Axis2, Middleware for Next Generation Web Services

Srinath Perera[1], Chathura Herath[1], Jaliya Ekanayake[1], Eran Chinthaka[2], Ajith Ranabahu[2], Deepal Jayasinghe[2], Sanjiva Weerawarana[2],Glen Daniels[2]
1. Indiana University, Bloomington, USA 2. Apache Software Foundation, USA
{hperera, cherath, jekenaya,}@cs.indiana.edu, {chinthaka, ajith, deepal, sanjiva, gdaniels} @apache.org

Abstract

Axis2, the next generation of Apache Web Services middleware, is an effort to re-architecture Apache Web Service stack to incorporate the changes in Web Services. Among many improvements, Axis2 provides first class Messaging and SOAP extension supports together with a novel lightweight streaming based XML processing Model. The architecture is build on top of a simple and extensible core that provides the basic abstractions for the rest of the system. We present the design and the thought process behind the key abstractions by breaking down the architecture in to three topics, XML Processing Model, Extensible SOAP processing model and Messaging Framework. This Paper explains the overall architecture while concentrating on the three topics, and demonstrate how they all fit together to yield Axis2.

1. Introduction

Axis2 is an effort to redesign Apache Web Services stack in order to address the major changes took place in Web Services stack since Axis1. Axis2 amalgamates experience from two generations of Web Services middleware, Apache SOAP and Axis, nevertheless redesigned from scratch to suit the next generation of Web Services.

Web Services are no longer just a promise for the future; rather it is a reality now. While being used in the real world, Web Services has evolved to suit the requirement of that world. This touch with the reality has redefined the values of Web Services, while deepening our understanding about the realm. Axis2 is motivated by such changes and knowledge gained as a result. Following are few of the influential factors.

1. Adaptation of messaging semantics in contrast to RPC semantics.
2. Improvements and wider understanding related to WS-Addressing
3. Understanding gained by implementing WS-Extensions like WS-Security [14], WS-Reliable Messaging[15] and WS-Transactions, and the proposals for a better SOAP processing Model
4. Availability of WS-Policy[4] and WS-Metadata [5] exchanges
5. Performance considerations of Axis1[16], and proposals for better XML processing
6. Availability of WSDL 2.0

Further more these developments have changed the expectations of the users. As a result the Web Service middleware developed 3 - 4 years back simply can not live up to the new expectations of the users. This is not because they are poorly designed; rather the assumptions that were made at the time of their development are no longer true. For example, three years ago synchronous request-response Web Services were the defacto standard. Axis1 supports only request-response interactions, an obvious design decision at the time. However such assumptions no longer hold and new requirements had to be met with "not so smooth" techniques. Supporting new requirements gracefully call for major architectural reforms.

Axis2 incorporate those developments and restructure the architecture around three topics.

1. XML Processing
2. SOAP Messaging
3. SOAP Extensions

Hence we shall focus our discussion on these three topics. Rest of the paper is organized as follows; the next section provides related works and Section 3 presents Axis2 architecture in general. The sections 4, 5, 6 explain the three topics and sections 7 conclude the discussion with summery and future works.

2. Related Work

Axis2 architecture is influenced by Axis1, feedback from users and experience gained from maintaining Axis1 for four years. Further more Apache SOAP [17] and related Apache Web Service projects like Sandesha [18], WSS4J [19] and Kandula [20] provide invaluable insights to the architecture. According to our observations we found Axis1 is limited in following three basic areas.

Axis1 defines the Handler architecture as a mechanism for extending the SOAP Processing Model. Handlers are components that can be registered with the Axis1 framework on per service basis. And they will be executed before sending or processing SOAP Messages. When executed, each Handler is provided with a reference to the current SOAP Message, and handlers may process the SOAP Headers. However the interfaces

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00 © 2006 IEEE

and the Services provided to the Handlers turn out to be insufficient and, to counter that, implementation of WS-* specifications need to do additional work, thus making those architectures complicated.

