WS-Aggregation: Distributed Aggregation of Web Services Data

Waldemar Hummer, Philipp Leitner, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, 1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

Recent trends of Web-based data processing (e.g., service mashups, Data-as-a-Service) call for techniques to collect and process heterogeneous data from distributed sources in a uniform way. In this paper we present WS-Aggregation, a general purpose framework for aggregation of data exposed as Web services. WS-Aggregation provides clients with a single-site interface to execute multi-site queries. The framework autonomously collects and processes the requested data using a set of cooperative aggregator nodes. The query distribution is configurable using strategies, e.g., QoS-based or location-based. We introduce WAQL as a specialized query language for Web service data aggregation that is based on XQuery. 3-way querying is a possibility to optimize requests by reducing the amount of data transferred between aggregator nodes. A Web-based graphical user interface facilitates composing aggregation requests. Our performance evaluation, which comprises aggregation scenarios with different settings, shows the good scalability of WS-Aggregation.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based Services; H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Distributed Data Aggregation, Web Services Data

Keywords

Web Services, Aggregation, Distribution, Query Language

1. INTRODUCTION

During the last years, the Web has emerged as a vast collection of resources that provide information or data both in the form of (static) documents and in the form of services [19, 22, 5]. Companies and individuals publish content on websites, and major search engines store and index the contained information. Enterprises have adopted the Service Oriented Architecture (SOA) [14] and use Web services as the building block for loosely coupled distributed applications. Ongoing trends in Web-based data processing focus on combining data from different sources to provide a new functionality and to generate added value. This creates the demand for techniques to collect and transform data from related resources in preparation for a certain business application. Collection and transformation of data are collectively referred to as data aggregation [20]. In order to achieve scalability in both the number of data sources and the size of the provided documents, it is advisable to divide aggregation problems into a set of sub-tasks, which can be performed in parallel by individual nodes.

Web data aggregation has previously been tackled in various ways. In its simplest form, data is collected from a set of predefined, homogeneous, fixed-location sources and displayed to the user. For instance, an RSS reader client application is configured with a set of URLs, from which it retrieves news feeds (in a well-defined format) in regular intervals and checks for new and updated content. More sophisticated applications, in which users are able to query and filter the content they are interested in, include distributed RDF retrieval systems [1, 16], products and services indexing systems, or "Data-as-a-Service" [22] providers, which offer financial, demographic and other business relevant data, usually priced by the amount of queries or analytics run.

Aggregation of Web services data is sometimes achieved with languages from the Web service composition [7] domain (e.g., WS-BPEL). However, service composition usually targets the process aspect of composite applications and falls short of providing explicit support for composite data. As a concept that evolved from service composition, service mashups [6] are applications that scrape together data from one or more existing (external) sources in order to create a new functionality. Contradicting the SOA paradigm, service mashups are often tightly coupled: developers who manually combine Web APIs or use a mashup platform such as *WSO2 Mashup Server*[1] mostly hard-code the endpoints of participating data resources. This leads to the problem that the mashup definition may need to be frequently updated for dynamic scenarios with changing data providers.

We argue that there is still a lack of generic frameworks that support loosely coupled, distributed aggregation of heterogeneous data published in the Web. In this paper we present the WS-Aggregation framework as a generic solution to this problem. WS-Aggregation employs a set of intercon-

[1] http://wso2.org/projects/mashup

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

nected aggregator nodes, which cooperate with each other to execute client requests. Following the SOA paradigm, WS-Aggregation decouples the aggregation platform from the target data services. The endpoint information of the system participants is stored in the VRESCo service registry [12] and service selection and binding takes place dynamically at runtime.

The remainder of this paper is structured as follows. We define the addressed problem based on three illustrative scenarios in Section 2. Section 3 presents a formalization of our approach to aggregating Web services data. We discuss conceptual and architectural aspects in Section 4, and implementation details in Section 5. Section 6 covers a performance evaluation of data aggregation in different settings. Section 7 discusses existing related work, and Section 8 concludes the paper with an outlook for future work.

2. MOTIVATING SCENARIOS

The concepts of this paper will be discussed in the context of three illustrative scenarios, which involve collecting and processing of data from different sources:

1. A *portfolio analysis* system calculates the combined value and risk of a set of stocks or other financial instruments. To that end, historical prices are queried from data providers such as *xignite*[2] and *Yahoo*[3].
2. Hotels that participate in a *hotel booking* system provide a RESTful Web service [17] to check the availability of rooms. A booking platform enables users to search for free rooms among all hotels and returns the 20 best results (e.g., ordered by price or user rating).
3. In an online *voting* system (e.g., for an international song contest), users submit their vote to a regional voting Web service. The votes are aggregated to regional and national results and the overall result constitutes the majority decision in favor of a certain participant.

With the goal of creating a generic solution that fits all three scenarios, we identify and address the following main challenges.

Multi-site query vs single-site view: The client should be provided with a single query interface, whereas the execution comprises several aggregation steps (as in scenario 3) and involves numerous service endpoints. Aggregation requests specify which data should be queried from which source(s), and the aggregation platform takes care of the query distribution.

Heterogeneity: The query process involves various providers, which publish data sources with diverse data formats and protocols; scenarios 1 and 3 partly use SOAP Web services and partly simple Web documents, and the booking system in scenario 2 uses RESTful services.

