# Distributed Cross-Domain Configuration Management

Liliana Pasquale[1], Jim Laredo[2], Heiko Ludwig[2], Kamal Bhattacharya[2], and Bruno Wassermann[3]

[1] Politecnico di Milano, Italy
pasquale@elet.polimi.it
[2] IBM TJ Watson Research Center, USA
{laredoj,hludwig,kamalb}@us.ibm.com
[3] University College London, UK
b.wassermann@cs.ucl.ac.uk

**Abstract.** Applications make extensive use of services offered by distributed platforms ranging from software services to application platforms or mere computational resources. In these cross-domain environments applications may have dependencies on services or resources provided by different domains. A service management solution based on a centrally managed configuration management database (CMDB) is not viable in these environments since CMDB federation does not scale well to many domains. In this paper we propose a distributed configuration management approach by applying standard technologies (e.g., REST services, ATOM feeds) to provide access to and distribution of configuration information. A domain exposes individual configuration items as RESTful web service resources that can be referred to and read by other domains in the context of service management processes. Using this distributed approach, organizations can engage in effective service management practices avoiding the tight integration of CMDBs with their service providers and customers.

## 1 Introduction

Applications make extensive use of services offered by distributed platforms hosted in different domains. These platforms range from software services (Software-as-a-Service, SaaS), to application platforms (e.g., facebook.com) to mere computational resources (e.g., Amazon Elastic Compute Cloud). Often, applications make use of different services from different providers, e.g., for storage and application platforms, and may be also integrated with in-house, dedicated software. Hence applications may depend on services or resources provided by different organizational domains. In such a loosely-coupled environment, providers are not even aware of the set of other organizations currently using their services. Furthermore, the wide adoption of web standards to consume and provide services facilitates the easy establishment and the change of these cross-domain configuration relationships. If providers conduct changes independently of their

L. Baresi, C.-H. Chi, and J. Suzuki (Eds.): ICSOC-ServiceWave 2009, LNCS 5900, pp. 622–636, 2009.
© Springer-Verlag Berlin Heidelberg 2009

clients, the clients services may be disrupted. For this reason clients need to understand on which external configurations they depend on.

Configuration management plays a crucial role for other service management processes, e.g. incident management, change management, or process management, whose activities depend on configuration information of the environment. Hence, management activities have to take into account the distribution of configuration information across organizational boundaries due to the presence of inter-domain dependencies. Moreover, when a configuration changes it is necessary to provide some mechanisms to manage these changes, notifying interested clients. This becomes of high importance especially in those environments in which an outage caused by an unmanaged configuration change may be propagated along a chain of dependencies.

Current service configuration management approaches rely on a centrally managed configuration management database (CMDB) [1], which collects the state of hardware and software entities, represented by Configuration Items (CIs) [2]. When changes happen in a CI, specific operations need to be performed on other CIs that depend on it. A service management solution based on a central CMDB is not viable in cross-domain environments since CMDB federation does not scale well to many domains and different organizations are often reluctant to provide direct access to their CMDBs.

In general there are different issues configuration management must address for distributed, loosely coupled environments:

- **Discovery:** The lack of scope and access to resources of other domains makes hard to discover CIs outside ones' own management domain.
- **Dependency management:** Detect the management domains an CI depends on is not an easy task.
- **Cross-Domain configuration analysis:** It is not always feasible to aggregate and combine configuration information of different domains in a straightforward way, to ease management activities.

In this paper we propose a distributed configuration management approach by applying standard Web technologies (e.g., REST services, ATOM feeds) to help to solve the issues described above and provide access and distribution of configuration information. A domain exposes individual CIs as RESTful web service resources that can be referred to and read by other domains in the context of service management processes. Domains can manage dependencies on outside resources in the form of URLs. Using this distributed approach, organizations can engage in effective service management practices while not requiring tight integration with their service providers and customers. This approach and the specific application to change management has been shown in [3] and [4].

The paper is organized as follows. Section 2 analyzes the problems of cross-domain configuration management using an example. Section 3 gives an overview of the architecture of our solution. Section 4 explains the approach of Smart Configuration Items, including their publication, consumption, and format. Subsequently, section 5 illustrates how configuration information can be aggregated

across domains. Finally, section 6 discusses implementation, section 7 summarizes related approaches, and section 8 concludes the paper.