Axis1 XML processing model works on top of SAX events. In order to provide a model that allows random access to the SOAP Message, Axis1 has to record the SAX events. Since SAX does not allow event flow to be paused once started, resulting XML processing model must record all the events. This was the major reason for the performance problems of Axis1.

Thirdly as mentioned in the introduction, Axis1 assumed Request Response synchronous interaction thus lack messaging support.

Further more, among the Web service middleware following projects can be considered peers.

XSUL2[10] developed at Indiana University provide a SOAP processing model based on XML Pull parsing and achieved impressive performance results [11]. Further more XSUL2 provides support for one-way, request-response interactions with synchronous / asynchronous behavior.

Colombo project [12] developed by IBM research provides a framework for service oriented applications that include a Web Service stack with transactional reliable and secure interactions support.

New version of the JAX-RPC specification, JAX-WS [23] and the reference implementation incorporates messaging, WS-Security, improved data bindings and annotation support as notable changes. Axis2 will provide a JAX-WS compatible layer in due course.

Among commercial products, following two provides the most comprehensive Web Services frameworks.

Windows Communication Framework (WCF – formerly known as Indigo) by Microsoft corporation provides a base for service oriented architectures with support for transactional reliable and secure interactions.

Systinet Server provides a Web Service stack with security and reliable messaging support. The server is available in both C++ and Java languages.

3. Axis2 Architecture

Axis2, being a community project, should address wide range of scenarios and interests. Resulting architecture tends to be complex, and hence maintaining simplicity and elegance of the system is a major challenge.

This challenge is addressed by defining a minimal but extensible core that includes SOAP processing Model, XML processing, Messaging frameworks and abstractions to implement other aspects like transports and deployment. These “non-core” aspects are layered and have minimal correlation with the core architecture. For example, we consider data binding as a separately solved problem that is orthogonal to the Web Services architecture. Axis2 supports multiple data bindings yet the core is not aware of its existence.

A system can be explained in terms of functionality, subsystems and API provided for the users. In next section, we shall look at Axis2 in each of these angles.

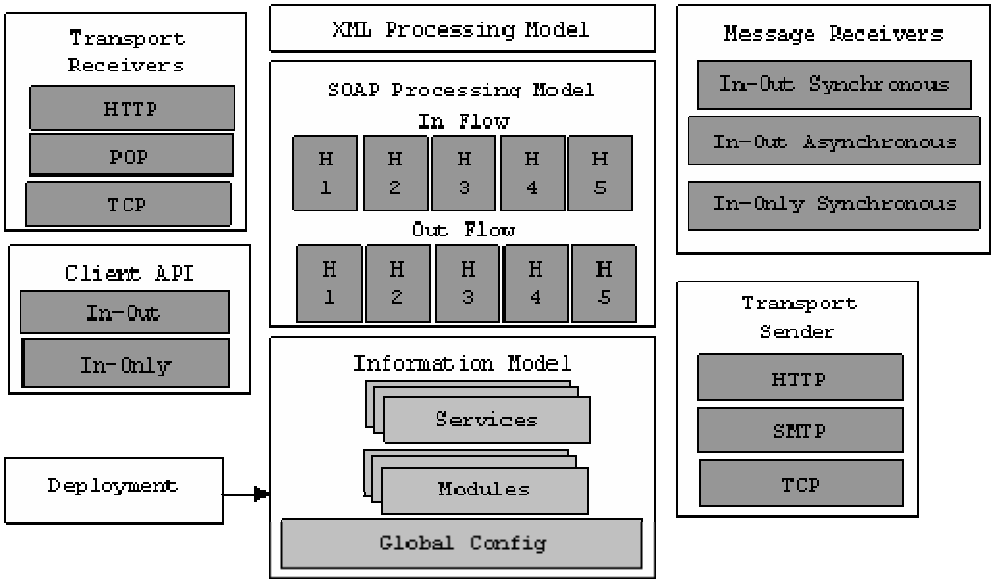

Figure1: Axis2 Architecture

3.1 Functionality of Axis2

Axis2 is not just a tool kit for sending and receiving SOAP Messages, rather it provides the foundation for the Apache Web Services stack. It provides abstractions and services that are used in every aspects of the Web Services stack. The key functionalities provided by Axis2 are enumerated below.