Variable input data: We assume that the services of scenarios 2 and 3 each receive the same input, whereas different requests (with different stock ticker symbols) have to be issued in scenario 1 in order to retrieve the prices of all stocks in the portfolio.

Self-Adaptation: Particularly in scenario 2, new participants (hotels) may dynamically join the system and existing services may be removed. The aggregation platform should be flexible and able to adapt to these environment changes.

[2]http://xignite.com

[3]http://finance.yahoo.com

Performance: All three scenarios demand for scalability with respect to 1) the number of involved target services, 2) the size of the transferred documents and 3) the number of parallel aggregation requests.

3. AGGREGATION MODEL

In the following we present a formalization of our approach of Web services data aggregation, which serves as the basis for the discussion of WS-Aggregation. We make use of the VRESCo service model [12], which uses the notion of *Services*, *Operations* and *Features*. A feature is an abstract description of a capability (e.g., `GetStockPrice` with inputs `symbol`, `startDate`, `endDate`), which is implemented by the operations of one or more concrete Web services (e.g., the finance services of *xignite* and *Yahoo*). In our formalization, F_{all} denotes the set of available features, S_{all} is the set of services entered into the VRESCo service registry, $\mathcal{P}(O_{all})$ is the power set of available service operations, and the function $o : F_{all} \rightarrow \mathcal{P}(O_{all}), f \mapsto \{o_{(f,1)}, ..., o_{(f,n)}\}$ maps features to implementing service operations. The function $s : O_{all} \rightarrow S_{all}$ returns the service to which a specific operation belongs.

The problem addressed in this paper is to aggregate (i.e., request, collect and process) XML-formatted data from a set of service operations $O_\alpha = \{o_{(f_1,1)}, .., o_{(f_1,n_1)}, \ldots, o_{(f_m,1)}, .., o_{(f_m,n_m)}\}$, which implement features $F_\alpha = \{f_1, \ldots, f_m\}$, over a single-site query interface. We define an execution of such an aggregation as a function $\alpha : (InputsMap, WAQL, Topology) \rightarrow XML, (i, q, t) \mapsto r$ that receives a certain input, queries data from all relevant services and produces a resulting XML document r. The inputs which are used for issuing the service requests are defined via the input function $i : F_\alpha \rightarrow \mathcal{P}(Input), f \mapsto \{i_{(f,1)}, ..., i_{(f,n)}\}$. The inputs may be encoded either in XML (for SOAP and RESTful Web services) or using URL encoding (for HTTP requests to websites), and contain an arbitrary number of HTTP headers. The variable q denotes a WAQL (Web service Aggregation Query Language) query expression. WAQL is a query language based on XQuery [21], which is used to select and transform elements from the individual service results to construct the final aggregation result. The features of WAQL will be discussed in Section 4.1. Finally, the parameter t identifies the topology context information required for distributed computation of the result. More details on aggregation topologies is given in Section 4.2.

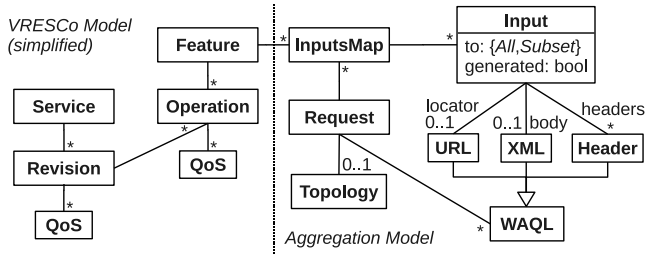


Figure 1: Aggregation Request Model

Figure 1 illustrates a simplified version of the VRESCo service model and the data aggregation model (separated by a dashed line). Note that VRESCo models service versioning using a *Revision* element, which we do not consider in our formalization. One aspect that requires closer con-

sideration is the way in which clients provide the input for an aggregation α, which requests data from the operations O_α, which implement the features $F_\alpha = \{f_1, \ldots, f_m\}$. Let us consider some input $i_x \in i(f_1)$ specified by the function i for the feature f_1. We distinguish the following input types:

- *Fixed* versus *Generated*: Fixed input is provided as either XML markup or a URL encoded query string. On the other hand, input messages in WAQL can also be generated, which is useful if the sub-requests only slightly differ from one another (e.g., requests for different stock symbols in the portfolio example). In this case, the inputs are specified using a WAQL expression, which, upon execution (χ), generates the actual input messages $\chi(i_x) = \{i_x^1, ..., i_x^k\}$.
- *ToAll* versus *ToSubset / ToOne*: This distinction concerns the input distribution strategy. If the input i_x is tagged *ToSubset=num*, it is sent to a subset of *num* (randomly selected) service operations $O_{i_x} \subseteq o(f_1)$, $|O_{i_x}| = num$, which implement the feature f_1. *ToOne* is a special case of *ToSubset*, where $num = 1$. Conversely, if the input i_x is of type *ToAll*, a request with this input is sent to all service operations which implement the feature f_1, i.e., $O_{i_x} = o(f_1)$. Use cases for *ToAll* and *ToOne* are obvious, and *ToSubset* can be used in scenarios where data should be queried redundantly from multiple sources to enforce data integrity.

All combinations of the above variants are possible. For instance, the portfolio scenario generates requests for a subset of service operations implementing a certain feature (*generated, ToSubset*), and the voting scenario uses a fixed input to retrieve votes from all voting services (*fixed, ToAll*). Section 4.1 provides more details and presents a sample generated input. The purpose of *ToAll* and *ToSubset* is further discussed in Section 4.2.

