Collecting and Querying Distributed Traces of Composite Service Executions

Marie-Christine Fauvet[1]*, Marlon Dumas[2], and Boualem Benatallah[1]

[1] School of Computer Science & Engineering, The University of New South Wales
Sydney NSW 2052, Australia
email: fauvet@imag.fr, boualem@cse.unsw.edu.au
[2] Centre for Information Technology Innovation, Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
email: m.dumas@qut.edu.au

Abstract. The development of new Web services by composition of existing ones is becoming a widespread approach to realise business-to-business collaborations. The composite services obtained in this way are then eventually used in other compositions. Given the dynamic nature of the Web, this recursive composition of services rapidly leads to intricate dependencies between them. On the other hand, businesses need to track the executions of their composite services in order to ensure explainability in case of failure and to support decision making. This paper deals with the issue of tracing composite service executions over the Web. It describes a model and an XML representation of service execution traces, an approach for collecting and storing these traces in a distributed environment, and an approach to evaluate queries over distributed repositories of traces.

1 Introduction and motivation

The connectivity generated by the Internet is re-shaping the way organisations architect their collaborations with other organisations, as well as their interactions with their customers. Organisations of all sizes are profiting of this connectivity to form online alliances by inter-connecting their services for the purpose of providing one-stop shops to their customers.

In this setting, the idea of developing new services by composition of existing ones is becoming the keystone of the next generation of Internet systems. A service is seen as an abstraction of a set of activities involving a number of resources (e.g., data sources, application programs, business processes), intended to fulfil a class of customer needs or business requirements. In order to satisfy complex needs, services are inter-connected among them, thereby forming *composite services*. Examples of composite services include a travel management service combining flight and accommodation booking services, or an account aggregation service that integrates banking, tax declaration, and financial services.

In order to satisfy current users and to attract new customers, organisations need to pay special attention to the quality of their services. In particular, they need to trace executions of these services in order to ensure explainability in case of failure or auditing,

* On leave from LSR-IMAG, University of Grenoble, France.

as well as to support decision-making aimed at improving the structure and dynamics of the services. These traces of ongoing and past executions of services provide the information required to answer queries for the following purposes (among others):

Customer feedback: to explain specific failures. A query in this context would be *"Retrieve the traces of all executions that have been triggered for a given client"*.

Quality assessment: to detect services whose executions tend to fail, like for example in *"Retrieve the executions of a given service that have been stalled since more than 30 minutes"* or to make a report on past service executions as in *"Retrieve the components of a composite service whose executions take the most time on average"*.

Monitoring and control: to adapt the service to the actual requirements by identifying, in the context of a given service, some patterns of its component executions. An example would be *"In how many executions of the service S, the execution sequence of the service A, then B and finally C has been observed?"*. Also, the ongoing execution of a service could be adapted *on the fly* by analysing what has happened so far. For example, the choice of which component to trigger at a given point of an execution, could be based on information extracted from the traces of the composite service.

Audit: to conduct routine or ad-hoc checks involving the executions of a service, like for example when validating the bills issued by the providers of a service.

This paper presents a framework for the collection and management of traces about either past or ongoing executions of composite services. The proposed framework includes: (i) a generic model of traces of composite services; (ii) a concrete representation of traces in XML; (iii) an approach to collect and store these traces in a distributed environment; and (iv) a method for evaluating queries over these traces. The framework addresses the following issues:

- The traces are distributed: querying the traces of a service's executions may therefore require multiple sub-queries to be sent to the providers who have hosted the execution of the component services. This issue is different to the one addressed by classical approaches in the context of distributed query processing. These approaches typically rely on a centralised knowledge of the meta-data describing the topology of the network where the data are distributed. In contrast, the partitioning of the execution traces across service providers can only be incrementally discovered when browsing the traces themselves.
- The number of providers can be large and continuously changing: the provider of a component within a composite service may be dynamically selected based on various factors. As a result, the service providers involved in a composite service varies from one execution to another. In addition, providers of component services may join and leave a composite service at any time.
- The traces are heterogeneous: although conforming to a common generic interface, each provider will offer its own service interfaces, with a different set of states and observation points than those of other providers. This means in particular that traces must be treated as semi-structured data, which motivates the choice of XML.

The rest of the paper is organised as follows. Section 2 introduces the basic concepts of the proposed framework. Section 3 deals with the collection of traces represented in XML. In section 4 we discuss and illustrate the evaluation of queries in a distributed

environment. Finally, section 5 compares the proposal with similar or complementary ones, while section 6 concludes.

2 Design Overview

In this section we introduce the framework that we adopt for service composition and execution, and for querying traces. In order to ensure a broad applicability, this framework is intended to be independent of specific service implementation technologies (e.g., J2EE, .Net), service description languages (e.g., WSDL) and service registration and discovery infrastructures (e.g., UDDI).

2.1 Service composition

We distinguish between *elementary* and *composite* services. Elementary services are pre-existing or native services that should be treated as black boxes from the perspective of other services or application programs. A composite service is an aggregation of other (either composite or elementary) services, which are referred to as its *component services*. At a very abstract level, a composite service is modelled as a graph whose nodes are labelled with invocations to the component services. The edges between these nodes capture data and control-flow dependencies. Control-flow dependencies determine which nodes (if any) need to be entered after the service invoked by a given node completes its execution. Control-flow dependencies also establish timing constraints, signal sending and processing, etc. Data-flow dependencies on the other hand determine the data items that must be passed from one node to another when a control-flow link is taken.

