AMPol: Adaptive Messaging Policy

Raja N. Afandi, Jianqing Zhang, Munawar Hafiz and Carl A. Gunter*
University of Illinois Urbana-Champaign

April 3, 2006

Abstract

Large scale distributed systems often cannot impose a uniform policy on all participants. This can be addressed by making it possible for parties to use diverse policies and adapt to the policies of others in their interactions. In this paper we investigate how to achieve this adaptiveness in messaging systems using a service-oriented architecture called *AMPol (Adaptive Messaging Policy)*, which provides services for expressing policies, finding them, and carrying out system extensions to adapt to them. We implement this approach with a web service middleware that allows parties to use policies for features like attachments, payment, encryption, and signatures. Our implementation demonstrates how AMPol can enhance the function of email messaging by enabling automatic deployment and use of features like cycle exhaustion puzzles, reverse Turing tests and identity based encryption without the need for global deployment or changes to the baseline messaging system.

1 Introduction

Service Oriented Architectures (SOAs) use middleware with standardized interfaces, languages, and protocols to provide interoperability between heterogeneous systems with loose coupling. One challenge for for this objective is support for non-functional (QoS) features like security, availability or reliability without breaking the interoperability of the system. In a highly dynamic and varying environment these features and their constraints may change frequently with each change affecting interoperability and flexibility. Supporting non-functional features in a service-oriented environment is more complex than traditional distributed computing environments (based on DCOM/COM+, J2EE and CORBA) since such behaviors cannot be assumed by client applications and there may be no coupling between requestor and provider. Such systems can be made to function on a large scale in an interoperable way *by allowing participants to dynamically adapt to the requirements of others with whom they need to interact*. In order to fully implement non-functional features, we need an end-to-end adaptive mechanism to support and ensure them on both requestor and provider ends. Also frequent changes and additions of non-functional features require automatic and potentially frequent system updates. SOAs based on web services with meta data specifications offer a promising platform for realizing this kind of scalability and interoperability.

While a variety of studies have explored aspects of how the promise of end-to-end adaptive interoperable systems can be realized, new case studies are needed to reveal more of the requirements and design alternatives. This paper describes one such effort in which we explore an adaptive architecture for large-scale messaging systems such as Internet email, instant messaging, chat rooms, list servers, Wiki pages, blogs, bulletin boards, and so on. In many cases these protocols lack basic facilities for adaptation and this breaks their functionality with the introduction of new requirements. For instance, Internet email messages are

*Contact Information: Afandi, Zhang, Hafiz, afandi@illinoisalumni.org, {jzhang24, mhafiz}@uiuc.edu, Gunter http://cs.uiuc.edu/cgunter

often discarded by email relays for reasons that are unknown to senders. In some cases these policies are secrets of the relays or recipients, such as many anti-spam filtering techniques, but, in other cases, they could have been advertised to potential correspondents to facilitate reliable messaging. Examples include rules for the allowed sizes of messages, types of attachments, origin guarantees such as DNS listing, required signatures or encryption, and so on. If these policies can be communicated to senders and they can adapt to them easily, then the overall messaging system can be made more secure and efficient without sacrificing convenience.

In this paper, we propose a reference architecture for adaptive interoperable messaging based on advertisable QoS requirements in a form of policies and a case study of this approach for email based on web services. Our architecture is called *Adaptive Messaging Policy (AMPol)* and it is based on the idea that message senders should learn and adapt to the policies of potential recipients. This is achieved through three fundamental architectural components: (1) a way to describe QoS requirements and constraints of the entities' adaptive behaviors in form of policies (such as the allowed size of messages) called the *policy model*, (2) protocols to publish, discover and merge such policies (such as retrieving them from servers) called *policy discovery*, and (3) means to add capabilities (*e.g.*download and install plug-ins) and enforce policy rules on messages called *Extension and Enforcement (EE)*. Our validation case study realizes AMPol for WSEmail, an approach to Internet email in which legacy protocols such as SMTP, IMAP, and S/MIME are replaced by families of web service (SOAP) calls and email messages are XML documents. We aim to validate AMPol by showing how potentially useful email services could be facilitated by the AMPol architecture to support QoS features in an end-to-end adaptable manner. In particular, our implementation is able to automatically support addition of new QoS requirements for availability and security by deploying and using plug-ins for puzzles (to raise burdens for spammers) and identity-based encryption (which allows senders to encrypt mail for recipients based on email addresses). All this can be done with secure and seamless operations that can realize adaptively in a large-scale messaging system.

