AMPol: Adaptive Messaging Policy

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

Abstract

Interoperability in a large-scale distributed system is challenged by by the diversity of node policies. We introduce AMPol (Adaptive Messaging Policy)*, a service-oriented architecture that facilitates policy-aware, end-to-end adaptive messaging between multiple parties. AMPol provides services for expressing non-functional QoS 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 this objective is to support non-functional quality requirements like security, availability or reliability constraints 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 policy requirements 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 requester and provider. Such systems can be made to function on a large scale in an interoperable way *by allowing requesters to dynamically adapt to the policy requirements of others with whom they need to interact*, we call it *policy adaptation.*

In SOA, the term Quality of Service (QoS) refers to non-functional properties which affect the definition and execution of a service, while QoS policy refers to a set of constraints of the non-functional behavior of a service. In this paper, we extend this definition to include service consumers because there may be non-functional constraints from a target service which affect the execution of a client, we call this feature *two-way policy*. Furthermore, the scope of these QoS constraints may not be limited to immediate service consumers and service providers; it may involve all the intermediate entities or end parties (*e.g.* brokers, composite services, message relays or end node message recipients). We call this feature *end-to-end policy*. End-to-end policy supports adaptive messaging between the requester and the service provider including all the intermediaries. Current approaches [8, 28, 21, 9, 24, 29] lack this end to end and two-way notion of policies. Also, adaptation based on non-functional quality requirements would require system updates. SOAs based on web services with meta data specifications and adaptive middleware approaches 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 requirements and design alternatives. This paper describes one such effort in which we explore an adaptive architecture to support QoS policies for large-scale messaging systems, such as email, instant messaging, chat rooms, list servers, Wiki pages, blogs, bulletin boards, and so on. In many cases, these systems lack basic facilities for adaptation, which breaks their functionality with the introduction of new policy requirements. For instance, email messages are often discarded by mail relays for the reasons unknown to senders. In some cases, these policies are secrets of the relays or recipients, such as many anti-spam filtering techniques, while 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

*Contact: afandi@illinoisalumni.org, {jzhang24, mhafiz}@uiuc.edu, http://cs.uiuc.edu/cgunter

DNS listing, required signatures or encryption, *etc.* If these policies can be communicated to senders in an easily adaptive way, the overall messaging system can be made more secure and efficient without sacrificing convenience. This domain-specific focus leads us to appreciate the need for new features in the general solutions. For instance, in messaging scenarios we need to consider the assurance of policies at multiple nodes in a manner that is not just a generalization of the two node client-service model. Also, there is considerable benefit in our application to enabling dynamic system updates. Although the case study is domain-specific, our adaptive service-oriented architecture should be applicable to a range of other types of systems.

In this paper, we propose *Adaptive Messaging Policy (AMPol)*, a reference architecture for adaptive interoperable messaging based on advertisable QoS requirements in a form of policies. AMPol is based on the idea that the entities participating in message processing should learn and adapt to the policies of each other in an end-to-end manner. This is achieved through three fundamental architectural components:

(1) *Policy model:* describes QoS requirements of the entities' dynamic behaviors in the form of policies.
(2) *Policy discovery:* encapsulates protocols to publish, discover and merge such policies.
(3) *Extension and Enforcement (EE):* adds policy conformance capabilities and enforces policy rules on messages.

Our validation case study realizes AMPol based on WSEmail [18], an approach to 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 formatted in XML. We aim to validate AMPol by showing how email services could be facilitated by the AMPol architecture to support QoS policies in an end-to-end adaptable manner. In particular, our implementation is able to automatically support addition of new QoS constraints for availability and security by deploying plug-ins for puzzles (to raise the burden 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 integration in a large-scale messaging system.

AMPol's goal is to support dynamic and complex QoS policies and maintain the interoperability of the systems. Our solution is based on the idea of two-way policy adaption, end-to-end policy and an agile middleware to support these policies. This paper's main contributions are its end-to-end and two-way policy solution, adaptive and generic distributed policy framework, reference architecture of adaptive middleware for messaging systems, and application of this middleware for WSEmail.

The paper is divided into seven sections. Sections 2, 3, 4 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 5 describes a case study on WSEmail to validate the AMPol solution. Section 6 discusses some of the work related to adaptive messaging and policy. Section 7 concludes.