4. WS-AGGREGATION FRAMEWORK

As a generic solution for the scenarios presented in Section 1, WS-Aggregation constitutes a data aggregation platform that allows clients to run multi-site queries of heterogeneous XML data sources against a single-site interface. The coarse-grained architecture is depicted in Figure 2.

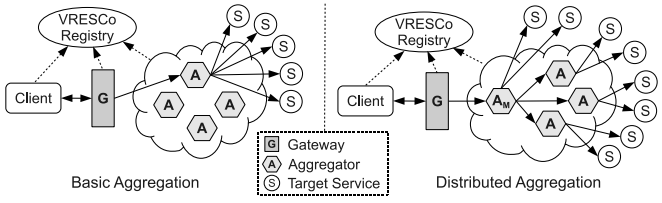


Figure 2: WS-Aggregation Architecture

The *Gateway* (G) is a Web service that acts as the single endpoint the clients communicate with. A number of *Aggregator* (A) nodes serve incoming aggregation requests. The aggregators are depicted in a cloud because they are invisible to the clients and may be added to and removed from the system transparently. Besides basic aggregation, WS-Aggregation focuses on distributed construction of the final result. As illustrated in Figure 2, distributed aggregation makes use of a tree topology among aggregator nodes. The root node in the topology is denoted *master* aggregator, A_M. Each aggregator retrieves data from one or more target services and passes the collected, intermediate results on to its parent node. One reason for distributed aggregation is to reduce the load imposed on a single aggregator node; however, WS-Aggregation also considers other distribution strategies (e.g., location-based), which are further discussed in Section 4.2.

WS-Aggregation follows the SOA paradigm and focuses on the use of a service registry, which fosters a decoupled and flexible architecture. The gateway, aggregator and target services are published in the VRESCo service registry. Clients use the registry to discover the gateway instance, and the gateway finds all active aggregator nodes contained therein. New aggregator nodes can be seamlessly integrated and existing nodes can be taken off the system. The aggregators themselves determine the endpoints of the data services, for which purpose the registry allows to query services by feature. Faulty or nonresponsive aggregators are automatically unregistered when detected by the gateway or a partner aggregator. Furthermore, the VRESCo registry stores metadata such as QoS or physical location of services, which serves as the decision basis for query distribution (see Section 4.2).

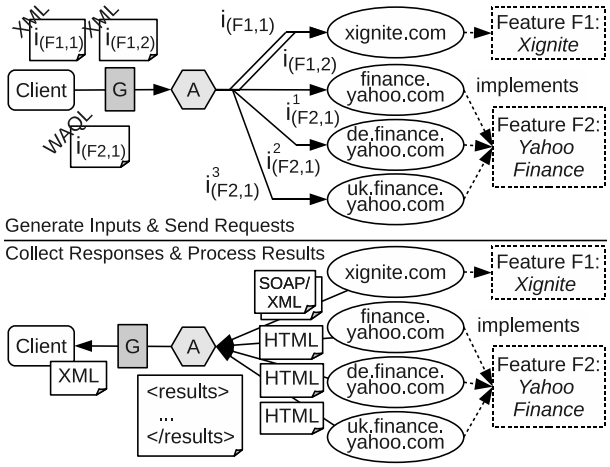


Figure 3: Aggregation in Portfolio Scenario

An aggregation example taken from the portfolio scenario is depicted in Figure 3. The client provides two fixed inputs (XML data), which target the *xignite* market data Web service, and one input in the form of a WAQL expression, which is targeted at the *Yahoo Finance* feature. The WAQL input defines the rules to generate the actual inputs that are sent to the target services. The results of the service invocations is a mix of SOAP messages and HTML pages, which are combined into a single XML document and sent back to the client.

4.1 Requesting and Querying Data with WAQL

In the following we present WAQL, the *Web service Aggregation Query Language* used to formulate aggregation requests in WS-Aggregation. WAQL is used both for 1) defining templates and rules for constructing generated request inputs as defined in Section 3, and 2) expressing how the response documents should be combined and transformed into the final result. WAQL is based on the XML query language

XQuery, which provides powerful functions to arbitrarily select, transform and generate nodes in XML documents. Owing to its power and universality, XQuery expressions tend to grow big and complex even in small scenarios. Hence, WAQL extends XQuery by a set of convenience language constructs which we deem important in the context of Web service data aggregation.

4.1.1 Templates for Generated Inputs

WAQL simplifies the specification of generated inputs that have a similar structure. The WAQL query in Figure 4 is used to generate requests for the *xignite* Web service (portfolio sample). The user-specified input contains three lists (starting with a dollar sign, '$'), which represent alternative values that should appear in the generated inputs.

