MoSCA: Seamless Execution of Mobile Composite Services

Lucia Del Prete
Dept. of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK
L.DelPrete@cs.ucl.ac.uk

Licia Capra
Dept. of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK
L.Capra@cs.ucl.ac.uk

ABSTRACT

We envisage tomorrow's services to become increasingly pervasive, being deployed within buildings, transport systems, markets, as well as people portable devices. Such services will be, by their own nature, simple and fine grained; as a consequence, service composition will become crucial to deliver rich functionalities that satisfy end users' requests. The higher the dynamic nature of the environment, the higher the chances that services will move out-of-reach before the composition completes, causing the service as a whole to fail. We argue that, in order to enable the successful provision of compound services in mobile environments, the reliability of the composition must be measured and reasoned about. In this paper, we present MoSCA, a middleware that facilitates the rapid development and deployment of reliable composite services. At design-time, a MoSCA Service is uniquely identified within an OWL-S ontology, and described as a composition of further MoSCA Services, which can themselves be composite or basic. At run-time, whenever a (composite) service is invoked, MoSCA selects the providers, among those currently available, that are capable of collectively delivering the (composite) service with the highest reliability. Reliability is estimated by reasoning about providers' historical colocation patterns, that are learned over time. Unforeseen changes to such patterns are being monitored as well, potentially triggering re-bindings during service execution.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Design, Reliability

Keywords

Service Composition, Mobility, Middleware, Dynamic Adaptation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ARM 2008, December 1, 2008, Leuven, Belgium.
Copyright 2008 ACM 978-1-60558-367-9/08/12 ...$5.00.

1. INTRODUCTION

In the last few years, two major trends have been observed: first, the enormous evolution (and market penetration) of mobile technology. Mobile phones have seen their computing capabilities increase according to Moore's law; they have been enriched with additional functionalities (e.g., digital cameras, MP3 players, GPS receivers), and integrated with a variety of wireless network technologies of increasing bandwidth (e.g., Bluetooth 2, Zigbee, WiFi and WiMax), thus enabling the on-the-fly creation of networks of devices in proximity. Second, the Internet has seen a proliferation of blogs and personal content spaces, revealing a transformation of users from traditional consumers to active producers of content. It will not be long before these two trends will converge, thus creating an integrated environment where, besides traditional services delivered by powerful server machines accessible via wide area networks, new services and content will be offered by users to users via their portable devices. A user may, for example, access an information service made available locally within a building, she may use the navigation system of another user in reach, in exchange for her higher bandwidth Internet connectivity, and so on. These fine grained services, attached to people and the environment, will need to be composed to deliver more sophisticated functionalities to the end user.

In order to give users a positive mobile experience, such composite services will have to be perceived as supplied by a unique entity, that is reachable and available for the duration of the service. This opens up significant challenges for the application developer: as services are indeed mobile to each other, the higher the dynamic nature of the environment, the higher the chances that services will move out-of-reach before the composition completes, causing the service as a whole to fail. In order to promote the rapid development and seamless deployment of composite services and applications, we propose MoSCA, a middleware model and framework that provides application engineers, as well as end users, the abstraction of a MoSCA Service as a single, locally available service. Upon service request, MoSCA transparently binds the service to the set of available providers that are capable of collectively delivering the (composite) service with the highest reliability. By reliability, we refer to the probability that a composition will successfully complete before any of its components moves out of reach. To do so, MoSCA reasons about the composition semantics and the dynamically-learned co-location patterns with other providers. Unforeseen changes to such patterns are being monitored, potentially triggering re-bindings.

In the reminder of the paper, we describe a scenario that exemplifies a variety of mobile service compositions (Section 2). We then present MoSCA service composition model and middleware (Section 3). We illustrate the programmatic complexity of the framework by means of case studies (Section 4), before positioning ourselves with respect to ongoing research in this domain (Section 5), and presenting our concluding remarks (Section 6).

2. SCENARIO

Let us consider a user Alice, who owns a next generation mobile phone on which she has installed the Smart Media Player application. This application streams music and video for free from other devices, mocking the functionalities of radio and TV channels. Advertisements are injected from time to time, either interrupting the music/video streaming, or by means of interactive banners. The content to be played, as well as the adverts to be shown/reproduced, are selected based on what is currently available in the environment, taking into consideration Alice's profile. For example, adverts can be gathered from Internet services, when connectivity is available, as well as local advert broadcasters.

