Supporting Policy-driven behaviors in Web services: Experiences and Issues

Nirmal K Mukhi
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
nmukhi@us.ibm.com

Pierluigi Plebani
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
ppleban@us.ibm.com

Ignacio Silva-Lepe
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
isilval@us.ibm.com

Thomas Mikalsen
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
tommi@us.ibm.com

ABSTRACT

The Web services platform is gaining popularity as the technology of choice for integrating applications in diverse and heterogeneous distributed environments, such as the Internet. It is widely recognized that one of the barriers preventing widespread adoption of this technology is a lack of products that support non-functional features of applications, such as security, transactionality and reliability. Supporting such features in a service-oriented environment is more complex than traditional distributed computing environments since such behaviors cannot be assumed by applications, but some persistent representation of the behavior has to be discovered dynamically. WS-Policy has been touted as a possible future standard way to specify these features and associate them with services, but the multitude of related proposals resemble a poorly manufactured jigsaw puzzle more than a coherent vision.

This paper clarifies this vision by presenting a working prototype that demonstrates how the quality features can be expressed as *service policies* using WS-Policy. These policies are enforced by a policy framework that allows dynamic association of such non-functional features with applications on a per-interaction basis, as well as modification of these features. The paper discusses the issues involved in the creation of such a policy framework, and how the requirements of having to support transactional and reliable services guided the design. We hope that this work encourages other efforts to create viable products based on these emerging standards and advances the science of Service Oriented Computing.

ICSOC'04, November 15–19, 2004, New York, New York, USA.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Design, Reliability, Security, Standardization

Keywords

Service–oriented computing, Web services, Metadata exchange, policy

1. INTRODUCTION

Service Oriented Computing (SOC) is gaining prominence as the need for dynamic discovery and binding, a feature of service oriented architectures, is seen as a core requirement for the next generation of business applications. A particular standards-based instantiation of Service Oriented Computing, Web services [5], is seen by many as a viable platform for integrating scientific as well as business applications that operate in distributed and heterogeneous environments. The promise of platform independence, interoperability and a realization of SOC concepts has led to this popularity. However, the platform has lagged in delivering important features such as security, transactionality and reliability.

The important differentiator between supporting such features in SOC environments versus in traditional, insular distributed environments is that services are forced to make no assumptions about each other. In many other distributed systems, information about which applications interact with each other and how is well known at development time. By contrast, in SOC there has to be an up-front, declarative way of specifying the behavior that a service follows and expects from services it interacts with. For example, the algorithm used for signing a message, or transaction protocol used to safely conduct a business operation cannot be decided by service developers in an out-of-band or ad hoc manner. Metadata about services thus plays a central role in enabling SOC interactions, as described in [6]. The focus on metadata enables dynamic discovery and binding, and meta-

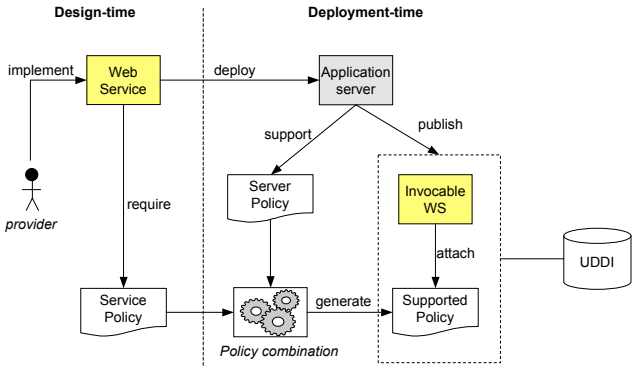

Figure 1: Policy management during design and deployment time

data based on standards enables interoperability. Metadata is generated at various points in the development and runtime lifecycle of an application, and needs to be communicated to possible partners. Thus the problem is twofold:

1. To define a metadata format for specifying non-functional features of applications and

2. To create a runtime framework designed around the above metadata that can support secure, reliable and transactional services.

Middleware research has addressed dynamically incorporating changes in non-functional behavior of applications to different degrees (for example, see [2] for a solution based on metaobjects, [8] and [10] for CORBA-based solutions, and [4] for context-aware middleware in a mobile computing environment), but these works do not assume an SOC platform, which comes with its unique challenges. Existing Web service products have piecemeal support for such features if at all. The existing support also lacks a complete view and focuses on a particular set of features (for example security). It does not take into account the common requirements across all non-functional features, such as the need for a common representation so we can reason about all the behaviors of a service, or the need for dynamic exchange of such metadata between services. This paper describes a working system that attempts to present solutions to many aspects of this problem. The contribution of this paper is to dissect the issues involved in creating such a system, and to describe our system design following the iterative process we went through during development. We hope our experiences help other researchers who are developing systems that operate in the emerging SOC world.