Each node in a composite service is associated to an organisational entity which is responsible for handling the service invocation associated to that node. The organisational entity associated with a node can be either an individual provider or a community of providers. In the former case, the designated provider is responsible for executing all the instances of this service. It may eventually partially or totally delegate the execution of these instances to another provider, but this delegation is hidden to the users of the composite service. On the other hand, a community of providers will systematically and transparently delegate the execution of a service to its members. This delegation is carried out by the *representative* of the community, which effectively acts as a service broker. The means by which a community's representative chooses a member to execute a request, is specified via a selection policy [1].

One way of concretely describing the control and data-flow dependencies of a composite services is to use an existing process modelling language, and especially, one of those that have proven to be suitable for workflow specification. There are numerous workflow specification languages based upon different paradigms. In fact, each commercial Workflow Management System implements its own specification language, with little effort being done to provide some degree of uniformity between products. In this respect, the Workflow Management Coalition [5] has defined a set of glossaries and notations that encompass some of the constructs used in existing workflow specification languages. Unfortunately, this standardisation effort has not yet led to a standard

language for process modelling, which could be applied for the specification of control and data-flow within a composite service. Recently, WSFL[3], XLANG [4], and the ebXML Business Process Specification Schema (BPSS)[5] have been proposed as candidate languages for this purpose. At present, standardisation efforts are underway based on these proposals, but no consensus has been reached yet.

For the purpose of this paper and to keep the model general enough, we choose to specify control and data-flow dependencies using statecharts [9]: a widely used formalism in reactive systems which has been integrated into the Unified Modelling Language (UML) [17]. Statecharts offer constructs for modelling sequence, loops, branching, concurrent threads, and communication between threads based on signals. Since these are the basic concepts found in most process specification languages, we expect that our results can be adapted to other composition languages such as WSFL, XLANG, and BPSS.

The statechart in Figure 1 specifies the control-flow dependencies of a composite service S. S1 and S2 are invoked first and executed in parallel. When both finish, either S3 or S4 is executed according to the condition C. Then S5 is finally executed.

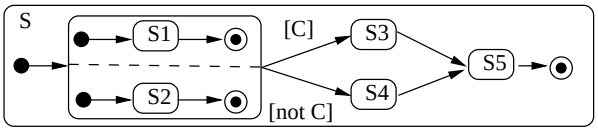

Fig. 1. Control-flow statechart of a composite service.

2.2 System Architecture

The basic entities of the framework architecture are “wrappers”, “schedulers” and “multiplexers”. The entities are described in turn below.

Wrappers. A provider of a service has to supply a *wrapper*. The wrapper of a service ensures that a native service can be invoked regardless of its underlying data model, message format and interaction protocol. For this purpose, a service’s wrapper handles (among other things) data conversion between the data model of the service interface and that of its implementation [1]. Other issues that wrappers can address include security management and protocol heterogeneity. In our tracing model, the service wrapper is also responsible for recording facts about each execution of the wrapped service. These facts are stored locally by the wrapper in a repository of traces and made available through a query interface as discussed later in the paper.

Schedulers. The provider of a composite service hosts a *composite service scheduler* for that service. Interactions among components of a composite service are implemented

[3] http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
[4] http://www.gotdotnet.com/team/xml_wsspecs/xlang-c
[5] http://www.ebxml.org

by a *composite service scheduler* (a scheduler in short). A scheduler is responsible for orchestrating the executions of the composite service by triggering the executions of the component services according to the control-flow dependencies associated with the composite service. The scheduler is also responsible for handling and processing data according to the data-flow dependencies encoded within the statechart.

The scheduler of a composite service can be either located in a central location (the *centralised orchestration approach*) or implemented as a set of distributed processes that cooperate in a peer-to-peer manner (the *P2P orchestration approach*). In the centralised approach, the scheduler of a composite service S is implemented as a single software module as in [11], [2] and [21]. This scheduler is responsible for initiating the execution of the components of S according to the control-flow statechart associated with S. To do so, the scheduler of S invokes each of the components of S according to the control-flow dependencies of the composite service.

In the P2P approach, the scheduler is implemented as a collection of software modules communicating with each other directly as in [16], [7] and [4]. Each participant in a composite service hosts one of these software modules, that we call a *local scheduler* in the sequel. On the other hand, the provider of the composite service hosts another software module that we call the *global scheduler*. When the global scheduler receives a request to start an execution, it sends messages to the local schedulers of those participants that need to start their executions in the first place. Each of these local schedulers invokes the underlying service through its wrapper, waits until the execution resulting from this invocation is completed, and when this happens, it sends a message to the local schedulers of those participants that need to be executed next according to the control-flow dependencies of the composite service. These peer-to-peer exchanges between local schedulers continues until eventually one of the local schedulers indicates to the global scheduler that the overall composite service execution has completed. A more detailed description of this model and its implementation can be found in [1].

Query multiplexers. Each service provider hosts a software module call the *query multiplexer*, which is responsible for: (i) receiving a query from a requester and preprocessing it, (ii) identifying the eventual sub-queries and if any, (iii) dispatching them to the corresponding providers, (iv) receiving the sub-results for the providers, (v) merging local and remote results, and finally, (vi) sending back the overall result to the requester. The features of the query multiplexer are detailed and illustrated in section 4.

