Dynamic Composition of Cross-Organizational Features in Distributed Software Systems

Stefan Walraven, Bert Lagaisse, Eddy Truyen, and Wouter Joosen

DistriNet, Dept. of Computer Science,
K.U.Leuven, Belgium
{stefan.walraven,bert.lagaisse,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract. Companies offering software services to external customer organizations must ensure that the non-functional requirements of all these customer organizations are satisfied. However, in such a cross-organizational context where services are provided and consumed by different organizations, the implementation of features, for example security, is scattered across the services of the different organizations and cannot be condensed into a single module that is applicable to multiple services.
In this paper we present an aspect-based coordination architecture for dynamic composition of cross-organizational features in distributed software systems such as systems of systems or service supply chains. The underlying approach of this architecture is to specify the features at a higher level that abstracts the internal mechanism of the organizations involved. A coordination middleware dynamically integrates the appropriate features into the service composition, driven by metadata-based user preferences.

Key words: Cross-organizational, Feature-oriented, Service engineering, Dynamic composition, AOSD

1 Introduction

A software service is used by several customer organizations simultaneously, possibly each of them with their own end users. Each customer organization may have different – possibly conflicting – requirements with respect to the features provided by the service. As non-functional requirements are often application-specific and pervasive, a software service is required to allow on-demand integration of tailorable features. To fit those varying requirements recent trends in service engineering aim to combine the benefits of feature-based and service-based approaches [3,13,1,16]. This combination offers a modular and pluggable solution that increases the reusability of services and supports on-demand customization tailored to the user's needs.

The building blocks for these customizations consist of modular features. A feature is a distinctive mark, quality, or characteristic of a software system or systems in a domain [11]. Features define both common facets of the domain as well as differences between related systems in the domain. They make each system in a domain different from others.

However, services are mostly used in a service composition consisting of services from different organizations. In such a cross-organizational context, service implementations are black boxes, implemented and deployed by different organizations on possibly different service platforms, and only the interface descriptions are publicly available [1]. In this cross-organizational and heterogeneous context, a problem arises when a feature implementation cannot be contained within the service provider, but crosscuts the service provider and service consumer (see Fig. 1). Because such a logical but distributed feature cannot be condensed into a single feature any more, on-demand service customization tailored to the user's preferences is hard to achieve in a cross-organizational context.

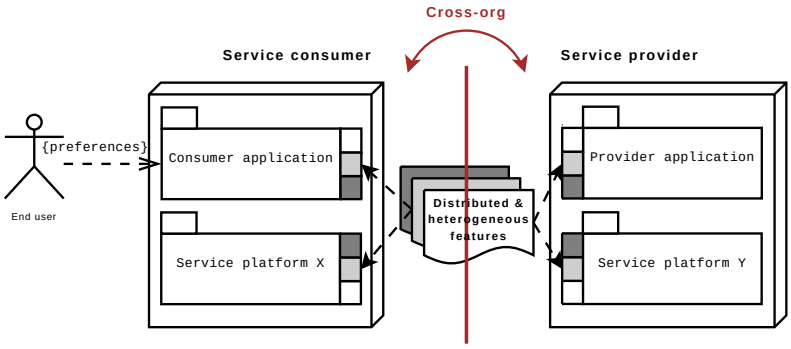

Fig. 1. Problem context of cross-organizational feature composition.

A typical example of a cross-organizational crosscutting feature is security in distributed e-finance software, such as online stock trading systems. When implementing an access control concern (authentication and authorization) in such an application, security actions need to be performed for every interaction between the distributed subsystems as presented in Fig. 2. In this cross-organizational context it is difficult to defend that a single feature module encapsulates the implementation of the internal security mechanisms of the organizations involved as well as the global security policy governing how security must be addressed in the overall interaction between different organizations. The latter security policy belongs to a level of abstraction above the internal security mechanism, allowing different implementations.

The problem of cross-organizational customization of services has not been well addressed in the current state-of-the-art. On the one hand, existing coordination architectures for cross-organizational service provisioning (GlueQoS [29], BCL [18], T-BPEL [26]) focus on dynamically establishing and monitoring agreements for enacting service delivery, but fail to support the coordination and dynamic composition of system-wide software features. On the other hand, state-of-the-art dynamic composition technologies such as dynamic aspect weaving fail to offer the right mechanisms for expressing customizations at the right level of abstraction.