The paper is structured as follows: in Section 2, we describe the issues involved in solving the problem. In Section 3, we describe the model we use for representing non-functional features of services, based on WS-Policy[3][1]. In Section 4 we describe our system design, detailing our experience of design refinement as we discovered the needs of the

[1]Since the WS-Policy proposal is in flux, we define our own model for expressing service policies based on WS-Policy. So while our model is not strictly conformant to the current proposal, we believe that the modifications and interpretations we impose are necessary and will likely be addressed by future versions of WS-Policy

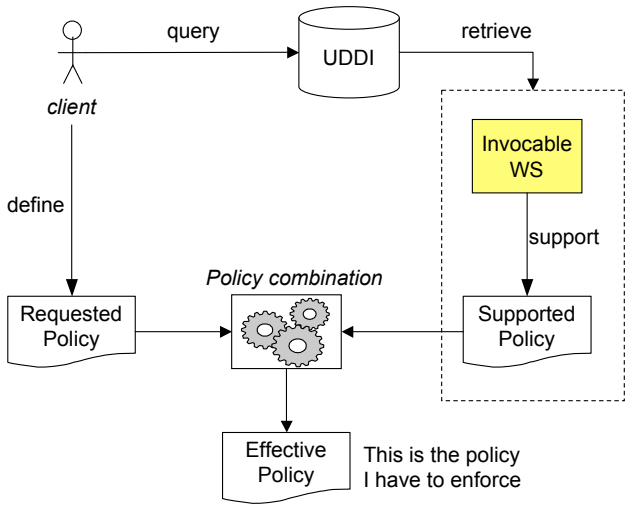

Figure 2: Policy management during run time

domains we addressed, viz. transactionality and reliability. In Section 5 we describe a simple scenario involving service interactions for a banking application which demonstrates the usefulness of our prototype. We conclude in Section 6 with a discussion of open issues and future work.

2. ISSUES

A solution that is aware of the non functional features of services first requires a standard way to describe such features and secondarily a set of mechanisms to handle such specifications. The next paragraphs present a discussion about this two main aspects.

2.1 Representing non-functional service behavior

2.1.1 Using WS-Policy

In Section 1, we described the need for a persistent metadata representation for non-functional behaviors. WS-Policy, being promoted by some industry players, is a natural fit since it is designed to specify the non-functional features of a service. WS-Policy does not rely on a precise model for describing non-functional features, but it is an extensible framework in which multiple feature domains can be included. As a brief description, WS-Policy defines a set of possible feature configurations using three main operators: *ExactlyOne*, *OneOrMore* and *All*. For example, consider a service that can be associated with two features: the possibility to encrypt (*enc*) and to digitally sign (*dsig*) a message. The semantic of the three operators will be:

1. *ExactlyOne(dsig, enc)*: only one of the two mechanisms must be used at same time,

2. *OneOrMore(dsig, enc)*: at least one of the two mechinisms must be used and

3. *All(dsig, enc)*: both the mechanisms must be used.

Naturally these operators can be nested and combined in order to create more complex expressions. It is worth noting that WS-Policy does not define the feature vocabulary for

a particular domain, but is responsible only for grouping features in order to identify a policy that can be attached to a service. WS-Policy defines the guidelines for how a policy assertion, specified in the vocabulary of a particular feature domain, should be written. For details the interested reader may refer to [3]; here we only mention that all policy assertions, compliant to the WS-Policy specification, must define when an assertion is *required* (the feature must be always present) or *rejected* (the feature must be absent).

2.1.2 Granularity of policy attachment

A policy assertion can specify features for a particular element of the service. For instance even if the digital signature feature is associated with all the messages handled by the service, the encryption feature may be associated only with a particular message which contains a confidential data. WS-PolicyAttachment[7] defines a way to attach a policy to a service, along with the possible granularities of that attachment. Policies attached to finer-grained service artifacts (such as a message type) need to be combined with those attached to coraser-grained artifacts (such as an operation within which that message of that type is exchanged) to compute the policy that is in effect.

