

# Policy-based Network Service Collaboration

Tomohiro IGAKURA Toshio TONOUCHI

Networking Laboratories, NEC Corporation

1-1, Miyazaki 4-Chome, Miyamae-ku, Kawasaki, Kanagawa 216-8555, Japan

Tel: +81 44 856 2314 Fax: +81 44 856 2229

E-mail: {igakura, tonocuhi}@ccm.cl.nec.co.jp

## **Abstract**

Web service becomes a promising technology to achieve a service collaboration over networks. However, it is difficult for users to develop service collaboration programs using the current version of Web service, because customers must take account of many diverse aspects of currently available service components. In order to cope with this difficulties, we propose an automatic network service collaboration scheme by introducing a policy server. In proposed scheme, a request message issued by users includes properties representing their requirements, and each service component registers request admission control policies and subsequent request generation policies with the policy engine. By iterating a matching procedure between policies and requested properties, the policy server is able to decide an appropriate set of service components and their execution sequence automatically. As a result of this, users easily have various benefits through creating network services customized for their own. We have validated the effectiveness of proposed scheme through prototyping both the policy engine and its applications.

## **Keywords**

Policy, Service collaboration

# Introduction

- ◆ Service Collaboration is getting common as Web-service technology is getting wide-spread.
- ◆ Users should , instead of Service Providers, implement Collaborations satisfying their ad-hoc requirements.
- ◆ For an user to realize a service collaboration is, however, difficult because
  - there are several services satisfying the user's requirements are considered. and
  - the user is necessary to know a lot of information of service components.
- ◆ Automatic implementation of service collaboration is, therefore, desirable.



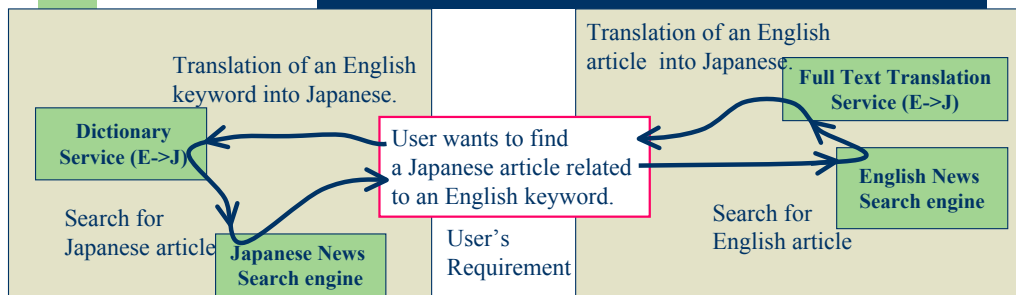
- ◆ We propose in this paper the policy-based approach for automatic implementation of service collaboration.

Web-based technologies have become very popular; clients can easily use services provided by servers distributed worldwide. Web service technology, which extend web-based technology, make each service possible to be available as a service component and to collaborate with other components. This collaboration can provide users with various kinds of services exceeding the sophistication of individual service components.

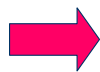
One hand, service providers can easily implement collaboration programs among these service components through these web-service technologies only if they know detail information about available service components and know a specific algorithm satisfying users requirements. On the other hand, if users would like to satisfy their individual requirement, they should implement collaboration programs instead of service providers, because the service providers cannot cope with each user's individual request. We note that it is difficult for users to develop service collaborations, because there may exist various combinations of available service components that can satisfy their requirements. Furthermore, in case of implementing collaboration programs, although the users need to know detail specifications about related service components, it takes much effort on collecting appropriate information located in distributed way.

As described in the above, we insist that an automation of implementing service collaborations is strongly desirable from users. Because of this, we propose an automatic service collaboration technique: policy-based service collaboration in this paper.

# Service Collaboration



- ◆ To implement a service collaboration for satisfying user's requirement,
  - the requirement must be divided into sub-requirements that are satisfied by service components,
  - The same requirements can be divided into different set of sub-requirements, and many parameters influence which set to be selected.
- ◆ Since the user, therefore, must take a lot of factors into account in order to describe how services collaborate one another, it is difficult for the user.



