AtomServ Architecture:
Towards Internet-scaled Service Publish, Subscription, and Discovery

Chen Wu, Elizabeth Chang
Center for Digital Ecosystem and Business Intelligence
Curtin University of Technology, Perth 6845, Australia
{Chen.Wu, Elizabeth.Chang}@cbs.curtin.edu.au

Abstract

With the surge of SOA-based infrastructure and applications, increasingly end users and small-medium-enterprises directly participate in the service publish and discovery across the Internet. The recent shutdown of public UDDI exposes critical problems of existing Internet-based service discovery. Hence, public service discovery becomes a central SOA issue. In this paper, we present a light weight service discovery architecture built upon widely-adopted WWW technologies and proven software architectural styles. Firstly, it provides a handy discovery facility for personal web services providers and consumers, who would not be expected to able to use complex UDDI specifications with dedicated endpoint computing capability. Secondly, it widens the adoption of service discovery by allowing simple and uniform web user interfaces (e.g. Internet Explorer7.0 and Firefox1.1) to subscribe and access frequently changing business services. This undoubtedly lowers the entry barrier for end users to play the role of service providers or consumers in a sheer Service-Oriented Environment across the Internet.

1. Introduction

Service discovery is a key aspect in the SOA research community. As an essential SOA activity, it paves the way for conducting service binding, sharing, reusing, and composing in a dynamically changing business environment. With the increasing number of service providers, service discovery is even more crucial for those small-and-medium enterprises as an ephemeral business opportunities could be the most significant factor leading to their ultimate successes. Moreover, with the enhanced personal computing capability (e.g. wireless device), a myriad of online personal service providers and consumers are overwhelmingly playing their important parts in the SOA practice. As a result, service discovery has become a central issue for this large population of service 'players' that are distributed across the Internet.

In contrast, the primary service discovery mechanism – the UDDI registry – has shut down permanently since January 12, 2006 [1]. However, this does not imply that some form of 'public' service registry is not necessary. It reflects that existing public service discovery supported by the UDDI registry has some issues [2, 3] that cause its suspension. One of the reasons suggests that the public registry UDDI is "too complex" for the end users, since the specification is more driven by its primary members than the feedback from the real world end users. This hinders its ubiquitous adoption among Internet communities.

In this paper, we provide a light weight service discovery architecture – AtomServ. It is built upon widely-adopted WWW technologies and proven software architectural styles. We introduce Atom-based service feeds to realize the publish/subscribe service discovery paradigm. Therefore, the practice of publish/subscribe interaction style is extended from RPC-based LAN to the HTTP-based Internet. As a result, existing web browsers, news aggregators, and feed readers can easily access and discover web services metadata. This without doubt improves the service advertisement capability given the prevailing use of RSS/Atom feeds and weblogs. Furthermore, the REST architectural style is firmly entrenched in the AtomServ design so that established web architectural principles ensure the feasibility when deploying the AtomServ architecture in an Internet scale.

The rest of this paper is structured as follows. Section 2 briefly reviews the related work in Atom-based web services research. Section 3 elaborates three types of architectural elements. This is followed by three corresponding architectural views provided in Section 4. Three architectural styles are then summarized in Section 5. Section 6 introduces implementation work. The paper concludes in Section 7, which identifies future research directions.