3 Modelling, representing, and collecting traces

3.1 Modelling traces

Simplifying assumptions. For the sake of simplicity, we assume that the wrapper and the scheduler of a composite service share a common time line. This can be achieved using well-known clock synchronisation protocols such as NTP [10]. We also assume that all temporal values (time instants, durations and intervals), are expressed at the same level of granularity (e.g., at the granularity of the minute or of the second). Under this assumption, instants and durations are unambiguously represented as integers, while an interval is represented as a pair of integers corresponding to its bounds.

Life cycle of a service instance. Throughout its life cycle, a service execution goes through a series of statuses. The following statuses are predefined by the tracing model: *enabled*, *running*, *stalled*, *completed*, and *cancelled*. These predefined statuses can be specialised (or refined) by a given service provider in order to accommodate application specific semantics. For example, the provider of a service "Currency Converter" can declare that a service specialises the status "running" into 3 sub-statuses: *getting data*, *processing data*, and *displaying results*. When an execution of this service is in the "running" status, in can be in either of these three sub-statuses as well.

Every service is associated to a *life cycle statechart*[6] that models the possible statuses through which the executions of this service can go, and the possible transitions between these statuses. The transitions of this statechart are labelled with the events that fire them. These events can be internal to the service execution (e.g., the service starts running), or external (e.g., the user sends a cancellation message). In both cases, an event occurrence within an execution is processed by the wrapper of the service, which determines which transition in the life cycle statechart needs to be fired (if any), and records the new status in the trace of the service execution.

The *standard life cycle statechart* defined by the tracing model is depicted in Figure 2. When an execution of a service is started, it enters the running status. While on this status, the service can be suspended due to an external request, or stalled because a resource required for the service execution is temporarily unavailable. This is notified to the wrapper through an event *stall*. From the *stalled* status, the service instance can subsequently either move back to the running state or to the cancelled status. From the running status, it can move either to the completed status or to the cancelled one.

The states of this statechart can be refined by a given service provider in order to incorporate application-specific statuses, transitions, and events. For example, the "running" service of the standard life cycle statechart can be refined into a statechart with 3 states connected in sequence: *getting data*, *processing data*, and *displaying results*.

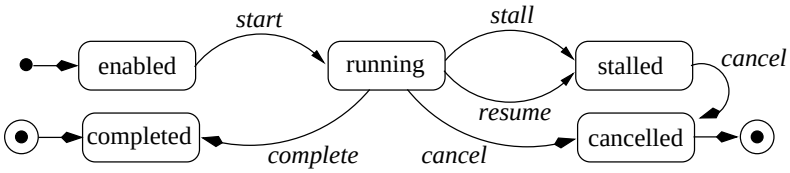

Fig. 2. A statechart modelling the life cycle of a service execution.

The tracing model does not impose any explicit relationship between the life cycle of a composite service and those of its components. For example, a composite service execution can very well be in the "running" status even if one or several of its components are in a "stalled" status. When an execution of a component service reaches a particular status, if any change of status has to be propagated to the composite service execution, a notification message is sent to the wrapper of the composite service, who determines whether a change of status at the composite service level is required.

[6] The life cycle statechart is not to be mistaken with the control-flow statechart of a composite service (see section 2.1), which determines the order in which its components are triggered.

Status history. A status history is a log of the life cycle of a service execution, that is, the statuses through which this execution went through, and the times of the transitions. At an abstract level a status history is defined as a function from a set of instants to a set of status values. At a concrete level a status history is represented by an ordered set of interval-timestamped statuses. For example, the status history [<[3..3], enabled>, <[4..7], running>, <[8..8], completed>] indicates that the execution was enabled at instant 3, then it ran from instant 4 to instant 7 before being completed at instant 8.

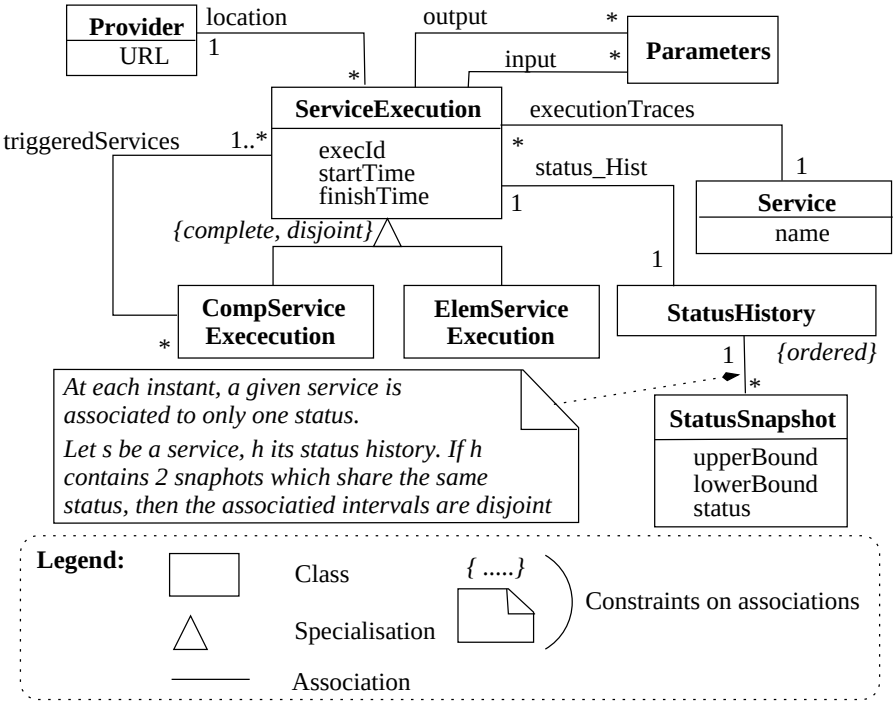

Fig. 3. UML class diagram for service execution traces.