AMPol is an end-to-end adaptive middleware based solution for supporting dynamic QoS features while maintaining the interoperability of the system. AMPol's main contributions are its end-to-end solution, adaptive and generic distributed policy framework, reference architecture for adaptive middleware for messaging systems, and application of this middleware for email system based on web services. There have been a number of attempts to exploit service-oriented architectures to make distributed applications more adaptive and interoperable. AMPol differs from these studies in its focus on exploring an end-to-end solution that incorporates all of the necessary adaptive support features. There are few attempts which claim to propose an end-to-end solution for adaptive support of QoS features, but either these efforts are not end-to-end or do not have the flexibility of AMPol.

The paper is divided into seven sections. Section 2 provides some background including some motivation, basics of puzzles and IBE, and an overview of our WSEmail middleware platform. Sections 3, 4, 5 are the core of the paper. They describe our policy model, policy discovery, and EE components respectively by providing the designs of the components and reviewing these with respect to our implementation of the case study. Section 6 discusses some of the work related to adaptive messaging and policy. Section 7 concludes.

2 Background

Consider the challenge of telling potential email correspondents some rules concerning the email you wish to receive. For example, you may wish to specify that attachments must be less than 500KB in size and must be of certain extension types, and that messages from certain parties, like banks and mutual fund companies, must be encrypted using IBE. This is difficult to do with current Internet email facilities because there is no standardized format for expressing these requirements or widely accepted protocols for senders to deal with them. However, such policies could be quite helpful in increasing security, decreasing spam, and avoiding a

lot of annoying email mysteries arising from policy conflicts. Quite a bit can be done by filtering, and there are limits on what can be advertised (for instance, it makes little sense to tell spammers what criteria you are using to identify them as such), but, in many cases, it would be helpful to just let potential correspondents know what protocol you would like them to respect in sending you a message. Dually, it would be useful to learn the policies of potential recipients and conform to these (when acceptable) with as little fuss as possible. In short, it would be handy to have more adaptive messaging systems.

Our validation of AMPol as an adaptive messaging system is based on our prototype for the case study of adaptive messaging policy with Puzzles and IBE for a WSEmail messaging system. We describe the background for these here. Their design and implementation in AMPol will be described in subsequent sections.

WSEmail [13] uses the emerging suite of W3C standards and service-oriented computing concepts as a foundation for messaging, rather than trying to design on top of the existing SMTP legacy protocols. It provides for a service-oriented Mail User Agent (MUA) client and Mail Transfer Agent (MTA) server which are designed to provide extensible messaging with plugins that works for both the MUAs and the MTAs. The existing system does not satisfy important requirements of messaging such as a protocol to discover policies or a policy model for messages. One of our key objects was to supply the functional components of adaptive messaging with very few extensions of the WSEmail platform.

Puzzles [10] are a mechanism to prevent DoS attack; in particular, puzzle-based anti-spam email systems have been studied for many years [8]. They aim to increase the cost of sending email in order to exploit an asymmetry between valid senders and spammers. There are two general type of puzzles. One type is a *cycle exhaustion puzzle* such as *hashcash* (`www.hashcash.org`). Another type of puzzle is a *Reverse Turing Test (RTT)*. If the recipient demands puzzle based anti-spam, the sender needs to know what kind of puzzle is required, what the puzzle problem is and where it can get the puzzle plug-in package to resolve the problem. AMPol is to provide for mechanisms to address all of these issues.

Identity Based Encryption [5] is a technique for addressing some of the burdens of key distribution that have made public key encryption of messages less widely used than PKI vendors had hoped. As in the scenario of the puzzle based anti-spam, the sender needs to know that the message should be encrypted by IBE and what IBE tool the sender needs to install. As with puzzles, our goal is to show how AMPol can aid the deployment of IBE without requiring universal adoption of IBE by users.