This paper proposes an aspect-based coordination architecture for dynamic composition of cross-organizational features in distributed software systems such

as systems of systems or service supply chains. The underlying approach of this architecture is to provide support for cross-organizational service customization by leveraging some of the principles of cross-org coordination architectures. First, a high-level feature ontology is specified to agree about a technology-independent feature model among the organizations within a specific contract domain or service network. Second, service consumers are able to express a desired feature configuration through the specification of a feature policy that is based on the vocabulary of this ontology. Third, each organization has to map their part of the feature ontology to a specific aspect-based implementation platform. The underlying coordination architecture is responsible for managing the feature ontology, its organization-specific mapping to aspects and the deployment of user-specific feature policies within various scopes such as per service binding, per session and per message.

The remainder of this paper is structured as follows. In Sect. 2 we further motivate and illustrate the importance of cross-organizational feature composition in distributed software systems. Section 3 shortly discusses related work. We describe our aspect-based coordination architecture for dynamic composition of cross-organizational features in Sect. 4 and evaluate the performance overhead of our prototype in Sect. 5. Section 6 concludes the paper.

2 Problem Motivation and Illustration

We present an example in the e-finance domain to further motivate and illustrate the importance of cross-organizational feature composition in distributed software systems (see Fig. 2). Banks offer their customers a stock trading service to inspect, buy and store stock quotes. To be able to provide this service, those banks cooperate with the stock market, which in turn cooperates with a settlement company. Such a cross-organizational service composition allows each participant to take up two roles: *service consumer* and *service provider*. For example, the bank company is a server for the bank customers, but consumes the `QuotesOrderService` of the stock market.

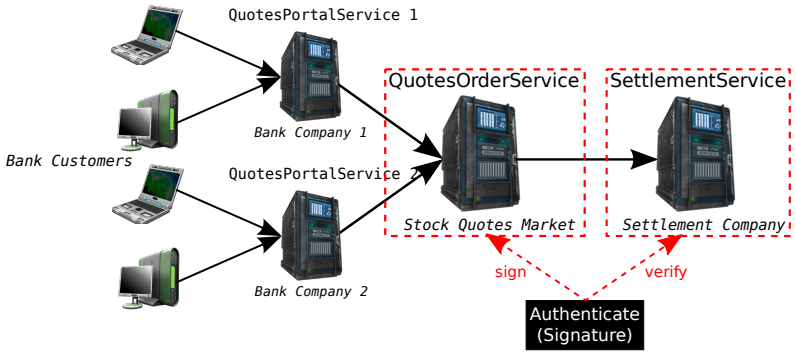

Fig. 2. Illustration of the stock trading service composition including the signature-based authentication feature as a single module.

Since different clients have different needs, the service providers must ensure that the different and varying non-functional requirements are satisfied, for example security, transaction support, load balancing, priority processing and stepwise feedback. In our example the bank customers can obtain different variants of the stock trading service composition by selecting features tailored to their preferences. For instance, a bank can offer to their customers several signature-based authentication options from which they can choose one, based on the different algorithms (e.g. `SHA1withDSA`). In the same way the stock market will negotiate with the different bank companies about the message carrier, the used protocols or which trusted third party (TTP) will be used. The stock trading service in Fig. 2 includes a signature-based authentication feature, applied on the connection between the stock market and the settlement company. This feature affects both the service consumer, to sign the messages, and the service provider, to perform the verification of the signatures. This clearly illustrates that a single feature module, consisting of client and server functionality, can affect multiple services in a service composition.

However, each company in a cross-organizational service network has its own IT administration and trust domain, and will not allow external parties to add or update feature implementations. The services provided by the different partners are black boxes, independently maintained by the company's own administrators. This black-box scenario hinders the feature modularization and composition in a cross-organizational context [1]. Moreover, since the services are loosely-coupled, the different parties can use different service platforms, programming languages and feature composition techniques (e.g. Java and .NET platforms).

Therefore a feature cannot be condensed into a single module any more and should be applied in a *distributed and heterogeneous* way. Cross-organizational features need to be split up into consumer- and provider-side parts that respectively fulfill the service consumer and service provider responsibilities. A uniform high-level representation of those features is necessary to be able to share them in a cross-organizational and heterogeneous application domain or service network. Further we need a coordination middleware to dynamically activate the appropriate feature implementations throughout the service composition in a consistent way.