2. Policy Model

The entities involved in adaptive messaging would need to declare their policies up-front to allow others to discover the policies. The AMPol policy model provides the framework for specifying a set of rules or constraints describing the QoS requirements. The policy constructs should be distinct, modular and supportive of various rule combination logics, making the policy language expressive and unambiguous. The policy model should unambiguously define which entities the policy is applied to and which entity enforces the policy. The language for expressing the policy should be generic so that the policy scheme and core policy engine do not need to be modified when new assertions are added. The AMPol policy model is illustrated in Figure 1; the rest of section shows how it satisfies these requirements.

The basic unit of the policy is the construct Rule. Each rule describes an operation or process requirement for a message, *e.g.* encryption and signature. Each rule has two parts. One is the *action*, which specifies the operation (*e.g.* encryption). The other one is the *property*, which specifies parameters of this operation (*e.g.* IBE encryption). The Rules are combined into a RuleSet with connectives AND, OR, EXACTLY-ONE, *etc.*. One or more RuleSets form a Policy. A Policy is associated with the application level usage (using the App attribute). For example, in a messaging system, the policies can be specified as ingress, egress, or local policies covering messages entering, leaving, or within the local domain. Finally, PolicySet contains the Policy constructs.

Policies are classified as static or dynamic. Static policy is defined by each entity before the communication occurs. It defines rules that are effective for all conversations. Dynamic policy, which is based on static policy, is a set of run-time rules for a particular conversation session. If there are multiple entities involved in a session, each entity needs to know the static policies of others and then generate the dynamic policy. Section 3 describes how to obtain dynamic policies.

AMPol uses meta-specification, *i.e.*"the policies of a policy", to elaborate the policy definitions. For example, each Rule or RuleSet needs meta-information, *e.g.* who will *perform* the required operation and who will *check* whether the policy constraint is satisfied. In the AMPol policy

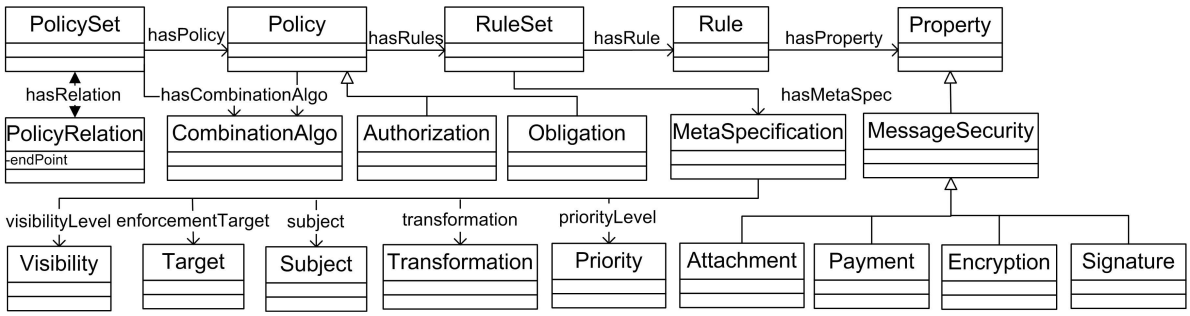

Figure 1. AMPol Policy Model

model, the rules are applicable to the Subject entity and the rules are enforced by the Target entity. In a distributed system, the creator of the rule or the policy might not be the entity who will verify or enforce the policy. So it is necessary to indicate the target explicitly. To conform or enforce the rules of a policy, the entity needs some extended functionality to perform the operation. We define Transformation for a Rule that contains all the information to identify and download a particular plug-in. Policy engines can parse the transformation information and pass it to system extension 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 and system extension module do not need to change.

In service oriented systems, clients and services have a built-in logic to enforce or conform to policy constraints. Even if these logics are independently provided, the underlying policy frameworks are not flexible enough to dynamically bind policy rules to external processing components to handle (*i.e.* parse, comply, enforce or verify) policy rules. In AMPol, we have decoupled the policy conformance and policy enforcement logic from the core policy engine and provided these logics in the form of pluggable extensions. The binding logic of a policy rule with a processing plug-in is specified in the meta-specification of a policy rule itself instead of hard coding it in a core policy engine or application logic.

All of the above meta-information is encapsulated by MetaSpecification, which also contains entities to indicate the visibility (Visibility) and the priority (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.

Domain specific policy rules can be defined by extending the AMPol policy model. Our case study on messaging systems is based on a set of rules called *APES* for *Attachment, Payment, Encryption and Signature*. The Encryption and Signature rules specify the cryptographic parameters required for encryption and signature. For availability, the Payment rule specifies the type of cost (puzzles) imposed on the message senders. The Attachment rule specifies the patterns of the attachment files, which are the primary medium for spreading viruses. These rules are used to put constraints on different entities in the messaging system to achieve security goals.

