Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications

Valeria Cardellini, Stefano Iannucci
DISP, University of Roma "Tor Vergata", 00133 Roma, Italy
{*cardellini, iannucci*}*@ing.uniroma2.it*

***Abstract*—One of the major current trends in service-oriented systems is the emphasis given to the need of introducing runtime adaptation features, so that the system can meet its QoS requirements in a volatile operating environment. In this paper we present the design and implementation of a service broker that supports the QoS-driven runtime adaptation of SOA applications offered as composite services to users. We describe the functionalities provided by the broker components and present their design and implementation according to two different versions we have developed and that are both based on open source products. The components of the first version have been developed in Java as Web services, while the second version takes advantage of OpenESB. Since the broker needs to sustain a traffic of requests generated by several concurrent users, we also present the replicated architectures of the two broker versions. We discuss the design tradeoffs and the lesson we have learned in developing the broker.**

***Keywords*-Service-oriented architecture; quality of service; runtime adaptation.**

I. Introduction

The introduction of self-adaptation and self-management techniques may significantly improve service-oriented systems, because such techniques can help tackle the increased complexity of the systems themselves and of their environment [1]. Specifically, runtime adaptation features can allow a SOA-based system to meet its quality of service (QoS) requirements even when operating in highly changing and evolving environments. Being able to effectively guarantee the QoS agreements at runtime is an important concern for the provider of a SOA application, because it may bring competitive advantage over other providers. Since a SOA application is constructed by orchestrating network-accessible services offered by a multitude of third-party service providers, the dynamic binding to the component services is a crucial task. Indeed, several competing services may coexist implementing the same functionality (we refer to the former as *concrete services* and the latter as *abstract service*) but with different QoS performance attributes (e.g., response time and reliability) and cost.

In this paper, we present the architecture and detailed design of the MOSES prototype. MOSES, which stands for *MOdel-based SElf-adaptation of SOA systems*, is a runtime adaptation framework for a SOA-based system architected as a service broker. The methodology at the basis of MOSES has been presented in [2]; we briefly review its distinguishing features prior to present the MOSES prototype.

The MOSES aim is to drive the self-adaptation of the SOA system it manages to fulfill non-functional QoS requirements of the composite service, such as its response time, availability, and cost. The QoS contracted by the users and the composite service provider must meet certain respective obligations and performance expectations which the parties agree upon in a Service Level Agreement (SLA). The distinguishing features of the MOSES approach to adaptation can be summarized as follows [2]. First, MOSES performs a *per-flow* adaptation to reduce the computational load under a sustained traffic scenario. Such approach jointly considers the aggregate flow of requests; to the contrary, most of the proposed adaptation methodologies (e.g., [3], [4]) deal with single requests to the SOA application, which are managed independently one from another. The second characteristic of MOSES regards the mechanisms used to perform the adaptation, because it combines *service selection* with *coordination pattern selection*. The goal of the first mechanism is to identify at runtime for each abstract service a corresponding concrete service, selecting it from a set of candidates (e.g., [3], [4]). To increase the offered QoS, MOSES also exploits the coordination pattern selection [2], which allows to bind at runtime each abstract service to a set of functionally equivalent concrete services, coordinating them according to some redundancy pattern.

The two major goals of the service broker supporting the MOSES methodology are: (1) its ability to manage in an adaptive and flexible manner the concrete services in such a way to guarantee for the SOA application the QoS parameters agreed in the SLAs with the users; (2) its scalability and reliability, because the broker needs to sustain a traffic of requests generated by several users without service discontinuity. To achieve the first goal, we have designed the MOSES architecture as an instantiation for the SOA environment of a self-adaptive software system, where the software components are organized in a feedback loop aiming to adjust the SOA system to changes during its operation. We present two implementations of the MOSES architecture: in the first version the MOSES components have been developed in Java as Web services, while the second version is based on OpenESB. To achieve the scalability and reliability goal, we have designed the replicated architecture of the two broker versions.