1. A Framework to develop, to deploy, to invoke, and to manage Web Services.
2. Extensible SOAP processing model and services need to develop WS-* specifications
3. Framework for modeling different Message Exchange Patterns (MEPs) and perform synchronous and asynchronous interactions
4. Pluggable Transports and data binding
5. WS-Addressing support
6. Message Transmission and Optimization Mechanism (MTOM) support
7. Representational State Transfer (REST) support

Figure 1 shows the subsystems of Axis2. Three subsystems, XML processing Model, SOAP processing model and Information Model are considered the Core, and they layout the foundation for rest of the systems. Non core sub systems are either pluggable, or can be replaced with minimal effort, where as the core subsystems are indivisible with Axis2.

Information Model contains states of Axis2 and is usually populated by the deployment mechanism. Starting Axis2 involves initializing the information model and transport Receivers.

Incoming SOAP message is received by a Transport receiver and handed over to SOAP processing Model, at the end of the processing; the message is handed over to the Message Receiver to invoke the business logic.

Outgoing SOAP Message is initiated by the Client API or the Message Receiver and handed over to SOAP processing Model, once processed; Message is passed to the transport Sender and sent to the target endpoint.

3.2 Axis2 User Interfaces and tools

Axis2 provides interface for three groups, Web Service users, Web Service Developers and Developers of the WS-* specifications. Basic interfaces can be listed as follows.

1. Client API for invoking Web Services
2. Server API for developing/deploying Web Services
3. WSDL2Java tool to code generate server or/and Client side code from a WSDL
4. Handler API for extending the SOAP processing model
5. Deployment API for configuring Axis2

The topics like WSDL, deployment, MTOM and REST are interesting by their own right. However we will not discuss them here as they are not central to the Axis2 core architecture. In the rest of this paper we will systematically discuss the three basic topics XML Processing Model, Extensible SOAP processing model and Messaging Framework.

4. XML Processing Model and Performance

4.1 XML processing and Web Services

XML processing, the decisive factor for performance, faces a conflict of interests inside a Web Service middleware framework. In one hand the framework should provide efficient XML processing, on the other hand it should provide users with a simple and easy to use representation of XML.

Handling an XML document involves managing documents in three different states.

1. The source form (input stream/object)
2. The intermediate form (usually a tree structure denoting the XML info set)
3. Processing specific format

The first and the last states are inherent to the SOAP processing and can not be avoided. By means of efficient XML processing we try to reduce intermediate form as much as possible since the usual way of transferring between state 1 and state 3 is via state 2.

Key for efficient XML processing lies in the ability to do sequential processing, without moving back and forth in the XML document. Before moving further we should look at the SOAP processing model and investigate the possibilities of sequential processing.

4.2 SOAP and XML processing models

SOAP message has two parts, the Headers that provides the information about the service behavior and the body that includes the payload. At SOAP Processing those two parts are processed in order. According to the SOAP processing rules, SOAP Header processing requires random access, because they need to move back and forth among data. However overhead introduced by the SOAP Headers is generally limited since headers are smaller compared to the SOAP Body. The SOAP Body processing usually involves data binding, which can be done in a streaming fashion.

When a SOAP message is being sent the SOAP Body is constructed from parameters in the programming language representation and SOAP Headers are filled by the SOAP processing model. Again SOAP Body construction can be done in streaming fashion.

But it is important to note that there are use cases like WS-Security [13] where the SOAP Header processing requires accesses to SOAP Body, and in those cases it is harder to do the SOAP Body processing in streaming fashion.