3. Policy Discovery

After specifying policies individually, the entities need to exchange them and negotiate a mutual acceptable policy set for the current session. We call this *policy discovery*. The policy discovery component consists of three functional sub-components, which provide the ability to advertise, merge, and query policies. This component is able to publish each entity's policy to everyone involved in the session and ensures the published policies are accessible remotely. Each entity is able to query the other's static policy or the final dynamic policy. Policy merging of two or more policies is supported and it has the potential to support multi-node policy discovery.

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 public subset of the static policy. Policies of a particular entity can be published at the local node (*e.g.* service itself), or at a dedicated policy server (registry or discovery service), or at any remote customized server. The main requirement is that the server is available and remotely accessible through some known protocols. From the policy consumer's perspective, the main issue is the protocol and the location to find the published policies. The protocol for finding (querying) 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. We assume policies are available somewhere on a remote server and can be queried for a particular system entity. AMPol has pluggable components for advertising and querying policies and these customized components can be plugged to the AMPol system

to configure it per application scenario.

Policies for email clients (senders or recipients) can be published at the mail server. Each mail server is responsible for providing services for uploading, updating, querying and downloading policies for a particular system entity. The entity that is publishing the policy also specifies how the policies are uploaded and maintained in a policy publishing server. This can be done in either the push or pull mode. In push mode, clients upload/update the policies to the server. In pull mode, policy server periodically (or when required) updates the policy for a particular client by re-acquiring the policy from the client.

As discussed before, the scope of policies may not be limited to immediate service providers only, it may involve intermediate entities (*e.g.* an email message may go through multiple forwarding entities). To discover and merge the policies of all the participating nodes and generate an end-to-end dynamic policy, we have added a policy dependency construct called *PolicyRelation* [2] in the AMPol policy model. When a node advertises its policies, it also provides a reference to its immediate dependent nodes. The policy relation information is used to discover policies of dependent nodes. This policy reference either directly points to another policy or provides meta information to discover the policies of dependent nodes. Discovery continues until policies of all the nodes are discovered and merged. The final policy is an end-to-end dynamic policy. Our current work on policy discovery is focused on static system topology. Our work [2] on global QoS involves exploring solutions for policy discovery for dynamic SOAs which dynamically discover and compose services.

Policy Merging is required to reconcile the policies of diverse parties in Policy Query Protocol exchanges (the protocol description is given later in this section). The propagated static policies are merged to generate the dynamic policy. The service requester (message sender) is the interested party to get the service's (message recipient) policy and generate the dynamic policy. The merging can happen at different nodes of the system that lie in the path of the requester and the service provider. For simple client-service interaction, policies are merged at client node. For two-way adaptation, the merged dynamic policy is also sent to the service along with the request message. The Policy and RuleSet constructs of the AMPol model have the CombinationAlgorithm structure, which specifies policy combination algorithms such as AND, OR, EXACTLY-ONE and so on.

PQP, the *Policy Query Protocol* is the fundamental protocol for generating dynamic policies from static policies when a message transmission is required. The initiator in this case is the message sender requester. We present PQP for 'four-node messaging' (for email systems) as illustrated in Figure 2.

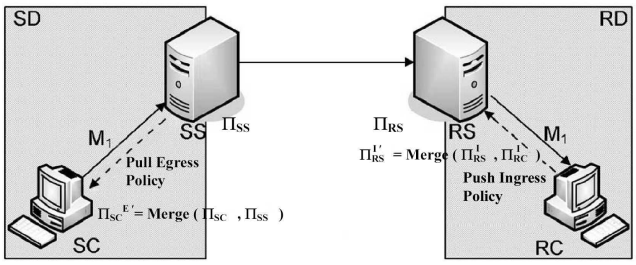

Figure 2. Policy exchange and merging

SC is the message sending client. It is connected with SS, also known as 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 $\Pi^{\mathrm{I}}_{\mathrm{RC}}$, the egress policy of sending server as $\Pi^{\mathrm{E}}_{\mathrm{SS}}$ and so on.