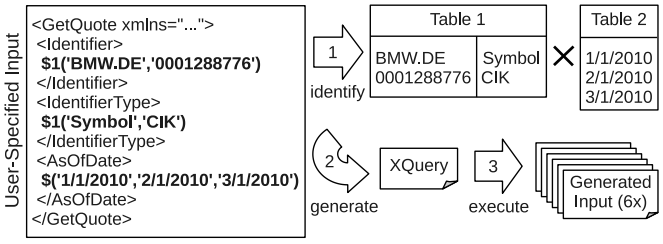


Figure 4: WAQL Input Preprocessing

The optional numeric identifier after the dollar sign links lists that are correlated, which means that the lists' items at the same index appear together in the generated inputs. In the example, `BMW.DE` is the ticker *Symbol* of BMW whereas `0001288776` is the *CIK* code (Central Index Key) of Google, Inc. For these values to appear in pairs, the two lists are linked to each other using the identifier '`1`'. Using this very light-weight syntax, value lists can be unambiguously parsed and detected inside an XML or XQuery expression. During query preprocessing, all such lists are firstly identified and grouped. Then an XQuery *FLWOR* (acronym for *for,let,where,order by,return*) expression is generated, which, upon execution, creates all possible combinations (see Section 5 for implementation details). Besides simple values such as strings or numbers, the lists may also contain complex XML elements.

4.1.2 Expressing Data Dependencies

Each aggregation query is divided into a number of subrequests that target different data services. WAQL provides the possibility to express data dependencies between subrequests. Each sub-request has a numeric identifier that is used as a reference. For instance, an occurrence of the string `$3{/foo}` (note the curly brackets) in the input of a sub-request `1` signifies that the element matching the XPath `/foo` needs to be extracted from the result of sub-request `3` and inserted into the input of `1`. WS-Aggregation constructs a dependency graph and ensures 1) that dependent requests are executed in the right order and 2) that independent requests are executed in parallel. If the identifier is left out (e.g., `${/foo}`) the element is nondeterministically extracted from the first matching result of any sub-request. The framework is able to identify circular and unresolvable dependencies. The former is checked by analyzing the query's dependency graph up front, and the latter occurs if no element can be selected to satisfy a depencency during execution.

4.1.3 Integration of Non-XML Data Sources

WAQL offers a number of data conversion functions to specify the result output format and to integrate non-XML data formats into WS-Aggregation. E.g., comma-separated values (CSV) files are a common means to encode spreadsheet data. The `csvToXML` function takes as argument a CSV-encoded string and transforms it to an XML "table" structure (i.e., `<row>` elements containing `<column>` subelements with text content). Such structures are rendered as HTML tables using `toHTMLTable`. Conversely, `xmlToCSV` converts table-structured XML to CSV. JSON (JavaScript Object Notation) is a light-weight data-interchange format natively supported by JavaScript, often used to retrieve data in dynamic Web 2.0 applications. A mapping from JSON to XML is achieved using the function `jsonToXML`. Moreover, a huge number of non-XML-compliant HTML pages exist across the Web, which can be converted to XML via `htmlToXML` to be XQuery-processable. The more exotic function `bibtexToXML` allows conversion of BibTeX files to XML.

4.2 Aggregator Topologies

In this Section we discuss the aggregator topologies used in WS-Aggregation. Apart from the general load distribution aspect, distributed aggregation is also useful from other viewpoints. Examples range from aggregators responsible for data nodes in different regions (see voting scenario), to specialized aggregators (e.g., high-performance server for large data conversions or to enforce WS-Security), to scenarios with access restrictions or trust issues.

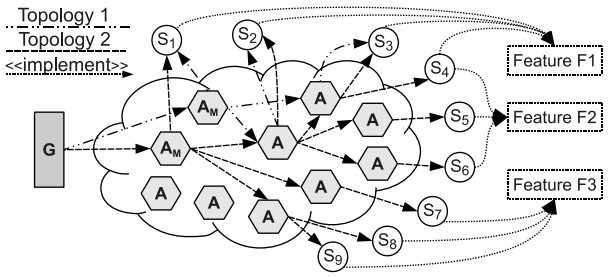


Figure 5: Aggregator Network Topologies

Each aggregation request α is associated with a *topology* (t) that determines how the query execution is distributed among the aggregator nodes $A_{all} = \{A_1, ..., A_n\}$. The topology specifies 1) the subset of aggregator nodes $A_t \subseteq A_{all}$ that are responsible for performing an aggregation α, 2) which partner (child) nodes an aggregator can rely on, and 3) which data service nodes each aggregator is responsible to collect data from. Figure 5 depicts an exemplary distributed aggregation scenario with two topologies. Topology 1 involves three aggregators and retrieves data from three (s_1, s_2, s_3) of the four (s_1, s_2, s_3, s_4) services providing feature *F1* (*ToSubset=3*), whereas topology 2 includes all services implementing features *F1*, *F2* and *F3* (*ToAll*). The master aggregator of each topology is denoted A_M. We distinguish three types of topologies (*basic, predefined, adhoc*), which are discussed in the following three paragraphs.

Basic: Each aggregation request is handled by exactly one aggregator. The single responsible aggregator is selected by the gateway at runtime according to a load distribution strategy. The strategies include random choice, round-robin

delegation and metadata based selection (taking into consideration the current load of the aggregators in terms of request queue length, CPU usage and memory consumption). This approach has the smallest overhead since no inter-aggregator communication takes place, but obviously it lacks advantages such as location- or performance-based aggregation distribution.

Predefined: Prior to issuing a data aggregation query, clients request the construction of a new topology with certain characteristics, including the required service features F_t and the size and structure of the topology (e.g., 7 aggregators in a tree of height 2 and branching factor 2). The Gateway then selects the subset of aggregators A_t, retrieves the endpoints of the target services $S_t = \bigcup_{\forall f \in F_t, o_f \in o(f)} s(o_f)$ and computes a suitable topology model, which maps aggregators to partner nodes ($A_t \rightarrow A_t$) and aggregators to target services ($A_t \rightarrow S_t$). The gateway passes this information to the affected aggregator nodes along with a unique topology identifier (ID). Finally, the client uses the topology ID in subsequent requests. Advantages of predefined topologies are the fast lookup of responsible aggregators and the reusability aspect.