2. Related work

Atom[4] (or RSS[5]) is an XML-based file format that allows lists of information, known as "feeds", to be synchronized between publishers and consumers. The current use case of Atom is to "syndicate" web content such as weblogs and news headlines to other web sites and end consumers. However, nothing prevents it from being used for other types of content such as web services metadata. This is one of the main motivations for our research. Nevertheless, utilizing Atom/RSS to facilitate SOA service discovery and interaction still lies in its infancy. To our best knowledge, [6] is the only existing work that explicitly integrate Atom feeds in service description. The author introduces and demonstrates a combined use of the Atom 1.0 and WS-Addressing 1.0 [7] specifications. It focuses on generating Atom feeds out of the Web Services for J2EE (JSR-109) specification file located in the specific service endpoint – IBM WebSphere Application Server. A small portion of the Section 3 in this paper corresponds to the work in [6]. However, this paper aims to proposes a service discovery architecture, which applies the publish/subscribe style through the atom-based service registry that supports service Atom feeds publish and subscription. From a broader integration perspective, the work in [8] defines the specification to enable loosely-cooperating applications to use RSS2.0[5] as the basis for bi-directional, asynchronous information sharing and communication. Although it did not give any implementation in terms of (web) service sharing and reusing, it does reaffirm the significance of Atom/RSS in the Internet-scaled computing environment.

3. AtomServ Architectural Elements

At the abstract architectural level, AtomServ comprises architectural elements that interact with each other. In this section, we explore each type of these architectural elements in detail so that special attentions can be given to the roles of elements and constraints of interactions among them in Section 4.

3.1. Data Elements

Data elements are xml documents enclosing useful request and replying message summarized in Table 1.

Table 1. AtomServ Data Elements

Data Elements	
Atom Feed and Entry	All the service publish and subscription data.
Topic Space Specification	Topic specifications and its subset
UDDI data	For interaction with UDDI server

3.1.1. Atom Feed and Entry. The key data element in AtomServ architecture is the atom feed and atom entry. Atom describes related information organized into lists termed as "feeds". An atom feed comprises several items known as "entries", each of which encloses a set of extensible metadata to self-describe the delivered information. For example, each entry has a title, an author, and a URI link, etc. While atom feed is originally used for news aggregation and syndication, we believe it is an expressive means to attach metadata for advertising web services and hence for facilitating the service publish and discovery. In AtomServ, an atom feed essentially contains the necessary metadata for a list of web services, each of which corresponds to an atom entry. In other words, the metadata of a web service is conceptually mapped to an atom entry, which is enclosed in an atom feed represented in the form of an XML document. By doing so, we are thus able to introduce atom publish and discovery mechanism into the world of web services. This is shown in Figure 1, where the arrows represent the mappings between service topic (see Section 3.1.2) and feed. Each entry in the feed is mapped to one web service belonging to a topic that corresponds to that feed. At the client side, web browser (e.g. Firefox1.0.7, IE7.0 beta) and feed aggregator are thus able to be notified of any changes (e.g. new web services are registered under certain topics) by polling the feed using common HTTP requests. Although the topics are organized in a hierarchy structure, it does not mean the feeds have to maintain such relationships. Thus, feed and topic have their own separate concerns. They do not need to interact with each other as long as their mapping is maintained via a valid URI that produces the representation of the web services metadata when being requested.

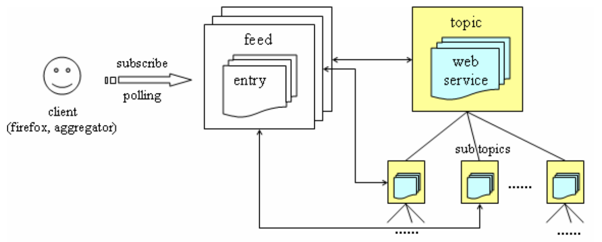


Figure 1. The mapping between feeds and web services

When narrowing down the atom feeds application into the specific area of web services discovery and publication, it is imperative to define the conceptual binding scheme that helps such a specialization consistent and compatible with the generalized syntax and semantics. Figure 2 illustrates the structure of the atom feed and entry, and their semantic binding in the area the web services publication and discovery.