4.3 XML processing model Architecture

Previous section concludes that the SOAP processing can not always perform in a streaming fashion (Due to WS-Security use case). But when we exclude such special use cases like the WS security use case, there are a number of instances where this optimization is possible and profitable.

Axis2 approach is to streamline the SOAP processing whenever it is possible and to handle the complexities transparently when the processing can not be streamlined.

Axis2 XML processing model tries to achieve this objective by limiting the intermediate form of the XML document as much as possible. Axis Object Model (AXIOM) tries to provide a simpler interface to users while handling complexities of the efficient XML Processing behind the curtain. Users may provide a source (input stream or a Java object) and create a virtual XML segment, where the source will not be read but just kept inside the XML Segment. When the user traverses the XML Segment, AXIOM reads data out of the source, but tries to read data as late as possible. We call this approach "on demand (lazy) parsing".

If so chosen, users of the tree model (XML Segment), can request for STAX [7] events for a selected part of the tree model and let AXIOM know that tree model is no longer required. Usually this happens at end of the SOAP Header processing, and request is for STAX events of the SOAP Body. In most cases, the Body part of the source has not been accessed and STAX events can be directly generated from the source. If the model is half built, the first part of the STAX events are generated from the half built model and the rest will be generated directly from the underline STAX parser.

When a SOAP Message is received, the Headers are processed first and then the Body is processed. Headers processing always involves moving back and forth among Headers, hence needs a tree Model. However if processing does not access the body, events will be read directly off the source, (which is an input stream in this case) and fed to business logic invocation which is usually done in streaming fashion.

Conversely, when a SOAP Message is sent, the information that needs to be sent in the SOAP Body is provided in some programming language representation (e.g. Java objects/DOM). Those objects act as the sources, and they could be directly written to the stream if the Header processing that follows does not accesses the information represented by the source objects.

In both these cases if SOAP Header processing accesses the SOAP Body, the intermediate object model can not be avoided. But AXIOM provides a model to build a lesser amount of intermediate model as far as possible. Furthermore when the optimization is not possible, it is handled transparently and upper layers do not need to worry about complexities of efficient XML parsing.

5. Extensible SOAP Processing Model

5.1 Introduction to SOAP processing

Web Services are defined by SOAP, WSDL and UDDI specifications and set of WS-* specifications that explained other aspects like Security, Reliability and Transactions. Each of this aspect is defined as a SOAP Extension, and a given service may associate one or more aspects with itself.

Those aspects are developed as plug-ins to SOAP processing model and a Service may enable aspects related to the plug-ins by associating them. For example Reliability and Security are two aspects, and could be implemented as plug-ins. Service can have reliability and security (or one of the two) by associating the corresponding plug-in(s) into the framework. Each plug-in should be able to work by itself as well as co-exists with the other plug-ins the service might associates with.

This approach simplifies the user's task to a great extent since each of this extension is available as a single entity, and they can be enabled by associating them. But on the other hand Web Service middleware has to solve more complex problems, as they should generalize these aspects and behavior. Axis2 provides an environment to develop those extensions, taking that generalization yet another step forward.

5.2 Challenges presents by SOAP Processing Model

Axis1 provides basic support for these SOAP extensions using Handler framework. A Handler is a component that can be registered with SOAP processing model in either global or per service basis. Axis1 SOAP processing model executes registered Handlers before sending an outgoing SOAP message or invoking business logic of an incoming SOAP message. The Handlers can be deployed via Axis1 deployment mechanism. This Handler architecture was an extension of Interceptor [6] pattern used in CORBA. They were first introduced by Axis1 and later adopted and standardized by JAX-RPC [9] specification.

But interaction and co-existence of SOAP extensions could be very complex. While using the Handler architecture provided by Axis1, the authors of the SOAP extensions and the users of those extensions, face following difficulties.