2.1.3 Multiple sources of policy documents

Policies may be created by different roles during the lifecycle of a service. Figure 1 and Figure 2 show how the policies are involved during the design, deployment, and run time. In particular, the designer may associate some policies, collected in a document called *service policy*, indicating the behavior the service implementation is designed for.

Some policies, included in a document called *server policy* may be associated at deployment time, by a system or server manager. These would dictate the features that all services deployed on that particular server are expected to have.

Finally, at runtime, policies associated with partner service, called *requested policy*, which affect the features of the service under question, may be discovered. These multiple policies also need to be combined in order to compute the effective policy for a particular message exchange.

2.1.4 Combining policies to create effective policy

All the policies defined during the service life-cycle can be combined in order to obtain a subset of policies able to satisfy what is requested by the different roles.

For example, as shown in Figure 1, the service policy and the server policy are combined to obtain the subset of policies, i.e. *supported policy*, required by the service provider also supported by the server which the service is deploying on. Policies have to be combined in a way that preserves the intended semantics. Doing this automatically requires a careful consideration of the interpetation of each of the operators in WS-Policy. Additionally, we have to deal with policies that have different vocabularies, and talk about different feature domains.

We also have to deal with priority and conflicts. Policies may be prioritized: for example, policies for a particular service may override general purpose server policies, and policies attached to a message type may override those attached to an operation that involves a message exchange using a message of that type. Conflict detection is needed to ensure that the effective policy does not compose incompatible policies.

The manner in which conflicts can be detected results in the following natural classification of policy incompatibilities:

1. Incompatibilities arising from to contradictory policy assertions (for example, a service policy that asserts the need for encryption of a message while at the same time rejecting encryption for the port type to within which the message belongs) and
2. incompatibilities arising from assertions whose contradictory nature is detected using knowledge of the domain-specific semantics (for example, requiring encryption using algorithm A while at the same time requiring message compression using algorithm B may not be technically achievable).

2.1.5 Negotiating runtime policy

The policy that is in effect for a particular interaction is not the product of the service provider's policy preferences alone. The service requestor might have its own policies that have to be taken into account in order to arrive at an agreed effective policy. Thus, a matchmaking phase has to be performed in order to identify (i) if the provider and requestor policy preferences are compatible and, (ii) an effective policy reached after negotiation between both parties. In this matchmaking process, the lack of a common feature vocabulary presents a problem, which we will describe in greater detail in the next section.

2.2 Issues in designing a policy framework

Open problems do not end at the specification and processing of service policy documents. A service platform to deploy services whose interactions are governed by such policies also needs to address a number of challenges:

1. Effective policies are not calculated until runtime in many cases, so we have to dynamically compute the logic needed to enforce a policy. Additionally, a policy may allow for options, for example a reliability policy may specify multiple possible qualities of service. The actual policy in effect may not be known until a requestor sends a message from which the choice of the policy can be inferred. Thus not only is the policy handling logic dynamically computed, but the framework has to verify during an interaction that the corresponding policy is in fact being enforced, and may have to refine the policy that is in effect based on the information exchanged.
2. Policy-related information (such as signatures or algorithm specification for security policy, transaction ID for a transaction policy, etc.) will be carried as headers in messages. This information will need to be manipulated by policy handling code which enforces the policy. The order in which headers are processed is important, since there may be dependencies between policies. For example, the message and its headers may need to be decrypted before anything else can be done. So if a transaction policy is in effect at the same time, the transaction headers cannot be processed prior to the security handler decrypting the message.
3. Changes in a policy document need to be reflected in the runtime logic that enforces it. An effective policy, once calculated, is not immutable. The process of

incorporating changes in policy into the enforcement logic is non-trivial. At what point is it safe to replace the existing logic with the newly computed enforcement logic? What happens to ongoing interactions that are using the previous version of the policy?