Technique for automatically deciding how services collaborate one another is required.

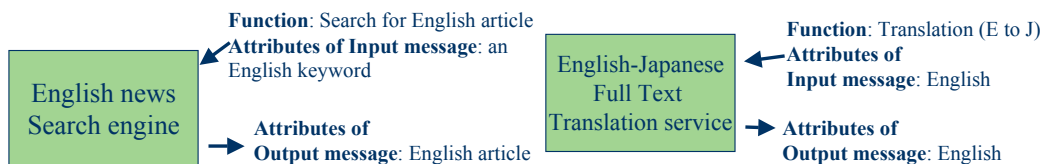
To achieve an automatic service collaboration, a user requirement should be translated into multiple service functions each of which can be supported by one of available service components. Consider the case that the user would like to read a Japanese article related to given English keywords. This requirements may be satisfied by two functions: an English news search engine is firstly used and then an English-Japanese full text translation service is used afterward. However, if the full-text translation service component does not exist, the user can not use this service collaboration. In such case, the user have to find alternate service collaboration ways such that: an English-Japanese dictionary service is firstly used and a Japanese news search engine is used afterward.

This example shows that the automatic service collaboration depends on not only user's requirements but also a set of existing service components. In addition to this, there may be alternatives for selecting a service components: some of the alternatives provide high quality translation with slow response, others are low quality translation with fast response. Moreover, there are many parameters that may influence the selection of service collaborations like a load of servers providing service components, a version of application programs, a network traffic/fault condition, and so on.

As discussed in the above, it is difficult for the users to directly specify appropriate service components that satisfy the user requirements, and a technique for automatically deciding how service components collaborate one another is required.

# Service Collaboration Policy

- Our approach: policy for describing how services collaborate one another.
  - A policy has two parts:
    - a pair of the function which a service component provides and the attributes of an input message
    - the rewriting rule of properties of an output message
    - A policy coordinates the service with another service component, considering both attributes of the input message and the output message.
  - A policy automatically decides how service components collaborate one another.



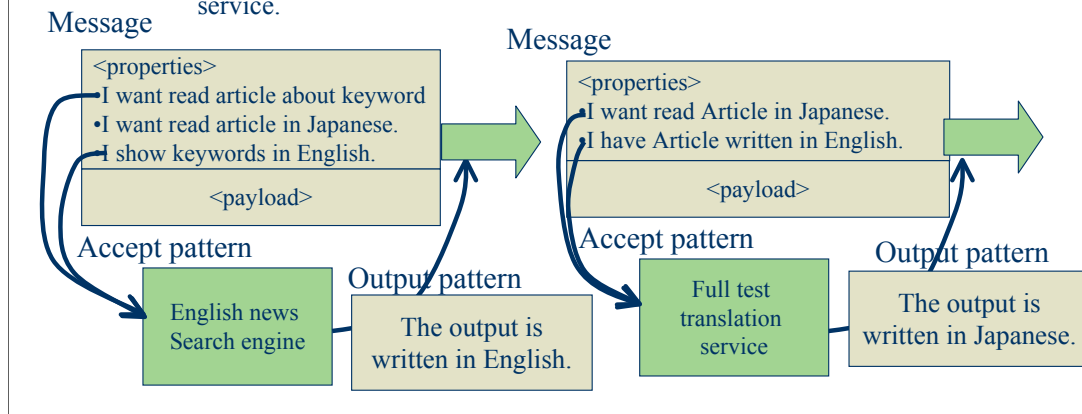
We propose a policy-based service collaboration technique for selecting appropriate service components and managing their execution order. In this technique, a message issued by a user is assumed to contain headers including user requirement properties.

A policy is given for each service component and the policy has two parts. The first part reflects what kind of function the service component provides and what kind of properties each service component requires. In short, this part specifies which kind of messages each service component accepts: A service component only accepts messages which require the function provided by the service component and messages whose parameters satisfy service component's requirements for the properties in the message.