1. Some SOAP extensions depend on other extensions and some are affected by the other, as a result execution order of extensions is critical.

2. The efficient implementations of SOAP extensions require more sophisticated services from the framework than just accessing the current SOAP message
3. Deploying and associating SOAP extensions with a Web Service is complex and long procedure.

There are interdependencies among SOAP extensions, for example most of the extensions depend on WS-Addressing, and hence the WS-Addressing extension should be executed before other extensions. Each extension should work independently and the framework should adopt itself when extensions co-exist. With Axis1 it is up to the Web Service developer to provide a total ordering of associating Handlers. However it is the SOAP extension developer who is aware of the dependencies among different SOAP extensions. As a result, providing a total order for the extensions needs deep knowledge on workings of SOAP extensions. This overloads Web Service developer thus limiting usability of SOAP extensions. This can be addressed by providing a mechanism for SOAP extension developer to express the partial order of each SOAP extension with itself and calculating the absolute order at the execution time.

Efficient implementations of most extensions require much more complex services from the framework than the mere access to the current SOAP message. Few of such services are as follows.

1. Pause the execution and restart it later
2. Add new operation to the current Service
3. Access the related messages grouped together
4. Start a new sub execution path

With Axis1 the interface between SOAP extensions and SOAP processing model is loosely defined. Hence installation and engagement of different Extension vary. As a result getting a SOAP extension to work is a demanding and time consuming task.

5.3 Processing Model Architecture

Axis2 extends the Axis1 Handler architecture to provide better support for the SOAP extensions. New architecture introduces the Module, a higher level abstraction for related set of Handlers grouped together. A Module typically represents a SOAP extension and in addition to Handlers, includes rules (Phase Rules) that express the partial order of the Handlers with in the Module and operations that should be added to the service the Module is associated with.

Module is given an archive format by Axis2 deployment model. The Module is bundled according to this archive format and it can be deployed by copying the archive to the Axis2 repository. Once it is copied Axis2 deployment model parse it and make it available to the rest of the system.

Use of the Module is best explained by examples. A Module represents a SOAP extension like Reliable Messaging or Addressing. Module contains all the information that needs to make that SOAP extension enable with a service, and service developer can enable one or more SOAP extensions (e.g. Addressing, Reliable Messaging) by just associating Modules with services. The crux of this approach is to develop self contained set of Modules that can be used with services by just associating them with services. The association of service with a Module is termed as "the service engages the Module". For an example if a service engage addressing module, the configured addressing Handlers will be added to Handler chain and service will support addressing.

If a service engage two SOAP Extensions (Modules), the order of the Handlers from different Modules is important. For an example, if Reliable Messaging and addressing are enabled, the addressing Handlers should be placed before the Reliable Messaging Handlers because latter depends on the data processed by the former. The above mentioned rules are used to make sure the correct order is preserved.

When the Reliable Message is enabled, the engaged service need to have a new operations to create a message sequence and end a message sequence. This is viewed as the Module injecting an operation in to engaged service and named as "Operation injection". There are other extensions (e.g. Secure Conversation) that need similar functionality and as a result Operation injection is added to the Module definition.

When the archived module is loaded by the deployment system, the module is said to be available. Services or operations may engage them and associate the extensions provided by the module to itself. According to the rules and operations, Axis2 configure the Handlers and add them to the Handler chains associated with each service or operation.

For performance reasons, the Handler chains are calculated at the deployment time; however the handler chains can be recalculated to support the change in configuration done at runtime. For an example if service requester and provider using Metadata Exchange [5], decides that the security is needed to continue their interaction, Axis2 can add the security module in runtime and recalculate the handler chains.

5.4 RM Server side as a Case Study

This section provides an overview of Reliable Messaging implementation (Sandesha2 [18]) to demonstrate how Axis2 SOAP processing models functions.