## 2 Problem Analysis

In this section we discuss the main challenges of cross-domain configuration management using an example scenario, shown in Figure 1. A startup company, E-Shop, integrates different retailers, to advertise and sell their products. E-Shop relies on a distributed application infrastructure whose elements are owned and managed by different organizations. In our example, *Domain A* provides an application server (*AS-A1*), hosting the service which advertises the products to sell (*Advertise*). *Domain A* also hosts a database management system (*DBMS-A1*) which controls several databases (e.g., *DB-A1*, etc.). Both the application server and the DBMS are hosted on a virtual machine, represented through its address (*131.34.5.20*). Each machine can provide one or more file systems. The same situation holds for *Domain B*, which provides the service that performs payments (*Payment*), and some storage facilities.

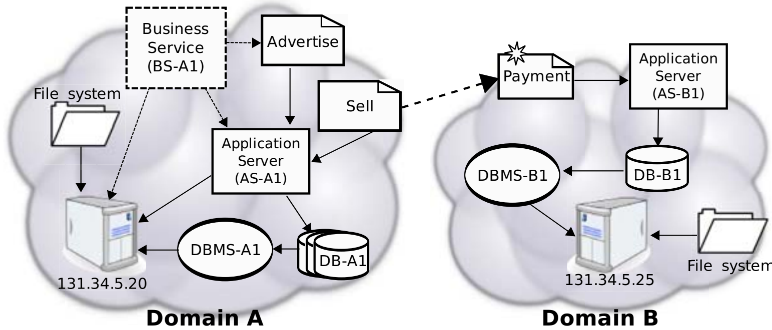


**Fig. 1.** Running Example

In the scenario we have intra-domain dependencies between CIs, represented through straight arrows, and inter-domain dependencies, represented through dashed arrows. For example, services provided by *Domain A* depend on the application server on which they are deployed. While file systems, application servers and DBMSs depend on the machine in which they are installed. Moreover, databases depend on the DBMS by which they are managed and application servers depend on local/remote DBs used by applications deployed on them (e.g., through a Web services connection). Furthermore each application may depend on services of another domain. In our example, application *Sell* depends on service *Payment*, provided outside its domain.

Finally, E-Shops marketing campaign is carried by several business services that can be considered as “abstract” CIs relying on “concrete” elements of the

infrastructure. Figure 1 shows a business service (*BS-A1*) that depends on those CIs that implement it (service *Advertise*, application server *AS-A1* and the machine *131.34.5.20*). This case highlights the need to trace properties and dependencies of CIs that do not correspond to an element provided by the underlying infrastructure (e.g., business services), since changes on the infrastructure may also impact on these abstract elements.

This example illustrates the main functional issues that need to be addressed by a cross-domain configuration management approach:

- **Publication of configuration information:** Management domain must select internal CIs relevant for other domains and provide them in a convenient way.
- **Identification of cross-domain dependencies:** When performing discovery in a domain, a configuration management system must identify those CIs that depend on external CIs and manage the dependency (e.g., receiving notifications when external CIs change).
- **Multi-domain configuration analysis:** In the course of service management processes, analysis is conducted through entire configurations, e.g., for root cause analysis. Organizations must be able to aggregate configuration information from multiple domains.

These functions enable a management domain to conduct configuration management in a multi-domain environment involving multiple service providers.

## 3 Overview of the Approach

Our approach deals with configuration information for each single domain of the infrastructure. This information is published on one or more web servers authoritative for a domain and can be consumed in a standard way through REST and ATOM [5] protocols. Local configuration management also provides distributed and cross-domain benefits, since information about the overall infrastructure can be easily published and obtained aggregating that available for each local domain. Figure 2 shows the application of our solution for our running example. It provides two main functionality: Smart Configuration Management and Cross Domain Aggregation.

**Smart Configuration Management.** All CIs are detected for each resource of a domain, through a discovery process *(1)*. We call these configuration items Smart Configuration Items (SCIs): they represent the properties and the inter- and intra-domain dependencies of an element of the infrastructure. Our discovery process is also able to resolve cross-domain dependencies, that in general are hard to identify, through the DSM Registry *(2.b)*. SCIs and their dependencies may also be established manually, when elements they represent cannot be detected through the discovery mechanisms (e.g., the business service we adopted in our example). Each SCI is associated with a feed document carrying on its changes. SCIs and feed documents generated after the domain configuration discovery