Alice leaves her office and walks to the nearest tube station. She is listening to some music played by her Smart Media Player. As she gets on the tube, she looses global connectivity, so the Smart Media Player application has to elect new service providers for music content. In order to do so, it first has to fetch Alice's music profile, and to examine what content and content sources are now available in the new environment. On the basis of those, the Smart Media Player elects the items to be played next, and streams them.

This simple scenario describes a variety of mobile services and introduces a variety of compositions semantics. For example, the media content selector and the advertising service both require to collect Alice's profile and context first; as such they need to be composed *sequentially* (in sequence) to an eventual context-aware user profiling service. Depending on the actual context and user preferences, advertising may be shown or played, either *in parallel* to the content selection and reproduction service, or *subsequent* to it. In order to enable the selection of one or the other strategy, a *choice* composition semantics will be needed. The Smart Media Player updates the list of the next-to-come songs or videos on a regular basis, so that the overall composition *loop* is started again. The full composition semantics of the Smart Media Player can thus be described as follow:

$$
\begin{array}{l}
\textbf{loop} < \textit{iterator}, \textit{guard condition} > \ (\\
\quad profilingService\ \textbf{seq}\ (\\
\quad\quad \textbf{choice} < \textit{guard condition} > \ (\\
\quad\quad\quad (contentSelector\ \textbf{seq}\ advertisingService), \\
\quad\quad\quad (contentSelector\ \textbf{parallel}\ advertisingService)\ ) \\
\quad) \\
)
\end{array}
$$

As the above scenario shows, pervasive services (e.g., Smart Media Player) are often compound services, provided by aggregating more basic functionalities according to a variety of semantics (e.g., sequential, parallel, choice, loop, etc.). Some of these services will be local to the client's device (i.e., the device who is consuming the compound service), while others will be available from a combination of stationary providers (e.g., those embedded in the local space) and mobile providers (e.g., those provided by other people personal devices). Given the dynamicity of the target scenario, with services appearing and disappearing all the time as perceived by the client's device, it becomes crucial to: (1) upon service request, bind to those providers that will maximise the chances of a successfully completed compound service; (2) upon environment changes, dynamically re-bind to providers currently available. In the next section, we present MoSCA, a service composition model and middleware that support this type of run-time reasoning, selection and adaptation, while hiding the exact topology of services making up a composition from both application engineers and end users.

3. MOSCA FRAMEWORK

MoSCA (Mobile Service Composition API) is a flexible and expansible framework that eases the development of applications requiring the dynamic and reliable composition of services in mobile environments. To illustrate how it achieves this goal, we begin by describing MoSCA Conceptual Model and its core concepts: Services, Bindings and Tokens (Section 3.1). We then illustrate the core components of the MoSCA Framework (Section 3.2), before discussing their current implementation (Section 3.3).

3.1 MoSCA Conceptual Model

MoSCA relies on the following three key concepts: a *Service Entity*, that statically describes the composition structure of a service, as well as its classification according to a pre-defined ontology; a *Binding Entity*, that captures the run-time association between a service entity and the node(s) who is(are) currently in charge of delivering such functionalities in the most reliable way; and a *Token Entity*, a container element that collects session information (e.g., input/output data) of a running service request.

The Service Entity

A MoSCA Service consists of a *classification* and a *composition structure*. Classification information consists of two unique identifiers: the identifier of the service taxonomy (or ontology) in use, and the identifier of the service type within such taxonomy. We are making the assumption that a taxonomy can be identified by a unique universal ID (e.g., an URI) and that each service type is univocally identified within such taxonomy by a local ID. Starting from basic services, composite services can be created and described using any of the MoSCA-supported composition semantics: `sequence`, `parallel`, `any order`, `choice`, `signal` and `loop`. The composition structure information is conveyed by the very same nature of the Service Entity, which is represented by means of the composite pattern depicted in Figure 1.

