Flexible Coordination Techniques for Dynamic Cloud Service Collaboration[1]

Gary Creaner and Claus Pahl
School of Computing, Dublin City University, Dublin, Ireland

ABSTRACT

The provision of individual, but also composed services is central in cloud service provisioning. We describe a framework for the coordination of cloud services, based on a tuple-space architecture which uses an ontology to describe the services. Current techniques for service collaboration offer limited scope for flexibility. They are based on statically describing and compositing services. With the open nature of the web and cloud services, the need for a more flexible, dynamic approach to service coordination becomes evident. In order to support open communities of service providers, there should be the option for these providers to offer and withdraw their services to/from the community. For this to be realised, there needs to be a degree of self-organisation. Our techniques for coordination and service matching aim to achieve this through matching goal-oriented service requests with providers that advertise their offerings dynamically. Scalability of the solution is a particular concern that will be evaluated in detail.

INTRODUCTION

Service-oriented architecture (SOA) is an architectural style that allows for business processes to be implemented by integrating various services. These services can be thought of as software components. Cloud computing builds up on service architecture as the platform, providing cross-organisational, externally hosted services.

Most current SOA implementations use web services as the technology platform based on message passing, a service registry and static service description respectively. This approach is rigid in that it requires services publish the details of their functionality and how to interact with them to a registry. This information must then be used by the requestor to bind to and invoke the service in the way in which the provider has published, usually using WS-BPEL. This is a property that does not meet the flexibility requirements of cloud computing, in particular if flexible service brokering and mediation is required where offered and requested cloud services are matched dynamically through a cloud brokering service. Intermediaries such as brokers that bundle and customise offerings in

[1] ***This chapter appears in "Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solution" edited by J. Cubo and G. Ortiz, Copyright 2012, IGI Global, www.igi-global.com. Posted by permission of the publisher.***

response to dynamic needs will become more important in the cloud domain in the near future. Currently, a cloud service user or broker would have to completely define what services are to be used, the order in which they would be used and how the input and output is passed from one service to another in order to implement a full business process.
Current approaches to service collaboration (Pahl, 2002) are web service orchestrations and choreographies like WS-BPEL (orchestration) and WS-CDL (choreography). Both require the services used to be specified prior to the execution of the process. This kind of static process specification does not lend itself to a dynamic, flexible approach in which provided cloud services could be used as part of a process without prior knowledge of the service.

In order to overcome the limitations described, a framework is needed allowing services to be chosen dynamically at run-time. Our framework introduces a coordination space for providers to collaborate their activities in order to fulfil requests. The coordination space consists of a tuple space where requestors can deposit their requests and providers can take on requests according to their capabilities (Doberkat et al., 1992; Li & Parashar, 2005; Pahl et al., 2011).

Ontologies can be used to add semantic descriptions to web services (Pahl, 2005; Pahl, 2007). There are ontologies available which offer ways of describing services in terms of their functionality. These will be discussed and the way in which they could be used in the context of service matching will also be explored. Matching requested and provided web services is possible based on these ontological descriptions (Klusch et al., 2006; Sirin et al., 2003; Sycara et al., 2003; Nixon et al., 2007). However, these have not been integrated in a tuple space as their coordination platform. We integrate goal-based service matching into tuple space coordination in order to add flexibility and allow in-exact matches.

The proposed framework would change the service coordination model from a pull model to a push model, whereby requests are published to the coordination space and providers search for requests that they would be able to fulfil. This means that requestors would be able to focus more on the definition of their request rather than on the services that are provided to them.

The next section will introduce core technologies used and discuss related work. A use-case scenario will then be introduced. There, we will also discuss how a tuple space architecture could be used to implement a coordination space for web services. Afterwards, we will give some detail on how the matching of providers and requestors is performed. We discuss on the scalability tests which were performed on the architecture in order to evaluate such an approach. An evaluation of the work discussing possible limitations and ways of overcoming these limitations is also provided.

BACKGROUND

Tuple spaces are widely used to support coordination activities (Johannson & Fox, 2004; Li & Parashar, 2005; Nixon et al., 2007). We present their principles, an overview of the chosen platform, and some background regarding a semantic extension of tuple matching. We also review literature on service composition and coordination.

Coordination and Tuple Spaces

The tuple space architecture was introduced in (Gelernter, 1985) as a means of communication in distributed programming. Tuples are constructs which consist of a collection of actuals. Templates, used for matching, consist of a collection of formals or actuals, or a mixture of both. *out()* and *in()* are the two key operations proposed for Linda. The *out()* operation takes a tuple as input and adds the tuple to the tuple space. The *in()* operation takes a template as input and searches the tuple space for a tuple which matches the provided template, returns it to the process that called it and removes it from the tuple space blocking access to it by another process. There is also a *read()* operation which will return a tuple matching the template, but will not remove it from the tuple space. This allows more than one process to access the tuple. With both the *in()* and *read()* operations, the calling processes will block until a matching tuple is found. If no matching tuples are in the space when the calls are made, the processes will block until a matching tuple becomes available.