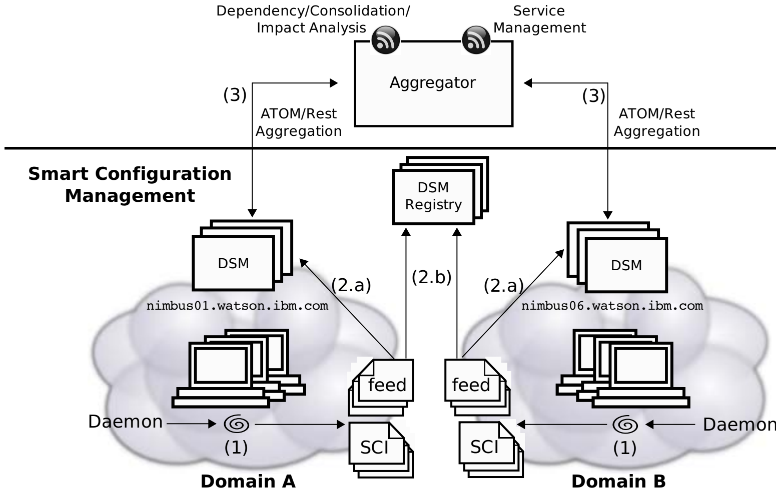


**Fig. 2.** Solution Architecture

are periodically published *(2.a)* on a authoritative web server known as Domain Service Manager (DSM), which serves local information via REST or as ATOM feeds to interested parties.

**Cross-Domain Aggregation.** Information about the overall infrastructure is obtained querying to an Aggregator that is in charge to communicate to all DSMs of the infrastructure *(3)*. This information is provided through a REST or an ATOM aggregation. The first one is synchronous and offers information about all SCIs of the infrastructure or all SCIs of a particular type. While, the latter is asynchronous and generates notifications if some changes happened in one or more SCIs that comply to specific features. Aggregation provides configuration information in a flexible way and eases the adoption of this information to perform several kinds of analysis (e.g. dependency, consolidation, impact analysis, compliance analysis) or to perform service management operation (e.g., change management, incident management, etc.).

The basic tenet of our approach is to use Web-based techniques for dealing with cross-domain management issues. The flexibility of the REST-based approach and the wide availability of tooling to create/consume SCIs and their associated feeds makes possible to easily manage configuration information also for cross-domain environments. For example, we can hypothesize to have listeners from *Domain A* for a change in service *Payment* of *Domain B*. In this case, after a feed listener in *Domain A* is notified about the change of parameter `AcceptedPayments` of service *Payment* it can trigger a new internal change process (supposed that this change is relevant), resulting in the participation of *Domain A* in *Domain B*'s change process.

# 4 Smart Configuration Items

DSM is the enabling element for domain configuration management. It tracks in its internal registry all available SCIs in a domain. Each SCI is associated with a unique id, a set of properties/dependencies (address, port, type, etc.) able to unambiguously identify it and two paths in the domain file system pointing respectively to the location of the SCI document and the feed document containing configuration changes. We also provide a DSM Registry, which associate each DSM with the hosts it is authoritative for. A DSM Registry may be available in a single domain and is in charge to communicate with other DSM Registries provided by other domains. The DSM offers RESTful services to retrieve, create, modify or delete discovered SCIs. Users can access to SCI information through a simple GET operation on the SCI URL constructed as follows:

```
http://<DSM_HostName>:8080/sci?id=<id>
```

where `<DSM_HostName>` is the address of the DSM and `<id>` is the identifier of the requested SCI. While the feed document associated with an SCI can be retrieved at the following URL:

```
http://<DSM_HostName>:8080/feed?id=<id>
```

Feeds can also be consumed through a standard feed reader. DSM also provides a graphical interface system administrators can use to perform several operations on local SCIs. It allows to visualize information about all SCIs available in a domain (URL, type, properties and dependencies). It also permits to recursively traverse the dependency chain of an SCI, with the possibility to reach SCIs involved in a inter-domain dependency, which are not local. For example, from the SCI associated with application *Payment* of *Domain A*, it is possible visualize information about its application server (*AS-A1*). This is still valid if the requested SCI is managed by another DSM, e.g., belonging to *Domain B*.

We also allow domain administrators to add a new SCI to represent configuration information that is not discovered automatically. This functionality is adopted when we need to add SCIs representing business services (e.g., *BS-B1*, in our example) that rely on infrastructural resources, but cannot be detected through the standard configuration discovery algorithms. The interface also permits to manually modify an existing SCI e.g., adding inter-domain dependencies when they cannot be discovered automatically.

## 4.1 Configuration Data Model

An SCI status is represented through an extensible XML document, able to address the descriptive requirements of different configuration domains.