Figure 2 illustrates an example of a MoSCA Service Entity. As shown, service S has a valid service type (sID)

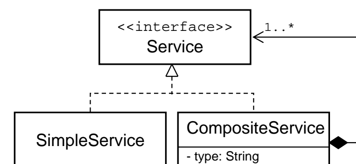

Figure 1: Service Entity

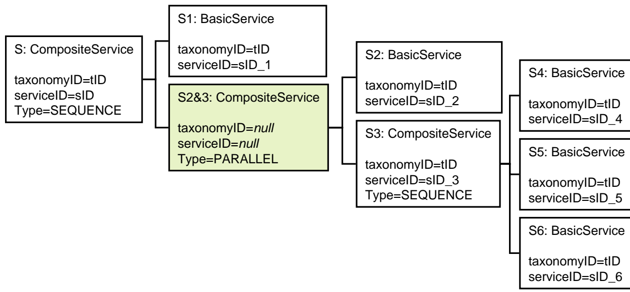

Figure 2: Example of a MoSCA Composite Service

within the taxonomy used (tID), and it can thus be delivered as a single service; however, S can also be decomposed as $S = S1$ **seq** ($S2$ **parallel** $S3$), where $S3$ can be further decomposed as $S3 = S4$ **seq** $S5$ **seq** $S6$. Note that ($S2$ **parallel** $S3$) does not correspond to any service type instead, and thus it can only be delivered as a composition of services.

The Binding Entity

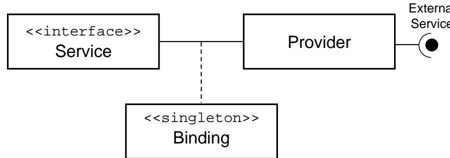

Figure 3: Binding Entity

A Service Entity is a static element. In order for a service to be executed, at least one service provider, delivering the functionalities of such service, must be available at service request time. When a request for service type S is received, a *Binding* to the most reliable provider of S (or set of providers collectively delivering composite service S) currently available in the surrounding is created (Figure 3). Note that Service Entities and Providers do not have direct access to each other; rather, access is provided by means of the Binding Entity, thus facilitating dynamic reactions to changes in the environment (e.g., updating the set of composing service instances delivering a composite service S at execution time impacts the Binding Entity only). Note also that Provider entities are effectively wrappers for services external to the MoSCA framework, and supplied by devices in the surrounding.

Let us revisit the previous example (Figure 2) of a service S that can be decomposed as $S = S1$ **seq** ($S2$ **parallel** $S3$), with $S3$ that can either exist alone or be further decomposed as $S3 = S4$ **seq** $S5$ **seq** $S6$. As exemplified in Figure 4, depending on the providers available in the surrounding, the executed composition may vary: in the example on the left, S is delivered as a single service by provider $P2$, while in the example on the right it is delivered by combining the services provided by $P4$, $P1$, and $P5$.

The Token Entity

A Session Token (or simply Token) Entity represents a running service request. It serves two purposes: first, it is used to capture and share, among the various component services, the status of the request (e.g., input and output parameters); second, it controls the service execution flow by means of a classic token passing protocol. Let us consider, for example, a request to sequentially execute services $S1$, $S2$ and $S3$ (Figure 5 top); upon receiving this request, the MoSCA framework creates a Token Entity and passes it to the first service in the composition, in this case $S1$. As a result of its execution, $S1$ updates the Token Entity with temporary results and session information; the same Token is then passed as input to the next service $S2$ to be executed, and so on until the last service in the composition has been completed and the result of the composition is extracted from the Token and returned to the entity that initiated the service request. At any time, a service can only be executed if it has control of a Token; depending on the actual composition semantic, a Token Entity may thus have to be split and later consolidated, for example, when executing parallel branches (Figure 5 bottom).