Ad-Hoc: As opposed to predefined topologies, ad-hoc topologies are dynamically created when handling a request. The gateway selects a master aggregator just as with basic aggregation. However, in this case the master may delegate (parts of) the request to partner aggregators. We distinguish different ad-hoc distribution strategies. *Self-centric* strategies consider the current state of the aggregator and delegate requests if, for instance, the own request queue grows too large or if the memory or CPU usage exceed a certain threshold. *Request-centric* approaches analyze the request itself and determine whether it contains inputs or WAQL queries that should be processed by certain nodes (e.g., a powerful aggregator machine should convert large CSVs to XML). Finally, A *target-centric* strategy analyzes the number and characteristics (e.g., location) of the target data service nodes. For instance, the system may be configured with a maximum number of target services that each aggregator invokes, before the request is split up and delegated.

Implementation details concerning topology construction are discussed in Section 5. Basically, each aggregator is configured with a list of distribution strategies, which split aggregation requests into atomic parts. For instance, consider an aggregator with three distribution strategies (*S1,S2,S3*). $S1$ (predefined topology strategy) assigns request parts to the predefined targets in case the client request contains a topology ID; if no ID is present, the request is passed on to $S2$ (location-based), which filters parts that need to be routed via a specific aggregator (see voting scenario); the remaining parts are handed to $S3$ (QoS-based) and are either dealt with immediately or delegated to partner aggregators.

4.3 3-way Result Construction

In many data aggregation scenarios the client is interested only in a small part of the data. For instance, the portfolio system calculates the average price of a stock, whose historical prices are contained in HTML tables on different pages. First collecting all documents and then applying the `avg` function at A_M is costly, since large amounts of data need to be transmitted between the aggregator nodes (indicated with thick lines in Figure 6). Therefore, WS-Aggregation offers the possibility to apply 3 queries successively.

- The *Preparation Query* is applied instantly to every document an aggregator receives from a target service.
- The *Intermediate Query* is applied to every partial result computed by an aggregator, prior to sending the data to the parent aggregator.
- The *Final Query* is applied before the master aggregator returns the collected data to the gateway.

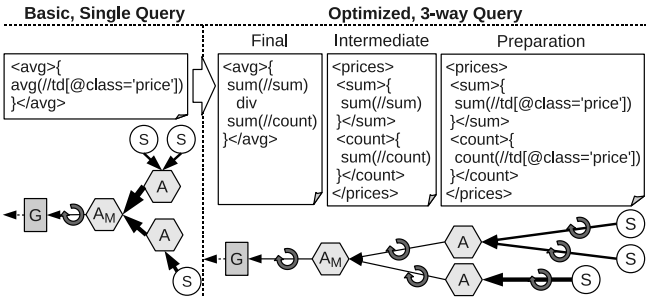


Figure 6: 3-way Result Construction

As can be seen from the example, the 3-way querying has two key advantages. Firstly, building partial aggregation results can take some load off the inter-aggregator network. Secondly, the size of the data, to which the XQuery is applied, gets reduced, which decreases the memory footprint of the aggregator nodes, particularly the master aggregator. In Section 6 we compare the performance of both approaches.

5. IMPLEMENTATION

In this section we discuss the prototype implementation of WS-Aggregation. The system components (client, gateway, aggregator) are implemented in Java, the interfaces of gateway and aggregators are accessible as SOAP Web services. The implementation of the gateway comprises mainly load-balancing, i.e., assigning requests to master aggregator nodes according to a configured strategy. The core component is the aggregator node, whose internal structure is depicted in Figure 7.

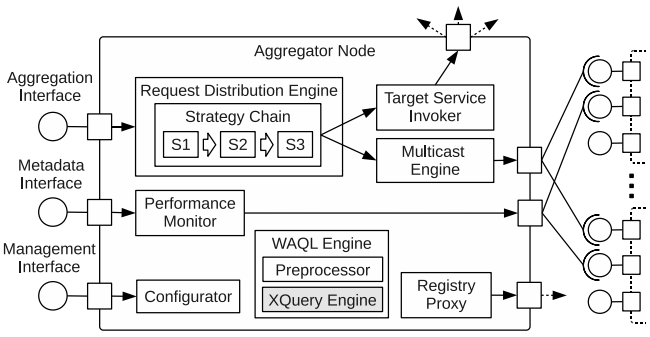


Figure 7: Structure of Aggregator Nodes

The aggregator receives requests via the *Aggregation Interface* and delegates them to the *Request Distribution Engine*. A *Strategy Chain* allows the aggregator node to be configured with pluggable distribution strategy components (*S1,S2,S3* in Figure 7), which inspect the request and split it into atomic parts. Atomic requests are either passed to the *Target Service Invoker*, or delegated to partner aggregators

via the *Multicast Engine*. The Multicast Engine itself makes use of the Aggregation Interface of partner aggregators.