In Figure 3 on the left we show the SCI logic schema, while on the right we propose an example of SCI document associated with application *Sell* of the proposed scenario. Each SCI is described through a set of mandatory attributes: `uri`, which represents the URL that uniquely identifies the SCI (on the DSM which is actually keeping it); `type`, which is the type of the represented item (DBMS, application server, database, etc.). In Figure 3 attribute `type` is set to `application`

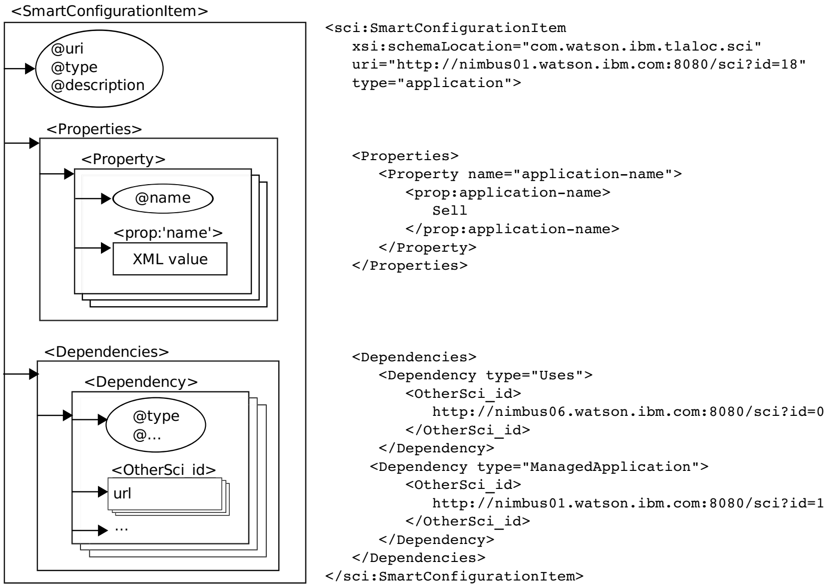


**Fig. 3.** SCI document schema and example

and attribute `uri` indicates that the SCI is kept in the DSM authoritative for *Domain A*, since it starts with hostname `nimbus01. watson.ibm.com`. SCIs can also have optional attributes (e.g., `description`, which gives a human readable description of the SCI).

An SCI can have any number of properties, defined by a name and an XML value. The property name is equal to the local name of the XML tag enclosing the property value. This mechanism allows users to define their own properties that can have values compliant to an arbitrary schema. In the proposed example the application is described through property: `application-name`.

An SCI has zero or more dependencies, specified by a type and a list of URLs identifying SCIs on which the item depends. Extension points are provided to insert new attributes and elements describing the nature of the dependency. In our example we have two kinds of dependencies: `ManagedApplication` and `Uses`. The first is on the SCI representing the application server in which application *Sell* is deployed. While the latter indicates a dependency on the adopted service (*Payment*). This last dependency is not local since the corresponding SCI is available on the DSM authoritative for *Domain B* (`nimbus06.watson.ibm.com`).

Besides the representation of the current SCI in the DSM, the discovery process produces a feed outlining SCI changes compared to the previous discovery. Possible changes are: add/delete/modify property, add/delete dependency, or add/delete a SCI pointer into a dependency. An example of the feed document associated to web service *Payment* is shown in Figure 4. It is updated after the input message of operation *PayOrder* change type from `tns:RPType`

to `tns:RPAllowedType`. Change descriptions are enclosed into element `<entry>` in the feed document. In the example we have two entries. The first is created when an SCI associated with web server *Payment* is added for the first time to the authoritative DSM, while the second one advertises the change of operation *PayOrder*. Change information is carried on by element `<property-change>` and is described through the following attributes: `type` that represents the kind of change happened (`ChangePropertyValue`); `xpath`, which points to the modified property/dependency (in this case, property `alias-name`); `uri`, that is the url of the corresponding SCI; and `feed-uri` that is the feed url. Each change is described through two sub-elements: `<old>`, which contains the previous value of the property/dependency and `<new>`, which contains the new value of the considered property/dependency. If the change is an addition or a deletion of a property/dependency, element `<old>` or `<new>`, respectively, are not inserted in the change description.

```
<entry>
   <title>SCI Added</title>
   <id>random id</id>
   <updated>2008-12-14T18:30:02Z</updated>
   <content type="TEXT">
      A new service was added to DSM nimbus06.watson.ibm.com;
   </content>
</entry>
<entry>
   <title>Modify Property Entry</title>
   <id>random id</id>
   <updated>2008-12-14T19:30:02Z</updated>
   <content type="XHTML">
      <!-- the element Property is modified -->
      <pc:property-change xmlns:pc="com.ibm.tlaloc.propEntryContent"
           type="ChangePropertyValue"
           xpath="//Property[@name='operationq']"
           uri="nimbus06.watson.ibm.com:8080/sci?id=0"
           feed-uri="nimbus06.watson.ibm.com:8080/feed?id=0"
           propertyName="operations">
              <pc:old>
                  ...
                 <wsd:operation name="PayOrder">
                    <wsdl:input message="RequstPayment" type="tns:RPType"/>
                  ...
              </pc:old>
              <pc:new>
                 <wsd:operation name="PayOrder">
                    <wsdl:input message="RequstPayment" type="tns:RPAllowedType"/>
                  ...
              </pc:new>
       </pc:property-change>
     </content>
  </entry>
```