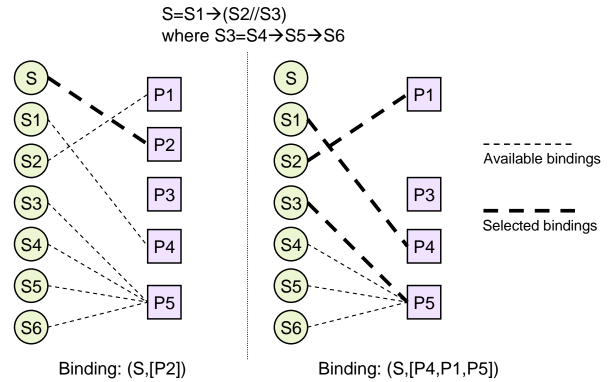

Figure 4: Example of MoSCA Bindings

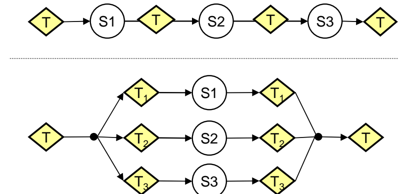

Figure 5: Token Passing Examples

3.2 MoSCA Components

Figure 6 provides an overview of the MoSCA Framework. As shown, MoSCA consists of four main modules: the *Service Manager*, the *Service Analyser*, the *Service Discoverer* and the *Service Coordinator*, that exchange objects of two types, *Service* and *Token*. As previously discussed, the former is used as a service descriptor, while the latter carries information related to an ongoing service request. Two further modules complete the framework: the *Mobility Predictor* and the *Semantic Reasoner*, whose role is to support informed assessment about the reliability of a composition, with consequent binding formation. We will now describe each of these core components and their interactions in more details.

Service Manager - The Service Manager component is the

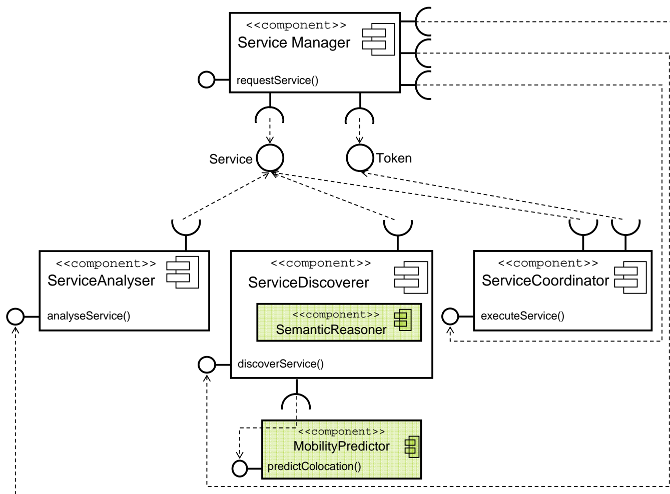

Figure 6: MoSCA Core Components

access point to the composition framework. It provides an interface to request services, leaving the invoking application unaware of whether the requested service is single or compound, and whether it resides on the same device or it is accessible from other devices in the proximity (in particular, at a single hop distance). Upon receiving a request for service S, the Service Manager creates a Service Entity object and passes it to the Service Analyser, whose goal is to 'understand' the request.

Service Analyser - The Service Analyser component provides two key functionalities: the ability to decompose a service according to its taxonomy, if the taxonomy is known, and the ability to compare services. In particular, upon receiving a Service object from the Service Manager, the Service Analyser decomposes S into component services $S_1, S_2 \ldots, S_n$, and returns the same Service object back to the Service manager, now enriched with the service (de)composition semantics (e.g., $S = S_1 \ \mathtt{seq} \ S_2 \ \mathtt{seq} \ldots \mathtt{seq} \ S_n$). Different decompositions are possible; the analyser favours those that rely on the minimum number of component services (i.e., services that appear higher up in the taxonomy), as this implicitly minimises the number of providers required to deliver a service. As soon as the first decomposition has been analysed, the updated Service entity is returned to the Service Manager, ready to be processed by the Service Discoverer. The Service Analyser keeps looking for alternative (more fine-grained) decompositions, in case there are no providers in the environment capable of delivering the more coarse grained ones. With reference to the example in Figure 4 (right), as there were no entities delivering S as a whole, providers of the more fine-grained $S1$, $S2$, and $S3$ had to be looked for; should a provider of $S3$ not be available, the Service Analyser would further decompose $S3$ into $S4$, $S5$, and $S6$, thus informing the Service Discoverer to look for providers of this fine grained services. In MoSCA, we assume that a pre-defined taxonomy (or ontology) exists to map a requested service S to a decomposition $S_1, \ldots, S_n$ with associated semantics (e.g., [10]); this taxonomy can be either universal (and built, for example, on OWL-S) or can be specific to a domain. The only requirement that MoSCA imposes on the taxonomy used is that it can be identified by a universal ID and that services there described can be recognized by a local identifier.