A prototype of our service request coordination architecture is based on the LighTS (Balzarotti et al., 2007) tuple space, which was designed as an open-source, lightweight, customisable tuple space framework. It provides support for the Linda operations described above and can also be easily extended or modified to change how the tuple space itself is implemented or how the matching is performed. We have added ontology-based matching and process-level coordination techniques.

The Semantic Web

The Semantic Web aims to give meaning to the web resources and describe the information provided in a machine interpretable way. The Resource Description Framework (RDF) is an XML based language used to create statements. These statements consist of a subject, predicate and object, or resource, property, and property value. The Web Ontology Language (OWL) is based on RDF. It provides constructs that can be used to aid reasoning about the ontologies. An application of OWL is OWL-S (Web Ontology Working Group, 2006), which is an OWL ontology to describe services in terms of functional and non-functional attributes. The non-functional description can include information such as the service name, and contact information of the provider. The functional description is used to describe the web service in terms of its inputs, outputs, preconditions and effects. OWL-S based matching of tuples containing service descriptions is our aim.

A number of techniques are used in our implementation. SWRL is a rule language for ontologies. The body and head of a rule consist of so-called atoms. Atoms can state that something is a member of a class, has a value for a certain property, is the same as or is different from something else. These atoms can be combined into atomlists. The atomlists can be used in the preconditions and effects part of the OWL-S service descriptions without the need for implication rules to be defined. SPARQL is a query language which can be used to implement queries on RDF graphs. It can be easily used to follow links between the various resources and provides SELECT, ASK, CONSTRUCT, and DESCRIBE queries. It also allows for WHERE clauses to be added to the queries. Jena is an open-source Java

framework for programming with OWL and SWRL, which provides a programmatic interface to access the various technologies listed above. We combined LighTS and Jena in our implementation.

Service Composition and Coordination

The work on semantic service descriptions has continued to support the discovery and composition of services using ontologies (Klusch et al., 2006; Sirin et al., 2002; Sycara et al., 2003), allowing matching between requests and provided services taking a variety of concerns into account. Recently, dynamic composition, particularly as a result of failure occurring at runtime, has been addressed (Cavallaro & Di Nitto, 2008; Moser et al., 2008; Wang et al., 2009). Failure handling is not our primary concern, but enabling a marketplace where negotiations can be automated through dynamic ontology-based coordination. Küngas and Popova (2011) describe a step in this direction, where objects are the central artefacts that are processed by processes dynamically. We add an ontology-based technique here.

A TUPLE SPACE ARCHITECTURE FOR SERVICE COORDINATION

Use Case

As a use case to illustrate our work, we have chosen a case where a company would like to store data on their products in the cloud, but would also like this data validated before it is stored. In order to formulate this into a request, the company would need to specify in their request that they would need both actions to be carried out.

There are many cloud based storage solutions available currently - Amazon's simple storage server is an example. This is quickly becoming an attractive solution to companies for storage and sharing of their data. Some providers offer cloud-based data validation for companies wishing their data to be compliant with the relevant standards. If a company is looking to share data on their products with other companies then it must be ensured that the data is compliant with the industry standards, e.g. to ensure their data conforms to standards like GS1.

In this situation, a broker might take on the initial request and then involve other cloud providers to validate and then store the data.

Tuple Spaces for Service Coordination

In using a tuple space for service coordination, we suggested in (Pahl et al., 2011) that the tuples could consist of three fields:

- an object field, which would contain the input of the request, such as the *data* in our use case.

- a goal field which would contain a description of what should be achieved by processing this request, such as *validated(data) -> stored(data),* which means that only successfully validated data should be stored.
- also an optional process field is included which would guide the choosing of services to complete the request, e.g. a conditional statement *if validate(data) then store(data)* that sequences validation and storage.

The tuple space is the central coordination component (Li & Parashar, 2005). Requestors deposit goal-based requests here (Andersson et al., 2005). Providers see if they can meet these requests. In doing so, they themselves can deposit further new requests into the tuple space. This allows the original goal to be broken into sub-goals and these to be deposited into the tuple space to be met by other services, forming a goal resolution process. In terms of the use case, this would involve a request tuple being created with the object field containing the data to be processed or perhaps a URI link to the data. The goal field would consist of an expression that data needs to be both validated and stored.