The MOSES architecture is inspired by existing implementation of frameworks for Web services QoS brokering

978-0-7695-4128-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ICWS.2010.77

(e.g., [5]–[7]). Menascé et al. have proposed a SOA-based QoS broker for negotiating QoS goals [7] but their broker do not offer a composite service and its components are not organized as a self-adaptive system. PAWS [5] is a framework for flexible and adaptive execution of business processes but some of its modules work at design time, while MOSES adaptation operates only at runtime. Erradi et al. have presented a policy-based middleware that performs runtime monitoring and adaptation of Web service compositions [6]. Proxy-based approaches, similar to that used in MOSES for the dynamic binding to concrete services, have been previously proposed for re-binding purposes [8] as well as for handling runtime failures [9] in SOA applications.

The remainder of the paper is organized as follows. In Section II we present an overview of the MOSES architecture, outlining the main tasks of its components. We describe the design and implementation of two MOSES versions in Sections III and IV, also discussing some performance results. Finally, in Section V we point out the lesson learned from the development of the MOSES prototype and give some suggestions for future work.

II. Overview of the MOSES Architecture

MOSES is architected as a *service broker*, which offers to its users a composite service with a range of different service classes. It acts as an intermediary between users and concrete services, performing a role of service provider towards the users and being in turn a requestor to the concrete services used to implement the composite service. Its main task is to drive the adaptation of the composite service it manages to fulfill the SLAs negotiated with its users, given the SLAs it has negotiated with the concrete services. Moreover, it also aims at optimizing a given utility goal.

Figure 1 shows the core components of the MOSES high-level architecture and their interaction. In this section we provide a functional overview of the tasks carried out by the MOSES components, while in Sections III and IV we discuss in details their design and implementation.

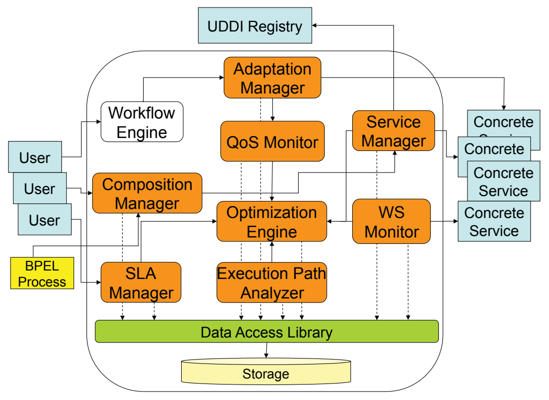

Figure 1. MOSES high-level architecture.

The *Composition Manager* is mainly responsible for the construction of the composite service model, which defines the set of abstract services forming the abstract composition, the set of concrete services, the parameters of the SLAs established by the broker with the users and the providers of the concrete services. Based on the workflow description in BPEL and with the cooperation of the Service Manager, the Composition Manager builds a behavioral model of the composite service and saves it in the storage layer to make it accessible to the other MOSES components. The Composition Manager is only invoked by those users who are responsible for MOSES administration and are allowed to publish new BPEL processes.

The *Workflow Engine* is the software platform executing the BPEL process and represents the user front-end for the composite service. It interacts with the Adaptation Manager to allow the invocation of the component services. For each invocation in the abstract composition (i.e., a BPEL `invoke` activity), the *Adaptation Manager* binds at runtime the request to the real endpoint that represents the abstract service. The real endpoint is identified by the solution of an optimization problem and can be either a single service instance or a subset of service instances coordinated through some pattern. The MOSES methodology (and therefore the prototype presented in this paper) currently supports the 1-out-of-n parallel redundancy and the alternate service coordination patterns [2]. With the former, the Adaptation Manager invokes the concurrent execution of the concrete services in the subset identified by the Optimization Engine, waiting for the first successful completion. With the latter, the Adaptation Manager invokes sequentially the concrete services in the subset, until either one of them successfully completes, or the list is exhausted. Therefore, in the envisioned architecture the Adaptation Manager is in charge of carrying out at runtime the adaptation actions.