Our case study uses Voltage's IBE package (`www.voltage.com`) for IBEs, uses a RTT package based on CAPTCHA [19], and a cycle exhaustion puzzle system based on hashcash [3]. All packages are wrapped as COM components. However, the aim of AMPol is to facilitate the deployment of any technique that users think will be effective without the need for a global consensus or changes to the baseline messaging system. For instance, we could also have explored the use of various 'postage' schemes [1, 18] that have been proposed.

3 Policy Model

To achieve the adaptive messaging, there has to be an up-front, declarative way of specifying the behavior that an entity follows and expects from other entities it interact with. Policy model is such a framework for specifying a set of rules or constraints which describe entities' requirements during the communication. To achieve strong expressiveness and unambiguity, the policy constructs should be distinct or modular and different types of rule combination logics should be supported. In a distributed environment, the policy model should be able to specify which entities the policy is applied to and which entity enforce the policy. Only by this, the message conformance (with policy) logic and policy enforcement logic for each individual quality requirement can be taken out of the core policy engine and the true adaptivity is realized. The last point is that policy languages should be generic enough so that the policy schema and core policy engine

do not need to be modified by addition of new assertions. The rest of section shows how AMPol's policy model satisfy these requirements.

The basic unit of the policy is the construct called a Rule. Each rule describes an operation or process requirement for a message, *e.g.*encryption and signature. Each rule has two main parts. The first is the *action* that specifies the operation (*e.g.*encryption). The second is the *property* that specifies parameters of this operation (*e.g.*IBE encryption). The Rule's are combined into a RuleSet with connectives AND, OR, EXACTLY-ONE, *etc.*. One or more RuleSet's form a Policy. A Policy is associated with the application level usage (using the App attribute). The policies are specified as ingress policies, egress or local policies which apply to messages in a local domain. Finally, the policies forms the PolicySet which is a set of the policy constructs and storage unit of AMPol policy model.

Policies are classified as static or dynamic. Static policy is defined by each entity itself before the communication occurs. It defines rules that are affective for all conversations. Dynamic policy, which is based on static policy, is a set of actual rules for a particular conversation session. If there are multiple entities involved in a session, each entity needs to know the static policies of the others and create or obtain a set of rules based on all the static policies from every involved entity. The final instantiated rules are dynamic policies; they are generated during policy negotiation. We shall give more details about how to obtain dynamic policies in Section 4.

Existing policy specification languages do not have the concept of meta-specification, *i.e.* "the policies of policy". For example, each Rule or RuleSet needs some meta-information, including who will *perform* the required operation and who will *check* whether the requirement is satisfied. In AMPol policy model, the Subject is the entity the rule or rules set will be applicable to and the Target is the entity enforcing the rule or rules set. In a distributed system, the creator of the rule or the policy might not be the entity who will check the enforcement of the policy. So it is necessary to indicate the target explicitly. When conform or enforce the rules or assertions of a policy, the entity may need some extended functionality to perform the operation. We define Transformation for a Rule, which contains all the information to identify and download a particular plug-in. Policy engine could parse the transformation information and pass it to system extensibility module. The latter can download and execute the plug-in. Thus, if different domain specific rules and corresponding plug-ins are introduced, the policy engine as well as system extensibility module do not need change, *i.e.*, the true adaptivity is realized. All of above meta-information is included in the MetaSpecification, which has extra properties to indicate the visibility and the priority of the rule. One aspect of meta-specifications is that they are applicable at different granularities, both at the rule level and the rules set level.

Our case study is based on a set of rules called *APES* (*A*ttachment, *P*ayment, *E*ncryption and *S*ignature). In APES, there are rules Encryption and Signature to specify the cryptographic parameters (key size, encryption algorithm, version of the encryption algorithm *etc.*) used for encryption or signature. For availability, spam is the main concern. There is a Payment rule that specifies the type of payment imposed on the message sender like RTT or hashcash. Another important security concern is the attachment, which is the primary medium for spreading viruses among email client hosts [14]. There is a Attachment rule that specifies the patterns of the content. An example rule might say that the recipient does not accept an attachment that has a '.pif' extension. (This does not necessarily mean that the sender will not send a .pif file without this particular extension, but proper security would prevent .pif processing of the attachment.)