**Fig. 4.** An example of configuration change

### 4.2 Domain Configuration Discovery

SCIs rely on a local discovery mechanism to report the dependencies and properties of each CI. The local discovery gives us another level of granularity removing the need of any centralized repository, ideally for a more distributed approach,

yet given the complexity of comprehensive discovery mechanisms it is necessary to make trade-offs as to how close to the CIs we can place the discovery engine given their resource requirements.

A discovery process must detect the main SCIs available on those virtual machines that it covers and, for each of them, it must find their main properties and dependencies on other SCIs (that can belong to that domain or to other domains). For example, it must discover the basic properties of virtual machines, e.g., their operating system and the hostnames associated with them. Moreover, a discovery process must find the servers installed on each host (e.g., DBMSs, application servers, http servers), their main properties (e.g., for a DBMS, the ports it listens to, its type and version), and dependencies (e.g. a DBMS is associated with the host in which it is installed). A discovery process must also detect SCIs managed by the servers installed on a host (e.g., applications managed by an application server). From the discovered properties and dependencies we also want to identify each SCI uniquely, among other SCIs of the same type. For example, a DBMS can be uniquely identified through the host in which it is installed and the ports it listens to. Finally we also require discovery to be performed periodically and automatically upon configuration change (e.g., with a specific periodicity or when something happens, for example a new component is installed or an existing one is upgraded).

Taking into account these requirements we demonstrated our approach using Galapagos [6], a lightweight discovery mechanism acting on a per virtual node basis. In particular we embedded in our discovery agent the Galapagos capability. The agent converts information discovered by Galapagos into several SCI state representations. The adoption of Galapagos satisfies our requirements since it is able to detect all basic elements provided by common virtual machines (file systems, http servers and their virtual hosts, databases, DBMSs, application servers, etc.). Furthermore, Galapagos is primarily tailored for IBM software (e.g., DB2, IBM HTTP Server, WebSphere Application Server, etc.), for which it can discover a wider set of properties. Finally we allow to perform discovery periodically depending on specific needs in terms of times and frequency of scans, or it can be triggered by particular events, like failures, software/hardware upgrades, etc.

### 4.3 SCI Dependency Resolution and Management

The discovery agent inspects all CIs starting from those that have no dependencies (e.g. a virtual machine) up to those that may have numerous dependencies (e.g. application servers, applications).

If we consider host *131.34.5.20* of *Domain A*, discovery will follow the following steps:

1. **host (mandatory):** It leverages data describing the host *131.34.5.20* in which discovery is performed to create an SCI of type `host` which has no dependencies and has at least two properties: `os`, which represents its operative system, with a name (Linux) a version (2.6.18 - EL5.02), etc., and `lan`,

which carries on hostnames associated with that host (`nimbus03.watson-.ibm.com`).