The *Optimization Engine* is the MOSES component that solves the optimization problem, which is based on the behavioral model initially built by the Composition Manager and instantiated with the parameters of the SLAs negotiated with the composite service users and concrete services. The model is kept up to date by the monitoring activity carried out by the QoS Monitor, the WS Monitor, and the Execution Path Analyzer. The optimization problem is formulated as a Linear Programming problem and is therefore suitable to be solved at runtime because of its efficiency [2]. Its solution provides indications about the suitable adaptation actions that must be performed to optimize the use of the available concrete services with respect to the broker utility criterion, within the constraints defined by the existing SLAs.

The *QoS Monitor* collects information about the performance and reliability levels (specified in the SLAs) perceived by the users and offered by the concrete services. The *WS Monitor* checks periodically the responsiveness of the pool of operations. The *Execution Path Analyzer* keeps up to date information about the composite service usage profile by examining the business process executed by the Workflow Engine; specifically, for every task of the abstract composition it determines the expected number of times the task is invoked by each service class.

Besides maintaining up to date the parameters of the optimization problem, the QoS Monitor, WS Monitor, and Execution Path Analyzer check and notify whether some relevant change occurs in the composite service environment. Changes to be tracked include: the arrival/departure of a user, an observed variation in the SLA parameters of the concrete services, the addition/removal of a concrete service, and a variation in the usage profile of the tasks in the abstract composition. Upon receiving a notification of a significant variation of the model parameters, MOSES finds out whether an adaptation action must be performed. To this end, it builds a new instance of the optimization problem, with the new values of the parameters. Based on this solution, the Adaptation Manager issues suitable directives, so that future concrete instances of the composite service workflow will be generated according to these directives.

One of the *Service Manager* tasks is to identify through the UDDI registry the set of concrete services (including the parameters of their SLAs) implementing the required functionalities of the composition. Its other task, shared with the *SLA Manager*, regards the SLA negotiation processes in which the broker is involved as intermediary. Specifically, the Service Manager negotiates the SLAs with the concrete services, while the SLA Manager is in charge of the user SLA negotiation and registration, that is, it can add, modify, and delete SLAs and users profiles. The SLA negotiation process towards the user side includes also the admission control of new users; to this end, the Optimization Engine is invoked to evaluate the MOSES capability to accept the incoming user, given the associated SLA and without violating already existing SLAs. If the requesting user is admitted, the user registration process adds the user profile in the MOSES storage layer.

Finally, the *Data Access Library* is used by most of the components to access the model parameters of the composite service operations and environment (including the abstract services and the corresponding concrete services with their QoS values, and the solution of the optimization problem).

The MOSES architecture represents an instantiation for the SOA environment of a self-adaptive software system [1], focused on the fulfillment of QoS requirements. The architecture of an autonomic system consists of a set of managers and managed resources [10]. Each manager communicates with the resources through a sensor/actuator mechanism and the decision is elaborated using the so called MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) cycle. This loop collects information from the system, makes decisions and then organizes the actions needed to achieve goals and objectives, and controls the execution. Most of the MOSES components are organized according to the MAPE-K cycle. In particular, the QoS Monitor, WS Monitor, and Execution Path Analyzer components play collectively the Monitor and Analyze roles, the Optimization Engine plays the Plan roles, while the Adaptation Manager plays the Execute role. Finally, the Data Access Library collects the knowledge about the system and environment status.

III. MOSES Version 1

MOSES 1 components have been entirely developed in Java as Web services. Java, Apache Tomcat, and Apache Axis are the core technologies used in the implementation. Furthermore, we used the open source ActiveBPEL Engine by Active Endpoints to realize the Workflow Engine. At the time of MOSES 1 implementation, for the BPEL engine we have considered and tested both Apache ODE 1.2 and ActiveBPEL 5. We selected the latter mainly for two reasons: first, only ActiveBPEL was fully compatible with BPEL4WS 1.1 and WS-BPEL 2.0 specifications; additionally, in preliminary experiments we found that ActiveBPEL had better performance and offered a more detailed documentation.

A. MOSES 1 Components

Each component has been developed as a Web service and deployed inside an Axis2 engine. Figure 2 shows the architectural overview of MOSES 1. We note that MOSES components and ActiveBPEL actually reside on different instances of the Apache Tomcat servlet container. Specifically, MOSES components use Tomcat 6, while for ActiveBPEL we were constrained to Tomcat 5.5, because it is the most recent version supported by the BPEL engine. The two Tomcat instances increase the resource consumption and complicate the global management of the prototype.