4 Policy Discovery

After specifying policies individually, the entities need to exchange them and negotiate a mutual acceptable policy set for the current session. Policy Discovery component fulfills this work. This component should be able to publish each entity's policy to everyone involved in the session and ensure the published policies

could be accessible remotely. Each entity should be able to query other's static policy or the final dynamic policy. Policy merging of two or multiple policies is required. It also should have a potential to support multi node negotiation. Our design comprises three functional sub-components, which provide the ability to advertise, query and merge policies.

Policy advertisement is used to publish static policies to other entities. The issues relevant to publishing are what to publish, where to publish, and how to publish. The publishable policies are determined using a meta level attribute of the policy model. The published policy is a subset of the static policy of the publishing entity. For the location of policy publication, the main requirement is the remote accessibility and availability. Policies of a particular client can be published at the client node, or at a dedicated policy server or at any remote server. In email systems policies for email clients can be published at a mail server. The entity that is publishing the policy also concerns how the policies are uploaded and maintained in a policy publishing server. This can be done in either the push or pull mode. In pull mode, policy server periodically (or when required) uploads/updates the policy for a particular client. In push mode, policies are uploaded/updated by individual clients. From the policy consumer's perspective, the main issue is the protocol and the location to find the published policies. The protocol for finding policies is implementation specific (*e.g.* HTTP, SSL, Web Service SOAP interface, LDAP, or custom built protocols). The location of a policy hosting node can be known in advance or can be learned by different types of discovery protocols such as UDDI or DNS.

Policy Merging is required to reconcile the policies of diverse parties in PQP exchanges (noted late). The propagated static policies are merged to create the dynamic policy. The message sender is the interested party to get the message recipient's policy and create the dynamic policy. The merging happens at different nodes of the system that lie in the path of the message sender and the recipient. The Policy and RuleSet constructs of the AMPol model have the *CombinationAlgorithm* structure, which specifies policy combination algorithms. If we have two policies with AND combination algorithm, the result of merging is a policy that has all the rulesets combined with an AND combination algorithm.

Policy Query Protocol (PQP) is the fundamental protocol for generating dynamic policies from static policies when a message transmission is required. We classify the entities involved in communication as the initiator and the respondent. The initiator is actually the message sender but in this context it is initiating the protocol. We present PQP for 'four-node messaging' as illustrated in Figure 1.

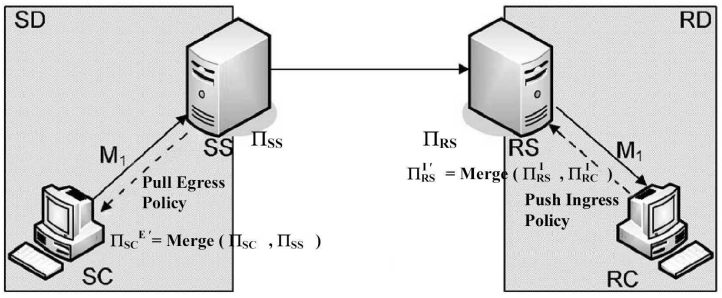

Figure 1: Policy exchange and merging

SC is the message sending client. It is connected with SS or the sending server. These two entities constitute the message sending domain. At the message recipient domain, RS is the recipient server that receives the message and forwards it to receiving client RC. The four entities involved have ingress and egress policies specified by the policy model. We denote the ingress policy specification of receiving client (RC) as Π^{I}_{RC}, the egress policy of sending server as Π^{E}_{SS} and so on.

In the policy advertisement mechanism for four node system, the policies can be advertised at the server

node. AMPol proposes either pull or push mecahnsim to load ingress policies from clients to a email server which serves as a policy server. The policy server then merges its ingress policy with the client ingress policies. This results in the merged ingress policy for the receiving server and client. It is denoted as: $\Pi^{I'}_{RS} = Merge(\Pi^{I}_{RS}, \Pi^{I}_{RC})$. These policies are stored per receiving client and can queried by using a unique identifier which can be fully qualified email address. This whole process is a part of policy advertisement.