Another part is the modification rule of the output messages replied by each service component: The properties of The reply message from the service component to which the pointer-to-service part refers to is modified according to the output part. The part is used for selecting a successor of the service component because the successor can be decided, based on this part, whether it can use the output message as an input message.

# Behavior of Policy Engine

- ◆ Policy Engine stores a set of policies, and it decides which policy is executed when it receives a message.
- ◆ A policy consists of an accept pattern and an output pattern.
  - An accept pattern specifies which message a policy accepts. The policy is executed when its accept pattern matches with properties in a message.
  - An output pattern specifies how a message is virtually forwarded to the other policies. The executed policy virtually sends a message to another service.



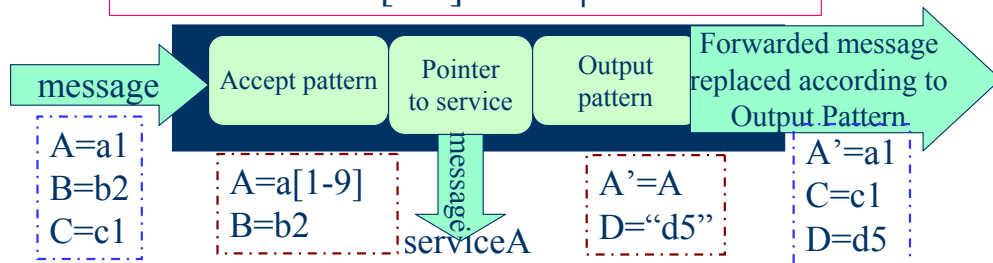
This page shows the behavior of the policy engine we have implemented. The engine stores a set of policies. A policy is registered for each service component. The policy engine receives messages sent by the user, and it decides which service component should be executed, considering the accept patterns of the policies. It forwards reply messages from the selected service component to other service components whose policy matches with the reply messages.

We will explain the behavior of the policy engine by using an example. We assume that the policy of the English news search engine specifies that the user requires a search function, that its input message contains an English keyword, and that its output message is written in English. In contrast, the policy of the English-Japanese full text translation service specifies that the input message contains English contents, and the output message contains Japanese. Considering that the user sends a message whose properties describes request, which is to search an article with an English keyword, and that he wants to read new articles in Japanese, the policy for the search engine is first matched with the request message. The reply message of the search engine is forwarded to the translation server, and, then, the user gets a Japanese article, as a result.

# Syntax of Policy Language

- ♦ A header of a message is assumed to have a list of properties, each of which is as “*label = data*”.
- ♦ A policy consists of three parts: an accept pattern, a pointer to a service, and an output pattern.
  - The syntax of a policy is “*service:accept\_pattern | output\_pattern*”.
  - The accept pattern specifies the properties of a message to be accepted.
  - The message is sent to the service which the pointer to the service specifies.
  - The output pattern specifies properties which is added to the message to be forwarded and properties which is modified in the message.

*serviceA : A=a[1-9] B=b2 | A'=A D="d5"*



In this page, the syntax of our policy language is mentioned. We assume that a header of a message has a list of properties. The syntax of each property is like “*label = data*”.

A policy consists of three parts: an accept pattern, a pointer to a service, and an output pattern. The syntax of a policy is “*service:accept\_pattern | output\_pattern*”. The **accept pattern** specifies the properties of a message to be accepted. The syntax of the pattern is a list of property patterns “*label = pattern*”. This matches with the property whose name is *label* and whose value is matched with the *pattern*, which is expressed in a regular expression. An access pattern matches with a message if and only if every property pattern in the access pattern matches with a property of the message.

The **pointer to a service** specifies a service component to which the matched message is forwarded when the access pattern of the policy is matched.