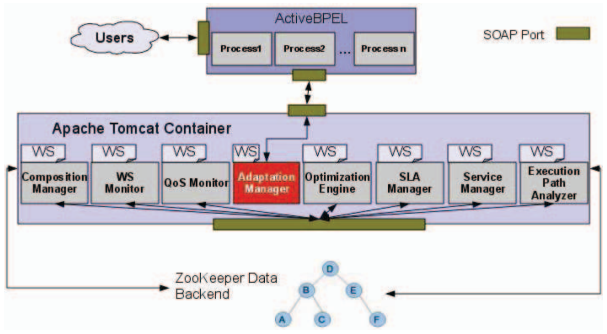

Figure 2. MOSES 1 architectural overview.

In MOSES 1, the request-response cycle begins with a request to a given business process deployed inside the BPEL engine; then, the invocations to the external concrete services involved in that process are captured by the Adaptation Manager, which applies one of the coordination patterns according to the current solution of the optimization problem. Therefore, for each service invocation the Adaptation Manager has to query the storage layer to know the optimal solution and then provide the response to the BPEL Engine. At the end of the process execution, a response is returned to the user. For the completion of the request-response cycle, there is the need to pass twice through a SOAP port for the BPEL engine, plus 8 times for each concrete service invoked through the Adaptation Manager (it would be 4 times without the Adaptation Manager). In addition, inter-modules communications use SOAP calls. A detailed description of all components follows, except for the Service Manager whose implementation is still incomplete.

1) Adaptation Manager: It provides a generic interface, so that it can be used by any business process for the invocation of any partner link. To reach this goal, the component has to manage any kind of SOAP message; therefore, its interface was designed with the maximum degree of flexibility. The tasks of the Adaptation Manager are: (1) to modify the namespaces found in the payload field of the request element, so that they can be compatible with the invoked concrete service; (2) to invoke such service according to the coordination pattern determined by the optimal solution. To accomplish the former task we used the Apache Axiom library, that provides a pull-based parser for SOAP messages and a set of API to act on them. The latter task is accomplished by reading the solution from the storage backend and then invoking the service by means of the Apache Axis libraries.

2) Composition Manager: The original BPEL process is not able to call the Adaptation Manager: therefore, we have to specialize the process to let it invoke our component instead of the concrete services. The Composition Manager performs this specialization by looking only at the process structure (e.g., the process name, the number of parallel invocations) and applies it transparently to the developer of the BPEL process. The extension of the original process is done at the process deploy time. After the business process specialization, the Composition Manager builds the behavioral model [2] of the process just deployed.

3) Optimization Engine: Its core is a MATLAB program (the latter is the only proprietary software used in MOSES). The Optimization Engine is actually a wrapper that makes it possible to invoke a MATLAB program as a Web service. To this end, we defined an interface that exposes different Web service operations, corresponding to the specific events that may trigger the solution of a new instance of the optimization problem: SLA creation, SLA deletion, status change for a Web service, changes in the QoS parameters (e.g., a SLA violation identified by the QoS Monitor), and deployment of a new process.

4) WS Monitor: It checks whether registered concrete services are up and running. Differently from the other MOSES components, it is implemented as a daemon: it starts immediately after the deployment and enters into an infinite loop, without being invoked by the other components. The WS Monitor task is to notify the Optimization Engine of variations in the availability of the concrete services (i.e., available services that become unavailable or unavailable services again available), in such a way that MOSES can react to the change by solving a new instance of the optimization problem. It executes periodically an "HTTP ping" to all the endpoints defined for each concrete service. Because of this simple implementation, the types of faults that can be identified are limited: the monitor can only determine if the application server that hosts the specified Web service is running, but it is unable to find out whether the Web service is actually and properly working. To overcome this issue, the monitor should perform a "SOAP ping", but this involves the creation of request messages that are both syntactically and semantically valid. While the former task can be easily accomplished by parsing the WSDL document of each Web service, the latter is harder because it assumes the usage of ontologies [11]. Rather than following this approach, a simpler improvement of the monitor could be the definition of online test cases for each concrete service.