Trace of a service execution. The trace of a service execution, whether elementary or composite, includes (i) a status history; (ii) a set of effective input and output parameters; and (iii) the location of the provider to whom the service execution was dynamically assigned. A composite service execution trace has an additional property modelling the set of other service executions that it triggered (i.e. its triggered components).

The UML class diagram in Figure 3 describes the data model for service execution traces. In this diagram, the main class is ServiceExecution, whose instances model traces of service executions. This class has two sub-classes: one for composite services and the other for elementary services. The status history associated to a service execution is modelled as a set of snapshots, each of which associates an interval (upper and lower bound) with a status.

3.2 Collecting traces

The responsibility to trace the executions of a composite service S is distributed across the wrappers of this service (as many wrappers as actual providers for the component services). The wrapper of a service S is responsible for:

- Creating and instantiating an object of the class ServiceExecution. This involves generating an identifier for the execution, and recording the start and the end times.
- During the course of the execution, processing any events that may change the current execution status, and record any changes by modifying the corresponding object's status history.
- If S is a composite service, instantiating the association triggeredComponents: for each of the component services that are triggered, the wrapper of S must obtain a reference to an object of the class ServiceExecution from the wrapper of the component service. Such reference is of the form: <provider's url>/<service name>/<execution id> is the identifier locally assigned by the provider of the component service. The provider's URL uniquely identifies the repository where the value of the object is stored.
- At the end of the service execution, returning the reference (<provider's url>/<service name>/<execution id> where <execution id>) to the application program or composite service wrapper that initially invoked the service S.

Hence, a wrapper is responsible for collecting traces about the execution it is supervising, and passing the resulting object reference to whoever initiated the execution. The tracing model defines two alternative approaches for collecting the object references from the component service wrappers: one for the centralised orchestration model, and one for the peer-to-peer orchestration model.

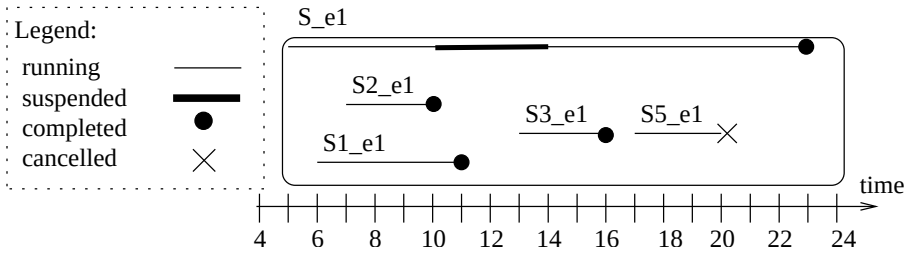

Fig. 4. An execution scenario for the service S.

To illustrate the two approaches to trace collection, let us consider again the service S depicted in Figure 1. Figure 4 describes an execution scenario where service S is executed. This execution is identified by S_e1. The execution e1 of service S started at time 5. It ran until time 10 (excluded) before being stalled from 10 to 14 (excluded). Then, the execution resumed and ran again from 14 to 23 (excluded) before being completed at time 23. The execution e1 of the component S1 was triggered at time 6; it ran until 11 (excluded) and completed at 11. We assume that S_e1 could complete even that S5_e1 has been cancelled. Such a recovery mechanism has to be implemented in S itself.

Given the execution S_e1 shown in Figure 4, Table 1 shows message passing between the central scheduler and the wrappers in the case of a centralised orchestration approach, while Table 2 shows message passings in the case of peer-to-peer orchestration approach. In both tables, the columns Sender and Recipient identify either wrappers or schedulers, while the column Message content shows information exchanged for the purpose of tracing only. A symbol of the form X_e1 (X $\in$ {S1, S2, S3, S4, S5}) denotes an instance of the class ServiceExecution corresponding to an execution of service X. For example, the object X_e1 is created by the wrapper of X at the beginning of the execution e1.

Time	Sender	Recipient	Message Content
10	S2_e1	S_e1	www.prov1.com.au/S2/e1.xml
11	S1_e1	S_e1	www.prov1.com.au/S1/e1.xml
16	S3_e1	S_e1	www.prov2.com.au/S3/e1.xml
20	S5_e1	S_e1	www.prov3.com.au/S5/e1.xml

Table 1. Messages between the central scheduler and the component services' wrappers during the execution of S_e1 (centralised orchestration model)

Time	Sender	Recipient	Message Content
10	S2_e1	S3_e1, S4_e1	{www.prov1.com.au/S2/e1.xml}
11	S1_e1	S3_e1, S4_e1	{www.prov1.com.au/S1/e1.xml}
16	S3_e1	S5_e1	{www.prov1.com.au/S2/e1.xml, www.prov1.com.au/S1/e1.xml, www.prov2.com.au/S3/e1.xml}
20	S5_e1	S_e1	{www.prov1.com.au/S2/e1.xml, www.prov1.com.au/S1/e1.xml, www.prov2.com.au/S3/e1.xml, www.prov3.com.au/S5/e1.xml}

Table 2. Messages between global and local schedulers during the execution of S_e1 (peer-to-peer orchestration model)

The 1st and 2nd lines of table 3.2 can be read as follows. At time 10 (respectively 11) S2's wrapper (respectively S1) sends its trace identifier in the form of a reference to a repository to both S3'wrapper and S4's wrapper. Because the boolean expression [C] evaluates to true, S4 is not required to be executed, so the local scheduler of the state that labelled S4 discards the collection of references that were sent to it by the local schedulers of S1 and S2. When S3 finishes (3rd line), its wrapper sends the collection of references that it has received so far, augmented with its own reference. The wrapper of S3 on the other hand does keep these collections of references and starts the an execution of S3. When this execution competed (3rd line of table 3.2), the wrapper of S3 sends to the wrapper of S the collection of references that it received from S1, merged with that received from S2, and augmented with its own reference to an object of the class ServiceExecution. At the end, as shown in the 4th line, the wrapper of S (through its associated global scheduler) receives all references to repositories describing the traces for S_e1 triggered components and populates its own repository based on them. A detailed description of the trace collection method can be found in [7].