The **output pattern** is a replacement rule of a reply message from the service component that the message has been forwarded to. Some properties of the reply message are replaced according to the output pattern, and the replaced message is forwarded to the next service component. You can use the value in the reply message as being the same values of properties of the forwarded message as well as you can specify directly a new value for a given property. For example, Output pattern “*A'=A*” means that a value of Property A in the forwarded message is a value of Property A of the reply message. The properties of the reply message are removed from the forwarded message if the output pattern explicitly specifies these properties. In short, the properties of the forwarded message must be specified in the output pattern of the matched policy.

# Behaviors of Policy Language

## ◆ Behaviors

- When there are two or more acceptable policies to a message, one of them is chosen in a non-deterministic manner.
- One policy is inserted only once in the loop with the output pattern.

## ◆ Supplementations

- NULL Service
  - There can be policies that don't have services.
- No Output Pattern
  - The policies that have no output pattern don't output messages.

In this page, we explain the behavior model of the policy language.

When a user sends a message, a policy with accept pattern that matches the message is selected. If there are two or more selectable policies to a message, one of them is chosen in a non-deterministic manner. We decided to choose non-deterministic mechanism because we considered that the speed of searching for a policy was important: the policy engine can choose any policy considering its performance.

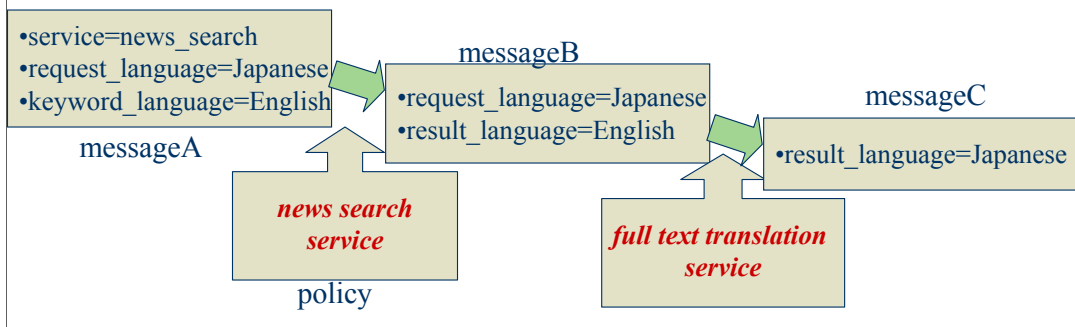
Policies generate new messages after the execution of service components. A policy with accept patterns appropriate for the new output message will be searched again and applied if it hasn't already been applied. We call this sequence a transaction. In our policy engine, a policy should not be selected more than once in a transaction because we thought that it would be important to guarantee that this transaction in the output patterns surely stops.

Our policy language have some extra features. One of them is a NULL service component. You can specify "NULL" in the pointer-to-service part of a policy. If the policies with the NULL service are selected, there output patterns generate new messages without actions. This feature is convenient when the input message are modified in accordance with the output pattern and the modified message is used for invoking another policy.

Moreover, policies without output patterns can also be described. The policies without output patterns don't output messages. If the policies without output patterns are selected, results of the service component of the policy are returned to users.

# Policy Example #1

- ◆ user's request
  - service=news\_search, request\_language=Japanese, keyword\_language=English
- ◆ policies
  - **news search service** : service=news\_search, keyword\_language=English | result\_language=English
  - **full text translation service** : request\_language=Japanese, result\_language=English | result\_language=Japanese



In this page, we show an example of describing policies.

Consider that a user requests “an article relevant to English keywords and written in Japanese.” In this case, the user’s message has the properties “service=news\_search”, “request\_language=Japanese”, “keyword\_language=English” (*messageA*) and the message has the keywords in payload. We assume that there are two service components “news search service” and “full text translation service”

The “news search service” registers a policy “news search service : service=news\_search, keyword\_language=English | result\_language=English.” into the policy engine, and the “full text translation service” registers a policy “request\_language=Japanese, result\_language=English | result\_language=Japanese.”

The message sent by the user first matches the policy of the “news search service.” Then, the “news search service” searches for news relevant to the keyword specified by the message contents. After the service ends, a new message is generated by the output pattern of the policy. In this message, the two properties “service=news\_search” and “keyword\_language=English” are deleted from the original message, and a property “result\_language=English” is added.