Service Coordination Architecture and Process

A service that provides any of the requested operations could chose to take on the processing of the request. It might then see if it can find another service that is able to fulfil parts of the task that it cannot fulfil. The service would then remove the tuple from the tuple space, blocking any other service from accessing it. This is a commitment to process the request and it should result in a completed tuple being entered in the tuple space later. Note, that in a negotiation process between cloud service users and providers, the handshake needs to be complete, i.e. the requestor would need to approve a provider. This would only require a simple coordination protocol (without significantly affecting the described tuple space functions) to be added and shall be neglected here.

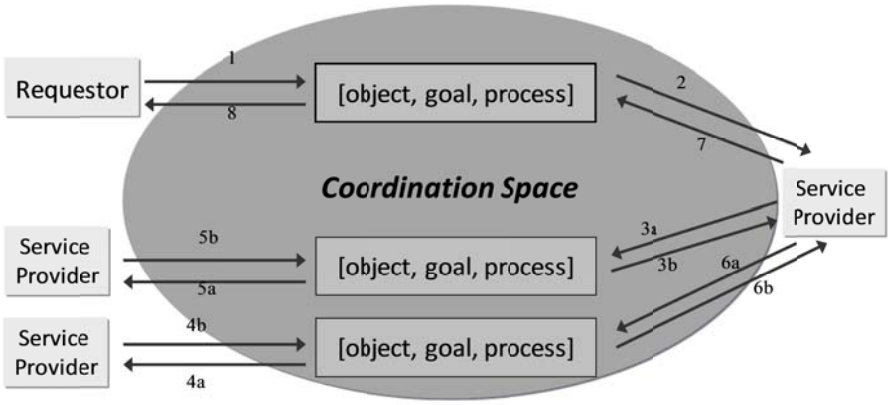

Figure 1. Coordination and Knowledge Space Architecture. (Adapted from C. Pahl, V. Gacitua-Decar, K. Yapa Bandara, M. Wang. Ontology-based Composition and Matching for Dynamic Service Coordination. Workshop on Ontology, Models, Conceptualization and Epistemology in Social, Artificial and Natural Systems Workshop ONTOSE'2011. 2011).

Overall, the services involved in completing a request do not need to be specified prior to the request being formed. This method is a much more flexible approach than current approaches. As the services are picked dynamically at run time, this allows providers to join and leave the cloud

market community as they wish. With other approaches like WS-BPEL, a service may be part of a BPEL process and the whole process could then fail if the service is no longer available. With the proposed method, a different service that provides similar functionality could take on that part of the process and the request could still be completed. The diagram in Figure 1 explains this in more detail. The numbered arrows in the diagram show the sequence of events.

- The original request is entered into the tuple space (1), relating to the use case. This would be the request for the data to be validated and stored.
- The tuple is then taken by a broker service (2) which can coordinate the full request. This would require a validation service (3a) and, if necessary, storage (3b) - both deposited as two more detailed requests with storage and validation of the data as the respective goals.
- The validation service would retrieve the data validation tuple from the tuple space (4a). The storage service (5a) would then store this tuple from the tuple space and complete the request.
- Both would then put completed tuples back into the tuple space (4b), (5b).
- These would then be returned to the broker for the original requestor (6) to retrieve its completed request from the tuple space (7), (8).

Implementation based on LighTS

LighTS supports the Linda operations. The data structure used for the tuple space is implemented using Java vectors. The tuples in LighTS are also implemented as vectors and consist of fields. Fields can be of any type. Any amount of fields can be added to a tuple. A valued field has a specific value and a typed field just has a type. For a tuple to be added to the tuple space, it must consist of only valued fields. Templates can consist of both valued and typed fields.

The *out()* method in this implementation takes a tuple as input, adds this tuple to the vector by calling its *addElement()* method. It then notifies any processes that are blocked to inform them that a new tuple has become available.

The *in()* method takes a template as its input. It compares the template to each element in the tuple space until it either finds a match or has tried every tuple and not found a match. If no match is found before the end of the tuple space operation, the process will be suspended in the while loop by calling a *wait()* inside a section of code that is synchronised on the tuple space. If a match is found, it will be assigned to the result tuple and this will cause the process to exit.

The matching in LighTS between a tuple and a template is performed by first checking if the tuple and template contain the same number of fields. If this is true, then the fields themselves are matched. For fields to match, the valued fields in the template must match the values in the tuple and the types of the typed fields in the templates must be of the same as the types of their corresponding tuple fields. The *in()* method removes the matching tuple from the tuple space.

The LighTS platform implements a strict form of matching that does not take flexibility regarding parameter typing and effect specifications for services into account. We add ontology-based tuple matching - introduced below.