3 Related Work

This paper will tackle the problem statement by combining and improving two bodies of work. On the one hand, the body of work on cross-organizational coordination architectures; on the other hand, dynamic software composition technology in particular aspect-oriented middleware.

Cross-organizational coordination architectures. A core tenet of the body of research on cross-organizational coordination architectures is a multi-layered architecture distinguishing between policy and mechanism. In previous work we have defined a reference model [28] for classifying the different approaches. In general, a cross-org coordination architecture consists of an agreement language

for specifying agreements (either contract- or policy-based) and a coordination middleware for establishing agreements and monitoring these for violations. An agreement between a service consumer and provider specifies the rules of engagement, that must be complied with by service consumer and service provider. For example, agreements specify the flow of interactions and message types to be exchanged (BCL [18]), modal constraints (i.e. authorizations, obligations, prohibitions, timings (Ponder [4])), QoS requirements (GlueQoS [29]), usage of protocols and standards (T-BPEL [26]). Secondly, the underlying coordination middleware supports establishing agreements between client and server dynamically, and to enforce the agreements or detect violations against it. These architectures, however, are not designed for user-specific customization of shared service instances and the consistent deployment of distributed and heterogeneous software features throughout cross-organizational service compositions.

Aspect-oriented middleware. Aspect-oriented software development (AOSD) [6] has been put forward as a possible solution to address the problem of crosscutting (often non-functional) concerns. In addition, AOSD is often applied to enable modularization and composition of features [17,15,14].

Aspect-oriented frameworks [21,9,24,12,23,22] have played a key role in the modularization of middleware platforms: these have evolved from a monolithic platform with a declarative configuration interface towards an architecture that is able to plug application-specific or user-specific extensions on demand. Current aspect-oriented frameworks also support dynamic aspect weaving in a reliable and atomic manner [19,27]. These AO techniques make these platforms therefore ready for deployment in usage contexts where a shared service instance can be dynamically customized to customer-specific requirements by dynamically weaving in desired features. In a cross-organizational context, however, current AO technology fails to offer coordinating mechanisms for deploying multiple aspect modules across multiple organizations and heterogeneous platforms.

4 Aspect-Based Architecture for Cross-Organizational Composition

This section describes the aspect-based coordination architecture enabling dynamic composition of cross-organizational features. Figure 3 illustrates our approach underlying the architecture. Similarly to the research on cross-organizational coordination architectures, our architecture assumes that a conceptual model for cross-organizational features is agreed upon between all organizations within a particular domain or a specific service network. This conceptual model defines a feature ontology, a feature model [11] in fact, shared by all organizations involved, for naming and defining the different features and their alternative implementation strategies. Next, the feature ontology is mapped to an aspect-based feature implementation within each organization which can use an AO-technology of its choice. Subsequently, a service consumer can then express user-specific feature preferences when binding to a service provider. An underlying coordination middleware will ensure that the appropriate feature im-

plementations are activated dynamically throughout the service composition in a consistent manner at the right moment, driven by metadata.

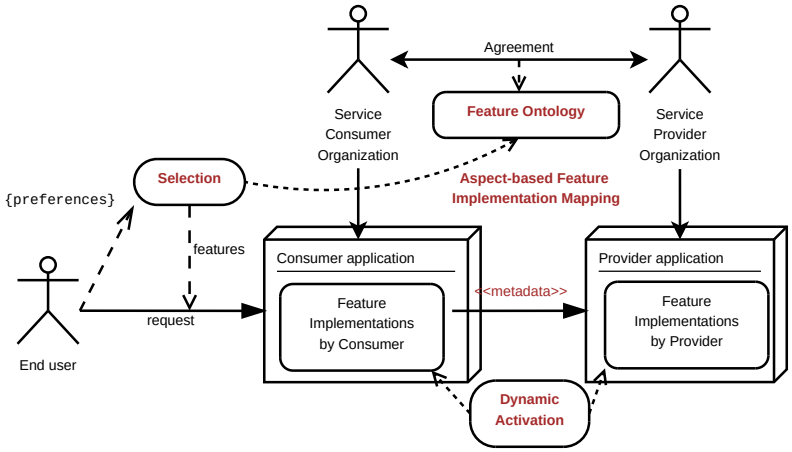

Fig. 3. Overview of the approach.

