Managing Distributed Service Environments: A Data-oriented Approach

Yann Gripay
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle
F-69621, Villeurbanne, France
yann.gripay@liris.cnrs.fr

Marian Scuturici
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle
F-69621, Villeurbanne, France
marian.scuturici@liris.cnrs.fr

ABSTRACT

Managing dynamic and distributed computing environments, a.k.a. pervasive or ubiquitous environments, is currently a major issue in many application domains. The abstraction of functionality as services, representing sensors, actuators, and all other devices ranging from small mobile handsets to powerful servers, is a common way to address this issue. However, managing a great number of dynamic distributed services is still a difficult issue.

In this paper, we present a data-oriented approach for distributed service environments. It relies on a model that homogeneously represents services producing, storing and/or consuming data and data streams, and providing computation and actuator functionality. This model enables the extension of one-shot and continuous query processing techniques to manage distributed service environments. We also describe a protocol of RESTful web services implementing this model, and service-oriented continuous query processing techniques on top of it.

1. INTRODUCTION

Computing environments evolve towards what is called pervasive or ubiquitous systems [10]: they tend to be more and more heterogeneous, decentralized and autonomous. On the one hand, personal computers and other handheld devices are largely widespread and take a large part of information systems. On the other hand, data sources may be distributed over large areas through networks that range from a world-wide network like the Internet to local peer-to-peer connections like for sensors.

Managing those dynamic and distributed computing environments is currently a major issue in many application domains: intelligent buildings, environmental monitoring, remote patient monitoring, *etc.*. The abstraction of functionality as services, representing sensors, actuators, and all other devices ranging from small mobile handsets to powerful servers, is a common way to address this issue [1, 11]. This abstraction enables several approaches to manage those environments: service discovery, service composition and orchestration, service-oriented architectures, *etc.* However, managing a great number of dynamic distributed services is still a difficult issue, in particular when dealing with heterogeneous service functionality that can handle data and data streams.

In this paper, we present a data-oriented approach for distributed service environments, which is an ongoing work within a French National Research Agency (ANR) project called Optimacs [6]. This approach relies on a model that homogeneously represents services producing, storing and/or consuming data and data streams, and providing computation and actuator functionality. Using concepts from the relational model, we aim at building a logical data model of such environments.

Moreover, this model enables the extension of one-shot and continuous query processing techniques to manage distributed service environments [9]. The integration of our service model within a complete data model, namely the SoCQ (Service-oriented Continuous Query) data model, is tackled in [3]. Applications can then be expressed as declarative queries [2, 8], leading to new opportunities for query optimization in this context. It is however out of the scope of this paper: in this paper, we just sketch the SoCQ data model and associated query processing techniques.

We also propose a protocol of RESTful web services that implements our service model, namely the SoCQ Data Service protocol [7]. This protocol is based on the HTTP protocol, but exploit it in a "relational way".

The rest of the paper is organized as follows. In Section 2, we present our service model through some definitions and notations, and also discuss some interesting extensions. We describe the service protocol in Section 3 and sketch the SoCQ data model in Section 4. We then conclude and discuss some perspectives in Section 5.

2. SERVICE MODEL

We propose a data-oriented approach for dynamic distributed services. We first state some definitions and then define some notations. We reuse standard definitions and notations from the relational model, borrowed from [5], for the following data-oriented concepts: attribute, schema, tuple, relation.

2.1 Definitions

In our approach, a service is seen as a container of capabilities, called service resources. A service is neither a function nor a datasource in itself, but is a provider of such resources. A resource is represented as a method, a relation, an output stream or an input stream. The same resource can be implemented by several services. Operations are the possible interactions with a resource implemented by a given service.