The *Preprocessor* of the *WAQL Engine* detects and processes WAQL-specific constructs in aggregation queries. To that end, first the data dependencies in a request need to be resolved and replaced with actual values obtained during the query execution. Note that a complete discussion of WAQL data dependencies is out of the scope of this paper. As discussed in Section 4.1.1, generated inputs are then transformed into valid XQuery expressions, which get executed by the *XQuery Engine* (third-party engine from Saxon[4]). To facilitate the actual parsing of WAQL requests, we defined the syntax rules using EBNF (Extended Backus Naur Form) and make use of *JavaCC*[5], a parser generator for Java.

The *Registry Proxy* queries and caches service metadata from the VRESCo service registry. The *Performance Monitor* continuously measures the system performance (memory/CPU usage, request queue length, ...). The total memory used by the Java Virtual Machine (JVM) is determined via `java.lang.Runtime`. Our implementation further makes use of the `java.lang.management` API, which allows CPU usage inspection on a per-thread basis. The Performance Monitor also retrieves the performance data of all other deployed aggregators via their *Metadata Interface*. This information is used for ad-hoc load balancing and query distribution. Via the *Management Interface* authorized clients are able to access the *Configurator*, which allows online modification of the aggregator's behavior (e.g., strategy chain, maximum queue length, resource limits, etc). A JavaScript-based Web 2.0 user interface (UI) serves as a convenient way to construct, execute and save aggregation queries. The Web UI uses the *XMLHttpRequest* object to exchange SOAP messages with the gateway, and displays the aggregation result as XML source code or rendered HTML.

6. EVALUATION

To measure the performance of WS-Aggregation we have set up a comprehensive test environment, which executes the Web service data aggregation scenarios discussed in Section 2. The main aspects of our evaluation are 1) query distribution, 2) parallel requests and 3) query optimization. The scenarios are implemented in a configurable manner to support testing in different settings (e.g., used aggregation strategy) and sizes (e.g., number of used aggregators, target services, parallel requests, ...). The key points describing the three scenarios are summarized below.

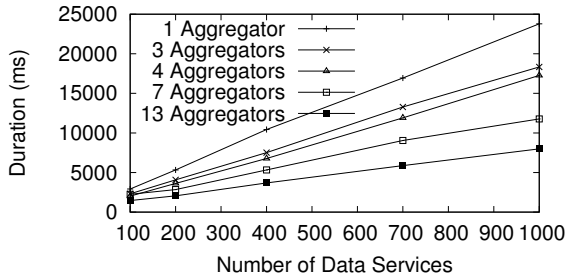
portfolio: The client provides a list of stock symbols to request the historical prices from finance Web services. The result contains the sum, count and average of each stock's prices. The test is implemented with both a non-optimized and an optimized 3-way query (see Section 4.3).

hotel booking: This test receives available rooms from hotel booking services. Each service returns 100 entries, results get sorted by price, an *intermediate query* is used to transmit only the (locally) best 20 rooms between aggregators.

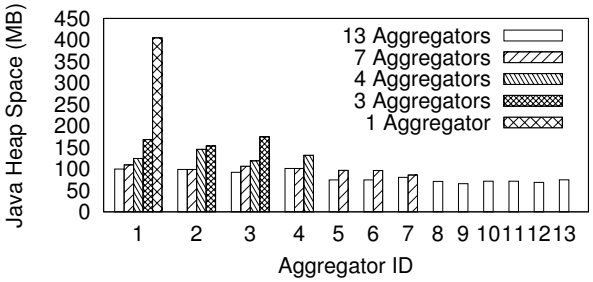
voting: The system receives voting points (1-10) for 10 candidates from voting services. We simulate 1) regional aggregators that collect votes from their region, 2) national aggregators which process the results from a nation's regions and 3) a master aggregator which combines all the data.

[4]http://saxon.sourceforge.net/

[5]https://javacc.dev.java.net/



(a) Aggregation Time



(b) Memory Consumption

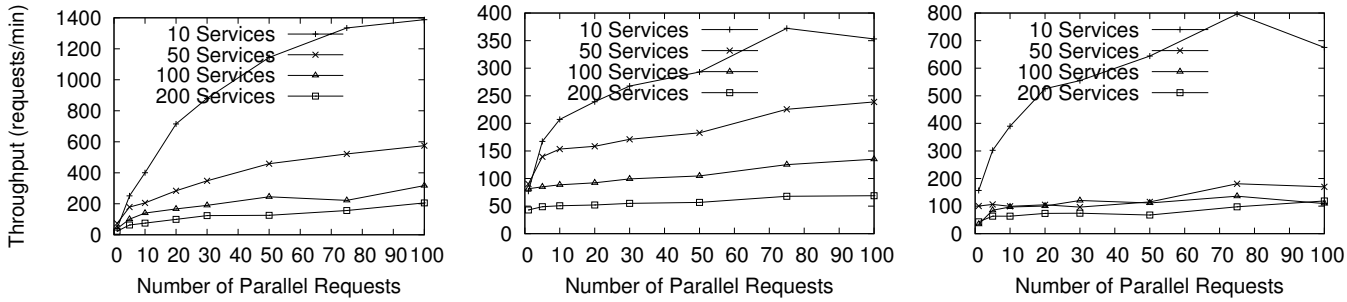
Figure 8: Performance of *Voting* Scenario

We launched 20 *Amazon EC2*[6] cloud instances (type 'S') and deployed 20 aggregator nodes (one per instance) and 1000 data services (50 per instance), which provide the functionalities of the three test scenarios. For the portfolio scenario, we simulated the behavior of the *Yahoo Finance* and *xignite* services in the test environment. All tests have been run 20 times, and the numbers in the following represent mean (execution time) and maximum (memory) values.