In the policy advertisement mechanism for four node system, the policies can be advertised at the server node. AMPol proposes either a push or pull mechanism to load ingress policies from clients to an email server which serves as a policy server. The policy server then merges its ingress policy with the client's ingress policies. This results in the merged ingress policy for the receiving server and client. It is denoted as: $\Pi^{\mathrm{I}'}_{\mathrm{RS}} = \mathit{Merge}(\Pi^{\mathrm{I}}_{\mathrm{RS}}, \Pi^{\mathrm{I}}_{\mathrm{RC}})$. These policies are stored per receiving client and can be queried by a unique identifier which can be fully qualified email address. This whole process is a part of policy advertisement. The push and pull mechanism and merging of policies in the server is done in the policy advertisement step to achieve better performance. Another advantage of this is that it does not require RC to be remotely accessible all the time.

Clients may use a pull mechanism to improve sending performance. For example, the sending client SC pulls egress policies from the sending server and merges them with its 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 during PQP. It is denoted as: $\Pi^{\mathrm{E}'}_{\mathrm{SC}} = \mathit{Merge}(\Pi^{\mathrm{E}}_{\mathrm{SC}}, \Pi^{\mathrm{E}}_{\mathrm{SS}})$, where $\Pi^{\mathrm{E}'}_{\mathrm{SC}}$ is the merged egress policy of SC and SS. The equilibrium state reached after exchange of these policies is shown in Figure 2.

The policy query phase can now be described. Step 1: Client SC wants to communicate with RC; so it queries the advertised ingress policies of its immediate node SS: SC $\rightarrow$ SS : Query for $\Pi^{\mathrm{I}}_{\mathrm{SS}}$ and $\Pi^{\mathrm{I}}_{\mathrm{RC}}$. Step 2: Server SS finds out that its ingress policy has a dependency relation with RS policy. RS is resolved from the RC (*e.g.* through DNS mapping) and SS relays the request to RS. SS $\rightarrow$ RS

: Query for Π^{I}_{RS} and Π^{I}_{RC}. Step 3: As the policies at RS are already merged with the recipient's policies, there are no further policy dependencies and the server RS sends the merged policy $\Pi^{I'}_{RS}$ to SS. RS $\rightarrow$ SS : $\Pi^{I'}_{RS}$. Step 4. Server SS sends the merged policy to SC. SS $\rightarrow$ SC : $\Pi^{I'}_{RS}$. Step 5. 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 $\Pi^{E'}_{SC}$.

For two-way adaptation (*i.e.* to enforce policies for the reply message) the above protocol can be easily extended to propagate the policies from SS to RS and RC in the first two steps or during the actual request processing. The merged ingress policy of SC and SS are sent to RS and RC ultimately so that all the nodes can comply to ingress policies of other nodes to send a reply message.

The above protocol uses distributed discovery mechanism in which PQP components at each node are configured according to the application scenario and system settings. In conventional Web services, the policies can be advertised at a UDDI or registry service. The service requester's PQP module queries the policy of immediate target service from the policy server, and if there is a policy dependency, the discovery continues until all the policies are discovered and merged into dynamic policy. AMPol's flexibility to adapt the policy discovery protocol for application specific setting helps achieving real adaptivity for any type of system (*e.g.* P2P, service oriented computing or Email system).

We have used a middleware based policy framework to implement all the components of AMPol. This middleware policy framework includes components for publishing, querying, merging, enforcing and complying to policies. It hides the implementation complexity from the core application logic and the functionality provided by the middleware can be reused by different applications. To achieve greater adaptivity we want automatic system extension mechanisms that do not require modifying baseline applications. Once the dynamic policy is negotiated, the AMPol middleware components at each individual node act autonomously to enforce or comply to policies. For one-way adaptation (client adapting to services or message recipients), we only need the AMPol middleware to be active at the client node. But for two-way policy adaptation we need AMPol middleware to be active at all the participating nodes.

The AMPol middleware can be integrated to a high level application by developing application specific hooks or interceptors. We need to identify the message (request/response) entry and exit points in an application and then use hooks to intercept these messages and only allow them to proceed further if they are successfully processed by AMPol underlying components. These hooks can be directly integrated into the source code of the application or dynamically plugged into the application if it provides a mechanism for this. In the case of aspects or pluggable hooks, we do not require source code of the application and integration will be relatively easy. Different types of applications and distributed system technologies (*e.g.* email clients and servers, web and application servers, web browsers, .Net COM+, J2EE *etc.*) provide frameworks to develop and plug interceptors, hooks or filters.

Our proof-of-concept discovery model for Web services 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 key-value pairs. Accordingly, a component is developed for client middleware to interact with the policy publisher. When the client requests a service, the AMPol middleware hooks intercept the request and initiate the PQP exchange. It retrieves policy for a particular message by calling the *policyQuery* web method of the 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.