Ontology-based Matching of Providers to Requestors

The matching of providers and requestors in the cloud service setting requires matching the declared goals of the providers to requestors. The simple matching semantics offered by the LighTS platform is not sufficient. The LighTS platform offers exact matching on the values of fields or matching on the types of the fields. For the architecture described in the previous section, there would need to be a more flexible partial matching provided. This partial matching is needed to allow a provider to determine if they can take on the request even if they cannot fully complete it and decompose the goal to involve other providers or if they can over-satisfy the requirements (Pahl & Zhu, 2006). More functionality must be added to the framework to enable providers to work out which part of the request they cannot fulfil. This will enable them to formulate a new request with a sub-goal which can be completed by another provider for the original request to be fully completed.

We used the OWL-S ontology to describe the services offered and their goals. OWL-S was also used to describe the request. OWL-S is based on the RDF format. The RDF format is a means of representing the resources in ontological format and the links between all the resources. It can be thought of as a graph or a collection of triples. The triples are composed of a subject, a predicate and an object. The subjects and objects are nodes in the graph and the predicate is the link between the nodes. For the given case study example,

validate(data) = GS1-compliant

or *[data,validate,GS1-compliant]* in triple notation, is a goal expressing the aim to have the subject *data* being *validated* (predicate) as *GS1-compliant* (object – the goal here).

OWL-S service descriptions can consist of non-functional properties, such as the name and contact information, but also describe the service in functional terms. The functional description is based on the inputs, outputs, preconditions and effects (IOPEs). It is this functional description that we use here to match the requestors and providers. SWRL expressions are used to define the effects of the services and the requested goal:

validated(data) -> stored(data)

In the implemented platform, the SWRL expressions in the effect descriptions are matched with each other. They are based on atomlists, which can be divided into atoms. Atoms can be thought of as representing ontology properties. The atomlists in the effect part of the OWL-S files can be thought of as describing the new state that the object will transition to when the service has completed.

The Jena platform offers a programmatic interface for interacting with RDF models using the Java interface, used for accessing and manipulating the ontology information (including SPARQL queries). This was used to extract the atomlists from the service and query descriptions. When these are extracted by the tuple space to see if they match, we attempt to match part of the query with the provider. If part of it matches, then the part that does not match is added - like *validate(data)* or *store(data)* - to a new tuple and this is added to the tuple space for another provider to take on this

sub-goal. If this sub-goal can be completed by another service, then the original goal can be completed by the original provider service. Jena OWL-processing is used to determine which part of the goals match and also to build the goals from parts that do not match.

The completed tuples are removed from the tuple space by using a field to indicate whether it is completed or not and once the request is placed in the tuple space, the process that deposited it should try to match it again with the completed field set to true.

Scalability of the Tuple Space Architecture

One of the main concerns with such a coordination architecture is its scalability. How the platform will perform with varying numbers of providers and requestors, with differently sized tuples, and matching different data types needs to be looked at to demonstrate the applicability beyond toy examples.

Ideally, the tests would have to be carried out by providers making an *in()* call and waiting for a request to enter the tuple space. However, as *in()* calls are blocking, it would require the creation of a separate thread for each provider. As the aim of the scalability tests is to assess the time needed for the matching of requests to providers, the processing overhead that managing these threads would incur, would greatly skew the results obtained for large provider numbers. To overcome this, requests were first placed in the tuple space and then *in()* calls were made. Although this is opposite to the order which would be expected to occur, requests would not be removed from the tuple space until matched with a provider. It does not affect the results as the providers wait for a tuple to be introduced to the tuple space when no match is found and are woken when a new tuple becomes available. Access to the tuple space is synchronous, so the providers would be allowed access to the tuple space in an order similar to the way in which the requests are searched in order using this method.

The steps involved in carrying out the tests were:

1. Create an array of initialised tuples and rearrange their order within the array so that they are not searched for in the same order they were entered.
2. Insert the tuples into the tuple space using the *out()* operation.
3. Create an array of templates to match tuples and again rearrange the order.
4. Remove the matching tuples from the tuple space using the *in()* operation.

The time taken to perform the *in()* operations was measured to evaluate the scalability of using a tuple space architecture for service coordination.

The tuples consisted of 3 fields. The first was used for value-based matching and tests were run with this as an integer, a short, a long, a float, a double and a string. The second field was a byte array. This was only matched on type, not value and its size was varied between 100 bytes, 1000, bytes, and 10000 bytes. The third was a string that was value matched and was the same for each tuple. Tests were run with 100, 1000 and 10000 requesters and providers, then doubling and tripling their respective numbers. Each test was run 10 times. Note, that we have displayed results as linear approximations for simplicity in Figure 2, although the actual functions might be non-linear.

As expected, the more tuples that were being matched, the longer the matching process took. The results in the graphs are taken from using integers as the matching type and the same number of providers as requestors. A typical example of such a graph from the data collected can be seen in Figure 2 (top left). The legend on the side indicates the size of tuples used for matching in the graphs.