Service Discoverer - The Service Discoverer component is responsible for the selection of the providers, within the current environment, that will be relied upon to carry out a service request. It receives a Service Entity object from the Service Manager, after it has been further annotated with decomposition information by the Service Analyser; its goal is now to choose providers $P_1, \ldots, P_m$, among those available in the current environment, that will be able to deliver services $S_1, \ldots, S_n$ and that will maximise the chances of successful service completion. With reference to the example in Figure 4, the Service Discoverer component will thus first look for a provider of service S (i.e., top-level service type, being found in the taxonomy); should this fail, it would then try to find providers of $S1$, $S2$, and $S3$ (note that $S2\&S3$ does not exist in the taxonomy, and thus a single provider of such service is not looked for). Should this fail too, the most fine-grained decomposition would be attempted, with the Discoverer looking for providers of $S1$, $S2$, $S4$, $S5$, and $S6$. Note that MoSCA aims to deliver only those services that are deemed reliable, that is, services which have high probability to complete successfully, before any of the providers involved moves out of range of the requester. It is the responsibility of the Discoverer to quantify the reliability of the currently analysed decomposition, pondering alternative bindings should more than one provider of the same service be available. Only the binding maximising the currently analysed decomposition, and for which the probability of successful completion is above a given threshold, is returned to the Service Manager, if any. In order to quantify the reliability of a composition, the Service Discoverer relies on two further components: the Mobility Predictor and the Semantic Reasoner. The former estimates for how long a given provider will remain colocated with the client's device (where the composition framework is running), based on historical co-location patterns; the basic observation underpinning this idea is that people show a high degree of regularity in their activities, often traveling to/from work on the same train, following routines during their working days, visiting the same pub or restaurant, and so on. Although the number of unknown devices we encounter will always be high, a non-negligible set of devices, either stationary or mobile, will be re-encountered regularly [5]; it is thus possible to learn human behavioural patterns from past activities and use them as predictors of future behaviours. The latter (i.e., Semantic Reasoner) then uses these predictions, together with the specific composition semantics, to determine if a composition can be attempted and, if so, what providers are best to rely upon. In [13], we have proposed algorithms for co-location prediction and semantic composition analysis, and have demonstrated their positive impact on the rate of successfully completed composite service requests on a large set of real human mobility traces. The Service Discoverer, Mobility Predictor and Semantic Reasoner components discussed in this paper function as architectural placeholders of these algorithms; different algorithms can be deployed (and re-deployed at run-time) to deliver these functionalities, while leaving applications developed on top of MoSCA framework unaltered.

Service Coordinator - The Service Coordinator component is in charge of executing the service request. It receives, from the Service Manager, both an annotated Service object,

containing information about the (de)composition semantics and the selected providers (via Binding entities), and a newly created Token object storing the input parameters to the service request. The Service Coordinator will then carry on the request, updating the data stored in the Token object, splitting and consolidating it if necessary at every stage of the composition as the request is being processed. Note that a service decomposition, and its bindings, are dynamically re-assessed during execution; more precisely, each service decomposition is annotated with 'aspects', that is, entry gates in the execution flow where a re-assessment of the current environment should take place (with potential re-binding of services) before the service execution proceeds. For example, an aspect within the Smart Media Player compound service described in Section 2 is the entry to the `loop`: before the execution of a new iteration, the Service Coordinator notifies the Service Manager in order to re-assess the reliability of the composition, with new bindings potentially being formed. If one or more providers become no longer available in the middle of a service execution (i.e., between re-assessment entry points), the Service Coordinator notifies the Service Manager that, depending on the annotated decomposition, determines if the overall service can still be carried on (e.g., with roll-back to the previous entry point and re-binding) or if a failure must be reported.

3.3 MoSCA Implementation

We have implemented the previously illustrated MoSCA Framework in Java Micro Edition (Connected Limited Device Configuration 1.1, Mobile Information Device Profile 2.0). A package has been developed for each main component. A Factory pattern has been used for the retrieval of a Service Analyser, a Service Discoverer and a Service Coordinator; in so doing, it is possible to configure MoSCA to use various implementations of these interfaces, thus making it easily and dynamically extensible (with changes to one component not impacting any of the others). As previously anticipated, the Service entity has been realised using the composite pattern, thus intuitively representing the recursive nature of the composition; moreover, Service objects are designed intentionally not to have a direct reference to their providers. Provider instances can be only retrieved by using the association singleton class Binding; MoSCA can thus keep monitoring and updating the best providers for the various services in an asynchronous manner, while executing other services. The environment is being monitored to find available services by means of an Observer pattern. Implementations of an Environment interface are used to discover services accessible via different technologies (e.g., WiFi and Bluetooth); other monitors can be implemented and added to MoSCA framework, and be dynamically deployed by means of reflection. In total, the implementation occupies 39 KB, making it suitable for most mobile devices.