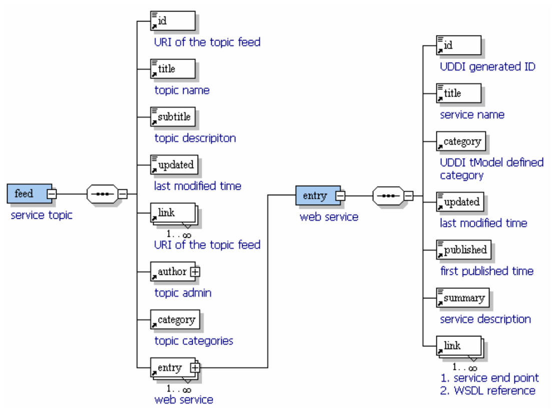


Figure 2. The Structure of Atom Feed and Entry

3.1.2. Topic Space Specification. A collection of related topics can be used to characterize and categorize subscription interests. Similar to the notion of group, a topic associates with it a number of related event contents, by which subscribers can subscribe or reason about notifications of interests. Such a topic-based notification mechanism is widely used in publish/subscribe systems [9]. In our research, we use the service Topic Space Specification (TSS) to store and specify all the topics, each of which in turn relates to a list of web services. The format of the specification is represented in XML document. As shown in Figure 3, a topic space organizes topics in a hierarchical structure where each topic has one parent element (either topic or the 'root' of the topic space), and may have several child topics. A topic corresponds to one atom feed by enclosing a sub-element named "feed". The feed element has an important attribute "href" identifying a valid URI that can be dereferenced to retrieve the representation of that Atom feed. In doing so, we establish the mapping between the service topic and feed illustrated in Figure 3. The description element is used to provide subscribers with some suggestions regarding the subscription options.

For example, we can define a sample specification presenting service topics in a domain "Online Financial Services". The topics are organized in a way that the two root topics are "Insurance" and "Loans".

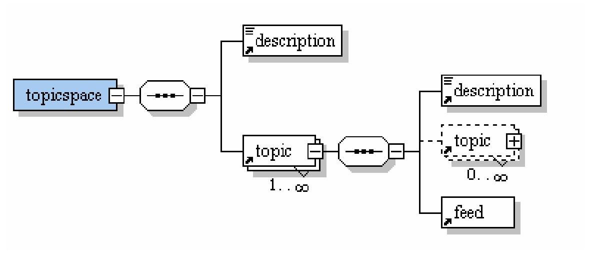


Figure 3. The Schema Diagram of TSS

In addition, the subscriber can use the Xpath expression to represent the subscription criteria. An example of topic subscription represented as Xpath is something like: "/Insurance/CarInsurance/StudentCar" or "/Loans/MortgageHomeLoans". Both topics can be included in the topic space "OnlineFinancialService". The reply of the subscription request is actually the subset of this TSS after calculating the Xpath against the whole specification document. It is very useful in replying the service topic subscription. At this stage, we assume each domain has one topic space and maintaining topics is out of the discussion of this paper.

3.2. Connectors

AtomServ consists of a number of architectural connectors listed as follows. At the protocol level, these connectors mainly leverage the HTTP to move data elements.

Service Provider Interface (SP). SP provides an interface for the SP to publish their service.

Service Requester Interface (SR). SR enables the SR to subscribe their interested topic, publish their requests, and organize their subscriptions. Most importantly, it provides the initial interaction probe for SR to interact with SP based on their subscribed feed entries. In addition, it provides the asynchronous notification to the user agent by employing the smart polling mechanism.

Service Cache (SC). SC stores the related service metadata locally in the client side. Although use of a cache adds some latency to each individual request due to lookup overhead, the average request latency is significantly reduced when even a small percentage of requests result in usable cache hits.

HTTPDispatcher. It receives the HTTP requests from the SP and SR, and forwards the modified requests to server components described in Section 3.3.

Atom Filter. All the filters share the same interface and form a chain to incrementally process the data.

Atom Cache. The atom cache keeps recent (most frequently-used) feeds in the memory to enhance the performance.

Admin. The Admin interface is used to maintain the AtomServ, i.e. to manage the topic tree, to configure the matchmaker, etc.

Topic Filter. The topic filter is used when topic retrieve requests are made to get the recent topic schema.

UDDI Client. It connects the UDDI server and performs the fundamental operations on UDDI.