We will now explain the architecture in more detail. First, the high-level, technology-independent feature ontology is presented. Next, we describe the aspect-based feature implementation mapping. Thereafter it is presented how users can specify customization requirements through feature policies. Finally we present the design of our coordination middleware. As a running example we will use the example of dynamically composing security features such as authentication and non-repudiation in the stock trading service composition.

4.1 High-level Feature Ontology

The conceptual model in our approach for specifying cross-organizational features consists of a high-level feature ontology. This feature ontology should be abstract and independent from the aspect-based implementation (the computational model) to enable that different organizations in the service network can implement the same features differently depending on their choice of implementation platform and composition technology.

Within a particular domain, for example e-finance domain, a standard for non-functional features (e.g. security) can be agreed upon. Figure 4 presents an example of a feature ontology for security features based on existing standards [8,20,2]. Security protection breakdowns into authentication, authorization, audit, availability, confidentiality, integrity and non-repudiation. We also itemized some possible feature implementation strategies, based on the different algorithms (e.g. SHA1withDSA for signature-based authentication).

A feature node within the ontology consists of a *feature identifier*, a unique name for referring to that feature, and a high-level, technology-independent *feature contract* about the intended behavior of the feature and the roles that different parties involved have to play. These roles are described by a name

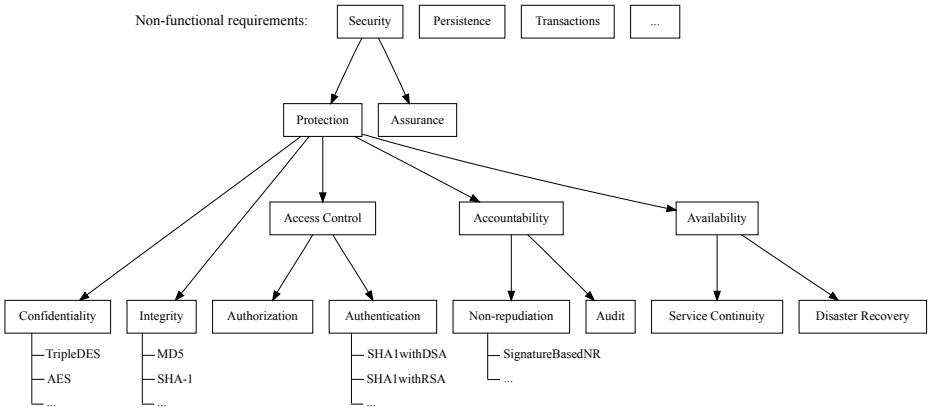

Fig. 4. Example of a feature ontology for the stock trading composition.

(e.g. Service Consumer) and a set of responsibilities that specify constraints on behavior and interfaces. Furthermore, composition rules can be specified that prescribe which features depend on other features and which features can't be executed during the same request due to feature interference.

For example, the `SignatureBasedNR` non-repudiation feature (see Fig. 4) defines two roles: a service consumer who retrieves the customer account information, and a service provider responsible for securely logging the customer account, the name and arguments of the request, and the cryptographic signature of the message. The service provider role requires a `CustomerAccount` attribute, which will be provided by the service consumer role. There is also a dependency rule necessary that prescribes that `SignatureBasedNR` requires the `SHA1withDSA` authentication feature to provide a `Signature` attribute.

Currently we have not yet designed a concrete language for representing feature contracts (We expect though that such feature contract language would be based on existing feature modeling techniques for services such as Service Diagrams [7].). We do offer a declarative specification language for representing feature identifiers and roles from the ontology and their mapping to specific feature implementations.

4.2 Aspect-Based Feature Implementation Mapping

The mapping between the high-level feature ontology and the aspect-based implementations is specified on the level of the internal processes and data, hiding the implementation details for external parties. By capturing the semantics of the features in a high-level feature ontology, the different features can be implemented independently by each of the service providers using their favorite service platform and AO-composition technology. Hence, the different services in the network may have their own optimized aspect-based implementations of the different features, and the most appropriate feature implementation in each service may depend on environmental circumstances. However, the feature implementations have to satisfy certain constraints, enforced by the feature on-

tology. In addition, the implementation of the different features and the software composition strategy are open for adaptation by each of the local administrators.

The use of AOSD [6] enables a clean separation of concerns, in which the core functionality of a service is separated from any feature behavior. Therefore features are implemented separately from each other as composite entities containing a set of aspect-components, providing the behavior of the features (so called advice). This advising behavior can be dynamically composed on all the components of a service – at consumer-side and at provider-side. The aspect-components of the features are composed by means of declarative specifications in the form of *AO-compositions*: these specify on which elements of the service platform the aspect-components must be applied.

