Also the

service specifies the three cost functions C_G, C_T and C_S for service activation, data-transfer and service execution, costs relative to the current execution site. The service activation and execution function report costs only when the execution site referred in the grounding parameter of the functions is the current execution site. Please note that the execution cost could include the cost of transfering parameter data between an execution site and a "local" service. The data-transfer cost function reports costs only when one of the two execution sites involved in the link is the current site and the other site involved in the link is a remote site. The meaning of the data-transfer cost-function $C_T(t, g_1, g_2)$ is relative to the current execution site: when g_1 is the current execution site and g_2 is a remote execution site, the function result represents the cost of sending the parameter data from the current site remotely; conversely when g_1 is a remote execution site and g_2 is the current execution site the function result represents the cost for the current execution site to receive the parameters data. We consider two different costs for sending and receiving data between execution sites, since communication links are frequently asymmetric both in what concerns the technical parameters and regarding multi-provider business agreements. The data-transfer cost function should reflect some measure of locality (e.g., for an execution site in Paris, the cost for invoking a service in Paris should be lower than the cost for invoking a service in Barcelona).

Finally each execution site returns a cost-annotated workflow CW in the form of a *Propose* message. As the initial CFP might specify also a deadline or timeout, execution sites not providing responses in the given time frame will be discarded from the computations done in the next steps. The process continues if at least one response is received; otherwise a failure is returned.

6   Computing an Optimal Trigger Assignment

For computing the optimal assignment of triggers, we first merge the grounding and cost information in the cost-annotated workflows received as responses to the CFP. Then from the initial set of steps and data-dependencies and the merged sets of groundings and merged cost-functions we create an integrated cost-annotated workflow CW. Finally from the integrated workflow we create and solve a Constraint Optimization Problem (COP), a particular case of Constraint Satisfaction Problem (CSP).

Merging the sets of groundings is straightforward – a new set is created as the union of the grounding sets in the received CGs, where the duplicates are discarded. Merging service activations and execution costs is also straightforward – the new cost functions will just aggregate the costs of activation and execution functions in the received CGs. For data-transfer functions $C_T(t, g_1, g_2)$ the merged cost function returns the sum of the costs of transferring data between execution sites as reported by the two sites specified by the groundings of the link binding. I.e., the total cost of data-transfer will be computed as the sum of the cost reported by the site in g_1 for sending the parameter data to g_2, plus the cost reported by the site in g_2 for receiving the parameters from the site in g_1.

Formally, we define a constraint optimization problem (COP) as the tuple $< X, D, C, R >$ where:

- $X = \{x_1, x_2, ..., x_n\}$ is a set of n variables.
- $D = \{d_1, d_2, ..., d_n\}$ is the set of domains of the n variables in X, each given as a finite set of possible values.
- $C = \{c_1, ..., c_m\}$ is a set of m constraints, where a constraint c_i is given as the list $(x_{i1}, ..., x_{ik})$ of variables it involves.
- $R = \{r_1, r_2, ..., r_m\}$ is a set of relations, one for each of the m constraints, where a relation r_i is a function $d_{i1} \times ... \times d_{ik} \to \mathcal{R}^+$ giving the cost of choosing each combination of values. Combinations that are not allowed have a very high cost (∞).

A *solution* is a combination of values $v_1 \in d_1, ..., v_n \in d_n$ such that the sum of the cost of the relations is minimal and different from ∞. Otherwise we consider that the COP has no solution.

We create a COP from the annotated workflows received as responses of the CFP as follows:

- X contains a variable for each step of the initial workflow (each execution of a service), for each data-dependency of the initial workflow (each parameter that needs to be transferred between services), and for each grounding in the new set of merged service groundings (each service that might need to be activated).
- The domains of the variables corresponding to executions of services contain as values the possible groundings for the respective service from the merged service grounding set. The domains of variables corresponding to data-dependencies between services contain as values tuples corresponding to all combinations of groundings for the two steps in the data-dependency, corresponding to all reported combinations of senders and receivers. For k execution sites, we have maximum $k(k-1)$ possible values. For example, in the case of three execution sites ES_{NY}, ES_{Paris}, $ES_{Barcelona}$, we have six possible values: $ES_{NY} - ES_{Paris}$, $ES_{NY} - ES_{Barcelona}$, $ES_{Paris} - ES_{NY}$, etc., corresponding to a data transfer between execution sites in NY and Paris, NY and Barcelona, Paris and NY, etc. The domains of service-activation variables corresponding to groundings in the set of merged groundings have two boolean values representing the fact that the service has to be activated or not in the current COP solution.
- In C and R we have two kinds of constraints and costs: activation and data transfer. These constraints link execution variables to activation and data-transfer variables, making sure that once a grounding is chosen for executing a service the grounding is going to be activated (the corresponding grounding activation variable is going to be true) and the required input parameters will be available (the values of the variables corresponding to the data-transfer links ending at the current step will have to have the target grounding as the grounding chosen for the execution step). For example, a variable corresponding to a node in the workflow is assigned the value ES_{Paris}. Then

the generated constraints ensure that the respective grounding has been acti-
vated and that all the data-dependencies (edges entering the respective node)
are fulfilled by being assigned values of the form $* - ES_{Paris}$ (incoming data
transfers).