3. POLICY MODEL

A policy document represents constraints over the service to which such a document is attached. The aim is that at runtime the service must never violate the behavior specified via the policy assertions. In this way a policy document can be viewed as a set of admissible configurations. Here and elsewhere we refer to a configuration as being defined by a particular set of non-functional features that must be exhibited by the service, as specified in the policy. For example, if a WS-Policy document attached to a service asserts *ExactlyOne(enc, dsig)* then the framework cannot apply neither, nor both of these security mechanisms at the same time for all interactions with this service. It is very useful to provide a boolean interpretation to the three WS-Policy operators. Assigning the XOR, OR, AND boolean interpetations respectively to the ExactlyOne, OneOrMore, and All operators, an the boolean NOT to rejected assertions, any WS-Policy document can be represented as a boolean expression. In particular the Disjunctive Normal Form (DNF) expression related to such a boolean expression explicitly identifies the set of all the admissible configurations for the service. For instance, referring to the example $ExactlyOne(enc, dsig) = enc \wedge dsig$ the DNF form is $\vee(\wedge(enc, \overline{dsig}), \wedge(\overline{enc}, disg))$. Here the DNF minterms represent the service *admissible policy configurations*. Since the DNF is a disjunctive form only one minterm can be valid at the same time.

Applying this interpretation to the multiple sources of policy as described in Section 2.1.3 in our model, we have three main policy documents:

- *Service Policy* (SP) applicable to a service or an element of a service,
- *Platform Policy* (PP), applicable to all services deployed on a platform, and
- *Requestor Policy* (RP), specified by a service requestor,

where the respective DNF represents:

- SP_{dnf} the set of admissible policy configurations independently from the platform in which the service will be deployed,
- PP_{dnf} the set of admissible policy configurations the platform offers to the services that are going to use this platform, and
- RP_{dnf} the set of admissible policy configurations the user requires from a service.

These three kind of policies, each of them with two different representations, are used during the deployment and invocation time in order to link together the platform the service and the user constraints. In particular combining SP and PP we obtain the Supported Policy (SPP), i.e. the policy supported by the service when it is deployed on a particular platform.

Such a combination is obtained according to the following formula:

$$SPP = SP \wedge PP$$

Actually it is more interesting combining the DNF representations. In fact the combination results in a selection of all the minterms included in both the combining expressions. In other words SPP_{dnf} represents the set of admissible policy configuration valid for both the service and the platform. According to that we can state that the service deployment will fail if the SPP is empty, i.e. no policy requested by the service is supported by the plaftorm. For example, if $PP_{dnf} = \vee(\wedge(enc, \overline{dsig}), \wedge(\overline{enc}, disg))$ and the $SP_{dnf} = \vee(\wedge(enc, transaction))$ the policies cannot be combined and the service cannot be executed in this particular platform (in this case the platform requires something the service cannot support, i..e transaction). Otherwise if $SP_{dnf} = \vee(\wedge(enc, \overline{dsig}))$ then $SPP_{dnf} = \vee(\wedge(enc, \overline{dsig})$. SPP, thus, describes all the admissible configurations for a service deployed in a particular platform independent of the service requestor.

In the same way, combining the SPP and RP we can obtain the effective policy (EP), i.e. the set of admissible policy configurations which should be valid while the user is invoking a service deployed in a particular platform. All the incoming and outgoing message, indeed, will have to satisfy one of the admissible policy configurations defined in the EP.

Even if this approach appears very simple, it relies on the assumption that the combining expressions are composed by the same assertions set, also called *vocabulary*. In other word if one expression does not specify anything about the digital signature, for example, and the other one does then any minterm can be equal since they differ at least for this assertion (if $SP_{dnf} = \vee(\wedge(enc))$ with the same SPP then the SPP will be empty). Actually in a service oriented environment this assumption is not always valid, and it is too restrictive, since all the actors define their own policies independently from each other. In this way the mismatch between the vocabulary can be discovered only when the combination phase starts. To solve this problem, the "absency is negation" approach is followed, i.e. if an expression does not specify an assertion the other expression includes then the missing terms are considered rejected (the NOT operator is applied). Using the same example, the expression used during the combination phase will be $SP' = SP \wedge (\overline{dsig})$ obtaining $SP'_{dnf} = \vee(\wedge(enc, \overline{dsig})$

4. POLICY FRAMEWORK DESIGN

The policy model now introduced represents the basis of a Policy Manager able to process and enforce the service provisioning according to the policy specifications. Figure 3 shows the main components of this Policy Manager totally written with Java and compliant to the subset of Ws-Policy specification specified in Section 2.1.1.