The pull and push mechanism and merging of policies in the server is done in the policy advertisement step to achieve improved performance. Another advantage of this is that it does not require RC to be remotely accssible all the time. The clients can also involve in a pull mechanism for added performance. The clients pull egress policies from the server and merge them with their own static policies. This step is done at the start of the PQP protocol. The merging of policies at the client node reduces the requirement of any merging done during PQP. It is denoted as: $\Pi^{E'}_{SC} = Merge(\Pi^{E}_{SC}, \Pi^{E}_{SS})$, where $\Pi^{E'}_{SC}$ is the merged egress policy of SC and SS. The equilibrium state reached after exchange of these policies is shown in Figure 1.

The policy query phase can now be described. Step 1: Client SC wants to communicate with RC so it queries the advertised ingress policies for the RC and RS. It sends this query to its own server. SC $\rightarrow$ SS : Query for Π^{I}_{RS} and Π^{I}_{RC}. Step 2: Server SS relays the request to the server of the recipient. SS $\rightarrow$ RS : Query for Π^{I}_{RS} and Π^{I}_{RC}. Step 3: Server RS sends the merged policy $\Pi^{I'}_{RS}$ to the sender's server. RS $\rightarrow$ SS : $\Pi^{I'}_{RS}$. Step 4. Server SS sends the merged policy to the sender client. SS $\rightarrow$ SC : $\Pi^{I'}_{RS}$. Step 5. Client SC merges the received policies with the sending domain's egress policies ($\Pi^{E'}_{SC}$) and sends messages complying with $\Pi^{I'}_{RS}$ and $\Pi^{E'}_{SC}$ via SS. This goes through RS to RC and RC accepts it because it is compliant to its ingress policy. SC $\rightarrow$ RC : Message complying with $\Pi^{I'}_{RS}$ and $\Pi^{E'}_{SC}$.

The protocol above is the base case in which the communication has no cached values for policies. There is a stream-lined version when there is caching since this improves performance. To support caching, the policies have lifetimes. After these expire, policies will be renegotiated.

The implementation of the policy discovery model consists of three main modules: *policy publisher*, *policy merger* and *PQP handler*. The policy publisher service is implemented as a C# .NET Web Service. It reads the stored static policy file(s). The loaded policy is mapped into policy objects and maintained in a key-value map with key as a unique policy identifier. The sender side PQP module initiates the protocol on behalf of the sender client. It retrieves dynamic policy for a particular conversation by calling a *policyQuery* web method of a policy server. It then calls policy conformance module of the EE component for conforming a message to the policy. There is a policy merger module at both server and client nodes as policy merging is required at all nodes.

The AMPol modules are integrated to a messaging application by application-specific hooks. The hooks are placed at appropriate message entry and exit points in an application to intercept these messages and only allow them to proceed further if they are successfully processed by AMPol underlying services. These hooks can either be directly integrated into a source code of the application or plugged into the application if it provides a mechanism for adding plug-ins.

5 Enforcement and Extension

Once mutual acceptable policies have been negotiated, the entities need to determine how to conform to the policies (if it is the sender) or how to enforce the policies (if it is the recipient). AMPol's Enforcement and Extension model deals with extensions to the system of the sender to accommodate recipient requirements if it has determined to do so, and enforcement processing by the recipient to assure conformance to policy. This component should not modify the core base implementation of each entity. It should have meta-level control over the adaptation process itself in order to ensure that changes are carried out effectively. Each

extension should be described by policies and be able to run in terms of the policies after interpreting them. Each extension must be implemented as a separate module that can be incrementally added to and removed from the core application by addition or removal of a rule/assertion from the policies. Next, we'll show how AMPol fulfills these requirements and achieve the true adaptability.