Finally we solve the formulated COP using a state of the art commercial
Java solver for constraint optimization problems. Still since there might not be
enough responses received from the execution sites by the deadline of the CFP
the COP might not have a solution in which case a failure is returned.

7   Defining Triggers

In this section we present more details regarding the installation of triggers.
In our description we use the following abbreviations for identifying services,
respectively triggers:

[**SID:**] **Service ID.** Globally unique identifier of a service to be executed. It
   consists of host, port, protocol, and local service identifier (e.g., service name
   and version number, depending on the protocol). SIDs are computed from
   concrete service descriptions (including grounding information).
[**PID:**] **Parameter ID.** Locally unique identifier for a service input or output
   parameter.
[**TID:**] **Trigger ID.** Globally unique trigger identifier. It consists of details
   regarding the underling execution site like host and port. It also contains
   a local trigger identifier (e.g., an integer number referring to an invocation
   trigger).

Below we present a simple API to deal with triggers in an abstract way:

[**CreateTrigger:**] Creates and installs a trigger.

   Arguments:
   – Destination of the trigger (host and port). `CreateTrigger` will ask the
      destination execution site to set up the desired trigger.
   – SID. The service to be invoked by the trigger.
   – Service input parameters to wait for. Each parameter is identified by its
      PID. A parameter may be required or optional. The trigger will fire as
      soon as all required parameters are available. As for a given parameter
      multiple values may arrive before the trigger fires (while still some of the
      required parameters are missing), the client has to define which values to
      preserve: `preserveLast` or `preserveFirst`. If values for optional input
      parameters arrive before the trigger fires, they will be passed to the
      service. After the trigger has fired, arriving inputs are discarded.
   – Optional: Input data. For each input parameter a default value may be
      provided. This value could be transmitted with `SendData` (see below),
      but including it in `CreateTrigger` may be more efficient and help to
      reduce network traffic.

– Output routing. For each output value (identified by a PID_O) generated by the service SID, the output routing defines a possibly empty list of pairs (TID_i, PID_i) to forward the output. That is, whenever the service SID returns an output value for the parameter PID_O, the trigger will forward it to all TID_i with the name PID_i, implementing a multicast. If there is a communication problem with a trigger TID_i, the trigger will retry to forward the data several times in order to overcome temporary network problems.

– Desired timeouts:
1. Timeout to wait for inputs, starting with the installation of the trigger. If not all required input data arrives before this timeout, the trigger will be discarded.
2. Timeout to wait for service completion, starting with the service invocation.
3. Timeout to wait for completed forwarding of service outputs, starting when the trigger receives the results of the service invocation.

– Optional: Destination for failure notification message (host, port, protocol). In the case of a failure (i.e., service returning a failure message or expiration of one of the timeouts mentioned before), a failure notification is sent before the trigger is discarded, including information concerning the current state of the trigger. The level of detail of this notification can be configured. The message may simply indicate the reason of the failure, or it may include service inputs resp. outputs the trigger has received so far. This information may help the client to recover from the failure.

Results:
– TID of the installed trigger (if the trigger was accepted).
– Granted timeouts. Each granted timeout may be the desired timeout or shorter.

[**RemoveTrigger:**] Explicitly removes a trigger. Normally, a trigger is removed automatically if either a timeout occurs or if the output routing task is completed, i.e., all outputs have been forwarded according to the trigger's routing information.

Arguments:
– TID. The trigger to remove.

[**SendData:**] Sends input data to a trigger. Normally, triggers receive results either by initialization (see the input data of `CreateTrigger`) or through other triggers (forwarded results from another service). However, a client may want to install a trigger and provide input data later on.

Arguments:
– TID. The trigger to send data to.
– Input data. For each input parameter a value may be provided.

[**Status:**] Returns information concerning the status of a trigger. I.e., whether the trigger is still waiting for required input, which input arguments have been received so far, whether it has already triggered the service, whether it is waiting for the service outputs, etc.

Arguments:
  − TID. The trigger to ask for its status.

Results:
  − Status information.