Definition 1 (Service). *A* service *is a logical entity identified by an URL. It is described by a name and a list of service resources it provides. A service is dynamic: at a given time, it is either available or unavailable.*

DEFINITION 2 (SERVICE RESOURCE). *A* service resource *is a logical component identified by a name. It is described by a type, an input schema and an output schema. There exist 4 types of resources:*

1. ***Method:*** *A* method resource *is a computation or actuator component. It represents a functionality that can be invoked. An invocation is parameterized by a tuple of data over its input schema and produces n tuples (with $n \geq 0$) of data over its output schema*[1]*.*
   Example: SENDMESSAGE(ADDRESS,MESSAGE):(SENT)
2. ***Relation:*** *A* relation resource *is a data storage component. It stores tuples of data over its output schema. Tuples can be inserted, deleted and retrieved.*
   Example: CONTACTS():(NAME,ADDRESS,JOB)
3. ***Output Stream:*** *An* output stream resource *is a continuous data producer component. It produces tuples of data over its output schema and sends them to its subscribers (a.k.a. consumers). A subscription is parameterized by a tuple of data over its input schema.*
   Example: TEMPERATURES(THRESHOLD):(TEMPERATURE)
4. ***Input Stream:*** *An* input stream resource *is a data sink component. It consumes tuples of data over its input schema.*
   Example: VIDEOINPUT(IMAGE):()

DEFINITION 3 (SERVICE RESOURCE OPERATION). *A* service resource operation *is an operation that can be performed on a service resource to interact with it. There exist 3 operations, which behavior depends on the type of the resource:*

1. ***retrieve-data:*** *It retrieves data from a resource. It retrieves stored tuples from a relation. It subscribes to an output stream, with some given input parameters, in order to continuously receive produced tuples. It invokes a method, with some given input parameters, and retrieves tuples from the invocation result (it may then be called an* ***invoke*** *operation). It is not defined for an input stream.*
2. ***insert-data:*** *It inserts data into a resource. It inserts some given tuples into a relation. It feeds an input stream with some given tuples. It is not defined for a method or an output stream.*
3. ***delete-data:*** *It deletes data from a resource. It deletes some stored tuples from a relation. It is not defined for a method, an output stream or an input stream.*

2.2 Notations

Within our data-oriented approach, we now define the structure of a data model for dynamic distributed services. It is based on five mutually disjoint infinite countable sets: the discrete time domain $\mathcal{T}$ of "time instants", constants (*i.e.*, data values) $\mathcal{D}$, attributes $\mathcal{A}$, services Ω, and resources (or "prototypes of functionality" [3]) Ψ.

[1]In a data-centric perspective, we suppose that every method has at least one attribute in its output schema. If a method does not have a "natural" one (*e.g.*, some actuator functionality like switching on a light or sending a message), its output schema should still contain one boolean attribute. An invocation of such methods should then return one tuple with the boolean value TRUE.

We also define the boolean domain $\mathcal{B} = \{true, false\} \subset \mathcal{D}$ and the service URL domain $\mathcal{U} \subset \mathcal{D}$. The set of resources is furthermore partitioned in 4 disjoint subsets representing the 4 types of resources: methods $\mathcal{M}$, relations $\mathcal{R}$, output streams $\mathcal{O}$, input streams $\mathcal{I}$.

A schema S represents a set of attributes, denoted by $schema(S) = \{A_1, ..., A_n\} \subset \mathcal{A}$. A tuple t over a schema S contains a data value for each attribute of S, *i.e.*, it is an element of $\mathcal{D}^{[S]} = Dom(A_1) \times ... \times Dom(A_n) \subseteq \mathcal{D}^n$, where $Dom(A_i) \subseteq \mathcal{D}$ is the domain of attribute A_i.

A service $\omega \in \Omega$ has an identifier (*i.e.*, its URL) denoted by $id(\omega) \in \mathcal{U}$. Its availability at instant $\tau_i \in \mathcal{T}$ is denoted by $available(\omega, \tau_i) \in \mathcal{B}$. Its set of resources is denoted by $resources(\omega) \subset \Psi$, where a resource $\psi \in \Psi$ has an input schema $Input_\psi$ and an output schema $Output_\psi$.