The module `Policy4J` is in charge to parse and store the policy document according to the boolean interpretation described above. In this way, every policy document is transformed in a tree form representing the correspondent boolean expression. The `Policy combination` module is invoked:

- during the service deployment to combine the service

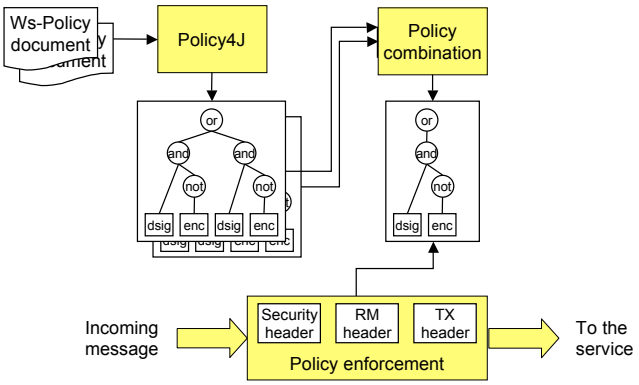

Figure 3: Policy manager architecture

policy and the server policy obtaining the supported policy, and

- during the service invocation to combine the supported policy and the requested policy obtaining the effective policy (the situation shown in the figure)

Once the effective policy is computed the `Policy enforcement` module is in charge to process all the incoming and outgoing messages in order to verify if all of them are compliant to what the effective policy expresses. In particular the enforcement module is composed by several policy headers each of them dedicated to a kind of policy. As it will be discussed in the following paragraphs each of these headers, even if follow the same way to process the messages, require a different approach to interact to the underlying infrastructure,

4.1 First Iteration

Given the issues to be addressed, our first policy framework was designed to accomodate message interceptors that would incorporate the logic required to enforce a policy. These interceptors could be chained together.

Enforcing a policy has two distinct pieces of logic: one that is responsible for translating relevant header information to and from wire format into a memory model for a message and its associated headers, and a separate piece of logic which actually processes that information to enforce the logic required by the policy. Thus, there were two processing chains in this framework: a header processing chain, and a policy handling chain. Each message exchange between a service and a partner had a potentially unique effective policy. We took the approach of having one common header processing chain for all such message exchanges, and then choosing an appropriate policy handling chain based on the policy in effect for that message exchange.

Policy documents from various sources and attached at different granularities are combined to create an effective policy for each message exchange. The framework then awaits the arrival of messages that are part of this message exchange. As messages arrive, the framework chooses from the available set of handlers an appropriate subset and chains these together to create a handler chain for this message exchange. As we noted in section 2, the policy may include options. Based on the incoming messages, the choice of policy eventually becomes clear and the configuration of the chain is then fully determined. All future messages that are part of this exchange will then be filtered through the same set of handlers chained in that particular configuration.

In our first design, we consciously avoided addressing issues regarding dependencies between header processors or handlers; i.e. we assumed that the framework creates header processing and handler chains with full knowledge of dependencies, so that those chains are correct by construction. Additionally, we avoided the issue of incorporating dynamic changes in policy, so once a handler chain for a message exchange was fully determined, it would not be modified even if a policy that contributed to the effective policy was modified.

4.2 Supporting transactions

The WS-Coordination and WS-AtomicTransaction specifications define message formats and protocols for interoperable distributed transaction processing in a Web services environments. Additionally, the WS-AtomicTransaction specification defines WS-Policy assertions to describe an endpoint's ability to support and participate in these protocols. The intended audience for such policies are other services. We define transaction policies that describe transaction processing requirements that a service imposes on it's environment (e.g., the container). Our transaction policies can be attached to both outbound and inbound conversations.

For example, the policy *PropagateTransaction* attached to an outbound conversation instructs the container to include the transaction context as part of the outgoing request. The policy *TransactionRequired* attached to inbound conversations instructs the container to process a message within the transaction context associated with the incoming request, or (if the request is not associated with a transaction) create a new transaction, and process the message within that context.

Such policies introduce additional requirements on our policy framework. One requirement is that a policy handler be able to alter the state of the system as opposed to simply transforming a message: e.g., the transaction policy handler needs to import, export, and demarcate transactions. Another requirement is that the policy "event space" be extended to include post-message processing events: e.g, the transaction policy handler needs to be notified both before and after a message has been processed by a service.

4.3 Second Iteration

The requirements described above highlight the limitations of the approach described in our first design iteration about the role of the policy handler. Indeed a simple listener, as we assumed the policy handler to be, cannot access internal resources or invoke external services. Actually such functions are required in order to perform operations like saving the current state of the service execution, as well as identifying and communicating with a transaction coordinator which is avaiable as an external service.