4. Enforcement and Extension

Once mutual acceptable policies have been negotiated, the participating entities need to determine how to conform to the policies of each other or how to enforce their own policies. AMPol's enforcement and extension component ensures that the sending client conforms to the policy of the recipient, and if required it extends the client system. The recipient side enforcement component verifies that the incoming messages comply with policy requirements. The extension component should not modify the core implementation of each entity. It should be able to control the adaptation process itself in order to ensure that changes are carried out effectively. Each extension must be implemented as a separate module that can be incrementally added to, and removed from, the core application by adding or removing a rule/assertion from the policies. We show how AMPol fulfills these requirements to provide adaptability.

The enforcement and extension model has three subcomponents, namely, *policy conformance*, *policy enforcement* and *system extension* (see Figure 3. For two-way policy adaptation, these components need to be active at each participating entity nodes. Extensions are realized as third party plug-ins and these extensible components can be dynamically added or removed from the AMPol system. The novel idea here is the decoupling of the conformance and enforcement logic in independent, dynamically plug-

gable processes. The pluggable process is called *extension* whereas the mechanism to locate, load, and execute these extensions is called *system extension*. The interpretation of policy on the service requester side is the execution of a series of extensions on a message to conform to the dynamic policy. For policy enforcement at the service end, the service enforces the policy by executing the corresponding policy verification functionality on the request message. Similarly, the service conforms the response message to the policies of a requestor and accordingly the requester verifies it. The policy conformance and the policy enforcement mechanisms follow classic pipes and filters architectural style, with the operations for enforcement applied in reverse order of operations in the conformance component.

Every participating entity must be able to comply to requested policy constraints to fulfill QoS policy requirements of other entities: we call this *policy conformance logic*. Here we want to distinguish between two types of conformance logics, pluggable and non-pluggable. Pluggable policy conformance logic can be supported independently without any significant change to the core application or policy framework, *e.g.* an encryption algorithm, executing puzzles, *etc.* Non-pluggable policy conformance logic cannot be supported by adding an external capability, *e.g.* processing time or network bandwidth. Generally qualitative policy constraints (e.g. security, privacy) are likely to be pluggable more often than quantitative ones (*e.g.* measurable QoS constraints). So a policy conformance concept can only be applied to pluggable policy logic. Policy conformance capabilities may be pluggable through extensions, while logic for policy verification and enforcement for both qualitative and quantitative features are easily pluggable.

Each AMPol extension modifies the original request message to comply with a particular policy constraint. An extension for a puzzle will generate an output message that will contain an original message appended with a puzzle result. Similarly an extension for encryption will generate a modified message which contains the encrypted original message. 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.

Our design and implementation of *AMPol extension framework* has been inspired by the WSEmail plug-in framework [18]. It is a white-box framework and is extended by inheritance. Figure 3 shows the steps followed by the conformance, extension and enforcement components of the sender and recipient.

The PQP module, after retrieving the dynamic policy, invokes the *policy conformance* module by calling the *conformance controller*, which coordinates all the processing steps. The controller first identifies the transformations for property rules using the *policy conformer*, then invokes the

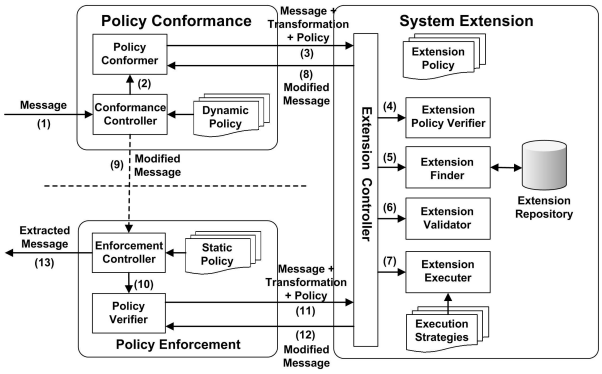

Figure 3. EE Components

extension controller and passes it the list of transformations along with the policy and the 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 target service or 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* coordinates 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. The same process is repeated for a reply message, in which the recipient executes conformance extensions to comply to the sender's dynamic policies.

The AMPol middleware suspends the initial functional request until the dynamic policy is created. This per-invocation multi-step approach does have a considerable performance overhead. Caching the dynamic policy is a simple solution to improve performance. A policy could have a lifetime, and, after it expires, the policy is renegotiated. However, it is not easy to set a suitable expiration time to achieve significant performance improvement. Renegotiation of policies requires a framework for propagating policy changes to the interested entities, and it is not feasible for highly dynamic policy constraints (*e.g.* conventional QoS constraints for availability and performance). We have addressed some of these issues in AMPol-Q [2].