Furthermore, the component functionalities can be enriched by letting it trigger adaptation actions to the Optimization Engine not only when needed (i.e., after the detection of an already unavailable service) but also in a proactive manner (i.e., on the detection of a soon-to-be unavailable service). As stated in [12], proactive fault management is the next challenge in fault handling.

5) QoS Monitor: Its implementation currently focuses on monitoring the response time and the availability of the invoked concrete services. These measures are collected by the Adaptation Manager as a result of the service invocations and are periodically requested by the QoS Monitor. Given a sliding time window T formed by k subintervals and for each concrete service, the QoS Monitor calculates the number of subintervals v in each of which the concrete service's mean response time (availability) violates the corresponding SLA value; if $v/k > \alpha$, where $0 < \alpha \leq 1$ is a given threshold, the QoS Monitor invokes the Optimization Engine with the measured QoS parameters.

6) SLA Manager: It is in charge of the creation and deletion of the SLAs with the users, the admission control of new users and their possible registration in case of acceptance (see Section II). The current implementation provides a naive automatic renegotiation of the SLA parameters assuming that the broker offers ordered service classes with fixed parameters (e.g., gold, silver, and bronze): in case the user cannot be admitted, the SLA Manager invokes repeatedly the Optimization Engine degrading the service class until it finds a suitable one (if any), and proposes the renegotiated contract to the user. We plan to enhance our automatic negotiation mechanism, using an approach similar to [13].

7) Execution Path Analyzer: It is basically a simple file parser: after a periodic analysis of the log file produced by

ActiveBPEL, it collects information on the number of visits for each `invoke` activity in the business process, updating subsequently the QoS model in the storage layer.

8) MOSES 1 Storage: Storage is a critical component of a distributed system, because the right tradeoff between responsiveness and other performance indexes like availability, reliability, and scalability, should be found. We have investigated various alternatives to implement our data layer, focusing on MySQL and Apache ZooKeeper. The first is a well-known relational database, while the latter is a distributed coordination mechanism for distributed applications. ZooKeeper provides synchronization primitives as well as a shared tree data structure that frees the developer from the burden of managing the data distribution among the system nodes [14]. In MOSES 1, we have decided to use Zookeeper 3.1.1 for two reasons. First, MOSES is a distributed application and it could be limiting to use a single relational database as data backend, especially considering that our application does not need sophisticated data operations (e.g., complex joins). Furthermore, ZooKeeper provides near-local read performance and it easily scales when read/write ratio is over 10 [14], as in the MOSES case. To allow the MOSES future developers not to know ZooKeeper internals, we have developed a data access library, named MOSES Data Access Library (MDAL), that completely hides the data backend. This library implements a ZooKeeper specific logic, but its interfaces can be implemented with other logics.

B. Replicated Architecture of MOSES 1

To evaluate the overhead introduced by the Adaptation Manager on the management of the composite service, we have compared the response time of a BPEL process served by MOSES to that of the same BPEL process managed by a stand-alone BPEL engine. The response time includes the BPEL execution time and the invocation and execution times of dummy partner Web services, while network times are negligible because the tests have been executed in a Gigabit LAN environment. Each test had a duration of 180 seconds, where the first 30 seconds were discarded. The tests were executed on 3 homogeneous computers, the first acting as user machine, the second with only ActiveBPEL, and the third running ZooKeeper, all the MOSES components except the Workflow Engine, and the concrete services. Figure 3 shows the mean response times obtained for increasing request arrival rates. We observe that at a rate of 1 req/sec, the response time of the BPEL process served by MOSES is 266% higher than that of the process served without MOSES. We have investigated the reason and found that it is due to the data serialization inside the ZooKeeper storage: to gain in flexibility we have used XML, but XML parsing wastes over 75% of the Adaptation Manager execution time. From Figure 3 we observe that the saturation point occurs with approximatively 30 req/sec without MOSES and with approximatively 25 req/sec with MOSES serving the business process. Since a sustained rate of 30 req/sec is too limited for a service broker operating in a real world scenario with several concurrent requests, our next effort was to design the MOSES 1 replicated architecture.