Thus, a refinement of the policy framework design described above is needed. In particular all the policy handlers can be considered as normal deployed services which can access to the resources like other services do. As before, all policy handlers should implement a standard set of methods used by the policy manager for lifecycle management and message notifications.

4.4 Supporting reliability

A WS-Reliable Messaging (WS-RM for short) policy handler provides support for delivering Web Service requests and responses as messages with delivery assurance guarantees that include: at most once, at least once, exactly once and in order. The first three guarantees are mutually exclusive, while the last can be combined with one of the other three (e.g., at least once and in order).

A WS-RM handler takes one of two forms. A WS-RM Source handler assigns sequence numbers to outgoing service request messages and inserts them into sequences. It keeps track of what sequence numbers have been acknowledged by processing sequence acknowledgements conveyed in incoming service response messages (or in messages not associated with service responses as well), and by sending acknowledgement request messages if necessary. A WS-RM Destination handler keeps track of incoming service request messages that are delivered to a service provider. If in order delivery applies, the Destination handler may need to suspend delivery of a service request message until all preceding service requests have been delivered. When an incoming service request has been delivered, the Destination handler sends an acknowledgement for the corresponding sequence number, as part of an outgoing service response (or in a message not associated with a service response as well).

Although the behavior to support WS-RM has been described in terms of a policy handler as a message interceptor, some WS-RM protocol messages (e.g., sequence acknowledgement or create sequence) may need to be conveyed as service invocations in their own right. Thus, a WS-RM policy handler also needs to behave as a managed service. In addition, given that the WS-RM Destination handler may need to suspend delivery of an incoming service request, the Destination handler may turn out to interrupt the chain of policy handlers itself. The chain will then need to be resumed at the interruption point for any suspended service request message that becomes ready for in order delivery. This last requirement led to our third design iteration, described below.

4.5 Third Iteration

The need for for the WS-RM handler have the ability to interrupt a policy handling chain led us to the following observation: policy handlers need have reflective behavior, i.e. they need to have the ability to modify the behavior of the framework that they are a part of.

So far, the reflective API we designed merely allows handlers to suspend and resume processing of a message. This makes it possible, for example, for the WS-RM handler to suspend processing of an out-of-order message. Once missing messages arrive and it is safe to continue processing the suspended message, some other actor in the system can initiate the resumption of processing. This is sufficient for the WS-RM handler; in future we aim to extend the reflective API so that handlers can inspect the chain they are a part of, and modify the structure of that chain. This could be a viable strategy for enforcing dependencies between handlers in a chain.

5. SCENARIO

Summing up all the involved element discussed so far, the following example presents how the policy driven behavior should be used in the typical bank case (see figure 4). Here a

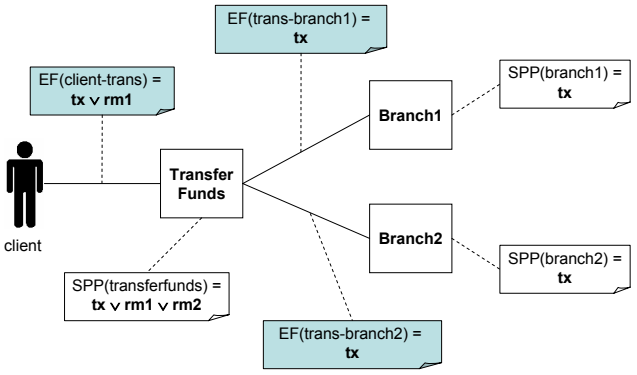

Figure 4: Funds transfer showing dynamic discovery and enforcement of service policies

`transferfunds` service relies on two bank account services, viz. `branch1` and `branch2`, to transfer money from one bank account to the second. So we have three different services in this scenario, running on their respective platforms, connected by Internet and communicating via the operations each provides. The functional interfacesof the services are appropriately described by WSDL documents. As described above the service description is also enriched by the non functional description through a WS-Policy compliant document and, in the same way, the supported policies of the hosting platforms are defined. Following the policy model we described in Section 3, let us suppose that the supported service policy for the three service, i.e. the service policy combined with the platform policy, are:

- $SPP(transf) = \vee(tx, rm1, rm2)$
- $SPP(branch1) = tx$
- $SPP(branch2) = tx$

where tx means "transaction is required", rm1 means "reliable messaging with exponential backoff with particular inactivity timeout and ack interval specified", rm2 means "reliable messaging with exponential only ack interval specified".