3.3. Components

AtomServ contains a number of important architectural components – Atom Server, Topic Server, Matching engine, and the UDDI server.

Atom Server. This component mainly deals with the access and management of Atom files.

Topic Server. Topics are the major publish and subscription mechanism. Topic manager handles the management of all the service topics.

Matching Engine. This component maps the appropriate services for the service requesters. It achieves this by either searching the Atom files space or looking up in the UDDI server.

UDDI Server. It is used as service metadata storage in the RDBMS. In AtomServ, UDDI is not the major service discovery instrument; rather it works as a service repository.

4. AtomServ Architectural Views

In this section, we provide several architectural views [10] to illustrate how elements presented in Section 3 collaborate with each other to form an architecture. Three types of view – process, connector, and data – are particularly helpful in demonstrating the nature of AtomServ architecture.

4.1. Process View

A process view focuses on the data flow though the architectural components and some aspects of the connections between these components concerning the data elements [10]. Since we believe an architecture portrays the run-time system characteristics as well as the static system structure, we provide a process view that captures a 'snapshot' of a AtomServ at a particular point during which the server is responding to five different service requests conveying different data types. Such a process view is depicted in Figure 4, where the user agent uses the client-side connectors to initiate five different types of requests (from **a** to **e**), which are then sent to the HTTPDispatcher to interact with the server components. We now examine each request in more detail.

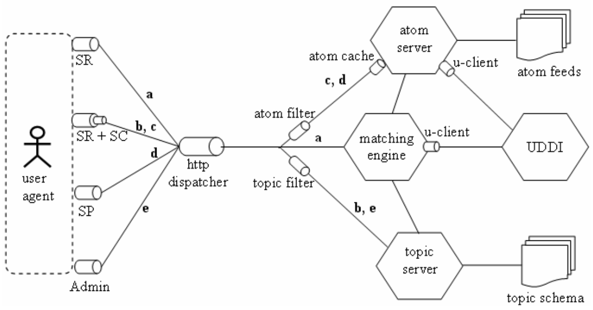


Figure 4. Process View of AtomServ architecture

Request a – find feed. This request encloses some criteria to actively seek appropriate services (providers). Since its response is not cacheable to the service cache, it is directly forwarded to the HTTPDispatcher. The HTTPDispatcher checks the request URI, and dispatches the request to the matching engine for query processing. The matching engine provides further access to Atom Server (e.g. service metadata search), Topic Server (e.g. topic-based search), and UDDI server (e.g. tModel-based categorization search). All the search results are populated using atom feeds maintained in the Atom Server. This process complies with the "synchronous read" service discovery pattern.

Request b – subscribe feed. The user agent, via the SR connector, subscribes to some interesting service topics against the topic schema. This part of request is sent to the HTTPDispatcher as HTTP GET, and is then forwarded to the topic filter connector. Since it is a subscription request, the topic filter simply passes the subscription criteria to the Topic Server, which is responsible for analyzing the subscription and returning a subscription tree – a subset of the topic tree defined in the topic schema. Once the SR obtains the local cache of subscription tree, it parses the tree and converts into a list of URIs that represents the corresponding feeds for the subscribed service topics. For all the upcoming interactions, the SR uses this local cache unless it obtains newer version through the regular polling.

Request c – notify feed. The SR connector simply performs the polling to each one of the URIs in order to get the updated service topics. Each polling request is passed towards the Atom Server by the HTTPDispatcher. However, the atom filter first examines the requested URI and decides whether it is necessary to send it to the server. If it is not, the atom filter simply bounces back the request using the *Not Modified* HTTP response. Otherwise, the atom cache attempts to retrieve a copy from the memory based on the requested URIs. If the cache version is not available, the Atom Server is resorted to find the feed entries from the atom feed files systems. Considering the connection performance, the multiple pollings share the same HTTP1.1 persistent connection with the AtomServ. Request b and c together form the “asynchronous read” discovery pattern.