Listing 1.1. Example of a feature implementation mapping.

```
featureImplMapping SHA1withDSAImpl {
 implements: SHA1withDSA;
 role: ServiceConsumer;
 ao-composition {
  id: SHA1withDSASigning;
  pointcut {
   kind: execution;
   componenttype: *;
   componentinstance: *;
   interface: ITransport;
   method: sendMessage;
  }
  advice {
   comptype: SHA1withDSASignature;
   interface: ISignature;
   method: sign;
}}}
```

Each *feature implementation mapping* within a specific organization is described by means of a declarative specification that specifies: (i) the feature and role that is implemented and (ii) a set of AO-compositions to integrate the feature into the internal processes and data of the organization. Such an AO-composition specifies a pointcut and a set of advices to apply. Listing 1.1 presents an example of a feature implementation mapping for the service consumer role of the `SHA1withDSA` signature-based authentication feature. The AO-composition specifies that this feature imposes on the transport layer of the service platform (pointcut) to digitally sign the messages before sending by means of the `SHA1withDSASignature` aspect-component (advice).

4.3 Expressing User-Specific Preferences Through Feature Policies

The feature ontology is accessible to the end users of the service application and allows them to select the desired set of features. A service consumer selects a set of features by instantiating a feature policy. A *feature policy* is a declarative configuration that specifies per service binding which features are desired for that particular binding (see Listing 1.2). A service binding simply identifies the URI and interface of the service provider in question. When user-specific preferences dictate that a particular feature must be applied, the feature implementations of

the consumer and provider side will be dynamically composed for every message exchanged through that service binding.

Listing 1.2. Example of a feature policy.

```
servicebinding {
 URI: http://www.stocktradingexample.be;
 port: StockTradingServiceSoapEndpoint;
 features: SHA1withDSA, SignatureBasedNR;
}
```

4.4 Coordination Architecture

In order to process user requests in a consistent manner throughout the cross-organizational service composition, coordination is needed between all participating services. To achieve this coordination, every request is tagged with extra meta information, specifying the set of features corresponding to user-specific preferences. This metadata propagates with the message flow initiated by that user request. As such, knowledge about the desired combination of features travels with the message flow. The coordination middleware also ensures that the appropriate feature implementations are activated dynamically when required.

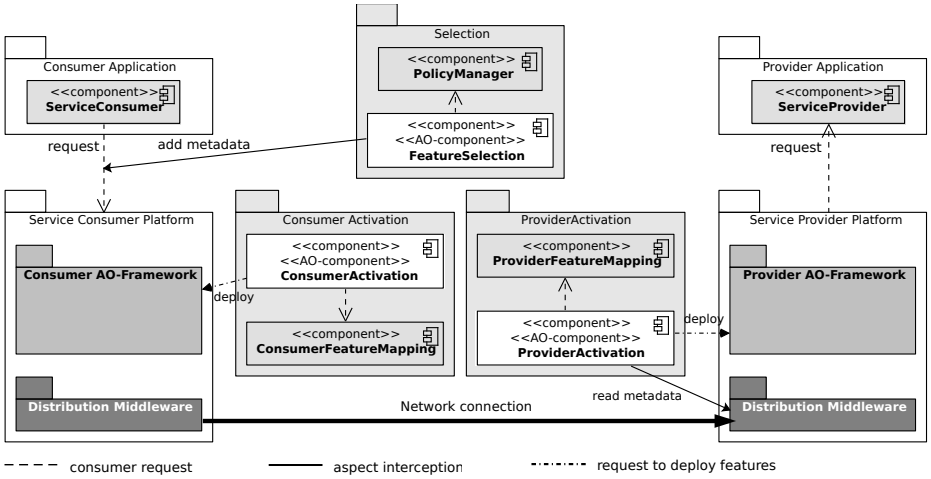

Fig. 5. The coordination middleware architecture.

Figure 5 presents our coordination middleware, built on top of an AO-framework. The coordination middleware relies on this AO-framework to support dynamic AO-composition. The runtime composition of cross-organizational features throughout service compositions consists of two main phases: *selection* and *activation*. These phases are represented in the architecture as modular packages: the `Selection` and `ConsumerActivation` package are part of the consumer service platform; the `ProviderActivation` package is included into the provider service platform. In the rest of this subsection we first explain the selection phase. Then the general structure and operation of the activation phase are described.