2. **File Systems (mandatory):** It transforms information regarding mounted file systems into an SCI of type `file_system`, which is described by the following properties: `fs-device` (file system device), `fs-mount-point` (mount point), `name` (file system name), `fs-mode` (read only/write mode). It also depends on the host providing its mount point, represented by dependency `HostedBy`. This dependency is within the domain and the corresponding SCI is detected at step 1.
3. **DBMSs:** An SCI of type `dbms` is created for `DBMS-A1`, found during discovery. It is characterized by a hostname (property `host-name`) and a set of ports it listens to (property `ports`). Each dbms depends on the SCI created at step 1 and associated with the host in which it is actually installed (dependency `HostedBy`).
4. **Databases:** Database `DB-A1` found during discovery is transformed into an SCIs of type `db`, described through a database name (property `databasename`) and an alias name (property `alias-name`). It also depends on the SCI associated with its DBMS (`DBMS-A1`). For this reason, dependency `ManagedDB` is created: it is within the domain and the corresponding SCI is created at step 3. Discovered DBs may also depend on other databases they refer to (dependency `Uses`) which can be managed on other hosts (this last case is not illustrated in our scenario).
5. **Application Servers:** Application server *AS-A1* found during discovery is associated with an SCI of type `application_server`, we already shown in Section 4. This SCI also has an inter-domain dependency on databases hosted on other domains of the cloud.
6. **Applications:** Applications *Advertise* and *Sell* found during discovery are associated with an SCI of type `application`. They are described through their name (property `application_name`). Furthermore they may be composed of several ejb/java/web modules (dependency `ComposedOf`). They depend on the application server on which they are deployed (dependency `ManagedApplication`). Both these dependencies are within the domain and the corresponding SCIs are created in the previous steps.

During discovery it is necessary to identify URLs of SCIs that are referenced in the dependencies. These SCIs can be local to the domain or they can belong to other domains. An SCI URL can be automatically constructed knowing the hostname of its authoritative DSM and the id through which the DSM reference it in its internal table. Hence, when a dependency refers to a local SCI (which has the same authoritative DSM of the depending item) it is only needed to know its id. This id can be retrieved from the local DSM giving in input some properties/dependencies inferred during discovery. The DSM searches in its table the rows that have properties/dependencies matching those given as input and returns the associated ids. When an SCI is not local, it is also necessary to know what DSM maintains it.

For example, in our scenario we need to identify URL of the SCI associated with service *Payment* on which application *Sell* depends. Information retrieved

during discovery about service *Payment* is its endpoint `http://131.34.5.25/-FlexPayService.wsdl`. From this property we know the host on which service *Payment* is deployed (`131.34.5.25`). At this point the discovery needs to know what is the DSM authoritative for the SCIs of host `131.34.5.25`. Discovery process gets this information from DSM Registry, issuing the query below:

```
http://nimbus06.watson.ibm.com:8081/machine?address=131.34.5.25
```

It is worth to note that each host of the domain knows the address of the authoritative DSM Registry, since it is given to the discovery process as a configuration parameter.

The DSM Registry returns the hostname of the required DSM (`nimbus06-.watson.ibm.com`) that keeps the SCI of service *Payment*. Finally, what the discovery needs to do is to request to the DSM the SCI id of service *Payment* through a query of this type (single URI):

```
http://nimbus06.watson.ibm.com:8080/
  sciRegistry?type=web_service&
  properties=<property name=ws-endpoint>
    <prop:ws-endpoint>
      http://131.34.5.25/FlexPayService.wsdl
    </prop:ws-endpoint>
   </property>
```

The DSM Manager returns the id of the SCI associated to service Payment (i.e., 0). This way the discovery process is able to construct the URL of the SCI associated with service Payment as follows:

```
http://nimbus06.watson.ibm.com:8080/sci?id=0
```

Before terminating discovery the set of detected SCIs is given as input to the DSM authoritative for that domain. DSM keeps the set of SCIs already detected in the previous discovery phase. Hence it compares discovered SCIs with the previous ones grouping them into three sets: ADDED (new SCIs that were not discovered previously), DELETED (old SCIs that are not detected in the last discovery phase) and MODIFIED (pairs of SCIs detected in two subsequent discovery phases). Association between SCIs that refer to the same component in two subsequent discovery phases are detected as follows: the DSM checks if the properties/dependencies that allow to uniquely identify an SCI are still the same. For example, to uniquely identify an SCI associated with a db among all SCI of type db, we need property `database-name` and dependency `ManagedDB` (the corresponding dbms). If a previous SCI is detected with properties/ dependencies matching those given as input, both the previous SCI and the new one are inserted in the set MODIFIED. Otherwise the new SCI is put in the set ADDED. Old SCIs that do not have a corresponding new SCI, are put in set DELETED.

For each SCI in set ADDED the DSM adds a new entry in its internal table with a unique id, the discovered attributes that allow to uniquely identify it and the paths to the locations of the configuration information. A new feed document is also created and associated to that SCI, with an entry that advertise

its creation. For all SCIs in set DELETED, DSM adds a new entry in their feed documents to advertise their deletion. DSM also marks as "deleted" the row state in its internal table pointing to that SCI. Configuration files will be deleted after a certain time for space reasons. All couples of SCIs put in the set MODIFIED are compared to find differences in the SCI documents that reveal possible modifications. If a modification is detected a suitable entry is added to the feed document associated with that SCI to advertise the change.