Request d – publish. The user agent publishes its services to the AtomServ via the SP connector. This HTTP POST request contains a normal atom feed entry that entails enough service meta-data information including service endpoint, WSDL file location, and links to some other semantic description, etc. The HTTPDispatcher forwards the request to Atom Server, bypassing both the atom filter and the atom cache since it is a fresh publication request. The Atom Server stores the feed at the appropriate location according its specified URI that represents the service topic feed location, and appends the entry content to the service topic atom feed file. Moreover, if the UDDI server is available, the Atom Server parses the atom entry and produces a UDDI publication request that is then sent to the UDDI server via the UDDI client connector. Request d follows “synchronous write” pattern.

Request e – change topic. The user agent sends the update HTTP PUT request to the discovery agent via the Admin connector. The request is in effect a new version of the topic schema file in the form of an XML document. The Topic Server receives the request and updates the topic schema appropriately. The updated topic schema will be returned when it is requested by the following HTTP pollings initiated from any SR, SP, or Admin connector. This follows “synchronous write” interaction pattern.

4.2. Connector View

Since connectors move data around from component to component, they exhibit some properties required by the data elements [10]. Such properties are examined with particular focus from the connector view in this section. Furthermore, the AtomServ connectors also induce two overall architectural properties: scalability and extensibility.

Section 3.1 illustrates two types of data that connectors are transferring: atom feeds and the topic specification. The naïve polling of atom feeds can cause one well-known issue: the network traffic increases exponentially under the circumstances that all the service requesters perform regular polling to the server components. Addressing this issue essentially requires two properties for the connector: lower bandwidth and fewer roundtrips. For the first property, the connectors have to be ‘smart’ enough so that only necessary atom feeds or entries are transferred during the polling. In AtomServ, this is achieved by adding filters to the data flow between sender and receiver connectors. Each filter examines one particular aspect of the information, removes redundant data, and passes on the reduced data. Once the information has gone through all the filters, only needed data are actually traveling over the Internet. The bandwidth problem is thus alleviated. For example, in our implementation work, we add one filter that implements the delta-encoding specification stated in [11]. This filter can retrieve the delta part of the feeds against some criteria such as “*Modified Since*”. Moreover, such an filter-based processing also considers two top level requirements specified in [12].

Scalability. Scalability indicates the capability of the architecture to increase the load – large numbers of components, or interactions among components – without degrading performance (e.g. the server response time). One principal approach to improve scalability is to adopt the “scale-out” strategy [13], where the high loads are evenly distributed across a number of server nodes. Following this strategy, the dedicated filter that deals with the ‘delta’ feed in fact distribute the workload from one centralized server to multiple components without affecting the overall functionality. This enhances the system scalability within the Internet environment.

Extensibility. Extensibility refers to the ability of the architecture to dynamically accommodate changes without impacting the system. One important approach to achieve extensibility is to create loosely-coupling relationships between components that are subjective to frequent changes. Embedding the delta processing component within the server component is obviously deviates from loosely-coupling by mixing both business logics into a monolithic big component. We move additional delta calculation into dedicated filters, each of which reads data from its inputs and produces to its outputs the augmented data that is internally processed by the filter. Such a chain of filters systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be removed in an ad-hoc manner without changing the system architecture, design, and binary compilation.

The roundtrip issue is solved by implementing the HTTP1.1 *Condition Get* in the client connector so that the cache and the proxy server can ensure the actual polling will not be performed unless the actual changes are made and relevant to the service providers (e.g. new services are registered, existing services are updated, or unregistered).

4.3. Data View

A data view exposes the application state whilst information flows throughout the components [14].

This can be achieved by capturing the "application-oriented properties" [10], which describes the states of a data structure that are of significance to those architectural components. In this paper, we summarize application-oriented properties in Table 2.