Selection of Cross-Organizational Features. The machinery for the selection phase of the coordination middleware consists of the `FeatureSelection` aspect and the `PolicyManager` component. FeatureSelection is imposed onto the application layer where it intercepts the requests to remote services that are subject to a certain feature policy (cf. Listing 1.2). The PolicyManager processes the feature policies and stores which features apply to each service binding. These data structures are hash maps with constant access time. The FeatureSelection aspect queries the PolicyManager to determine which features apply for a given service binding, and annotates the intercepted messages with the feature identifiers of the required features. It also keeps track of which service bindings have already been customized: only for new service bindings the PolicyManager is queried for required features.

Activation of Selected Cross-Organizational Features. After the necessary cross-organizational features are selected, those features need to be activated. The activation phase of the coordination middleware consists of the `ConsumerActivation` and `ProviderActivation` aspects and the `ConsumerFeatureMapping` and `ProviderFeatureMapping` components. The feature mapping components store a hash map of feature implementations and their associated AO-compositions. This allows to query the appropriate feature implementation based on a given feature identifier. ConsumerFeatureMapping and ProviderFeatureMapping handle respectively the service consumer and service provider roles of the features. The consumer-side ConsumerActivation aspect imposes on the consumer service platform; at the server-side the ProviderActivation aspect intercepts all incoming messages. These locations are the first joinpoints – thus the entry points – in the call flow of the remote requests, at the consumer-side middleware stack as well as the provider-side middleware stack.

ConsumerActivation intercepts all messages and checks them for metadata with selected features. The selected features are compared with the features currently applied on the particular service binding to see whether any changes are necessary. If a change is necessary, ConsumerActivation queries the ConsumerFeatureMapping component for the descriptions of the AO-compositions for the selected features and uses this information to compose the necessary aspect-components. The activation aspect will then send a request to the AO-framework to deploy those aspect-components.

The last step in the feature activation mechanism is notifying the provider-side about the feature change. This way we ensure the features are applied throughout the service composition in a consistent manner. Therefore the feature updates are added as a piggyback to the message.

At the provider-side, the ProviderActivation aspect intercepts the incoming requests and checks them for feature updates. If necessary, the new features are activated. The activation at the provider-side is analogous to the consumer-side. Concretely, ProviderActivation first inspects the currently activated cross-org features of the service binding, then it applies the feature changes using the feature implementation mappings retrieved from ProviderFeatureMapping. Incoming requests are also verified to be compliant with the feature policies: the

messages are inspected for unsupported or missing features. These messages are not accepted, and an exception message is returned.

Since the activation aspects check all requests, cross-organizational features can easily be deployed within a per message scope. Therefore the FeatureSelection aspect will need to query the PolicyManager for all intercepted messages. The activation aspects will ensure that features will be applied only once per service binding.

5 Evaluating the Performance Overhead

The evaluation aims to measure the absolute and relative performance overhead that our coordination architecture introduces on the overall responsiveness of service applications. In particular we want to measure the overhead introduced by the Selection and Activation components. Before we present the evaluation of our architecture, we briefly discuss our prototype implementation.

As a proof of concept, a prototype of the coordination architecture has been implemented as a framework using Java SE 6. It is built on top of our own aspect-component framework which offers support for dynamic AO-composition. The Java dynamic proxy technology is used to intercept invocations and call advices. For all declarative specifications we used XML files.

The coordination architecture is developed as an extension to the service platform, enabling the dynamic composition of cross-organizational features in distributed applications. Since it is a modular and aspect-based extension, the coordination middleware can be omitted or removed when needed.

Our current prototype relies on an aspect-component framework that also supports weaving various middleware features in the distribution layer. We measured the roundtrip by comparing a version of our distribution layer where all features are statically composed, against one where all features are dynamically woven using our coordination architecture. In particular we focused on one security feature for the distribution layer: the implementation of a signature-based authentication feature (for signing and verifying method invocations) as described in the running example of this paper.