3.3 XML representation for traces

A trace of a service's execution trace is represented as an XML [23] document supplied by the provider who has hosted the service execution. The provider's URL combined with the document name and the service execution id is used as an URI (Universal Resource Identifier) to locate the service execution trace.

The choice of XML as a language for externally representing and exchanging traces is mainly motivated by two reasons:

- Although conforming to a common generic interface, each provider will offer its own service interfaces with a different set of states and observation points than those of other providers. XML provides mechanisms (e.g. namespaces and mixed elements) to deal with this form of controlled heterogeneity.
- The traces are intended to be exchanged between different sites both during service execution and during trace querying. The use of XML enables service providers to internally store these traces using (e.g.) relational databases, and to dynamically translate them to and from XML using well-known tools.

The structure of XML documents is directly derived from the class diagram depicted in Figure 3. Given the execution scenario depicted in Figure 4, the XML document below contains sample data collected during the execution of services that have been hosted by the provider foo.com.au:

```
<traces>
  <serviceExecution name="S" execId="e1" loc="www.foo.com.au/S/e1.xml">
    <time start="5" finish="23"/>
    <inputs> <input name="X" value="100"></input> </inputs>
    <outputs>
      <output name="Y" value="20"></output>
      <output name="Z" value="500"></output>
    </outputs>
    <triggeredComponents>
      <serviceExecution name="S1" execId="e1" loc="www.prov1.com/S1/e1.xml"/>
      <serviceExecution name="S2" execId="e1" loc="www.prov1.com/S2/e1.xml"/>
      <serviceExecution name="S3" execId="e1" loc="www.prov2.com/S3/e1.xml"/>
      <serviceExecution name="S5" execId="e1" loc="www.prov3.com/S5/e1.xml"/>
    </triggeredComponents>
    <statusHistory>
      <statusSnapshot status="running" lowerBound="5" upperBound="9"/>
      <statusSnapshot status="suspended" lowerBound="10" upperBound="13"/>
      <statusSnapshot status="running" lowerBound="14" upperBound="22"/>
      <statusSnapshot status="completed" lowerBound="23" upperBound="23"/>
    </statusHistory>
  </serviceExecution>
  <serviceExecution> ....
</traces>
```

As discussed earlier, the provider of a service has the right to specialise the predefined statuses by defining sub-statuses (e.g., defining sub-statuses of the status "running"). These sub-statuses can appear in the traces of a service execution within sub-snapshots of the snapshots involving predefined statuses. This approach is similar to the

one discussed in [20]. For example, if we assume that the composite service S defines 3 sub-statuses of the status “running”, namely “searching”, “displaying” and “booking”, then the XML elements representing snapshots involving the “running” status, can have children elements representing sub-snapshots involving these 3 sub-statuses. Hence, the italicized line in the XML code above could then be refined as follows:

```
<statusSnapshot status=”running” lowerBound= 5 upperBound= 9>
  <subSnapshot substatus = ”searching” lowerBound = 5 upperBound = 6/>
  <subSnapshot substatus = ”displaying” lowerBound = 7 upperBound = 8/>
  <subSnapshot substatus = ”booking” lowerBound = 9 upperBound = 9/>
</statusSnapshot>
```

4 Querying traces

This section describes and illustrates a mechanism to split a query on the execution traces of a composite service into subqueries to be executed by providers of the (direct and indirect) components of the composite service. The results of these subqueries are then collected and merged in order to build the result of the initial query. Queries are expressed in Quilt query language [3] a dialect of Xquery language [25]. XPath expressions are used as means to navigate through hierarchy of nodes [26].

4.1 Towards a query multiplexer

The query multiplexer of a service provider is responsible for processing queries regarding all the execution traces hosted by that provider. The scope of the queries that the query multiplexer of a provider P can handle is modelled as a tree. The root of this tree contains an XML document with a sequence of elements serviceExecution, describing all the service executions hosted by the provider P (see Section 3.3). A node other than the root contains an XML document with a single element serviceExecution describing an execution hosted by another provider than P, and linked to upper nodes through the “composite service–component service” relationship. An edge of the tree therefore models the invocation of a service: an edge from a node n1 to another n2 denotes the fact that the service execution described in n2 was triggered in the context of the (composite service) execution described in node n1.

At an abstract level this tree can be seen as a single XML document that contains the data required to answer any query related to the services hosted by P, at any level of detail. This abstract representation is obtained by replacing the elements serviceExecution in the root, with the contents of the XML file referenced in the attribute loc (see section 3.3). This expansion mechanism has to be recursively carried out starting from the root node, every time that the element serviceExecution is encountered . This mechanism is similar to the one implemented by XInclude [24]. In the sequel, we call the document obtained by expansion, *traces.xml*.