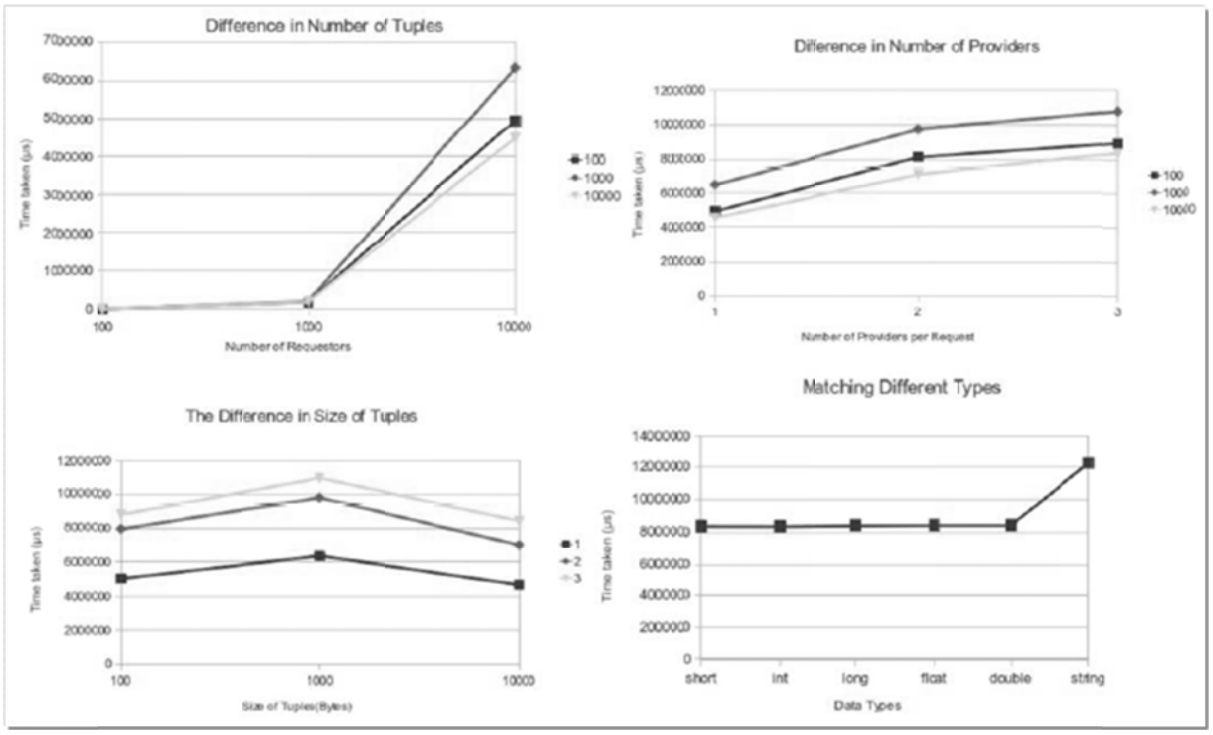

Figure 2. Performance Evaluation Results (different number of requests; different number of providers per request; differently sized tuples; matching different data types).

The difference between the number of providers per request can be seen in Figure 2 (top right). The more providers are available per request, the longer the matching process takes. While there are more matching tuples per request, there are even more non-matching tuples per request and, therefore, more incorrect matches to check before finding a correct match. For the tests shown in the graph, the matching was done using short as the type, 10000 requests were matched. The legend on the side indicates the size of the byte array in the tuple.

Figure 2 (bottom left) shows the results of varying the size of the byte array in the tuple. The results are for a double valued field for matching and 10000 match requests. The numbers in the legend indicate the number of providers per request. The trend observed in this graph is unexpected, but can be explained through platform functions like garbage collection affecting the smaller inputs unduly.

The graph in Figure 2 (bottom right) illustrates the difference in using different types for matching. They all produce similar results except for strings, because the string used is much longer than the other types, but resembles more the ontology-based triple elements that we aim to simulate here. LighTS uses the default *matches()* methods of the data type to determine whether they match. For

this graph, the time was taken from matching 10000 tuples with 3 providers per request and a byte array of 10000 bytes.

Overall, this demonstrates that the solution is scalable. In some situations, the performance is not linear anymore, but even the practically very high numbers still demonstrate acceptable performance.

Discussion of Ontology-based Matching

To evaluate the platform for the concrete use case, we created two OWL-S description files for two services - one which would offer the service of validation of data and one which would offer storage of the data. To do this, we created a data ontology using OWL which included properties *locatedAt* and *isValidated*. The *locatedAt* property was a link between the resources *Data* and *DataLocation* and the *isValidated* property was a link between the *Data* resource and a Boolean value to represent whether the data has been validated or not. Ontologies allow us to reason about described properties. This means that conclusions can be drawn about a resource's properties from other properties of the service resource. We created the properties that were needed for the services and requests and used the same properties to represent the same information for each.