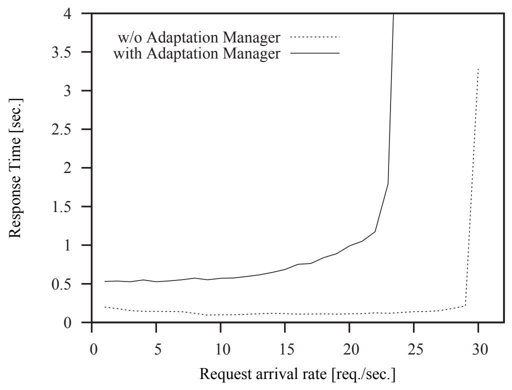

Figure 3. Response times with and without Adaptation Manager.

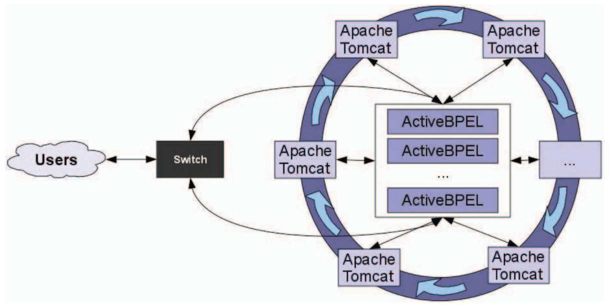

Figure 4. MOSES 1 replicated architecture.

Figure 4 illustrates the replicated architecture of MOSES 1, that has been designed to overcome various of bottlenecks, not only depending on the internal MOSES components, but also on the BPEL engine. The first change is the introduction of the new Switch component, which is in charge to select an available replicated instance of the BPEL Engine for the execution of the BPEL process using a simple stateless load sharing policy (random or round-robin). During the process execution, when the Adaptation Manager has to be invoked, its instance is selected from a list of available Adaptation Managers applying again a naive load sharing strategy.

The selection of which component to use is simplified by the distributed storage system: each MOSES 1 component registers itself in the storage system to inform the other components about its existence. Furthermore, the task of replicating the MOSES components is simplified by the presence of the distributed storage: we realized a new component, named MOSES Node, which registers itself on the storage and offers the deployment and undeployment of Web services to make it possible to add or remove MOSES component instances. Finally, we added two circular monitoring cycles: the first is among Tomcat instances (precisely, among MOSES Nodes instances); the second regards the MOSES components. With the MOSES 1 replicated version we are able to dynamically adapt the broker architecture, adding or removing Tomcat instances (i.e., MOSES Nodes), as well as changing the number of requested instances of a certain component (maximum one instance per kind per node) to increase the system scalability.

We have tested the replicated MOSES 1 using 2 ActiveBPEL nodes on 2 homogeneous PCs and 2 MOSES nodes on 2 additional homogeneous PCs (one of these hosted also the concrete services); a fifth PC was used for generating the requests. Figure 5 compares the response times of the replicated MOSES 1 with its non-replicated counterpart. For a request rate up to 22 (corresponding to the saturation point) the original MOSES 1 performed better than its replicated version because of the lack of the Switch dispatching. At a higher rate, we observe a significant performance improvement achieved by the replicated version.

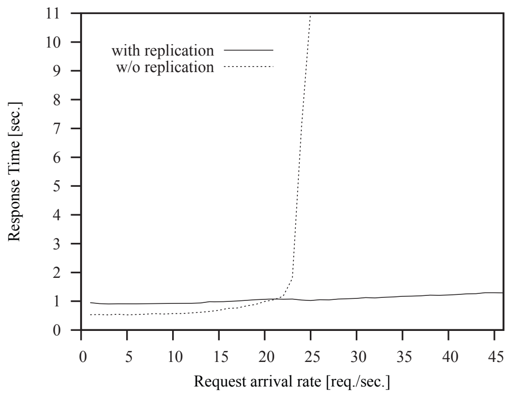

Figure 5. Response times of MOSES 1 with and without replication.