We represent service operations as functions:

1. Operation **retrieve-data** from resource ψ on service ω with input tuple t at time τ_i:

$$retrieve_\psi : \begin{array}{ccc} (\mathcal{U}, \mathcal{D}^{[Input_\psi]} \cup \emptyset, \mathcal{T}) & \rightarrow & \mathcal{P}(\mathcal{D}^{[Output_\psi]}) \\ (id(\omega), t, \tau_i) & \mapsto & r \end{array}$$

   The **function result** is a finite set of tuples over $Output_\psi$. For a method or relation resource ($\psi \in \mathcal{M} \cup \mathcal{R}$), the **operation result** is the function result at instant τ_i. The input tuple is however not expected for a relation (*i.e.*, $t = \emptyset$). For an output stream resource ($\psi \in \mathcal{O}$), the **operation result** is the sequence of function results for $\tau_j \geq \tau_i$, as long as the subscription lasts.

2. Operation **insert-data** into resource ψ on service ω with input tuple t at time τ_i:

$$insert_\psi : \begin{array}{ccc} (\mathcal{U}, \mathcal{D}^{[Input_\psi]}, \mathcal{T}) & \rightarrow & \mathcal{B} \\ (id(\omega), t, \tau_i) & \mapsto & b \end{array}$$

   For a relation or input stream resource ($\psi \in \mathcal{R} \cup \mathcal{I}$), a tuple over the input schema $Input_\psi$ is expected. However, for a relation resource ($\psi \in \mathcal{R}$), the input schema is always considered to be the same as the output schema: $Input_\psi = Output_\psi$.

3. Operation **delete-data** from resource ψ on service ω with predicate p at time τ_i:

$$delete_\psi : \begin{array}{ccc} (\mathcal{U}, P_{[Output_\psi]}, \mathcal{T}) & \rightarrow & \mathcal{B} \\ (id(\omega), p, \tau_i) & \mapsto & b \end{array}$$

   $p \in P_{[Output_\psi]}$ denotes a predicate over the output schema $Output_\psi$ of the relation. It can only be applied on a relation resource ($\psi \in \mathcal{R}$).

The integration of those notations within a complete data model is tackled in [3] (cf. Section 4). The representation of services is focused on method and output stream resources, but can be smoothly extended to integrate relation and input stream resources as defined in this paper.

2.3 Ongoing Extensions

We are currently working on two extensions of our model. The first one is the definition of **derived resources**. A derived resource ψ' is a component of another resource ψ that enables other types of interactions with ψ. For example, an output stream resource can be derived from a relation resource so that subscriptions to the stream are notified of tuples inserted into and deleted from the relation.

The second one is a new type of resources: **context resources**. It can represent a set of static and dynamic properties describing a service (or another resource, through a derived resource). Static properties could be simply retrieved (*e.g.*, service name, device type), or also updated in order to configure a service (*e.g.*, period of sampling for a sensor). Dynamic properties could be subscribed to, like for output streams, in order to be notified of value changes (*e.g.*, service location, battery level).

3. SERVICE PROTOCOL

In order to implement our proposition, we propose a protocol of RESTful web services: the SoCQ Data Service protocol [7]. This protocol is based on HTTP requests. Service operations are handled in a "relational way", where input and output data are represented by tuples over input and output schemas.

3.1 Services and Resources

A service is identified by an URL. This URL allows to retrieve the description of a service, in particular the list of resources it provides.

```
http://ds.example.com/service1

Name: MyExampleService
Description: Example SoCQ Data Service
Resources:
    sendMessage:Method
    contacts:Relation
    temperatures:OutputStream
    videoInput:InputStream
```

A resource is identified by its name within a service: it can be simply accessed by the URL of the service augmented by the name of the resource. It then allows to retrieve the description of the resource.

```
http://ds.example.com/service1/sendMessage

Name: sendMessage
Type: Method
Description: Send a mail message to the given address
Schema.Input:
    address:String
    message:String
Schema.Output:
    sent:Boolean
```

3.2 Data Representation

In this protocol, we use five basic data types: STRING for character strings, INTEGER and REAL for numbers, BOOLEAN for boolean values and BINARY for other data types (like images, sound files). Input and output schemas are a list of attributes, with their name and their data type.