We have implemented all the AMPol middleware components in C# .NET and packaged the code in DLLs. To develop pluggable extensions we have also provided AMPol extension framework which is packaged in a separate DLL.

5. Validation and Case Study

In current email systems, 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 attachment must be less than 500KB in size and must be of certain extension types, and the messages from certain parties, like banks and mutual fund companies, must be encrypted using IBE. Such policies could be quite helpful in improving security, resisting 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 the potential correspondents know what protocol and policies you would like them to respect when they are sending you a message. Our case study shows how an AMPol solution can support the use and deployment of complex policies without requiring universal adoption and changes to the baseline system.

We have created a prototype of AMPol with Puzzles and IBE for a WSEmail messaging system. WSEmail [18] 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 a service-oriented Mail User Agent (MUA) client and Mail Transfer Agent (MTA) server which support extensible messaging with plug-ins that work for both the MUAs and the MTAs. One of our key objects of this case study is to supply the functional components of adaptive messaging with very few extensions of the WSEmail platform.

Puzzles [13] are a mechanism to prevent DoS attack; in particular, puzzle-based anti-spam email systems have been studied for many years [10]. There are two general types of puzzles. One type is the *cycle exhaustion* puzzle such as *hashcash* [11]. 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. *Identity Based Encryption (IBE)* [7] 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 expected. 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. AMPol provides mechanisms to address all of these issues.

Our case study is based on the APES policy rules. We have used the Encryption rule to specify the cryptographic parameters of IBE. Payment rules are used to specify the RTT or hashcash type of payment imposed on the message sender. Another important security concern is the attachment, which is the primary medium for spreading viruses among email client hosts [19]. The Attachment rule is used to specify the patterns of the content, for example the rule 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; proper security would prevent delivery or .pif processing of the attachment.)

There are four entities involved in the system, the sending client (Sender Mail User Agent, SMUA), the sending server (Sender Mail Transfer Agent, SMTA), the receiving server (Recipient Mail Transfer Agent, RMTA) and the receiving client (Recipient Mail User Agent, RMUA). MTAs advertise their clients and their own policies, which are merged with client policies for simplicity. MTA policies also contain information for policies of dependant entities (Relays or RMTAs). In this scenario, we do not consider intermediate relays; we send a message to one recipient and it is transferred through an SMTA and an RMTA to the recipient (SMUA $\Longrightarrow$ SMTA $\Longrightarrow$ RMTA $\Longrightarrow$ RMUA). Also, there are trusted third-party plug-in servers to host the extensions. For the current setup, we show how an SMUA can automatically adapt to the target policy constraints of services (SMTA, RMTA and RMUA). Figure 4 shows the

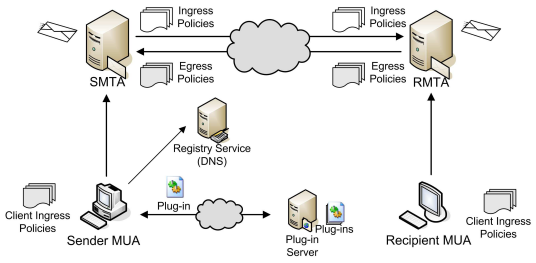

Figure 4. WSEmail Case Study: Architecture

high level system configuration of the case study.

The AMPol middleware at the SMUA starts the policy query protocol by intercepting messages at a *send* request. After determining the dynamic policy, the message sent by the SMUA is verified against the policy and, accordingly, conformance extensions are downloaded and executed to conform the message with required policy constraints. At SMTA, the received message is first verified and then processed by SMTA if the verification succeeds. When the message is relayed to the RMTA, it is again verified and

forwarded to the RMUA. The QoS policies are verified, adhered or enforced on point-to-point basis but eventually they all comply with end-to-end policy constraints and requirements. Pluggable extensions for policy discovery, conformance and enforcement are automatically downloaded from a trusted third party plug-in server.

Our case study uses IBE package from Voltage for IBE [26], an RTT package based on CAPTCHA [27], and a cycle exhaustion puzzle system based on hashcash [4]. All packages are wrapped as COM components. However, the aim of AMPol is to facilitate the deployment of any technique that is 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, 23].