The Enforcement and Extension model has three sub-components, each heavily reliant on Extensions. The *policy conformance component* is active at the sender side and the the *policy enforcement component* is active at the recipient side. The *system extension component* is active at both sides. Extensions are realized as third party plug-ins and extensible components that can be dynamically added or removed from the AMPol system. The novel idea behind this model is that we have extracted the logic of conforming a message to a policy rule (policy conformance and message compliance) and logic for enforcing a policy rule (policy enforcement) in an independent and dynamically pluggable process. This pluggable process is called an *extension*. The mechanism to locate, load and execute these extensions is called *system extension*. The interpretation of policy on the sender side is the execution of a series of extensions on a message to conform to the dynamic policy. For policy enforcement at the recipient's end, the recipient enforces the policy by executing the corresponding policy verification functionality on the messages. The policy conformance and the policy enforcement mechanism follows classic pipes and filters architectural style, with the operations for enforcement applied in reverse order.

Each extension application modifies or appends something to the input message. An extension for an RTT puzzle will generate an output message that will contain an original message appended with a puzzle result. Similarly an extension for IBE will generate a modified message which contains the encrypted original message appended with an encrypted symmetric key. These output message formats are defined in meta specification information of both extension and the Transformation part of the meta-specification for a static policy rule.

The enforcement and extension component is implemented using three corresponding modules: *policy conformance*, *policy enforcement* and *system extension*. We have designed and implemented an *AMPol extension framework* inspired by the WSEmail plugin framework [13]. It is a white-box framework and is extended by inheritance. Figure 2 shows the steps followed by the conformance, extension and enforcement

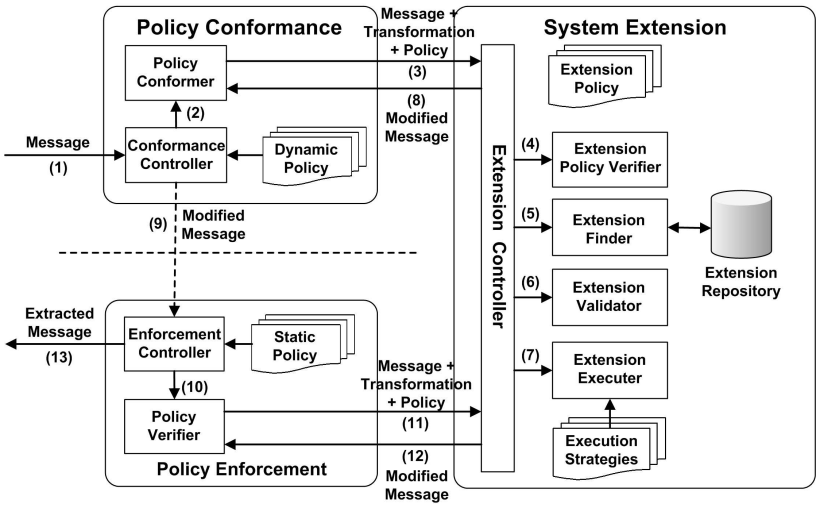

Figure 2: AMPol Conformance, Enforcement and Extensions Components

components of the sender and recipient. They are important parts of PQP.

The PQP module, after retrieving the dynamic policy, invokes the *policy conformance* module by calling the *conformance controller*, which co-ordinates all the processing steps. The controller first identifies the

transformations for property rules using the *policy conformer*, then invokes the *extension controller* and passes it the list of transformations along with the policy and a message.

The *extension controller* verifies the transformations against system extension policy. If the verification succeeds, the *extension controller* calls the *extension finder* to find the required extensions. If required, it downloads the extension using the meta-information in the transformations. Then the *extension controller* calls *extension validator* to validate the authenticity of all required extensions. Finally, the controller calls the *extension executor* to execute the extensions. If any extension fails then the whole process aborts with failure. The resultant modified message is returned to the *conformance controller* which returns it to the PQP module.

The message returned to the PQP module is sent to the recipient by the underlying application specific transport mechanism. At the recipient end, the *policy enforcement* module enforces all of the constraints on a received message. The *enforcement controller* co-ordinates all the policy enforcement tasks. It first performs the same policy compliance check as the sender node. The identified list of transformations is passed to the *enforcement controller*, which extracts the original message from the received message.