For our implementation, the focus for the matching was put on the functional properties of the services. As explained previously, the effect of the services was expressed using SWRL expressions which consist of atomlists. The atomlists offer *first* and *rest* properties to describe what happens. For our implementation, we limited the services to only contain a *first* property. This means that each service must only offer one function. The request can have a *first* and a *rest* property. For the use case, the validation service's effect states that it changes the *isValidated* property changes from false to true. The storage service's effect states that it changes the *locatedAt* property changes from the input data's location to the location that will be returned. The request for validation and storage consists of an effect that specifies its *first* property should change *isValidated* from false to true and the *rest* indicates that *isLocated* should change from the input location to the output location.

When searching the tuple space for a provider to match the specified request, it first checks if they fully match. This means that the *first* property of the provider matches that of the request and that the request's *rest* property is empty. If this is not the case, then it checks if it is a partial match. This is true when the *first* properties match, but there is also a *rest* property in the request. If this is the case, a new goal model will be returned with the *first* property set to what the *rest* property of the original request was. This is then entered into the tuple space as a request to be met by a different provider. The provider then acts as a requestor and waits for the request to be returned to the tuple space with an indication of completion. If this is the case, the original provider can then remove the request tuple from the tuple space, blocking further access to it. It will then proceed to complete the request in the same manner in which it decided if it could meet the request.

FUTURE RESEARCH DIRECTIONS

These experimental executions realise a process that demonstrates the enhanced matching capabilities. However, we now discuss some implementation aspects, which demonstrate key

difficulties in implementing a full-scale solution. The prototype currently does not allow for the case where a provider may provide a more complicated service that cannot be described by only a first property in its atomlist. However, the platform could be easily extended through more manipulation of the goal model before deciding how much of a match it is. Another current limitation of the model is that only the effect is considered when matching providers to requestors. It is assumed that for the input to all services needed, the process will be the same for the use case. However, some processes would need the output of one service to be the input of the next - validation is a requirement for storage. The OWL-S service descriptions contain resources that describe the inputs and outputs of their service and their types. This would have to be incorporated into a final model and some more work done in creating the subgoal request tuples and some more manipulation of the service descriptions in order to determine if the requests can be met. It would have to be ensured that one service can produce as its output the input needed by the next service to meet the sequencing constraint.

In terms of a cloud service brokerage or mediation solution, we have demonstrated the scalability of a flexible coordination model and the feasibility of ontology-based request matching. There are still areas for further investigation. Greater control over what providers are used to achieve the goal could be incorporated in the form of a simple negotiation protocol - the process element of the request triples can serve this purpose here. As a basic solution, a ranking system for order of preference can be utilised. In the framework discussed in this chapter, the first matching service is the one to obtain the request. However, a negotiation process could be added whereby the matching services compete to see which one should be given the request. This could include the requestor specified control described above, but other properties could also be taken into consideration. If requestors are being charged for using services, the price of those services could be taken into account. Security characteristics of the web services are other non-functional properties that could influence this.

CONCLUSION

Current methods of service collaboration show limitations. We discussed limitations regarding the flexibility for the dynamic collaboration of the services involved in the process and introduced a framework for service collaboration that is suitable for cloud computing settings. A Linda tuple space model was implemented as a coordination space that service providers could use to advertise their capabilities. We have demonstrated the scalability of the solution through extensive tests. We also looked at the feasibility of an ontology-enhanced matching technique. We have focussed here on stateless coordination, as our aim was to explore advanced matching techniques. The original Linda coordination model was extended to include an ontology-based goal matching aspect, as the matching semantics used by the original model would not be sufficient. It has been shown that this is an effective way to offer flexible, dynamic collaboration of services where the providers have the freedom to advertise and withdraw their services when needed. Its ability to deal with a large number of providers and requesters working concurrently with a large number of differently typed and sized requests demonstrates the suitability of the approach for large multi-user service coordination environments such as cloud computing.

KEY TERMS AND DEFINITIONS

- Cloud service: a cloud service is a software service (typically a Web service) offered as part of a cloud platform (at SaaS, PaaS, or IaaS layer).
- Service coordination: refers to protocols, languages and tools that support the coordinated interaction between service users and service providers.
- Tuple space: the tuple space architecture is a means of communication in distributed programming that allows a flexible coordination of participants.
- Coordination space: an infrastructure that acts as a mediation tool between service requestors and service provider through the provision of standard operations to deposit and retrieve coordination data.
- Semantic service description: usually ontology-based annotation of services to cover functional and non-functional properties for automated processing.
- Dynamic service matching: refers to the association of a provided service to a service request at runtime that satisfies the requirements of the requestor.
- Cloud broker: a mediator that matches service provider and requestor according to their needs and supports the negotiation of an agreement.