IV. Moses Version 2

The design of the replicated version of MOSES 1 has allowed us to achieve a significant improvement in terms of performance, reliability, and scalability, but at a certain point of the development process we had to face new issues. First, Active Endpoints has discontinued the open source BPEL engine we have used so far; second, we have experimented stability problems with Apache ZooKeeper; finally, masquerading the BPEL processes with the Switch component implies the exposition of a unique interface that is the union of all possible process interfaces. Therefore, we decided to rearrange the MOSES architecture in such a way to let it use as many industrial standards as possible. Specifically, the architecture of MOSES version 2 was redesigned with the use of the Java Business Integration (JBI) standard.

JBI is a messaging-based plug-in architecture [15], whose components are described in WSDL. It provides an architecture and enabling framework that facilitates dynamic composition and deployment of loosely coupled composite applications and service-oriented integration components. The key components of the JBI environment are: (1) Service Engine (SEs), enabling pluggable business logic; (2) Binding Components (BCs), enabling pluggable external connectivity; (3) the Normalized Message Router (NMR), which directs normalized messages from source to destination components according to specified policies.

After thoroughly comparing the available and stable open source implementations for JBI, we decided to focus on *OpenESB*, developed by an open source community under the direction of Sun Microsystems, because it is an implementation and extension of the JBI standard. It implements JBI because it provides the key components (SEs, BCs, and NMR); it extends JBI because it enables a set of distributed JBI instances to communicate as a single logical entity that can be managed through a centralized administrative interface. GlassFish application server is the default runtime environment, although OpenESB can be integrated in several JEE application servers.

A. MOSES 2 within the JBI Environment

The OpenESB-based architecture of MOSES is depicted in Figure 6. Each MOSES 2 component is executed by one Service Engine, that can be either Sun BPEL Service Engine for executing the business processes logic and internal orchestration needs, or J2EE Engine for executing the business logic of all the components but the BPEL Engine. Developing components with J2EE Engine improves the flexibility, because they can be accessed either as standard Web services or as EJB modules through the NMR.

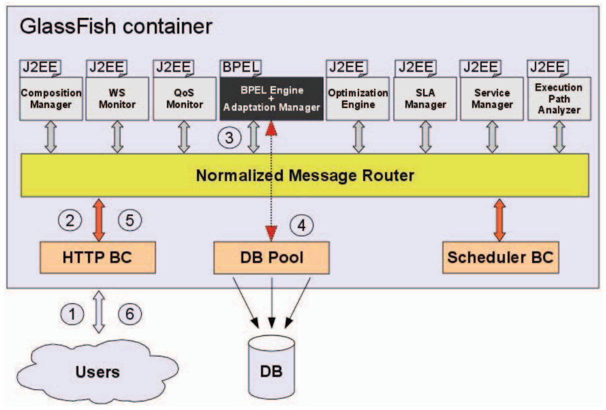

Figure 6. OpenESB-based MOSES architecture.

The typical execution flow is illustrated in Figure 6. In (1), a user issues a standard SOAP request to the MOSES front end, that is the HTTP BC. The request format follows what expected by the BPEL process. In (2), the HTTP BC normalizes the HTTP request and sends it to the BPEL Engine through the NMR (3). When the BPEL Engine receives the request message from the NMR, it de-normalizes the message and starts its execution. At this point, the request could be rejected because MOSES does not own sufficient internal resources to manage it. In such a case, an exception is forwarded to the user. (4) is accomplished whenever there is the need to read the solution of the optimization problem from the storage layer (i.e., for each `invoke` activity). Finally, (5) and (6) occur when the response is provided to the user: the BPEL Engine puts its response message on the NMR, the HTTP BC de-normalizes it obtaining a plain SOAP response message that is then forwarded to the user.