We have implemented all the AMPol modules and AMPol extension framework in C# .NET and packaged the code in DLLs. We have integrated AMPol modules to WSEmail through hooks developed for WSEmail servers and clients. To implement case studies for RTT and hashcash puzzles we have implemented two extensions for each type of puzzle by extending from the AMPol extension framework. The RTT Extension displays a dialogue box and user is challenged to write the displayed image text in a provided text box. The hashcash extension solves the hash collision problem and appends the result to the input message. In both extensions, the puzzle execution result is appended at the end of the message and the resultant message is returned back to the caller. At the recipient end the same type of extensions are again used for verifying the results. For the IBE case study we implemented the IBE extension in a similar way. The IBE extension first generates an AES symmetric key, and then encrypts the message using this key. The key is then encrypted using the IBE encryption algorithm using the recipient email address as an IBE encryption key. On the other end, the same IBE extension is used to verify and extract the original message. The IBE encrypted symmetric key is decrypted using the recipient's email address. The symmetric key is then used to decrypt the encrypted original message.

6 Related Work

There have been a number of attempts to exploit service-oriented architectures to make distributed applications more adaptive. Meta data approaches [12, 15] support the description, discovery and composition of services using languages such as XACML, WSPL, WS-Policy, and OWL-S to describe QoS requirements and policy constraints. Associated policy processing frameworks are used to enforce requirements for individual entities. Adaptivity is achieved by adding, customizing or replacing entities such as aspects [11], components [2] or concerns [9]. AMPol aims to integrate and extend these types of mechanisms to achieve an end-to-end solution that works in at least the domain of messaging systems.

Dynamic adaptation has been explored [17, 16, 7] in a service-oriented framework to deal with entities that have different QoS requirements on a per session basis. This work uses WS-Policy to describe the QoS requirements; policies are enforced by a framework that supports dynamic binding of non-functional features (such as security and performance constraints) with applications. It provides middleware to achieve cooperation and agreement of requirements between entities but does not provide concrete negotiation protocols and does not explicitly specify which system entity will enforce the policy. There is work on using policy framework and system extensibility to achieve end-to-end adaptability [4] but this work does not support negotiation of requirements and focuses more on system extensibility and policy framework. DySOA [6] is another effort to achieve an adaptive system. It provides a framework for monitoring the application

system, evaluating acquired data against the QoS requirements, and adapting the application configuration at runtime. It has a simple manual policy negotiation between the requester and the provider but does not support runtime negotiation. It does not address system extensibility and only re-configures the alternative variables for system parameters. GlueQoS [20] proposes a declarative language based on WS-Policy to specify QoS features and a policy mediation meta-protocol for exchanging and negotiating QoS features. One obvious limitation of GlueQoS is that it does not support dynamic system extensibility; it assumes that both ends have the capability to perform the required operations to fulfill QoS requirements.

AMPol differs from these studies in its focus on exploring an end-to-end solution that incorporates all of the necessary adaptive support features with the aim of customizing the solution to a specific domain (messaging). The goal is to improve validation and explore the extent to which adaptivity is constrained by domain-specific issues. We find that many important issues can be viewed as general concerns not particular to messaging systems, but the domain-specific focus leads us to appreciate the need for new features in the general solutions. For instance, all of the existing systems focus on adaptivity between pairs of nodes, whereas our messaging scenarios call for protocols that adapt the policies of four nodes in a manner that is not just a generalization of the two node protocols. While we are able to provide a treatment of this four node case (PQP), a fully general treatment is still lacking. Aside from this multi-node issue, the main distinctions of AMPol lie in its added flexibility.

Existing efforts assume a built-in logic to enforce policy constraints (QoS requirements) or have a static binding with external processing components to handle policy rules. In AMPol we have proposed an adaptive policy framework by taking out the message conformance (with policy) logic and policy enforcement logic for each individual policy rule out of the core policy engine and have provided this logic in a form of pluggable extensions. The binding logic of a policy rule with a processing plug-in is specified in a policy rule itself instead of hard coding it in a core policy engine. This enables AMPol to be more flexible in its adaptability than other approaches.