From the user’s point of view, queries are processed on the abstract document *traces.xml*. For efficiency reasons and given that this “abstract” document is a continuously evolving view, the document is not built a priori and stored in a central location. Instead, when a query is submitted to the multiplexer, it is locally analysed and split into

multiple subqueries. The result of this analysis is an XML document that contains tags indicating for each subquery the provider responsible for its execution. Each subquery is then sent to the corresponding provider whose query multiplexer in turn processes it and returns a result. When the results of all subqueries have been received, they are merged with the main result to produce the final output. This mechanism is carried out recursively each time that a subquery involves distributed traces. Similar mechanisms have been studied in the context of distributed query processing [13]. However, classical approaches in this area rely on a centralised knowledge of the topology of the network where the data are distributed. In contrast, the partitioning of the traces across the service providers is only discovered when browsing the traces.

Our splitting and merging mechanisms are formalised below. We adopt the following notations: Q is the set of queries (expressed in Xquery), X is the set of XML documents, and P is the set of service providers. T1 $\longrightarrow$ T2 stands for the type of all functions with domain T1 and range T2. {T} denotes the type of sets of T. ⟨T1, T2, ..., Tn⟩ designates the type of tuples whose i^{th} component is of type Ti ($1 \leq i \leq n$).

The multiplexer procedure for a query q ($q \in Q$) on an abstract root XML document d ($d \in D$), is captured by two functions $Split^d$ and $Merge^d$ defined below:

$Split^d$: Q $\longrightarrow$ ⟨ {⟨ P, Q ⟩}, Q ⟩
/* ⟨ ⟨ p_n, q_n ⟩, main ⟩ $\in$ $Split^d$(q) $\iff$ *the provider* p_n *is responsible for processing* q_n *according to the document* d*, and returning the result.* main *is the XML document which contains tags indicating for each subquery the provider who is responsible for.* */

$Merge^d$: ⟨ { X }, X ⟩ $\longrightarrow$ X
/* $Merge^d$ ($Split^d$ (q)) *is the XML document resulting from* q *processed on* d*.* */

Roughly speaking, the Split operator analyses the query given as parameter, and detects whether there is any navigation expression in this query containing the element trigerredComponents followed immediately by the element serviceExecution. If such a navigation expression is found, this means that the query must be split and executed in a distributed fashion. Accordingly, the Split operator evaluates the navigation expression up to (and including) the leftmost occurrence of the element trigerredComponents. This yields a collection of invocations to component services. The operator Split then retrieves the providers to which these invocations where assigned (through the provider attribute), and associates to each of them a query containing the rest of the navigation path (after the leftmost occurrence trigerredComponents), as well as any part of the original query involving a variable bound to the considered navigation expression.

The Merge operator on the other hand, performs embeds the query outputs that are given to it as parameter, into the output of the locally evaluated part of the query. It then applies any required aggregation function over the resulting document.

This approach is illustrated in Figure 5 and exemplified in the next sub-section.

4.2 Query examples

The following query illustrates the situation that arises when all the data involved in the query are locally stored by the provider who has received the query request.

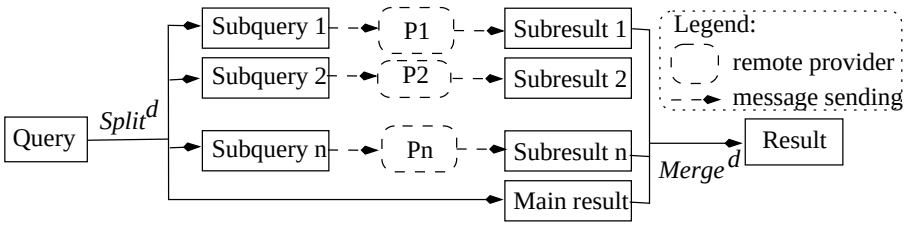

Fig. 5. Querying distributed traces: splitting, dispatching and merging sub-results.

Q.1: **Query locally processed**
For each component triggered in the context of the execution e1 of S, give its name, the Id of its execution instance, and the location where the execution trace is stored.

```
For $se in document("traces.xml")/serviceExecution[@name="S" and @execId="e1"]
   /triggeredComponents/serviceExecution
return <serviceExec name=$se/@name execId=$se/@execId loc=$se/@loc/>
```

In this query, the following expressions are used:

- document("traces.xml") is the root node of the document.
- between [and] is a filter: serviceExecution[@name="S" and @execId="e1"] selects elements whose value of the attribute name is S and for the attribute execId is e1.
- / provides access to the children of the current node. Therefore, /serviceExecution locates children of the root node. The result is a set of nodes, each one is an element containing information required by the query.
- the For ... return loop iterates over the set obtained by the previous step. The variable $se denotes a serviceExecution element.
- @ locates attributes of the current node. Therefore, serviceExecution/@name denotes the attribute name for a given serviceExecution element.