4. CASE STUDY

To gather a sense of the programmatic complexity of our framework, we consider the lines of code required to request the execution of a composite service within the MoSCA framework. An application willing to invoke a service using the MoSCA framework, would simply create a `MoSCAService` object, capturing the identifier of the taxonomy used, as well as the actual classification of the service within that taxonomy; it would then create a `Token` object, containing the parameters (as a list of attribute-value pairs) to be passed to the service. Finally, it would call the MoSCA Service Manager to start the service execution:

```
MoSCAService s = new MoSCAService(taxonomyID,
                                  serviceID);
Token token = new Token(``postcode'', ``WC1E 6BT'');
ServiceManager sm = new ServiceManager();
sm.requestService(s, token);
```

All services are instances of a `MoSCAService`: it remains transparent to the application (and its programmer) whether the service is indeed simple or composite, and whether it is being delivered by one or more providers, being them static or mobile. In case the requested service cannot be identified as a single service within a taxonomy, but can still be described as the composition of known service types, an explicit request can be made, which entails informing MoSCA about how the composition is to be carried out:

```
MoSCAService s1 = new MoSCAService(taxonomyID,
                                   serviceID);
MoSCAService s2 = new MoSCAService(taxonomyID,
                                   serviceID);
MoSCAService s = new MoSCAService(
                                   Service.SEQUENCE);
s.add(s1);
s.add(s2);

Token token = new Token(``postcode'', ``WC1E 6BT'');
ServiceManager sm = new ServiceManager();
sm.requestService(s, token);
```

As shown, the programmatic complexity involved in a MoSCA service request is minimal; should higher levels of control and adaptation be needed, MoSCA provides experienced developer with a rich set of directives that enable full (re)configuration of the framework (e.g., the Service Discoverer component and thus the algorithms it uses, the network interfaces used to listen to services being advertised, etc).

5. RELATED WORK

Service composition [4] has attracted a lot of interest since the advent of Web Services, and it has become a workable and broadly adopted technology thanks to real or defacto standards such as WSDL [17], SOAP [19] and UDDI [15]. This attention has mainly concerned the Internet and hence wired-environments, where the service providers are static and well known. In this domain, research has followed two main directions: one stream has focused on developing languages that aimed at adding semantic information (e.g., WSDL-S [18], OWL-S [11], METEOR-S [12]) and/or protocol information (BPEL4WS [1], WSCDL [21], METEOR-S [12], OWL-S [11], YAWL [16]) to service descriptions. A second stream has then focused on methodologies to aggregate these services, which can be broadly classified into: manual, semi-automatic and automatic [2]. Manual service composition entails the requester to browse a registry of services, find the desired service operations, and model their interactions into a flow structure (mainly with BPEL); the final service is then exposed as a unique service using WSDL. This methodology is effectively employed today within the Web Service Industry. Semiautomatic composition of services usually involves a service composition system that interacts with the requester in an iterative manner in order

to obtain information about the requested service, and to construct aggregate services out of the registered ones. The automatic composition, instead, demands the existence of a discovery agent that receives a service request and generates a structure of services/operations of some pre-registered services based on the information provided in the request. These automatic aggregation approaches rely on services to be richly described by means of the previously cited Semantic Web Service languages.

Service composition in pervasive environments requires this kind of automatic (de)composition. However, the very nature of the environment opens the door to new challenges that limit the applicability of current technologies. For example, resource limitations on the target devices have called for novel solutions to enable efficient ontology-based semantic matching of services [10]; more flexible approaches enabling the on-demand creation of an agreed ontology are being investigated too [20]. More closely related to our work are approaches that take into consideration mobility of devices, and thus services. In [3, 8] single service discovery protocols have been proposed that aim to find the device capable of delivering the best quality of service, given the dynamicity of the current environment; the discovery protocol has also been extended to consider multi-hop networks [14]. We argue that, while important, QoS reasoning must come *after* colocation reasoning, especially for services that require more than just a few seconds to complete. For the same reason, multi-hop service discovery and delivery is promising only for services that execute very fast, and/or are deployed in fairly stable scenarios. We first explored colocation reasoning in [9], where device mobility was considered when choosing from what single device to download content; we then extended our prior work by proposing algorithms that enable the mobility-aware discovery of, and reasoning about, composite services [13]. In this paper, we have completed the work with an adaptive and easy-to-use framework that realises such algorithms and that builds upon existing standards for service description to deliver a reliable service composition experience to the end user. Research on service composition in mobile environments is still in the very early days, and other issues, which were just marginal for infrastructure-based environments, must now be looked into, including session management [6, 7], on-the-fly adaptation, on-demand composition, dynamic execution monitoring, failure recovery mechanisms, to name a few.