We have integrated AMPol modules into the WSEmail implementation of [18] through hooks developed for WSEmail servers and clients. 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 are directly integrated into a source code of the client MUA application and plugged into application server at the service end as a filter. To implement RTT and hashcash puzzles we have implemented two extensions for each type of puzzle by adding to the AMPol extension framework. Each extension application modifies or appends something to the input message. An extension for an RTT puzzle generates an output message that contains an original message appended with a puzzle result. Similarly an extension for IBE generates a modified message which contains the encrypted original message appended with an encrypted symmetric key. Message formats are associated with policy rules through the Transformation part of the meta-specification.

The RTT Extension displays a dialog box and user is challenged to write the displayed image text in a text box; the correct user input is appended to the input message. The hashcash extension solves the hash collision problem and appends the result to the input message. 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 type of 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.

Timing delays might be an issue for AMPol. In our prototype, the delay could come from Web service overhead (compared to conventional email protocols), the process of publishing, fetching, finding, merging, adapting of the policies. In our previous work [18] on WSEmail, we carried out a performance evaluation of WSEmail through a series of experiments. These showed that a single WSEmail server should be able to routinely handle the normal load at some published benchmarks. This means our implementation is based on a plausibly efficient platform. The delay caused by policy operations can be mitigated by cache. The SMUA can fetch and cache the SMTA's static policies in advance. The SMTA or RMTA can cache static policies for other MTAs after the first interaction. Even dynamic polices can be cached. A mechanism to flush a "dirty" cache is needed for this solution. As for policy enforcement and system extension, the SMUA needs to download and install third party plug-ins only the first time it needs them. The flexibility of the AMPol middleware and the consequent timing delay is a trade-off that should be tolerable for many applications.

6 Related Work

There have been a number of attempts to exploit service-oriented architectures to make distributed applications more adaptive and policy driven. Meta data approaches [16, 20] support the description, discovery and composition of services. These approaches use languages like KAOS [25], REI [14], 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 components, aspects [15], or concerns [12]. 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.

Requirement-driven dynamic adaptation has been explored [22, 21, 9] in a service-oriented framework to deal with entities that have different QoS requirements on a per session basis. Their work uses WS-Policy to describe the QoS requirements; policies are enforced by a framework that supports dynamic binding of non-functional quality requirements with applications. The authors provides a middleware to achieve cooperation and agreement of requirements between entities. But their system does not have 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 [5]. But this work does not support negotiation of requirements and focuses more on system extensibility and policy framework. DySOA [8] 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 adapt-

ing the application configuration at runtime. Its manual policy negotiation mechanism is simple, but it does not have support for runtime negotiation. Moreover, DySOA does not address system extensibility and only re-configures the alternative variables for system parameters. GlueQoS [28] 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 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 supports all the necessary adaptive features for supporting policy-aware messaging. Our goal was to explore the extent to which adaptivity is constrained by domain-specific issues. We have found that many important issues can be viewed as general concern not particular to messaging systems, but the domain-specific focus has led us to appreciate the need for new features in the general solution. For instance, all of the existing systems focus on adaptivity in a two-node system. 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 protocol. We have provided a specific treatment of this four node case (PQP); our future work involves a fully general and efficient treatment. Aside from this multi-node issue, the main distinctions of AMPol lie in its added flexibility.

Existing efforts [8, 28, 25, 14], XACML, WS-Policy 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, 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 (or application). 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 adaptive enough to support new types of QoS constraints. 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-functional constraints), 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.

XACML fulfills most requirements of the AMPol policy model, but we have used our own light weight policy languages because XACML does not support the AMPol constructs for transformation, visibility, policy relations and others. In a related work [6] on messaging systems we explored using XACML for modeling policies for email systems. In this work policies are used for controlling access to messages and XACML is an ideal choice for this.

In other work [17], we explored an approach to merging general web service policies in a way that accounts for priorities and conflicts through the use of "defeasible policy merging". This work could be used to extend AMPol so it provides more sophisticated policy merging than we have used in our current system.

AMPol-Q [2] extends AMPol to provide a comprehensive solution to support and monitor global QoS for dynamic composite services. In AMPol-Q we proposed a semantic policy model similar to The AMPol policy model and implemented it by extending OWL-S and SWRL. The AMPol-Q middleware enables clients to dynamically discover, select, compose, and monitor services that fulfill end-to-end QoS constraints. Work on AMPol is based on systems with static binding and have a domain specific focus while AMPol-Q has a more generic treatment and application.