We have used a round-trip latency benchmark as presented in [5] which requires minimal processing time. This way the processing time influences the results as minimal as possible and the results show clearly the worst-case runtime overhead of the coordination architecture. As described in [5], the benchmark application is composed of two JVM applications: a client and a server. The client invokes a remote service which implements an empty `ping` method with no arguments and a void return value. The benchmark scenario is configured as follows. The remote service and the client run into two JVMs on different hosts [1] in the same LAN. 100 benchmark series are executed sequentially. For all series, 20000 warmup operations are performed. Next, 20 steps of 5000 invocations are

[1] The benchmark tests were performed on systems with an Intel Core 2 Duo 3.00 GHz processor and 4 GB memory, running Ubuntu 8.04 (hardy) and OpenJDK Runtime Environment version 1.6.0.

measured. For each step an average execution time is logged. So 2000 average measures (100 series * 20 steps) are obtained for each benchmarked distribution layer, representing 10 million operations.

This benchmark application is extended with the signature-based authentication feature and executed on the two versions of our distribution layer: the default version and the version extended with our coordination architecture. In both cases the authentication feature will be deployed from the beginning.

The results of the round-trip latency benchmark test are presented in Fig. 6(a) and indicate that the overhead introduced by our coordination middleware is negligible. We can achieve these good results because the Selection aspect in our coordination architecture keeps track of which service bindings have already been customized. In addition the feature activation mechanism only notifies the provider-side when there are updates. In the current benchmark setup where the feature is deployed from the beginning, this will result in very few updates.

To measure the relative performance overhead of our coordination architecture, we compared our prototype against standard middleware platforms: Java RMI [25] and JBoss Remoting [10], using the default benchmark application (without security features). Java RMI is a minimal distribution layer with almost no customization capabilities. JBoss Remoting is a distribution layer with a standardized interface for pluggable transport and serialization layers. The three middleware platforms all use Java serialization and TCP sockets, but differ in their composition architecture: hard-coded (Java RMI), object-based framework (JBoss Remoting), or aspect-component framework (our approach).

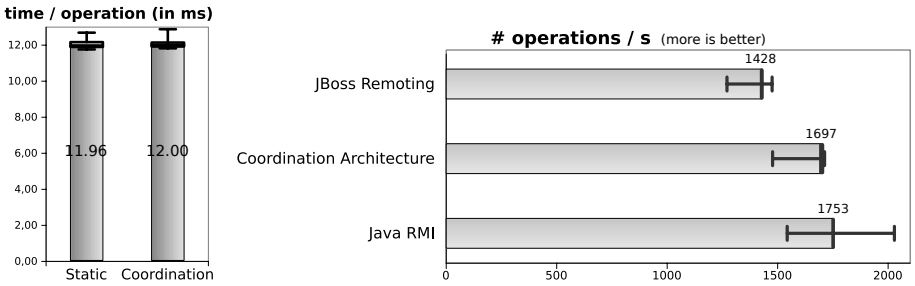

Fig. 6. Subfigure (a) presents the average time (in ms) needed to execute a `ping` operation in the two versions of the distribution layer, with the signature-based authentication feature activated. Subfigure (b) shows the average number of minimal method operations per second in the different distribution middlewares. The box plots at the end of the bars represent respectively the minimum, average and maximum during the different benchmarks.

The results of the second benchmark are used to calculate the average number of operations per second (see Fig. 6(b)) in the different distribution middlewares. These results show that Java RMI has the best performance (as expected), but more importantly, our coordination architecture and distribution layer introduce a relative overhead (in comparison to Java RMI) that is 6 factors smaller than that of JBoss Remoting. These initial results indicate that the overhead introduced by our coordination middleware is acceptable, especially in a cross-

organizational context where long-running and asynchronous interactions more frequently occur than synchronous interactions with timing constraints.

6 Conclusion

Cross-organizational customization of services has not been well addressed in the current state-of-the-art. In this paper we presented an aspect-based coordination architecture for dynamic composition of cross-organizational features in distributed software systems. A high-level feature ontology specifies for each feature the responsibilities of service consumer and provider. Using this feature ontology we are able to map feature identifiers to aspect-based feature implementations, allowing each organization to implement the features independently using an AOP technology of its choice. The underlying coordination middleware ensures that user-specific feature preferences are processed in a consistent manner throughout the cross-organizational service composition. Our architecture has been validated in a prototype. We have benchmarked this implementation and it shows acceptable performance.