## 5 Cross Domain Aggregation

SCIs availability in each domain via the authoritative DSM allows all interested stakeholders to get higher level views on the configuration of the overall infrastructure according to specific needs. These views transcend the perspective of a particular domain and are created through the combination and the reinterpretation of existing SCIs or feed documents. Cross-domain aggregation is enabled by the adoption of mashups relying on one or more Aggregators, which collects and aggregates the information exposed by each DSM. To aggregate SCIs and feeds coming from the whole distributed platform, Aggregators ask the DSM Registry what are the hostnames of all DSMs available in the infrastructure.

Aggregators provide overall information about items configuration and their changes through respectively a REST or an ATOM aggregation. REST aggregation allows to combine several SCIs according to some criteria. In our current prototype we provide the following aggregations we considered significant for service management processes:

*All SCIs available in the infrastructure.* It provides a global view of all items available in all domains of the distributed platform. For example, it can be useful when a cloud provider receives a request from a user who wants to deploy his/her applications. In this case, the provider needs a global overview of all SCIs of the infrastructure to know which machines of its cloud are more suitable to host those applications.

*All SCIs a business application relies on.* It is useful for business analysts who may want to retrieve SCIs a specific business application relies on.

*All SCIs associated with items of the same type.* It is useful for administrators who need to perform maintenance on items of the same type. For example, an administrator may ask for all SCIs of type `dbms` when he/she has to perform an upgrade to a next version of DB2, to all DBMS of the infrastructure. In fact he/she needs to view the version of all DBMS available in the infrastructure to know which of them has to be upgraded.

*A specific SCI together with those SCIs referenced in its dependencies.* During incident management processes, detecting the cause of a failure in an CI may require to inspect the configuration of other items it depends on.

ATOM cross-domain aggregation allows stakeholders to subscribe on changes that can affect any item of the overall infrastructure, without knowing the URLs of the feeds associated with each SCI. We provide some predefined criteria to aggregate feeds:

- *All feeds available in the infrastructure.* It eases change management processes. For example, interested users may be notified when an item in the infrastructure changes (e.g. service *Payment*) and, if this change is relevant for their business, they can perform maintenance actions on the affected items (e.g., change the parameters adopted to invoke service *Payment*).
- *All feeds associated with items of the same type.* It is useful, for example, when an administrator is interested in knowing all changes affecting all DBMS of the infrastructure, to perform suitable corrective actions.
- *A specific feed together with those feeds associated with SCIs an items depends on.* It shows configuration and changes relative to a specific SCI and its dependencies. If we consider a business service, it may be necessary to know changes in all items it depends on to perform impact analysis or activate change management processes.

Other SCI/feed aggregations may be offered easily since the infrastructure already provides all necessary configuration information. For example we may support aggregation that collects SCI/feeds of a component having particular properties, e.g., all DBMSs of type DB2, or we may want to collect feeds carrying specific kinds of changes to apply a suitable patch. We also provide a graphical interface to view aggregated SCI and feed documents.

Even if each DSM only keeps the current SCI version it is possible to go back to previous versions inspecting the corresponding feed document. This is important for incident management processes where stakeholders want to inspect configurations before a failure happened and analyze the cause of a problem. It is possible to retrieve the last SCI configuration, inspect the changes that happened after a particular time instant (that in which the failure happened), starting from the last one up to the first one and apply these changes in a backward way. For example, if an entry advertises a change in a property/dependency, it is sufficient to substitute the XML value of the property/dependency with that carried on by element `<old>` in the entry content. We may need to get the SCI version associated with service *Payment*, before its signature for operation `PayOrder` is changed. In this case we have to change the input parameter is changed from `tns:RPType` to `tns:RPAllowedType` (see second feed entry in Figure 4).

## 6 Implementation

The viability of the SCI approach was validated by implementing a prototype comprising the following components: the configuration discovery agent, the feed generator, the DSM, the DSM Registry and the Aggregator. The domain discovery process is a script that triggers the execution of Galapagos discovery and translates its results into a set of SCIs, and generates the ATOM feed entries associated to the detected changes. The DSM, the Aggregator and the DSM Registry are implemented through WebSphere sMash [7], a development and runtime environment for RESTful services and mash-ups.