Table 2. Application properties

has-outdated-feeds (**O-feeds**)	has zero or some feeds, some of which are not updated
will-have-updated-feeds (**WU-feeds**)	subscribed the feeds, but has not received the updated feeds yet
has-updated-feeds (**U-feeds**)	has the updated feeds against the server
has-outdated-topic (**O-topics**)	either has not downloaded any topics, or the topics are not updated against the changes
will-have-updated-topic (**WU-topics**)	Subscribed the changes to the topics, new changes have not been applied in the subscription tree yet
has-updated-topic (**U-topics**)	The subscription tree that the client has is updated against the version on the Topic Server

These states are solely captured from the "client-side" perspective. This is due to the reason that in order to enhance the scalability of the architecture, we intentionally let the client side maintain these applications state. The server focuses on stateless computation such as matchmaking, topic intersection etc. AtomServ concentrates all of the control state into the results received in response to the interaction. In doing so, the architecture undoubtedly exempts the server from managing complex states for each one of the numerous clients across multiple interactions. Hence, the scalability of the system is improved by 'scaling-out' the heavy workload from one centralized server to countless clients distributed across the Internet.

We now provide the visualization of such a states-oriented data view. As the process view and the data view are intertwined [10], we therefore examine these states according to the process view. For all possible processes, we walk-through all of its possible application states summarized in Table 2. This is shown in Figure 5, where the five requests studied in process view are included to demonstrate their corresponding state transitions. Each node represents a particular application state, which is the property combination of the two architectural data elements: the feed and the topic. The arrow edge between any two nodes presents the process that leads to the transition from one application state to another. Such a process is accomplished when the requests initiated from the client are all appropriately responded by the AtomServ server. In other words, the AtomServ reaches a steady-state whenever it has no pending requests and all of the responses to its current requests have been thoroughly received and interpreted.

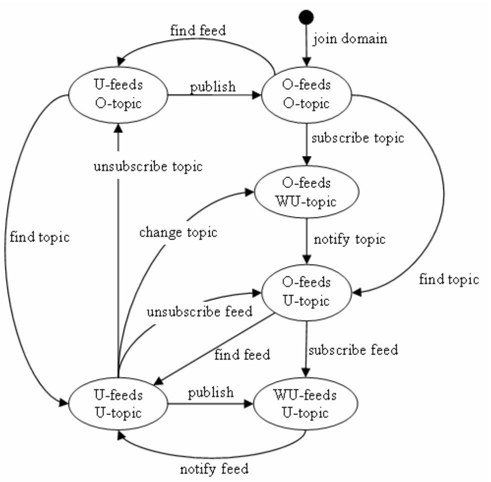


Figure 5. Application States transition

The data view directly supports the client design and implementation. A subset of such an application states transition in effect constitutes the core business logic of the client – service provider, service consumer, and the administrator. These clients are the most important stakeholders of AtomServ. The accomplishment of the state transition leads to the fulfillment of end user requirements. Moreover, it provides some hints on improving the application in terms of end user experience. Since the data view determines the whole application states, it also explicitly specifies the behaviour that the AtomServ server side is able to perform. Lastly, the combination of both data view and process view provides a consistent tool for designers to understand the AtomServ architecture.

5. AtomServ Architectural Styles

Software architectural style portrays a cluster of architectures that are related by shared structural and semantic properties. It can be used for multiple purposes such as categorizing architectures, defining common characteristics of architecture, and composing new architectures [15]. In this section, we demonstrate how three existing architectural styles are applied.