An input data tuple is sent within a HTTP request as a list of request parameters (with their value) corresponding to each input schema attribute. System parameters can be added, like `ds:encoding.input` to specify the input encoding of string and binary values (*e.g.*, base64 encoding).

```
// Input parameters for Method sendMessage
address=yann.gripay@liris.cnrs.fr
&message=Hello World!

// Same parameters, but with base64 encoding
address=eWFubi5ncmlwYXlAbGlyaXMuY25ycy5mcg==
&message=SGVsbG8gV29ybGQh
&ds:encoding.input=base64
```

Output data tuples are retrieved in the HTTP response body as a list of tuples. A tuple is itself a list of values for each attribute of the output schema. Each value is on a single line of the response. String and binary values are base64 encoded in order to keep the protocol format[2]. A human user may however specify to retrieve unencoded string values by using the system parameter `ds:encoding.output`.

```
// Tuples from Relation CONTACTS
name:WWFubg==
address:eWFubi5ncmlwYXlAbGlyaXMuY25ycy5mcg==
job:VGVhY2hpbmcgQXNzaXN0YW50

name:TWFyaWFu
address:bWFyaWFuLnNjdXR1cmljaUBsaXJpcy5jbnJzLmZy
job:QXNzaXN0YW50IFByb2Zlc3Nvcg==
```

3.3 Operations

Service resource operations are invoked using the HTTP GET operation on the service resource URL augmented with the name of the operation, with some request parameters:

1. **retrieve-data:** `<resource-url>/get`
   For methods and output streams, an input tuple is expected, but not for relations.
2. **insert-data:** `<resource-url>/insert`
   For relations and input streams, an input tuple is expected.
3. **delete-data:** `<resource-url>/delete`
   For relations, a predicate over the output schema is expected (more details in [7]).

The HTTP response for **insert-data** and **delete-data** operations is empty (the HTTP status code may however indicate an error). The HTTP response for **retrieve-data** is a document containing output data tuples. For methods and relations, it is a finite document with a finite number of tuples. For output streams however, it is an infinite document where output tuples are printed when they are produced by the resource: it represents a subscription to this resource that will end only when the client terminates the HTTP connection.

3.4 Ongoing Extensions

We are currently working on extending the data type system within our protocol. Besides our five basic atomic data types, we aim at integrating two **complex data types**: LIST(<TYPE>) (*e.g.*, LIST(STRING)) and OBJECT(<SCHEMA>) (*e.g.*, OBJECT(NAME STRING, ADDRESS STRING, AGE INTEGER)), that can also be nested (*e.g.*, LIST(OBJECT(NAME STRING, ADDRESSES LIST(STRING)))). Those data types would be handled like other data types within schema and tuples. In the protocol format for input and output tuples, complex data types could be represented like in JSON [4].

Moreover, we aim at integrating **refined data types** to build more precise data types, *e.g.*, STRING.URL.SERVICE, BINARY.IMAGE.JPEG, OBJECT.SPATIAL.COORDINATES(X REAL, Y REAL). This refinement could enable the integration of some existing type systems like MIME types.

Finally, in order to follow more closely REST architecture principles, the protocol will allow to invoke different resource operations on the same resource URL by using different standard HTTP methods like GET, POST, PUT, DELETE.

[2]Decoding encoded tuples from the example is left as an exercise for the reader... ;-)

4. QUERYING DATA AND SERVICES

Within our data-oriented approach, we have built a complete data model, namely the SoCQ data model [2, 3], on top of our service model. It provides a declarative query language to homogeneously handle data, data streams and services. In a similar way to databases, we have defined the notion of "relational pervasive environment" composed of several eXtended Dynamic Relations, or XD-Relations.