7 Conclusion

We have introduced an architecture to support QoS policies for SOAs, 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 to support complex policies. This provides one of the most complete studies to date of a proof-of-concept adaptive policy system based on Web services. Our future and current work includes support for multiple recipients (*e.g.* mailing lists), improved security measures such as sandbox protection, features to facilitate dynamic service composition, more sophisticated and semantically rich models for representing, discovering and negotiating policies, policy conflict resolution and performance testing. Our project web site [3] includes a video demonstration of adaptive messaging based on the implementation described in this paper.

Acknowledgements

We are grateful for help and encouragement we received from Anne Anderson, Noam Artz, Mike Berry, Jodie Boyer, Rakesh Bobba, Omid Fatemieh, Fariba Khan, Himanshu Khurana, Steve Lumetta, Adam Lee, Kevin D. Lux, Michael J. May, Anoop Singhal, Kaijun Tan. This research was partially supported by NSF CCR02-08996, CNS05-5170, CNS05-09268, CNS05-24695 and ONR N00014-04-1-0562, N00014-02-1-0715.

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] R. Afandi, J. Zhang, and C. A. Gunter. AMPol-Q: Adaptive middleware policy to support QoS. In *International Conference on Service Oriented Computing (ICSOC '06)*, Chicago, IL, December 2006.

[3] Adaptive Messaging Policy (AMPol). `http://seclab.cs.uiuc.edu/ampol/`.

[4] A. Back. Hash cash - A Denial of Service counter-measure, 1997. `www.hashcash.org/papers/hashcash.pdf`.

[5] F. Baligand and V. Monfort. A concrete solution for Web services adaptability using policies and aspects. In *WISE'03: Proc. of 4th Int. Conf. on Web Information Systems Engineering*.

[6] R. Bobba, O. Fatemieh, F. Khan, C. A. Gunter, and H. Khurana. Using attribute-based access control to enable attribute-based messaging. In *Annual Computer Security Applications Conference (ACSAC '06)*, Miami Beach, FL, December 2006. Applied Computer Security Associates.

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

[8] I. Bosloper, J. Siljee, J. Nijhuis, and D. Hammer. Creating self-adaptive service systems with DySOA. In *ECOWS'05, Proc. of the 3rd European Conf. on Web Services*.

[9] F. Curbera and N. Mukhi. Metadata-driven middleware for Web services. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*. IEEE Computer Society, 2003.

[10] 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.

[11] Hashcash.org. `http://www.hashcash.org/`.

[12] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, 1995.

[13] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *NDSS99: Networks and Distributed Security Systems*, 1999.

[14] L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin, and K. Sycara. Authorization and privacy for Semantic Web services. In *AAAI'04: Workshop on Semantic Web Services*.

[15] 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.

[16] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - A language for interoperability in relational multi-database systems. In *Proc. of the 22nd Int. Conf. on Very Large Data Bases*, 1996.

[17] A. J. Lee, J. P. Boyer, L. E. Olson, and C. A. Gunter. Defeasible security policy composition for web services. In *Formal Methods in Software Engineering (FMSE '06)*, Alexandria, VA, November 2006. ACM.

[18] K. D. Lux, M. J. May, N. L. Bhattad, and C. A. Gunter. WSEmail: Secure Internet messaging based on Web services. In *Int. Conf. on Web Services (ICWS '05)*. IEEE, July 2005.

[19] M. McDowell and A. Householder. Cyber Security Tip ST04-010: Using caution with email attachments. `www.us-cert.gov/cas/tips/ST04-010.html`.

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

[21] N. Mukhi, P. Plebanni, I. Silva-Lepe, and T. Mikalsen. Supporting policy-driven behaviors in Web services: Experiences and issues. In *ICSOC '04*. IEEE Computer Society.

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

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

[24] R. Shuping. A model for Web service discovery with QoS. In *ACM SIGecom '03*.

[25] A. Uszok, J. M. Bradshaw, R. Jeffers, M. Johnson, A. Tate, J. Dalton, and S. Aitken. KAoS policy management for Semantic Web Services. In *IIS'04: IEEE Intelligent Systems*, 2004.

[26] Voltage Identity Based Encryption. `http://www.voltage.com/ibe_dev/`.

[27] 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.

[28] E. Wohlstadter, S. Tai, T. Mikalsen, I.Rouvellou, and P. Devanbu. GlueQoS: Middleware to sweeten Quality-of-Service policy interaction. In *ICSE '04: Proc. of the 26th Int. Conf. on Software Engineering*.

[29] L. Zeng, M. Benatallah, B.and Dumas, J. Kalagnanam, and Q. Sheng. Quality driven Web service composition. In *WWW'03:Proc. of 12th Int. World Wide Web Conf.*