REFERENCES

Andersson, B., Bider, I., Johannesson, P. and Perjons, E. (2005). Towards a formal definition of goal-oriented business process patterns. *BPM Journal, 11*, 650-662.

Balzarotti, D., Costa, P. and Picco, G. P. (2007). The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems, 5(2)*, 215–231.

Barrett, R., Patcas, L., Murphy, J. and Pahl, C. (2006). Model driven distribution pattern design for dynamic web service compositions. In: *International Conference on Web Engineering ICWE'06*, 10-11 Jul 2006, Palo Alto, US.

Cavallaro, L. and Di Nitto, E. (2008). An approach to adapt service requests to actual service interfaces. In *Proceedings of SEAMS Conference*.

Doberkat, E.-E., Franke, W., Gutenbeil, U., Hasselbring, W., Lammers, U. and Pahl, C. (1992). PROSET A Language for Prototyping with Sets. In *Proceedings Third International Workshop on Rapid System Prototyping* (pp. 235-248).

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems, 7(1)*, 80–112.

Johanson, B. and Fox, A. (2004). Extending Tuplespaces for Coordination in Interactive Workspaces. *Journal of Systems and Software, 69(3)*, 243-266.

Klusch, M., Fries, B. and Sycara, K. (2006). Automated semantic web service discovery with owls-mx. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems AAMAS '06*, (pp. 915–922). ACM.

Küngas, P. and Popova, V. (2011). Artifact-Centric Service Interoperation. In *Estonian Information Society Yearbook 2010. Chapter 3.2 of Estonian Department of State Information Systems*.

Li, Z. and Parashar, M. (2005). Comet: A Scalable Coordination Space for Decentralized Distributed Environments. In *Proceedings of the Second international Workshop on Hot Topics in Peer-To-Peer Systems HOT-P2P*, (pp. 104-112). IEEE.

Moser, O., Rosenberg, F., and Dustdar, S. (2008). Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proceedings of the 17th International World Wide Web Conference (Web Engineering Track) WWW'08*.

Nixon, L., Antonechko, O. and Tolksdorf, R. (2007). Towards Semantic tuplespace computing: the Semantic web spaces system. In *Proceedings of the 2007 ACM Symposium on Applied Computing SAC'07,* (pp. 360-365). ACM.

Pahl, C. (2002). A Formal Composition and Interaction Model for a Web Component Platform. In *Proceedings ICALP'2002 Workshop on Formal Methods and Component Interaction*. Electronic Notes on Computer Science ENTCS, 66(4).

Pahl, C. (2005). Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. In Proceedings European Conference on Model-Driven Architecture – Foundations and Applications ECMDA'2005 (pp. 88-102). Springer-Verlag, LNCS 3748.

Pahl, C. (2007). Semantic Model-Driven Architecting of Service-based Software Systems. *Information and Software Technology, 49(8)*, 838-850.

Pahl, C., Giesecke, S. and Hasselbring, W. (2007). An ontology-based approach for modelling architectural styles. In: *The European Conference on Software Architecture ECSA'2007*. 24-26 Sept 2007, Madrid, Spain.

Pahl, C., Gacitua-Decar, V., Wang, M.X. and Bandara, K.Y. (2011). A coordination space architecture for service collaboration and cooperation. In *Proceedings CAiSE Workshops* (pp. 366–377).

Pahl, C. and Zhu, Y. (2006). A Semantical Framework for the Orchestration and Choreography of Web Services. In *Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005)*. Electronic Notes in Theoretical Computer Science 151(2), 3-18.

Sirin, E, Hendler, J. and Parsia, B. (2003). Semi-automatic composition of web services using semantic descriptions. In *Proceedings Web Services: Modeling, Architecture and Infrastructure Workshop at ICEIS 2003* (pp. 17–24).

Sycara, K., Paolucci, M., Ankolekar, A. and Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics, 1*, 27–46

Wang, M., Yapa Bandara, K. and Pahl, C (2009). Integrated Constraint Violation Handling for Dynamic Service Composition. In *Proceedings IEEE International Conference on Services Computing SCC 2009*. (pp. 168-175).

Web Ontology Working Group. (2004). OWL-S - Semantic Markup for Web Services. W3C. Retrieved 26 January 2012 from http://www.w3.org/Submission/OWL-S/.

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004). Web Services - Concepts, Architectures and Applications. Springer-Verlag, Berlin.