5.1. Publish/Subscribe Style

The Publish/Subscribe (Pub/Sub) paradigm[9] is a widely accepted, many-to-many asynchronous communication model used in distributed systems. In the context of service discovery, a typical scenario of Pub/Sub can be as follows: the service provider publishes its services in the discovery agent. Service requesters can then 'subscribe' by registering their interests with the discovery agent at anytime regardless of service providers. Whenever necessary, the discovery agent notifies related service requesters about the changes in terms of updated services (providers). The service requesters are then able to retrieve the service information based on the received notification. When thinking of applying Pub/Sub into AtomServ, we mainly concern one issue – scalability. In existing Pub/Sub styles, subscription server has to maintain a large number of subscriptions. Once any changes occur, the server compares them against each of the subscription, generate the notification messages, and reliably push notifications back to involved subscribers (i.e. the requesters). When subscription loads increase dramatically, it is unfeasible to scale properly without degrading the performance. To overcome this difficulty, we move part of the complexity from the AtomServ server to the service requesters. In particular, the notification (i.e. 'push') by the Atom Server is replaced with the passive update polling (i.e. 'pull') by the requester. Therefore, it is a requester's responsibility to keep itself updated to the latest changes. The Atom Server simply generates feeds of providers and makes them available for polling. The subscribers have to decide when and what to poll from the server based on their subscribed topic tree. Hence, the server frees itself from burdensome notification task and is thus more scalable when loads increase. Furthermore, loosely-decoupling is achieved in that the AtomServ server does not need to know who subscribes what. It is exempted from holding any references of service requesters, a tightly-coupled notification method suited for RPC rather than HTTP.

5.2. Pipe-Filter Style

Pipe-filter style is a very effective way to incrementally process a stream of data flow. Filters are completely independent with each other: they do not share state, control resources, or identity. Hence, it is very easy to add or remove different filters to change data processing dynamically according to the actual requirements. As stated in Section 4.2, atom filters and topic filters are used to reduce the web traffic during the feeds polling.

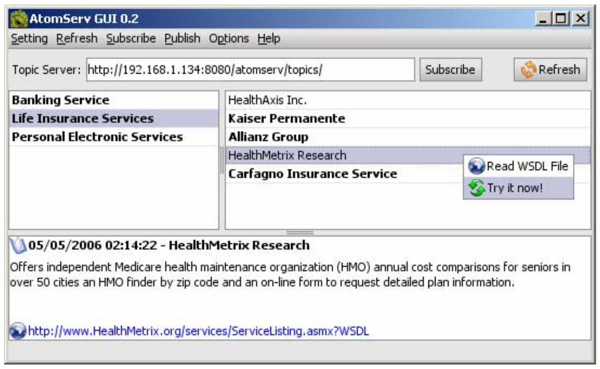
5.3. REST Style

REST architectural style [14] is in effect a constrain version of the current architecture of web, the most successful large-scale distributed system. One of the most important constrains that REST imposes on the interactions between connectors is to keep a small set of uniform interfaces (POST, GET, PUT, DELETE) supported by HTTP. It requires architects treat HTTP as an application protocol rather than a transport protocol. The latter case is, however, a common approach widely used in current web services research and practices. For example, the UDDI specification has defined over one hundred function calls, through which SOAP messages are transported via the HTTP. In this paper, we adopt this principle in order to achieve the Internet-scale architecture. This is reflected in the process view, where all the interactions are carried out solely by the HTTP operations and semantics.

6. AtomServ Implementation

Currently, we have implemented the proof-of-concept prototype for AtomServ. The main aim of this prototype is to verify the architectural design in terms of the styles application, the data elements arrangement. The prototype is built on Java and Windows platform. We believe various technologies can be applied to implement AtomServ. For the client connector implementation, we leverage the code base from open source ROME0.8[1], which provides basic feeds fetch utility. In addition, we provide an asynchronous proxy dealing with the polling on behalf of the user agent so that the user agent does not need to be aware the actual polling process. Rather, it behaves as if the notification is sent back from the server against its subscription. When a feed's updated version is retrieved by the proxy, it calls back interested feed listeners. In our implementation, the parameter of such a call back is a list of updated service entries, which are then highlighted as bold items in the user agent GUI as shown in Figure 6. The user agent GUI is based on the Thinlet[2], a very light weight GUI toolkit that can easily port our implementation to the J2ME-enabled wireless device.