First we evaluate the performance improvement achieved by distributing the aggregation query in the voting scenario. Figures 8(a) and 8(b) show the aggregation duration and the maximum level of allocated Java heap space, respectively. The number of services stored in the service registry has been gradually increased (100,200,400,700,1000). The test uses different predefined tree topologies t with branching factor $b \in \{2, 3\}$ and height $h \in \{0, 1, 2\}$, resulting in a total number of aggregators $|A_t| \in \{1, 3, 4, 7, 13\}$. The results indicate that the execution time decreases with increasing number of involved aggregators, particularly in scenarios where data from many services is requested. Furthermore, the memory usage per aggregator is cut from more than 400MB (1 aggregator) to roughly 100MB (13 aggregators).

In Figure 9, basic (a), predefined (b), and ad-hoc (c) topologies show different performance characteristics when it comes to parallel requests. The best overall results are in (a), where each request is handled by a single aggregator. In (b) we used a predefined topology of height 1 and branching factor 3. The results indicate that predefined topologies perform worse than (a) but are more stable and predictable than (c). With ad-hoc topologies in (c), each aggregator handles a maximum of 50 data service requests per aggregation request and delegates the remaining parts to partners, based on their current load (memory usage, queue length) and a random factor. Ad-hoc topologies uniformly distribute the system load, but entail an overhead for inspecting requests

[6]http://aws.amazon.com/ec2/



(a) Basic (b) Predefined (c) Ad-Hoc

Figure 9: Performance of *Booking* Scenario – Different Topology Types

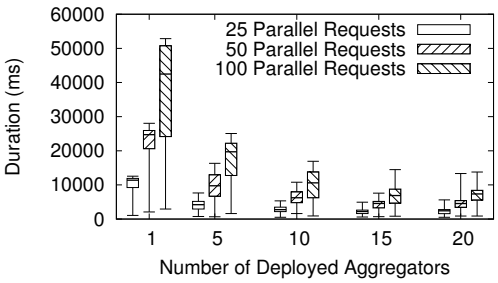
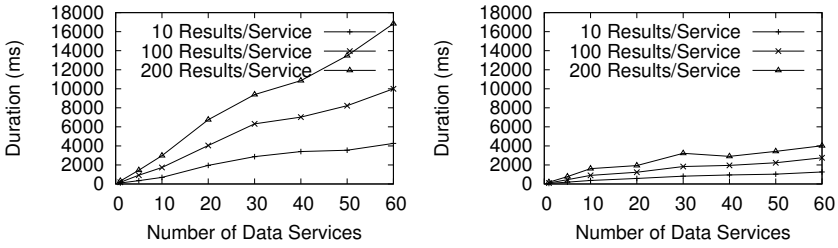


Figure 10: *Booking* – Nr. of Aggregators



(a) Unoptimized (b) Optimized

Figure 11: *Portfolio* Scenario – Unoptimized versus Optimized

and keeping aggregator performance lists up to date.

Figure 10 shows a boxplot of the runtime of the booking scenario, using 50 services and basic topologies. As the number of deployed aggregators increases (1,5,10,15,20), maximum and median decrease, especially when switching to 5 and to 10 aggregators. The minimum value stays near-constant since the requests that make it through first are executed fast and further requests are put to a queue.

Figure 11 compares the portfolio scenario execution time with a topology of 13 aggregators and using (a) the unoptimized and (b) the optimized query. The difference is astoundingly high (up to 16 seconds versus 4 seconds for 60 data services, which each produce 200 stock price results) and indicates that XQuery-processing and transmitting large documents between aggregators consume a large portion of the total aggregation time.

To sum up the individual results of the experimentation, WS-Aggregation provides good scalability with respect to both the size of aggregation requests and the number of parallel executions. Distribution of the query execution to several aggregators yields a performance gain, although the distribution strategies show different characteristics and should be tailored to the aggregation problem. The capability of 3-way querying helps the user to drastically reduce the execution time and the load for the inter-aggregator network.

7. RELATED WORK

A number of related approaches to Web service aggregation [10] and Web data integration [11] have been proposed. For instance, [9] presents an algorithm to aggregate data from different Web services. Whereas their approach attempts to perform schema matching among different service results, WS-Aggregation is a generic framework in which user-defined XQuery expressions are used to create an integrated view of different Web data sources. The authors of [3] present *DeXIN*, an XQuery-based framework for integration of Web data sources. Despite some obvious similarities to our approach, DeXIN targets the different goal of integrating SPARQL into XQuery in order to query RDF and OWL documents. WS-Aggregation, on the other hand, uses only structural and no semantic queries and focuses on distributed, cooperative execution of queries. In [15], the idea of ontology mediated Web service aggregation hubs (OMWSAH) is presented. An OMWSAH retrieves data from target services as well as from other OMWSAHs, similar to aggregator nodes in WS-Aggregation. In contrast to our solution of a single interface for generic queries, an OMWSAH dynamically alters its WSDL interface to cover all operations provided by the target data services. *Lixto* [8] is a platform for scalable Web data extraction processes. The logic-based language *Elog* is the basis for a visual specification framework to express data extraction processes. Lixto differs from our approach in that it focuses mainly on HTML documents, analyzes the semantics of documents (e.g., providing functions to determine whether a text node contains a date or currency) and provides no control over distributed execution of data extraction processes.