In other words the communication between the transferfunds service and its client can be ensured using one of the provided reliable messaging mechanisms. Moreover the transferfunds service can also execute its application logic over a transaction environment since the two service partners support this kind of policy. Now, suppose that the external client invokes the transferfunds service specifying the amount to move between the two accounts. Such a message is enriched with a set of headers also compliant to the policy the client wants the service support. In this way the tx and one of the two reliable message mechanisms, let say rm1, can be requested. So, the platform hosting the transferfunds service combines these policy requests with the supported policy in order to obtain the effective policy. The same operation will be done by the branch1 and branch2 services when the transferfunds starts to communicate with them, and in particular, requires them to be a part of a transactional protocol.

So, the obtained effective policies are:

- $EF(client - transferfunds) = \wedge(tx, rm1)$
- $EF(branch1 - transferfunds) = tx$
- $EF(branch2 - transferfunds) = tx$

Based on these effective policies all the involved platforms create a policy handler chain capable of processing the incoming messages. First of all, the handlers have to verify that the incoming message is compliant to the supported policy and, secondly, they must enforce what the effective policy requires. In particular the policy handler chain of the branch1 and that of branch2 is composed of only one handler, i.e. the transaction handler, whereas the chain on the platform which hosts the transferfunds service is composed of the transaction handler and the reliable messaging handler. In the latter case it is interesting to consider the the order in which the incoming message should be processed by the handlers belonging to the chain. In our case the reliable message handler has to be invoked first in order to provide to the transaction handler a complete and free error message.

6. CONCLUSION

This paper has described a working framework that uses supports transactional and reliable services within a dynamic SoC environment. The use of service policies to describe features, combined with the dynamic exchange of such policies between interacting services requires that the framework support message exchanges where the protocols and policies that apply are unknown until runtime. We have shown how, through the use of a policy model based on WS-Policy, we iteratively architected a solution for this problem. The strength of our framework is that it is based on emerging standards and hence our experience is relevant for other practitioners in the field.

However, there are a number of open issues we hope to address in future work. Matchmaking between services is a complex problem since it is difficult to reach an agreement when the vocabularies used in describing policies are different. The interpretation of extraneous or missing information in the policies is unclear. In future we hope that WS-Policy will address this issue. Our system did not support changing policies at runtime. This is certainly a requirement for most systems. The framework would have to keep track of what policies contributed to an effective policy in order to support this feature. Additionally, each handler would have to be able to tell the framework when it is safe to detach it from an existing handler chain, so that dynamic modification of the chain's structure would be possible. Our implementation assumes that dependencies between policy handlers are hardcoded into the framework logic, but of course in practice the framework cannot know about all policies and their relationships in advance. Instead, there could be a declarative way of specifying such dependencies; perhaps through a metapolicy. This problem is discussed in [9], where the authors discuss compositions of *microprotocols*, which are analogous to policy handlers in our system. A metapolicy-based solution is illustrated in the GlueQOS system[1]. An alternative solution could put the load on the developer of the policy handler to inspect the handler chain into which it is introduced and insert itself into the correct position within that chain. We briefly referred to this solution in section 4.5. We aim to explore both approaches in a future prototype.

7. REFERENCES

[1] Glueqos: Middleware to sweeten quality-of-service policy interactions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 189–199. IEEE Computer Society, 2004.

[2] M. Astley, D. C. Sturman, and G. Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, Apr. 2001.

[3] D. Box, F. Curbera, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web Services Policy Framework (WS-Policy Framework). Published online by IBM, BEA, and Microsoft at http://www-106.ibm.com/developerworks/webservices/library/ws-polfram,
2002.

[4] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, Kyoto, Japan, September 2001.

[5] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, Mar/Apr 2002.

[6] F. Curbera and N. K. Mukhi. Metadata-Driven Middleware for Web Services. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 278–286, Rome, Italy, December 2003.

[7] M. Hondo and e. Chris Kaler. Web Services Policy Attachment (WS-Policy Attachment). Published online by IBM, BEA, SAP and Microsoft at http://www-106.ibm.com/developerworks/library/ws-polatt/,
2003.

[8] P. Narasimhan, L. Moser, and P. Mellior-Smith. Using Interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, July 1999.

[9] B. Redmond and V. Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behavior. In *Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP 2002)*, pages 205–230, June 10–14 2002.

[10] E. Wohlstadter, S. Jackson, and P. T. Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 174–186, 2003.