The schema of an XD-Relation is composed of attributes that are either real or virtual. Virtual attributes do not have a value at the data level, but represent input and output parameters of service resources that may receive some values through some query operators. A schema of XD-Relation is further associated with binding patterns indicating which service resources are involved (with which input/output attributes), whereas service identifiers (*i.e.*, service URLs) are handled at the data level. An XD-Relation may be either finite (*i.e.*, like a standard relation) or infinite (*i.e.*, a data stream). Currently, binding patterns represent either invocations of a method resource or subscriptions to an output stream resource (STREAMING keyword).

```
RELATION contacts (
  name      STRING,
  address   STRING,
  messenger SERVICE,
  message   STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, message ) : ( sent )
);
```

name	address	messenger	message	sent
nicolas	nicolas@elysee.fr	http://.../mailer	*	*
carla	carla@elysee.fr	http://.../mailer	*	*
françois	francois@im.gouv.fr	http://.../jabber	*	*

```
RELATION sensors (
  sensor      SERVICE,
  location    STRING,
  temperature REAL     VIRTUAL
USING BINDING PATTERNS (
  getTemperature[sensor] ( ) : ( temperature ),
  temperatures[sensor] ( ) : ( temperature ) STREAMING
);
```

sensor	location	temperature
http://ds.example.com/sensor1	roof	*
http://ds.example.com/sensor2	corridor	*
http://ds.example.com/sensor3	office	*

Given the two examples XD-Relations CONTACTS and SENSORS, we can express an application of "temperature surveillance" by a simple service-oriented continuous query. The following query subscribes to the output streams of the sensors, and sends alert messages when a given threshold (36.0°C) is reached.

```
SELECT location, name, sent
STREAMING UPON insertion
FROM   sensors, contacts
WITH   message := concat("Alert in ",location)
WHERE  temperature > 36.0
  AND  location != "roof"
USING temperatures [1], sendMessage
```

In [3], besides standard relational operators that are redefined over XD-Relations, operators dedicated to virtual attributes and binding patterns are defined. Among them, the service discovery operator can build XD-Relations that represent a set of available services providing some required resources. For example, the XD-Relation SENSORS could be the result of such an operator and be continuously updated when new temperature sensor services become available and when previously discovered services become unavailable.

5. CONCLUSION

In this paper, we have tackled the issue of managing distributed service environments through a data-oriented approach. We propose a model that homogeneously represents services with the notion of service resources (methods, relations, output streams and input streams) and their associated operations (retrieve-data, insert-data, delete-data).

This model has been implemented as a protocol of RESTful web services respecting this data-oriented approach. In particular, input and output data for interactions with services are always tuples over input and output resource schema. This model has been integrated into a complete data model, leading to service-oriented continuous query processing techniques that can be implemented on top of the described protocol.

Through our data-oriented approach, "pervasive" applications and other user queries in dynamic distributed service environments can be expressed in a simple declarative way. It creates new opportunities for optimization techniques in such environments. We then aim at developing a benchmark for those environments to evaluate the performance of "hybrid queries" [6] involving data and services with objective indicators.

6. ACKNOWLEDGMENTS

This work is part of the French National Research Agency (ANR) project OPTIMACS [6] (ANR-08-SEGI-014, 2008–2011) involving the following institutions: LIG (Université de Grenoble, France), LIRIS (Université de Lyon, France), and LAMIH (Université de Valenciennes, France). The authors thank all the members of the project that contributed directly or indirectly to this paper.

7. REFERENCES

[1] D. Estrin et al. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[2] Y. Gripay, F. Laforest, and J.-M. Petit. SoCQ: a Framework for Pervasive Environments. In *ISPAN 2009*, 2009.

[3] Y. Gripay, F. Laforest, and J.-M. Petit. A Simple (yet Powerful) Algebra for Pervasive Environments. In *EDBT 2010*, 2010.

[4] JSON (JavaScript Object Notation). `http://www.json.org/`.

[5] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[6] Optimacs Project. `http://optimacs.imag.fr/`.

[7] SoCQ Data Service. `http://liris.cnrs.fr/ds/`.

[8] SoCQ Project. `http://liris.cnrs.fr/socq/`.

[9] G. Vargas-Solar et al. *IGI 2010*, chapter Querying Issues in Pervasive Environments. 2010.

[10] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1991.

[11] F. Zhu, M. Mutka, and L. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.