According to Reliable Messaging protocol, a Reliable Messaging enabled Service will provide two operations, *Create Sequence* and *Terminate Sequence* to the Client. Client will initiate a Message sequence using the create sequence Operation and send messages to the Server. Server makes sure they are executed in correct order

regardless of the order they are received. For each successful message the server send back an acknowledgment message.

Sandesha2 is implemented as a Module that has a Reliable Messaging Handler and two operations *Create Sequence* and *Terminate Sequence*. When a service engages the module the two operations are injected to the Service and made available as operations of the service. The Clients may initiate or terminate a sequence using those two operations.

When a Message arrives, the Reliable Messaging Module sends back an acknowledgment using a new outgoing Pipe (More on Pipes will be covered in the last section). The order of the Messages is examined and if they are in order they shall be executed. If they are not in order, execution is paused and the state is saved. Later when successive messages complete the order, execution for each message is resumed.

Reliable Messaging may depend on WS-Addressing and WS-Security extensions and those requirements are provided as "Phase Rules". SOAP processing model manages order of Handlers accordingly.

6. SOAP Messaging support

6.1 Introduction to Messaging

Three years ago request response interactions were defacto standard for Web Services. But distributed interactions have different considerations than their local counterparts and are best addressed with messaging.

SOAP Messaging is built on two basic operations, Message sending and receiving. Some of those messages are related, and grouped together to create Message Exchange Patterns (MEP). Simplest MEP has a single Message and the most common MEP is In-out MEP which represents a Request-Response interaction. However it is also possible to define more complex patterns like publish-subscribe interactions.

Transports used for messaging are two types, one way transports (e.g. SMTP, JMS) and two way transports (e.g. HTTP, TCP). One way transports are unidirectional where as two way transports are bidirectional. The two way transports can effectively act as a one way transport by utilizing only one direction, good example is HTTP replying with "HTTP 200 ACEPTED" to say that there is no content in the response.

Behavior of SOAP Nodes between two related Messages defines synchronous and asynchronous nature of the interaction. But the term synchronous/asynchronous is used to denote Client API level as well as transport level, and is a common source of confusion. Former is decided by whether client side thread of execution wait for the invocation to be complete, and the latter is used to explain behavior of the transport between the related Messages. If sent Messages and received Messages are transported in different transport connections they are Transport level asynchronous, and if they are transported in same connection they are transport level synchronous.

To capture these different aspects of messaging the architecture need to be inherently message centric. Due to lack of messaging support in Web Service middleware, those problems have been tackled on top of Web Services middleware adopting a layered approach [21], [22]. But we believe that those problems are best addressed inside core of Web Service middleware itself, where the intimate detail about the messaging is available. We shall look in to the Axis2 messaging architecture in the next section.

6.2 Messaging Architecture

Axis2 messaging architecture provides a framework to model different MEPs, and their synchronous and asynchronous aspects. In this section we shall discuss the architecture in detail.

Different types of MEPs are infinite, hence only way to support them is to provide building blocks that can be put together to model arbitrary MEPs

The basic building blocks of messaging, the operations sending and receiving are implemented as Pipes. Following figure denote three layered Axis2 Messaging architecture. The bottom layer of the figure represents Handler architecture that defines the Pipes. The Pipes consists of Axis2 defined inbuilt handlers, as well as user defined custom Handlers.

Each pipe does not assume anything about correlation with the other pipes, and those correlations are implemented by the next layer that composes basic pipes and create complex message interaction. At the second layer of the figure, Operation Clients (Client API) and the Message Receivers (Service API) know detail about current MEP. They are not part of the Axis2 core, and are pluggable.