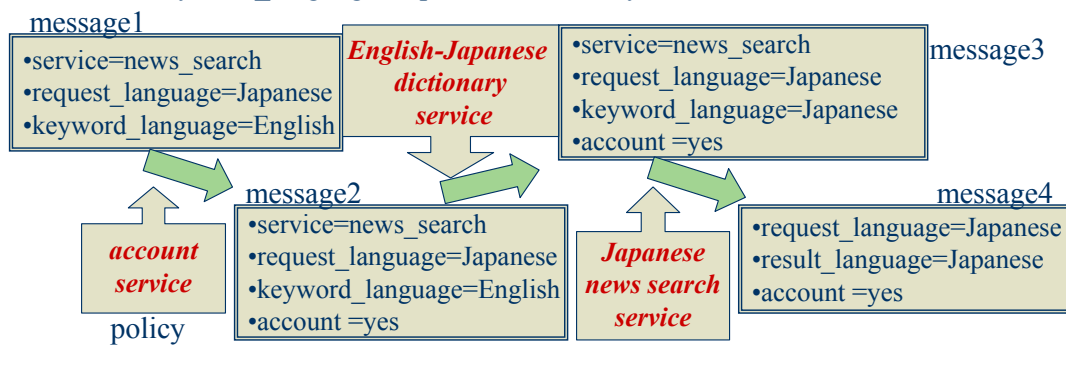
As a result, the message has two properties “request\_language=Japanese” and “result\_language=English” (*messageB*).

The “full text translation service” policy is applied to the new message, and the message is translated into Japanese. The newly generated message is given the property of “result\_language=Japanese.”

No policy accepts this message and the result of the “full text translation service” is returned to the user.

## Policy Example #2

- **account service**: | account=yes
- **Japanese news search service**: service=news\_search, keyword\_language=japanese, account=yes | result\_language=Japanese, account=yes
- **English-Japanese dictionary service**: request\_language=Japanese, keyword\_language=English, account=yes | keyword\_language=Japanese, account=yes



As a second example, we introduce a system that has an "account service", a "Japanese news search service" and a "English-Japanese dictionary service".

The user sends a message exactly the same with in example #1, that is, "an article relevant to English keywords and written in Japanese."

The policy of the "Account service" is "| account=yes." Because the policy has an empty accept pattern, the policy accepts any messages.

The policy of the "Japanese news search service" is "service=news\_search, keyword\_language=Japanese, account=yes | result\_language=Japanese, account=yes." And the policy of the "English-Japanese dictionary service" is "request\_language=Japanese, keyword\_language=English, account=yes | keyword\_language=Japanese, account=yes."

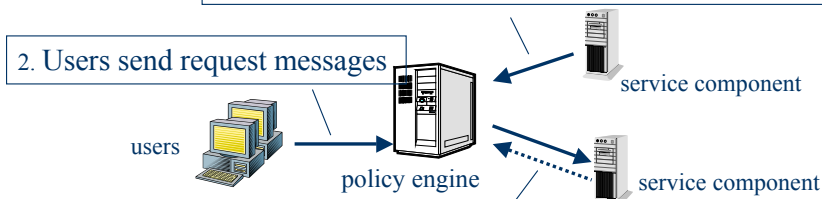
The accept patterns of the policies of the "Japanese news search service" and the "English-Japanese dictionary service" require the property "account=yes". That is, these two policies only accept output messages of the "account service." And the output messages generated by policies the "Japanese news search service" and the "keyword translation service" have a property of "account=yes" because this property is also in output pattern of the policies.

The message is processed as follows. First, user's message is processed by the "account service" and a new message *message2* is generated. After the "keyword translation service" accepts message2, *message3* is generated. And after the "Japanese new search service" accepts message3, message4 is generated.

The "account service" policy can accept message2, message3 and message4, but because of the behavior rule that "a policy should not be selected more than once in a transaction", the policy isn't selected again.