The platform was tested in a laboratory environment using scenarios like that outlined in section 2. The tests showed that the platform permits to maintain

configuration information automatically. Configuration exchange among different domains takes place easily, by simply retrieving or aggregating XML documents using Web browsers and feed readers. Service management processes or interested stakeholders can access configuration information using common tools. Finally, the application of filters to customize the SCIs/feeds aggregation offers to service management processes the information they exactly need.

## 7 Related Work

Distributed system management is the central focus of two standards: Web Services Distributed Management (WSDM) [8] and Web Services for Management (WS-M) [9]. Both propose the idea to expose management information as Web services and represent resource information through extensible models. To trace associations among resources WSDM provides the concept of relationship, which includes our notion of dependency. While, even if WS-M proposes a rich configuration model, i.e., CIM [10], it does not support dependencies. Furthermore, WSDM and WS-M provide limited discovery capabilities. Our solution represent a significant improvement over these standards because it offers a global approach that continuously maintains resources after they are discovered, updates their configuration when changes are detected, and notifies interested users about these changes. WSDM and WS-M support allow users to subscribe on events generated after resources' changes and being notified according to respectively WS-Notification [11] or WS-Eventing [12] standards. These standards do not provide a clear way to represent resources changes and their low diffusion, discourages their adoption. Instead, our approach adopts ATOM/RSS feeds, offering a standard way to represent changes (encoded into a feed entry), and consume them through any feed reader, with the possibility to rely on its subscription and filtering capabilities.

CMDB federations [1] are an approach to use CMDBs across domain boundaries, enabling access to information held in different CMDBs. This approach has high setup costs since all parties must establish explicit relationships, which it is not feasible in loosely coupled environments. Treiber et al. [13] proposed a concrete information model to represent both static and dynamic changes in web services and encapsulate them in atom feed entries. The authors also relate each change to its cause and to the stakeholders who may be interested in. Despite our approach focuses on static configuration properties, it has the main advantage of dealing with cross-domain environments, representing intra- and inter-domain dependencies among CIs. Moreover our solution keeps the information model light, enabling different business analysis through several cross-domain aggregations.

## 8 Conclusions

Loosely coupled applications spreading an SOA over multiple management domains requires a configuration management approach that takes into account

the the absence of central service management and a central CMDB. The SCM approach proposes to decentralize configuration management in a way in which service providers can expose configuration information to their users in a standard format based on domain discovery information while service users are able to discover and trace CIs outside their own management domain boundaries. The use of RESTful interfaces to CIs and ATOM feeds to distribute updates on configuration changes enables the use of very commonly available tools to expose and process configuration information. The feasibility of the approach was demonstrated in a proof-of-concept implementation. As next steps we will further validate the approach and work on improvements related to interaction with existing discovery technology, selective publication of SCIs, and programming models for aggregation. We also plan to remove the architectural bottleneck generated by the DSM Registries organizing them in P2P networks.

## References

1. Clark, D., et al.: The Federated CMDB Vision: A joint White Paper from BMC, CA, Fujitsu, HP, IBM, and Microsoft, Version 1.0. Technical report
2. IBM: Tivoli: Change and Configuration Management Database, `http://www-01.ibm.com/software/tivoli/products/ccmdb/`
3. Ludwig, H., Laredo, J., Bhattacharya, K., Pasquale, L., Wassermann, B.: REST-based management of loosely coupled services. In: Proceedings of the 18th International Conference on World Wide Web (2009)
4. Wassermann, B., Ludwig, H., Laredo, J., Bhattacharya, K., Pasquale, L.: Distributed cross-domain change management. In: Proceedings of the International Conference on Web Services (2009)
5. Network Working Group: The Atom Syndication Format (2005), `http://www.ietf.org/rfc/rfc4287.txt`
6. Magoutis, K., Devarakonda, M., Joukov, N., Vogl, N.: Galapagos: Model-driven discovery of end-to-end application-storage relationship in distributed systems. IBM Journal of Research and Development (2008)
7. IBM: Projectzero, `http://www.projectzero.org/`
8. OASIS: Web services distributed management: Management using web services
9. DMTF: Web Services for Management (WS-Management)
10. DMTF: Common Information Model (CIM) Specification, Version 2.2
11. OASIS: Web Services Base Notification 1.3 (WS-BaseNotification) (2006)
12. Box, D., et al.: Web Services Eventing (WS-Eventing) W3C Member Submission
13. Treiber, M., Truong, H.L., Dustdar, S.: On analyzing evolutionary changes of web services. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 284–297. Springer, Heidelberg (2009)