The above query is locally processed since it only involves executions that have been locally hosted. According to the XML document of Section 3.3 the result is:

```
<result>
   <serviceExecution name ="S1" execId="e1" loc="www.prov1.com/S1/e1.xml"/>
   <serviceExecution name ="S2" execId="e1" loc="www.prov1.com/S2/e1.xml"/>
   <serviceExecution name ="S3" execId="e1" loc="www.prov2.com/S3/e1.xml"/>
   <serviceExecution name ="S5" execId="e1" loc="www.prov3.com/S5/e1.xml"/>
</result>
```

Q.2: **Query involving multiple remote sites**
For each component triggered in the context the execution e1 of S, retrieve its name, the IDs of its execution instance its duration, and details about its triggered components (name, execution identifier, and location where the trace has been stored).

```
For $se in document("traces.xml")/serviceExecution [@name="S" and @execID="e1"]
   /triggeredComponents/serviceExecution return
      <service>
         <name> $se/@name </name> <execId> $se/@execId </execId>
```

```
  <duration> $se/time/@start - $se/time/@finish </duration>
  <triggeredComponents> $se/triggeredComponents/serviceExecution
  </triggeredComponents>
</service>
```

In the above query, expressions $se/time/@start, $se/time/@finish and $se/triggeredComponents/serviceExecution cannot be executed locally. The scope of this query includes XML documents remotely stored by providers prov1.com (who hosted execution e1 of S1 and e1 of S2), prov2.com (who hosted execution e1 of S3), and prov3.com (who hosted execution e1 of S5). The processing of this query is described below.

– The first step is to split the query into 4 sub-queries, and to execute the main query. This results in an XML document that contains subqueries to be executed remotely and contains for each of them, the URL of the provider who is responsible for its processing. In the sequel we detail only the part of the document dedicated to the service S1:

```
<result> <service>
  <name> S1 </name> <execId> e1 </execId>
  <query> <recipient> www.prov1.com </recipient>
    <queryText> <duration>
      document("S1/e1.xml")/serviceExecution/time/@start
      - document("S1/e1.xml")/serviceExecution/time/@finish
    </duration> </queryText>
  </query>
  <query> <recipient> www.prov1.com </recipient>
    <queryText> <triggeredComponents>
      document("S1/e1.xml")
        /serviceExecution/triggeredComponents/serviceExecution
    </triggeredComponents> </queryText>
  </query>
</service>
...
/* Subqueries related to S2, S3 and S5 are similarly described */
</result>
```

– The second step, consists in sending each subquery to the corresponding provider who executes it and returns the result:

```
Q1: <duration> document("S1/e1.xml")/serviceExecution/time/@start
  - document("S1/e1.xml")/serviceExecution/time/@finish </duration>
to prov1.com
```

whose result is (see Figure 4): <duration> 5 </duration>.

```
Q2: document("S1/e1.xml")
      /serviceExecution/triggeredComponents/serviceExecution
to prov1.com
```

whose result is: <triggeredComponents> <!– empty –> </triggeredComponents>
Subqueries related to other services (respectively S2, S3 and S5) are processed similarly except they are sent respectively to prov1.com, prov2.com and prov3.com.

– Finally results received for remote subqueries are merged in order to produce the overall query result:

```
<result>
  <service>
    <name> S1 </name> <execId> e1 </execId> <duration> 5 </duration>
    <triggeredComponents> </triggeredComponents>
  </service>
  <service>
    <name> S2 </name> <execId> e1 </execId> <duration> 3 </duration>
    <triggeredComponents> </triggeredComponents>
  </service>
  <service>
    <name> S3 </name> <execId> e1 </execId> <duration> 3 </duration>
    <triggeredComponents> </triggeredComponents>
  </service>
  <service>
    <name> S5 </name> <execId> e1 </execId> <duration> 3 </duration>
    <triggeredComponents> </triggeredComponents>
  </service>
</result>
```

5 Related Work

The issue of collecting traces of Web service executions is addressed in [18]. The authors present a mechanism for tracking messages exchanged between Web services. Traces are represented as pads added to XML messages. The trace of a composite service execution goes from the first component service to be executed to the last one through all the intermediate components that incrementally enrich the traces with data describing their own execution. At the end, the overall trace is stored by the provider who was responsible for executing the initial component service. This peer-to-peer communication for trace collection is very close to the one proposed in our approach. Unlike the present proposal however, [18] does not address the issue of storing and querying traces in a distributed environment. Instead, the entire trace of a composite service execution is stored in a single site.

The issue of tracing the execution of Web services is closely related to that of workflow tracing, which has been addressed in [15] and [12]. [15] presents an approach for tracing the execution of workflows expressed as statecharts. Specifically, the authors show that the process of tracing a workflow execution can itself be seen as a workflow. Consequently, by merging a workflow W, with the workflow dedicated to tracing the execution of W, one obtains a “self-traceable workflow”. Unlike our proposal however, [15] does not discuss the issue of tracing process executions in a distributed and inter-organisational environment, which is the kind of environment where Web services are typically executed. Also, the work reported in [15] differs from ours in that it does not address the issue of querying the traces of process executions.

In [12] the authors assume that workflows are executed in a distributed environment, and that each node (in our context: each provider) maintains the history of its

task executions (in our context: its service executions). Within this context, the authors present several strategies for evaluating queries such as "retrieve the history of a given process instance". In [12], the set of entities participating in the execution of a workflow is assumed to be fixed, whereas our approach caters for runtime provider selection. Our approach also differs from the above one in that we consider traces stored in XML, whereas [12] relies on an object-oriented database supporting OQL.

As discussed in the introduction of this paper, the traces of service executions can be used for different purposes: audit, monitoring, optimisation, etc. In particular, a number of research efforts in the area of workflow management have been directed towards developing techniques for predicting exceptions and preventing deadline expirations by analysing process execution traces (e.g. [8, 6]). [8] studies the use of data mining techniques to analyse (centralised) workflow execution logs, in order to predict and prevent exceptions of various kinds, such as deviations from the optimal or acceptable process execution that hinder the delivery of services with the expected quality.