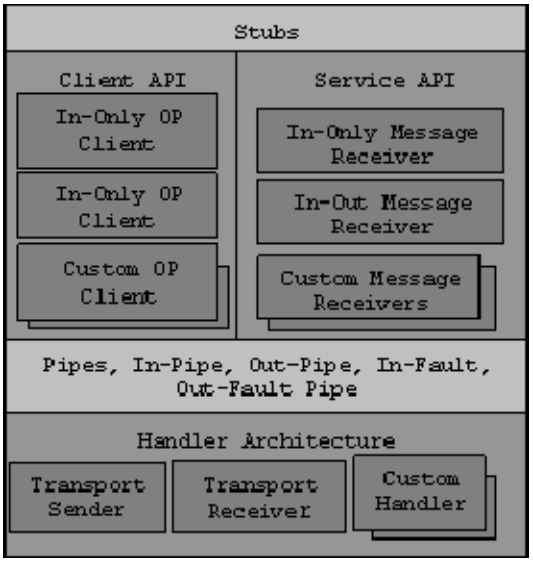

Figure 4: Messaging Architecture

For example In-Out Operation client knows there will be a response Message and prepare the system for accepting the response message. Similarly the In-Out Message Receiver knows there should be a response Message and initiates an Out-Pipe at end of the In-Pipe. Axis2 provides an implementation for In-Only and In-Out MEPs, but users can write new ones and registers them. The correct MEP is picked up by the deployment configuration and usually those configurations are built base on WSDL information.

When utilizing its full Messaging capabilities, Axis2 became a peer-to-peer system. To accept related messages that are transport level asynchronous and do not use polling, message initiator needs to set up a Transport Listener. In one hand this could be a simple standalone client starting a simple HTTP server from Java, or on the other hand the client could be a J2EE application running on a J2EE server and might use a Servlet as the means of receiving those messages. Then both Server and Client would have transport Listeners running, and would play role of sending and receiving Messages in same manner.

Axis2 Server and Client are identical at architecture level. They share same execution mechanism (Pipes) and same information model. When a SOAP Message is being sent (transport level asynchronous) and if there are related messages to be arrived, Axis2 registers an operation that would match with dispatching properties of the related messages. The related SOAP messages are processed in same way the incoming SOAP Messages are processed in the Server. If WS-Addressing Relates-To property is present, the Message is automatically correlated and added to correct message group of the information Model.

A deployed Web Services can not be identified as (transport level) synchronous or asynchronous, and that is decided by Service Requester. Depending on value of WS-Addressing Reply-To property the service Provider should send the related messages (if there are any) to correct destination.

Following this philosophy, a Service deployed in Axis2 is neither synchronous nor asynchronous. The Service Requester may decide it by specifying WS-Addressing Reply-To Headers of SOAP Message. Incoming SOAP Message triggers an In-Pipe; the execution that follows may generate related messages according to the Message Exchange Pattern. If WS-Addressing Reply-To Header is present and its value is not anonymous, related messages will be sent to the Reply-To address and otherwise anonymous address is assumed and related messages are sent back via return channel of the transport the SOAP Message was received.

Axis2 handles transports via Transport Senders and Transport Receiver interfaces. The core architecture is not aware of finer details about each transport, and as a result new transports can be plugged in via the deployment model. Different Axis2 supported transports are categorized as one way and two way transports. However difference and complexities are hidden behind Client API and as a result user is unaware of such complexities.

In summary Axis2 messaging architecture is based on Pipes that can be composed by upper layers to create complex message interactions. Those compositions are pluggable so user can define new MEPs. The Message path is decided based on WS-Addressing and the related messages are sent in conformance to WS-Addressing properties. The resulting architecture provides a peer-to-peer model that supports multiple MEPs as well as both transport and client API level synchronous asynchronous behavior.

7. Summary and future work

Axis2 amalgamates the experiences gained by developing Axis1 and Apache Web Services projects. The Axis2 core architecture is comprises of an XML processing Model, SOAP processing model and Messaging framework.

Axis2 has just released version 0.94 as and in the process of testing and incorporating user feedback. The development effort is continuing and currently the team is concentrating on performance and standard conformance.