Besides approaches in academia, a number of commercial data aggregation platforms exist. The *ecrion* Data Aggregation Server[7] allows an integrated view of an organization's heterogeneous data sources. Ecrion is more focused on the visual mapping of data sources and lacks query distribution and scalability. *Informatica*[8] employs a multi-step data aggregation platform including validation, normalization, mapping and other features. Also *Stylus Studio* sup-

[7]http://www.ecrion.com/Products/DAS/Overview.aspx
[8]http://www.informatica.com/solutions/data_aggregation

ports XQuery based aggregation of Web data [18], based on a graphical mapper for XQuery expressions. All mentioned platforms support Web services (SOAP and REST), and Informatica and ecrion integrate a palette of non-XML documents. None of the platforms support strategy-based query distribution, generated inputs, and SOA-based loose coupling as provided by WS-Aggregation.

Different approaches exist in the area of Web mashups, which aim at creating *"new Web applications by combining existing Web resources utilizing data and Web APIs"* [4]. The authors of [6] perform a thorough analysis of different mashup platforms such as Microsoft Popfly[9] and Yahoo Pipes[10]. The Enterprise Mashup Markup Language[11] (EMML) is an effort to unify mashup in a single domain-specific language (DSL). In contrast to WS-Aggregation, which solely relies on XQuery for specifying how the aggregation should take place, EMML is a rather heavy-weight language with resemblance to service composition languages such as WS-BPEL. Even though EMML programmers may specify activities that can be executed in parallel, the specification does not define how data aggregation is performed collaboratively across several nodes. Furthermore, as opposed to most mashup tools, WS-Aggregation supports loose coupling because the mapping between features and service operations is defined in the VRESCo registry and endpoints are dynamically exchangeable.

Wireless Sensor Networks (WSNs) [13, 2] build a network topology among individual sensor nodes, which monitor physical or environmental conditions. The data generated from the individual source nodes is collected, aggregated and sent to one or more sink nodes. The data flow occurs mainly event-based in the sense that sensor nodes send data to the sink in regular intervals or when changes in the monitored environment occur. This is adverse to WS-Aggregation, where the single data sources do not send data themselves, but are queried upon request.

8. CONCLUSION

We have addressed the problem of distributed aggregation of Web services data. Based on three motivating scenarios and a formal description of the aggregation model in our approach, we presented the architecture and prototype implementation of WS-Aggregation. The discussed highlights of the framework include the specialized query language WAQL, distributed query execution using configurable aggregator topologies, and user-defined query optimizations. The extensive evaluation targets the main aspects of 1) aggregation query distribution, 2) parallel requests and 3) query optimization. Our ongoing work focuses on modularization and reuse of aggregation queries and on data dependencies between sub-requests. Moreover, we are working on aggregation query patterns and new features for the WAQL query language.

9. REFERENCES

[1] G. Adamku and H. Stuckenschmidt. Implementation and Evaluation of a Distributed RDF Storage and Retrieval System. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 393–396, 2005.

[2] K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325 – 349, 2005.

[3] M. I. Ali, R. Pichler, H. L. Truong, and S. Dustdar. Dexin: An extensible framework for distributed xquery over heterogeneous data sources. In *11th International Conference on Enterprise Information Systems*, pages 172–183, 2009.

[4] D. Benslimane, S. Dustdar, and A. Sheth. Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[5] A. Dan, R. Johnson, and A. Arsanjani. Information as a service: Modeling and realization. In *SDSOA*, 2007.

[6] G. Di Lorenzo, H. Hacid, H. Paik, and B. Benatallah. Data integration in mashups. *ACM SIGMOD Record*, 38:59–66, 2009.

[7] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[8] G. Gottlob et al. Lixto data extraction project: back and forth between theory and practice. In *Symposium on Principles of Database Systems*, pages 1–12, 2004.

[9] E. Johnston and N. Kushmerick. Web service aggregation with string distance ensembles and active probe selection. *Information Fusion*, 9(4), 2008.

[10] R. Khalaf and F. Leymann. On Web Services Aggregation. In *Technologies for E-Services*, 2003.

[11] M. Lenzerini. Data integration: a theoretical perspective. In *Symposium on Principles of Database Systems*, 2002.

[12] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3(3):193–205, 2010.

[13] E. Nakamura, A. Loureiro, and A. Frery. Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Computing Surv.*, 39, 2007.

[14] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40, 2007.

[15] G. Piccinelli et al. Dynamic service aggregation in electronic marketplaces. *Computer Networks*, 37, 2001.

[16] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *European Semantic Web Conference*, 2008.

[17] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly, 2007.

[18] Sonic Software. Building XQuery with Stylus Studio: Web Service Aggregation and Reporting. http://www.stylusstudio.com/whitepapers/building_xquery_with_stylus.pdf, 2003.

[19] H.-L. Truong and S. Dustdar. On analyzing and specifying concerns for data as a service. In *Asia-Pacific Services Computing Conference*, 2009.

[20] R. van Renesse. The importance of aggregation. In *Future Directions in Distributed Computing*, 2003.

[21] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2007.

[22] F. Zhu et al. Dynamic Data Integration Using Web Services. In *Int. Conference on Web Services*, 2004.

[9] http://www.popfly.com (discontinued in August, 2009)
[10] http://pipes.yahoo.com
[11] http://www.openmashup.org/omadocs/v1.0