[1] https://rome.dev.java.net
[2] http://thinlet.sourceforge.net



http://192.168.1.134:8080/atomserv/topics/

Banking Service
Life Insurance Services
Personal Electronic Services

HealthAxis Inc.
Kaiser Permanente
Allianz Group
HealthMetrix Research
Carfagno Insurance Service

05/05/2006 02:14:22 - HealthMetrix Research

Offers independent Medicare health maintenance organization (HMO) annual cost comparisons for seniors in over 50 cities an HMO finder by zip code and an on-line form to request detailed plan information.

http://www.HealthMetrix.org/services/ServiceListing.asmx?WSDL

Figure 6. User Agent GUI

7. Conclusions and Future work

In this paper, we have proposed a light weight service discovery architecture that lowers the entry barrier for end users to play their role of service providers or consumers in the Service-Oriented Environment. From the architectural perspective, two architectural properties – scalability and extensibility – are thoroughly considered and induced in our architectural design by leveraging the polling pub/sub and pipe-filter architectural styles. The REST style is also firmly rooted so that all the interactions comply with the current WWW architectural principle, which directly supports the Internet-wide scalability in AtomServ architecture. Finally, AtomServ architecture also supports the standard UDDI service discovery when necessary, thus keeping the compatibility with existing web services architecture. For the future work under this project, we aim to achieve two major goals. First, extensive experiments are to be carried out to test the two architectural properties: scalability and extensibility. To test the scalability, certain simulation designs are also needed. Second, following the Internet principle, we aim to leverage existing web search engines to discover the service feed. Some proven and successful web search engines (e.g. Google) have already started to support the news feed discovery. Since our architecture is based on the atom feed, it is very natural to link them together, thus utilizing the powerful search capability provided by Google to discover service feeds maintained by the AtomServ server. One research implication from doing this can also be arguably stated as: transforming service discovery paradigm from registry-based to index-based, where the index does not centrally control the information that it references. This is, however by far, the most successful mechanism for current WWW resource discovery.

8. References

[1] http://www.theserverside.net/news/thread.tss?thread_id=38136, 2006.
[2] U. Ogbuji, "UDDI 3.0? Who really cares?," Oreilly, 2005.
[3] D. Chappell, "Who Cares About UDDI?," Addison Wesley, 2002.
[4] M. Nottingham and R. Sayre, "The Atom Syndication Format," in *RFC 4287*: The Internet Society, 2005.
[5] "RSS2.0 Specification, http://www.rssboard.org/rss-specification," 2005.
[6] J. Snell, "Advertise Web services with Atom 1.0, http://www-128.ibm.com/developerworks/webservices/library/ws-atomwas," in *IBM developerWorks*, 2005.
[7] D. Box and F. Curbera, "Web Services Addressing (WS-Addressing)," *http://www.w3.org/Submission/ws-addressing/*, 2004.
[8] J. Ozzie, G. Moromisato, and P. Suthar, "Simple Sharing Extensions for RSS and OPML," in *Microsoft MSDN*: Microsoft Corporation, 2006.
[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Survey*, vol. 35, pp. 114-131, 2003.
[10] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40 - 52, 1992.
[11] RFC 3329, "Delta encoding in HTTP," 2002.
[12] D. Austin, A. Barbir, C. Ferris, and S. Garg, "Web Services Architecture Requirements," *W3C Working Group Note*, 2004.
[13] Microsoft Corporation, "Implementing a Scalable Architecture," *Microsoft Windows Server 2003 White Paper*, 2002.
[14] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," *PhD Dissertations, University of California, Irvine CA, USA*, 2000.
[15] E. D. Nitto and D. Rosenblum, "Exploiting ADLs to specify architectural styles induced by middleware nfrastructures," *Proceedings of the 1999 International Conference on Software Engineering*, pp. 13 - 22, 1999.
[16] S. Cheshire and M. Krochmal, "DNS-Based Service Discovery," in *Internet Draft*: Apple Computer, Inc., 2005.
[17] Microsoft Corporation, "Enterprise UDDI Services: A Synopsis," *Microsoft Windows Server 2003 White Paper*, 2003.