Ankolekar, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D., McDermott, D., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T.R., Sycara, K. (2002). DAML-S: Web service description for the semantic web. In: Horrocks, I., Hendler, J. (eds.). Proceedings ISWC 2002 (pp. 279–348). Springer-Verlag, LNCS vol. 2342, Heidelberg.

Arroyo, A. and Sicilia, M.-A. (2008). SOPHIE: Use case and evaluation. Information and Software Technology. 50(12), 1266-1280.

Baader, F., McGuiness, D., Nardi, D., and Schneider, P.P. (2003) The Description Logic Handbook. Cambridge University Press, Cambridge.

Brogi, A. and Popescu, R. (2006) Automated Generation of BPEL Adapters. In *Proceedings ICSOC'06* (pp. 27-39). Springer-Verlag, LNCS 4294.

Buyya, R., Broberg, J. and Goscinski, A. (Eds.) (2011). Cloud Computing - Principles and Paradigms. Wiley, Hoboken, NJ, US.

DAML-S Coalition (2002) DAML-S: Web Services Description for the Semantic Web. In *Proceedings First International Semantic Web Conference ISWC 2002* (pp. 279-291). Springer-Verlag, LNCS 2342.

Dingwall-Smith, A., Finkelstein, A. (2007). Checking Complex Compositions of Web Services Against Policy Constraints. In Proceedings 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems.

Doberkat, E.-E., Hasselbring, W. and Pahl, C. (1996). Investigating Strategies for Cooperative Planning of Independent Agents through Prototype Evaluation. In *Proceedings First International Conference on Coordination Models and Languages* (pp. 416-419). Springer-Verlag, LNCS 1061.

Doberkat, E.-E., Franke, W., Gutenbeil, U., Hasselbring, W., Lammers, U. and Pahl, C. (1992). PROSET - Prototyping with Sets, Language Definition. Software-Engineering Memo 15, Universität GH Essen.

Erl, T. (2005). Service-oriented Architecture – Concepts, Technology and Design. Prentice Hall.

Gacitua-Decar, V. and Pahl, C. (2009). Automatic Business Process Pattern Matching for Enterprise Services Design. In *Proceedings 4th International Workshop on Service- and Process-Oriented Software Engineering (SOPOSE-09)*. IEEE Press.

Hayes, B. (2008). Cloud computing. Communications of the ACM, 51(7), 9-11.

Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C. and Fensel, D. (2005) Web Service Modeling Ontology. Applied Ontology, 1(1), 77-106.

NIST. Process Specification Language (PSL) Ontology - Current Theories and Extensions. National Institute of Standards and Technology, USA. Retrieved 26 January 2012 from http://www.mel.nist.gov/psl/ontology.html.

Pahl, C. (2001). A Pi-Calculus based Framework for the Composition and Replacement of Components. In *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'2001 - Workshop on Specification and Verification of Component-Based Systems*. ACM Press.

Pahl, C. (2005). A Conceptual Architecture for Semantic Web Services Development and Deployment. *International Journal of Web and Grid Services, 1(3/4)*, 287-304.

Pahl, C. (2007). An Ontology for Software Component Description and Matching. International Journal on Software Tools for Technology Transfer 9(2): 169-178.

Pahl, C. (2010). Dynamic Adaptive Service Architecture - Towards Coordinated Service Composition. In European Conference on Software Architecture ECSA'2010. Springer-Verlag, LNCS, pp. 472-475.

Pahl, C., Giesecke, S. and Hasselbring, W. (2009). Ontology-based Modelling of Architectural Styles. *Information and Software Technology. 1(12)*, 1739-1749.

Rao, J. and Su, X. (2004). A Survey of Automated Web Service Composition Methods. *In Proceedings Intl. Workshop on Semantic Web Services and Web Process Composition 2004* (pp. 43-54). Springer-Verlag, LNCS 3387.

Schaffert, S. (2004). Xcerpt: A Rule-Based Query and Transformation Language for the Web. PhD Thesis, University of Munich.

Semantic Web Services Language (SWSL) Committee (2006). Semantic Web Services Framework (SWSF). Retrieved 26 January 2012 from http://www.daml.org/services/swsf/1.0/.

Tsai, W.T., Xiao, B., Chen, Y. and Paul, R.A. (2006). Consumer-Centric Service-Oriented Architecture: A New Approach. In *Proceedings IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and Workshop on Collaborative Computing, Integration, and Assurance* (pp. 175-180).

Utschig-Utschig, C. (2008). Architecting Event-Driven SOA: A Primer. Oracle. Retrieved 26 January 2012 from http://www.oracle.com/technology/pub/articles/oraclesoa eventarch.html.

Wang, M., Yapa Bandara, K. and Pahl, C (2009). Integrated Constraint Violation Handling for Dynamic Service Composition. In Proceedings IEEE International Conference on Services Computing SCC 2009 (pp. 168-175)