References

1. Apel, S., Kaestner, C., Lengauer, C.: Research Challenges in the Tension Between Features and Services. In: SDSOA '08: 2nd International Workshop on Systems Development in SOA Environments. pp. 53–58. ACM (2008)
2. Beznosov, K.: Engineering Access Control for Distributed Enterprise Applications. Ph.D. thesis, Florida International University, Miami, Florida, USA (July 2000)
3. Cohen, S., Krut, R. (eds.): Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines. Carnegie Mellon University - Software Engineering Institute (May 2008)
4. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: POLICY '01: International Workshop on Policies for Distributed Systems and Networks. pp. 18–38. Springer (2001)
5. Demarey, C., Harbonnier, G., Rouvoy, R., Merle, P.: Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. In: CPM '04: The OOPSLA 2004 Component and Middleware Performance Workshop. pp. 7–24. Studio Informatica, Vancouver, British Columbia, Canada (2004)
6. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2004)
7. Harhurin, A., Hartmann, J.: Service-Oriented Commonality Analysis Across Existing Systems. In: SPLC '08: 12th International Software Product Line Conference. pp. 255–264 (2008)
8. International Organization for Standardization (ISO): Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 2: Security Architecture (ISO 7498-2:1989) (1989)
9. JBoss Community: JBoss AOP. `http://www.jboss.org/jbossaop/`
10. JBoss Community: JBoss Remoting. `http://www.jboss.org/jbossremoting/`
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. 21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)

12. Lagaisse, B., Joosen, W.: True and Transparent Distributed Composition of Aspect-Components. In: Middleware '06: 7th ACM/IFIP/USENIX International Conference on Middleware. vol. 4290/2006, pp. 41–62. Springer (November 2006)
13. Lee, J., Muthig, D., Naab, M.: An Approach for Developing Service Oriented Product Lines. In: SPLC '08: 12th International Software Product Line Conference. pp. 275–284 (2008)
14. Lee, K., Kang, K.C., Kim, M., Park, S.: Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In: SPLC '06: 10th International Software Product Line Conference. pp. 10–112 (2006)
15. Loughran, N., Rashid, A.: Framed Aspects: Supporting Variability and Configurability for AOP. In: Software Reuse: Methods, Techniques and Tools. pp. 127–140. Springer (2004)
16. Medeiros, F.M., de Almeida, E.S., de Lemos Meira, S.R.: Towards an Approach for Service-Oriented Product Line Architectures. In: Third Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL). pp. 151–164 (2009)
17. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: SIGSOFT '04/FSE-12: ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 127–136. ACM (2004)
18. Milosevic, Z., Linington, P.F., Gibson, S., Kulkarni, S., Cole, J.: Inter-Organisational Collaborations Supported by E-Contracts. In: Building the E-Service Society. pp. 413–429. Springer (2004)
19. Nicoară, A., Alonso, G.: Dynamic AOP with PROSE. In: ASMEA '05: Workshop on Adaptive and Self-Managing Enterprise Applications. pp. 125–138 (2005)
20. OMG: CORBA Security Services Specification. `http://www.omg.org/cgi-bin/doc?formal/02-03-11.pdf` (March 2002), Version 1.8
21. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible solution for aspect-oriented programming in Java. In: REFLECTION '01: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. pp. 1–24. Springer-Verlag (2001)
22. Rouvoy, R., Eliassen, F., Beauvois, M.: Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In: SAC '09: ACM symposium on Applied Computing. pp. 1021–1028. ACM (2009)
23. Söldner, G., Schober, S., Schröder-Preikschat, W., Kapitza, R.: AOCI: Weaving Components in a Distributed Environment. In: DOA '08: Distributed Objects and Applications. pp. 535–552. Springer (2008)
24. SpringSource: AOP with Spring. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html`
25. Sun Microsystems: Java RMI. `http://java.sun.com/javase/technologies/core/basic/rmi/`
26. Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N., Rouvellou, I.: Transaction Policies for Service-Oriented Computing. Data & Knowledge Engineering 51(1), 59–79 (2004)
27. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for Distributed Adaptations in Aspect-Oriented Middleware. In: AOSD '08: 7th International Conference on Aspect-Oriented Software Development. pp. 120–131. ACM (2008)
28. Truyen, E., Joosen, W.: A Reference Model for Cross-Organizational Coordination Architectures. 12th International Conference on Enterprise Distributed Object Computing Workshops 0, 252–263 (2008)
29. Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I., Devanbu, P.: GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. In: ICSE '04: 26th International Conference on Software Engineering. pp. 189–199 (2004)