Three different protocols are involved in the communication with triggers and services:

− The trigger communicates with the service using remote procedure calls (e.g., SOAP RPC [8]). That is, the trigger is transparent to the service, it behaves as any other client.
− The communication between triggers is unidirectional. A trigger forwards results to another trigger. The messages sent from trigger T_A to trigger T_B contains at least one value for an input parameter T_B is waiting for. Even though the communication protocol between triggers need not necessarily comply with standards, SOAP messages are well suited for trigger communication. If the triggered service returns a fault message, the trigger does not forward the message on the normal output routing path, but it may generate a failure notification message (if specified in `CreateTrigger`). Subsequent triggers will notice the failure by a timeout.
− A dedicated, simple protocol supports the API primitives described before. For instance, `CreateTrigger` will try to deploy a trigger on the specified destination platform.

8   Failure Handling

In our approach composed services are executed in a completely distributed way. Therefore, it is not easily possible to monitor the progress of each service invocation. As the client will only receive the final results of the composed service, in general it will notice a failure only after a timeout. In this case, the client may restart the execution of the workflow.

If the used services are not reliable, this approach may result in bad overall performance, since intermediary values may have to be computed multiple times. Hence, the client should make use of the failure notification mechanism in order to collect partial results that had been computed before the failure has happened. Based on the failure notification mechanism, the client could exploit redundant execution plans in order to replace a failed service. As an alternative (but inefficient) solution, if the distributed execution of a composed service fails, the client could simply re-execute the workflow in a centralized fashion (fallback solution).

The client may also use the `Status` primitive of triggers in order to monitor the progress of the execution. Note that `Status` will fail if the trigger has already been removed (i.e., after a timeout or after completing its task). However, `Status` creates additional network traffic, therefore an excessive use of this primitive is not consistent with the principal idea of our approach to minimize the network traffic involving the client.

9   Related Work

The Internet indirection infrastructure i3 (`http://i3.cs.berkeley.edu/`) uses triggers to decouple sender and receiver [6]. In contrast to our triggers, i3 triggers work on the level of individual packets and do not support waiting conditions (synchronization) to aggregate multiple inputs from various locations before forwarding the data. i3 supports only a very limited form of service composition, where individual packets can be directed through a sequence of services. While our triggers are rather transient (used only for a single service invocation) and their placement is explicitly controlled by the client, i3 triggers are more persistent (they act as a longer-term contact point for a service) and are mapped to the Chord peer-to-peer infrastructure, which allows only a limited form of optimizing the routing (by selecting a trigger identifier that will map close to a desired location). Summing up, even though there are some ideas in common, i3 has different goals (indirection, supporting mobility, multicast, anycast) and works at a much lower level than our approach. Our focus is on the efficient routing of intermediary results during the execution of composed services.

Another relevant approach is the SELF-SERV system[5]. The system architecture identifies three kinds of service: elementary, composite, and community. The execution of composite services is managed by coordinators. The concepts are similar to our definition of compositions as workflows and to our infrastructure of execution sites and triggers. The infrastructure presented in [5] is rather complex but no clear details are presented regarding the instrumentation for managing the coordination services or the way of choosing the appropriate coordination service. In this paper we present a simple and clear approach for instrumenting the management of execution mediators. Still the main contribution of this paper is the generic approach for choosing services and execution sites. The distributed execution of the resulting workflow is optimal regarding the costs of service activation, parameter transfer, and service execution.

10   Conclusion

When composed services represented as workflows are executed in a centralized way, partial results might not be efficiently handled. In this paper we describe a multilayered architecture where service invocation is mediated by execution sites equipped with triggers – the equivalent of a service proxy. This approach is advantageous regarding both the number of messages that have to be sent for

executing the workflow and the possibility of having faster interactions between services that are "closely" located.

The main contribution of this paper is an approach for computing the distribution of triggers over a set of execution sites such that for a given cost-model taking into account service activation, parameter transfer, and service execution, the execution cost of the given workflow is minimized.

References

1. W. Binder, I. Constantinescu, and B. Faltings. Efficiently distributing interactions between composed information agents. In *Second European Workshop on Multi-Agent Systems (EUMAS-2004)*, Barcelona, Spain, December 2004.
2. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004.
3. S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, Apr. 22–25 2002. Morgan Kaufmann Publishers.
4. S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *11th World Wide Web Conference (Web Engineering Track)*, 2002.
5. Q. Z. Sheng, B. Benatallah, M. Dumas, and E. O.-Y. Mak. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *VLDB*, pages 1051–1054, 2002.
6. I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, Apr. 2004.
7. S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.
8. W3C. Simple object access protocol (SOAP), http://www.w3.org/tr/soap/.
9. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.