6. CONCLUSION

In this paper we have presented MoSCA, a service composition framework that enables the rapid development and deployment of reliable composite services and applications. MoSCA provides application engineers, as well as end users, the abstraction of a MoSCA Service as a single, locally available service. Upon service request, MoSCA transparently binds the MoSCA Service to the set of available providers that are capable of collectively delivering the composite service with the highest reliability. It does so by reasoning about the composition semantics and the dynamically learned co-location patterns with other providers. Unforeseen changes to such patterns are being monitored during service execution, thus triggering re-bindings during if necessary. Although we have shown that MoSCA is, by design, able to react to changes in the surroundings, its efficiency still needs to be proved; our next step thus entails a proper profiling of MoSCA, both in terms of computational and runtime memory overhead.

7. REFERENCES

[1] Business Process Execution Language for Web Services (BPEL4WS) Version 1.1. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf, May 2003.

[2] A. Brogi and R. Popescu. Contract-based Service Aggregation. Technical Report TR-06-12, Dept. of Computer Science, University of Pisa, July 2006.

[3] L. Capra, S. Zachariadis, and C. Mascolo. Q-CAD: QoS and Context Aware Discovery Framework for Adaptive Mobile Systems. In *IEEE Intl. Conference on Pervasive Services*, Santorini, Greece, July 2005.

[4] D. Chakraborty, A. Joshi, Y. Yesha, and T. Finin. Service Composition for Mobile Environments. *Journal on Mobile Networking and Applications*, 2004.

[5] N. Eagle and A. Pentland. Reality Mining: Sensing Complex Social Systems. *Personal and Ubiquitous Computing*, 10(4), 2006.

[6] C. Julien. Adaptive Preference Specification for Application Sessions. In *Proc. of the 4th Intl. Conference on Service-Oriented Computing*, pages 78–89, Chicago. 2006.

[7] C. Julien and D. Stovall. Enabling Ubiquitous Coordination Using Application Sessions. In *Proc. of the 8th Intl. Conference on Coordination Models and Languages*, pages 130–144, Bologna, Italy, June 2006.

[8] J. Liu and V. Issarny. QoS-Aware Service Location in Mobile Ad-Hoc Networks. In *Proc. of the IEEE Int. Conference on Mobile Data Management*, 2004.

[9] L. McNamara, C. Mascolo, and L. Capra. Content Source Selection in Bluetooth Networks. In *Intl. Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, Philadelphia, USA, 2007.

[10] S. B. Mokhtar, A. Kaul, N. Georgantas, and V. Issarny. Efficient Semantic Service Discovery in Pervasive Computing Environments. In *Proc. of the ACM/IFIP/USENIX 7th Intl. Middleware Conference*, Melbourne, December 2006.

[11] OWL-S: Semantic Markup for Web Services Version 1.1. http://www.daml.org/services/owl-s/1.1/overview/.

[12] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. METEOR-S Web service Annotation Framework. In *Proc. of the 13th Intl. World Wide Web Conference*, 2004.

[13] L. D. Prete and L. Capra. Reliable Discovery and Selection of Composite Services in Mobile Environments. In *Proc. of 12th IEEE Intl. Enterprise Computing Conference*, Munich, Germany. Sept 2008.

[14] F. Sailhan and V. Issarny. Scalable Service Discovery in MANET. In Proc. of the 3rd IEEE Intl. Conference on Pervasive Computing and Communications, Hawaii, USA, March 2005.

[15] The UDDI Technical White Paper. http://www.uddi.org/, September 2000.

[16] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[17] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[18] Web Service Semantics - WSDL-S. http://www.w3.org/Submission/WSDL-S/, 2005.

[19] SOAP: Simple Object Access Protocol. http://www.w3.org/TR/SOAP/, April 2007.

[20] A. Williams, A. Padmanabhan, and M. Blake. Experimentation with Local Consensus Ontologies with Implications for Automated Service Composition. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):969–981, July 2005.

[21] Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/ws-cdl-10/, November 2005.