At the same time effort to implement other SOAP extensions on Axis2 has begun; current release has an initial version of WS-Security bundled with it. Effort for WS-Reliable Messaging, WS-Secure conversation, WS-Transaction and WS-Policy is underway.

Apache Web Services project thrive to provide complete Web Service stack on top of Axis2, and to continue Apache legacy on Web Services.

8. Acknowledgement

Axis2 is a community effort; in addition to the authors of this paper, Davanum Srinivas, Aleksander Slominski, Dasarath Weerathunga, Venkat Reddy, Ashutosh Shahi, Jayachandra Sekhara, Thilina Gunarathne, Ruchith Fernando, and Saminda Abeyruwan have been key contributors to the design and implementation of the Axis2.

9. References

[1] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, ‘‘Simple Object Access Protocol (SOAP) 1.1,’’ http://www.w3.org/TR/SOAP, May 2000.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, ‘‘Web Services Description Language (WSDL 1.1),’’ W3C Note, http://www.w3.org/TR/wsdl, March 2001.

[3] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler, ‘‘Web Services Addressing (WS-Addressing),’’ W3C Member Submission, August 10,

[4] D. Box, F. Curbera, D. Langworty, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk, ‘‘Web Services Policy Framework (WS-Policy Framework),’’ 2002,

[5] F. Curbera and J. Schlimmer (Eds.) ‘‘Web Services Metadata Exchange (WS-MetadataExchange)’’ (September 2004),

[6] P. Narasimhan, L. Moser, and P. Mellior-Smith, ‘‘Using Interceptors to enhance CORBA,’’ Computer 32, No. 7, 62–68, July 1999.

[7] Christopher Fry et, al. Streaming API for XML (STAX) Specification, (October 2003)

[8] Jeff Suttor, Norman Walsh, and Kohsuke Kawaguchi JSR 206 Java API for XML (JAXP) Processing Specification, (December 2003)

[9] Roberto Chinnici et, al. Java APIs for XML based RPC (JAX-RPC) Specification, (October 2003)

[10] XSUL2 http://www.extreme.indiana.edu/xgws/xsul/

[11] Michael R. Head1, Madhusudhan Govindaraju, Aleksander Slominski, Pu Liu, Nayef Abu-Ghazaleh, Robert van Engelen3, Kenneth Chiu, Michael J. Lewis, “A Benchmark Suite for SOAP-based Communication in Grid Web Services”, November 2005.

[12] F. Curbera, M. J. Duftler, R. Khalaf, W. A. Nagy, , N. Mukhi, S. Weerawarana, “Colombo: Lightweight middleware for service-oriented computing”, IBM Systems Journal, Volume 44, Number 4, 2005

[13] David Chappell et, al. “Introducing Indigo: An Early Look”. MSDN library, (February 2005)

[14] Chris Kaler et, al. “Web Services Security Specification” , (April 2002)

[15] Christopher Ferris et, al. “Web Services Reliable Messaging Protocol”, (WS-ReliableMessaging), (February 2005)

[16] The Apache Project Axis Java, http://ws.apache.org/axis/.

[17] The Apache Project SOAP, http://ws.apache.org/soap/.

[18] The Apache Project Sandesha, http://ws.apache.org/sandesha/.

[19] The Apache Project WSS4j, http://ws.apache.org/wss4j/.

[20] The Apache Project Kandula, http://ws.apache.org/kandula/.

[21] Uwe Zdun et, Markus Voelter, Michael Kircher “Pattern-Based Design of an Asynchronous Invocation Framework for Web Services”, International Journal of Web Service Research , Volume 1, No. 3, 2004

[22] Geoffrey Fox, Shrideep Pallickara, Savas Parastatidis “Towards Flexible Messaging for SOAP Based Services”

[23] Roberto Chinnici, Marc Hadley, Rajiv Mordani, "The Java API for XML Web Services (JAX-WS) 2.0 Proposed Final Draft, (October 7, 2005)