The efforts discussed above use WS-Policy framework. which is not generic and adptive enough to support new types of QoS costraints. WS-Policy language is an assertion based declarative language which is based on domain-specific policy specifications such as WS-PolicyConstraints, WS-SecurityPolicy and WS-ReliableMessagingPolicy. These policy specifications have domain specific vocabulary elements and in order to provide a support for new type of assertion (non-functiona constraint), the policy schema needs to be updated and each underlying policy engine must install a new code module to understand and process the semantics of the new assertion type. In contrast, AMPol provides a generic model for rule specifications in which a policy rule is defined as an attribute-value pair. Rules are domain independent so parties extend the specification language without modifying the policy schema and underlying policy engine. Only the policy conformance and enforcement logic at the end nodes needs to be updated.

7 Conclusion

We have introduced an architecture to support adaptive messaging, explained a detailed design, and implemented a prototype to illustrate the concepts and usefulness of the idea. Our architecture, AMPol, is based on functional components for expressing policies, discovering them, and facilitating extensions to conform to and enforce them. Our case study validates the approach and implementation by showing how to deploy puzzles and IBE over a WSEmail platform. This provides one of the most complete studies to date of a proof-of-concept adaptive system based on web services. Our future and current work includes support for multiple recipients (that is, mailing lists), improved security measures such as sandbox protection, features to facilitate error handling, more sophisticated negotiation for policies, and performance testing. Our project web site `seclab.uiuc.edu/ampol` includes a video demonstration of adaptive messaging based on the implementation described in this paper.

References

[1] M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In *Proceedings of the 8th Asian Computing Science Conference*, 2003.

[2] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. *j-LECT-NOTES-COMP-SCI*, 791, 1994.

[3] A. Back. Hash cash - a denial of service counter-measure, 1997. `http://www.hashcash.org/papers/hashcash.pdf`.

[4] F. Baligand and V. Monfort. A concrete solution for web services adaptability using policies and aspects. In *WISE03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*. IEEE Computer Society, 2004.

[5] D. Boneh and M. Franklin. Identity based ecncryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003.

[6] I. Bosloper, J. Siljee, J. Nijhuis, and D. Hammer. Creating self-adaptive service systems with dysoa. In *ECOWS05: Proceedings of the Third European Conference on Web Services*, 2005.

[7] F. Curbera and N. Mukhi. Metadata-driven middleware for web services. In *WISE03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*. IEEE Computer Society, 2003.

[8] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *Proc. CRYPTO 92*, pages 139–147. Springer-Verlag, 1992.

[9] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, Feb.24 1995. URL: `ftp://www.ccs.neu.edu/pub/people/crista/papers/separation.ps`.

[10] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In S. Kent, editor, *Proceedings of NDSS '99 (Networks and Distributed Security Systems)*, pages 151–165, 1999.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[12] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A language for interoperability in relational multi-database systems. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the twenty-second international Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 239–250, Los Altos, CA 94022, USA, 1996. Morgan Kaufmann Publishers.

[13] K. D. Lux, M. J. May, N. L. Bhattad, and C. A. Gunter. WSEmail: Secure internet messaging based on web services. In *International Conference on Web Services (ICWS '05)*, Orlando FL, July 2005. IEEE.

[14] M. McDowell and A. Householder. Cyber Security Tip ST04-010: Using Caution with Email Attachments. `http://www.us-cert.gov/cas/tips/ST04-010.html`.

[15] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[16] N. Mukhi, P. Plebanni, I. Silva-Lepe, and T. Mikalsen. Supporting policy-driven behaviors in web services: Experiences and issues. In *ICSOC04*. IEEE Computer Society, 2004.

[17] N. K. Mukhi, R. Konuru, and F. Curbera. Cooperative middleware specialization for service oriented architectures. In *WWW'04*. IEEE Computer Society, 2004.

[18] F. Rideau. Stamps vs spam: Postage as a method to eliminate unsolicited commercial email. `http://fare.tunes.org/articles/stamps_vs_spam.html`.

[19] L. von Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *Proceedings of Eurocrypt*, pages 294–311, 2003.

[20] E. Wohlstadter, S. Tai, T. Mikalsen, I.Rouvellou, and P. Devanbu. Glueqos: Middleware to sweeten quality-of-service policy interaction. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004.