The above discussion is summarised in Table 3. For each approach, the column Collection states whether the trace collection is done through a central scheduler (centralised orchestration) or through peer-to-peer exchanges between the component services (P2P orchestration). The column Storage states whether the storage of the traces is centralised or distributed. The column Querying, when applicable, indicates the querying techniques used by the approach.

Approach	**Collection**	**Storage**	**Querying**
[8]	N/A	centralised	data mining
[12]	centralised	distributed	OQL
[15]	centralised	centralised	N/A
[14, 18]	P2P	centralised	N/A
our approach	centralised/P2P	distributed	Xquery

Table 3. Comparison of related work on tracing composite service executions

6 Conclusion

The work reported in this paper addressed the issue of tracing composite services. The main contributions are:

- A data model of traces of composite service executions.
- A representation of these traces in XML.
- Two approaches for collecting execution traces: one with a central scheduler, and one based on P2P interactions.
- An approach to store these traces in a distributed environment.
- An approach to execute queries over these distributed traces.

We have implemented a prototype of the collection and querying approaches. The communications between providers are implemented in Java RMI [22]. The query engine has been built on top of Kweelt [19]: a tool that implements Quilt [3] a dialect of

XQuery. The prototype supports most basic XQuery features, although it does not support advanced features such as the closure operator. Ongoing work is being dedicated to generalising the query multiplexer in order to tackle all Xquery expressions, and to design optimisation strategies aimed at minimising communication costs. An example of such optimisation is to group together all the subqueries to be sent to the same provider.

On the other hand efforts are being directed towards designing techniques for analysing traces of past executions in order to perform optimisations and self-tuning both statically and at run-time. In particular, the use of execution traces for run-time provider selection is being studied in the context of the SELF-SERV system [1].

Acknowledgments We thank the anonymous reviewers of CoopIS'02 for their valuable feedback.

References

1. B. Benatallah, M. Dumas, Q.-Z. Sheng, and A. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In IEEE Computer Society, editor, *Proceedings of ICDE'02 Conference*, San Jose, California, 2002.
2. F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eFlow. In *Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.
3. D. Chamberlin, J. Robie, and D. Florescu. Quilt: an XML query language for heterogenous data sources. In *Proc. of the Workshop on the Web and Databases (WedDB). In conj. with SIGMOD'00*, Dallas - Texas, May 2000. Addison Wesley.
4. Q. Chen and M. Hsu. Inter-enterprise collaborative business process management. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.
5. WorkFlow Management Coalition. Terminology and glossary. Technical Report WFMS-TC-1011, Workflow Management Coalition, Brussels - Belgium, 1996.
6. J. Eder, E. Panagos, and M. Rabinovich. Time constraints in workflow systems. In *Proc. of the 11th Conference on Advanced Information Systems Engineering (CAiSE)*, Heidelberg, Germany, 1999.
7. M.-C. Fauvet, M. Dumas, B. Benatallah, and H. Paik. Peer-to-peer traced execution of composite services. In *Proceedings of the International Workshop on Technologies for E-Services (TES 2001). In cooperation with VLDB.*, Roma, Italy, 2001.
8. D. Grigori, F. Casati, U. Dayal, and M.-C. Shan. Improving business process quality through exception understanding, prediction, and prevention. In *Proc. of the 27th VLDB Conference*, Roma, Italy, 2001.
9. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
10. Internet RFC-1305. Network Time Protocol Specification Version 3. http://www.landfield.com/rfcs/rfc1305.html.
11. N.R. Jennings, T.J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Autonomous agents for business process management. *Journal of Applied Artificial Intelligence*, 14(2):145–189, 2000.
12. P. Koksal, S. Arpinar, and A. Dogac. Workflow history management. *SIGMOD Record*, 27(1), 1998.
13. D. Kossman. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), December 2000.

14. V. Machiraju and A. Sahai. A peer-to-peer service interface for manageability. Technical Report HPL-2001-61, Hewlett-Packard Laboratories, 2001. http://www.hpl.hp.com/reports/2001/HPL-2001-61.html.
15. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow history management in virtual enterprises using a light-weight workflow management system. In *Proc. of 9th International Workshop on Research Issues in Data Engineering (RIDE)*, Sydney, Australia, March 1999.
16. P. Muth, D. Wodtke, J. Weissenfels, A.K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2), March 1998.
17. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
18. A. Sahai, V. Machiraju, J. Ouyang, and K. Wurster. Message tracking in SOAP-based Web services. Technical Report HPL-2001-199, Hewlett-Packard Laboratories, 2001. http://www.hpl.hp.com/reports/2001/HPL-2001-199.html.
19. A. Sahuguet, L. Dupont, and T.-L. Nguyen. Kweelt. http://kweelt.sourceforge.net/.
20. H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In *Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.
21. Hans Schuster, Donald Baker, Andrzej Cichocki, Dimitrios Georgakopoulos, and Marek Rusinkiewicz. The collaboration management infrastructure. In *Proc. of the IEEE Int. Conference on Data Engineering (ICDE)*, San Diego CA, USA, February 2000. System Demonstration.
22. Sun Microsystems Inc. Java RMI. http://java.sun.com/products/jdk/rmi.
23. W3C. Extensible Markup Language (XML). http://www.w3.org/XML/.
24. W3C. XInclude. http://www.w3.org/TR/xinclude/.
25. W3C. XML Query. http://www.w3.org/XML/Query.
26. W3C. XPath. http://www.w3.org/TR/xpath20.