# System Architecture

1. Service components register the policies in the distributor



3. The policy engine forwards the messages to service components.

- ◆ The users and the service components communicate with one another through the policy engine.
- ◆ The distributor manages the policies.
  1. Service components register the policies in the distributor when they start.
  2. Users send request messages to the policy engine.
  3. The policy engine forwards the messages to service components whose policies match the messages.

The system architecture consists of a policy engine that manages policies, service components that provide application functions, and users that use service components. The service components register each policy in the policy engine when the service components start to provide the application functions.

The users first send request messages to the policy engine. The policy engine searches policies whose accept patterns match the messages, and then forwards the messages to the service components of the policies. The service components send the policy engine replies to the messages.

Next, the policy engine generates new messages according to the output patterns of the policies and then, it searches for policies that match the new messages.

The characteristics of this architecture is that all the message go through the policy engine. Although the policy engine in this architecture flexibly controls the destination of all the message, the policy engine may suffer from high load. The performance issue is future work.

## Related Work

### ◆ Web Service

- SOAP: Standardization of communication protocol
- UDDI: Searching Service function
- BPEL4WS: Description Language for Service Collaborations.

Problem : BPEL4WS can describe fixed collaborations only.

### ◆ Who plans service collaborations?

- Users: It is difficult to the users to get service information that is required to plan service collaboration.

- Providers : It is impossible to plan collaborations for all Answerable requests.



**Answerable policy based service collaboration**

A series of technologies called Web services that make service collaborations has been proposed. The web service technologies contain SOAP, UDDI, BPEL4WS. SOAP[1] is a communication protocol that standardizes communications between service components. UDDI[2] finds appropriate service components, and BPEL 4 WS is a description language for the collaborations. With BPEL4WS[3], the planner must adjust the collaboration sequence to suit the user's requests. The BPEL4WS is assumed that the flow among service components are programmed, but it is difficult to implement the flow satisfying the users' ad-hoc requirements which is changeable time-by-time. In addition, available services in the network is also changeable.

In light of these problems, we believe that a policy based service collaboration that automatically generates service collaboration sequences is needed.

# Conclusion

- ◆ Requirement: flexible construction of service collaborations based on users' requests.



## Policy based service collaboration

- Policies reflect provided functions, required parameters and output formats of service components.

- ◆ Collaborations Policies

- accept patterns: specify acceptable messages
- output patterns: describe output messages



## Successively execute the service that provides functions required by the messages

- ◆ Future work

- Mechanism for trying another collaboration sequence, such as a backtrack mechanism
- High performance of Policy Engine

In this paper, we described policy based service collaborations: a method to generate collaboration sequences to suit user's requirements. In this method, each service component has a policy. The policies reflect information such as functions provided and parameters required by the service component.

Users send messages that contain properties that reflect their requirements such as required functions and languages. The policies have accept patterns and output patterns. The accept patterns specify the functions provided by service components and requirements to execute service components. The output patterns describe information of the service results, such as the language and format. Users' messages are accepted by policies with accept patterns that match the messages. The policies' output patterns generate new messages, which are to be accepted by other policies. Processing continues like this until all requirements written in the user's messages are satisfied.

This paper is concluded with Future Work. In the current architecture, if an error occurs, an error message is returned to the user. However, collaboration sequences to satisfy user requirements may not be unique. Another collaboration sequence may satisfy the requirement. In short, the mechanism for searching other collaboration sequences, such as back-track mechanism is required.

The other future issue is a high performance policy engine. In our architecture, all messages go through the policy engine, and, therefore, the fast policy engine is inevitably necessary. The mechanism of our policy engine is very simple, such as indeterministic behavior, and the engine can afford to achieve fast execution.

## References

- [1] D. Box et al, Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>
- [2] UDDI(UDDI Ver 3.0), 19 July 2002, [http://uddi.org/pubs/uddi\\_v3.htm](http://uddi.org/pubs/uddi_v3.htm)
- [3] Business Process Execution Language for Web Services (BPEL4WS) version 1.1, May 2003, BEA Systems, <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>