Alternative execution flows can be split in monitoring flows and administration flows. The former denotes each flow that is related to the resources monitoring and can trigger the execution of the Optimization Engine to determine a new optimal solution. The WS Monitor along with the QoS Monitor and the Execution Path Analyzer are invoked by the Scheduler BC at fixed intervals, and each of them can trigger the Optimization Engine in case it has detected a significant change in the system model. The Execution Path Analyzer in MOSES 2 is a simple porting from MOSES 1: when invoked by the Scheduler BC, it parses a log file. This off-line implementation introduces some delay to the update of the system model. Therefore, we are currently evaluating the component implementation using the Intelligent Event Processor SE, which provides the ability to process complex events as well as event streams. The Service Manager can be invoked either by the Scheduler BC or by the SLA Manager when a lookup of new concrete services is required. The SLA Manager and Composition Manager invocation patterns are unchanged with respect to MOSES 1.

All the inter-module communications take advantage from the NMR presence: message exchanges are faster than those based on SOAP communication, because they avoid to pass through the network protocol stack without losing the ability to expose every MOSES component as a Web service.

B. MOSES 2 Storage

MOSES 2 data storage has also been redesigned to use as much industrial standards as possible. Therefore, we have abandoned Apache ZooKeeper in favor of a more consolidated DBMS like MySQL. With this choice, we gain the opportunity to maintain a more structured data set, that allows us not to use XML for data storing with the subsequent performance speedup.

MySQL also offers interesting cluster capabilities that we have exploited into MOSES 2. We use MySQL cluster Network DataBase (NDB) to obtain performance improvements and high availability. The NDB storage engine allows the creation and the management of in-memory databases, replicated with hard-consistency constraints. Within the hard-consistency hypothesis, we have built a multi-master MySQL cluster, where load balancing among the DB servers is carried out by GlassFish through a random policy.

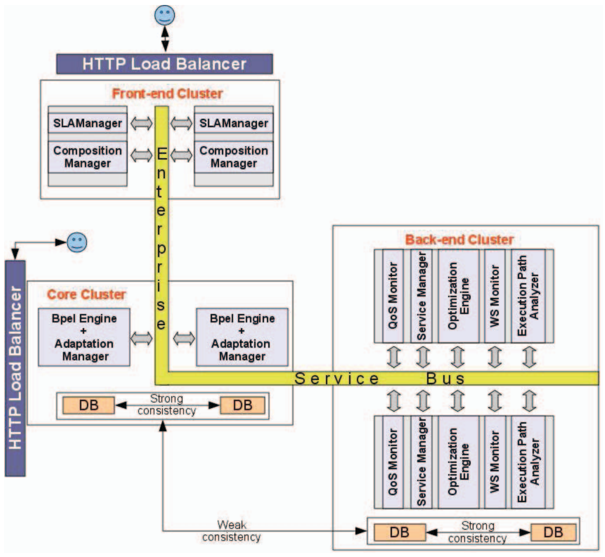

Figure 7. MOSES 2 replicated architecture.

C. Replicated Architecture of MOSES 2

MOSES 1 replicated architecture was quite flexible because its design allowed us to distribute the MOSES components at the finest level of granularity; but in practice we really did not need such flexibility. For example, why should the Workflow Engine call a remote Adaptation Manager rather than a local one? No reason, considering also that the Adaptation Manager load is lower than that of Workflow Engine. Therefore, it is preferable to consider them as a single logical unit and eventually replicate them in pair.

MOSES 2 requires that only two components must be up and running in order to complete the request-response cycle: the BPEL Engine and the DB. In case only these two components work, our broker may anyway orchestrate the services in a sub-optimal way, but still it succeeds in providing a response to the users. Figure 7 illustrates the MOSES 2 replicated architecture, where the BPEL Engine and the Adaptation Manager constitute the so-called *core* cluster. The other clusters provide additional features to MOSES 2 that are not mandatory for the basic execution: the *front-end* cluster provides to the broker the ability to receive new BPEL processes to deploy as well as the ability to negotiate SLAs with users. The *back-end* cluster comprises the components that allows the runtime adaptation capabilities. From a database point of view, the core cluster hosts its own high available DB server with strong consistency to make the execution of DB queries as fast as possible. The back-end cluster's DB is instead synchronized with the core cluster's DB using an external weak consistency policy and an internal strong consistency policy. Finally, the front-end cluster does not own a DB at all: we assume that the request rate it receives is much lower than that directed to the core cluster; therefore, we prefer to pay a penalty for the

DB accesses generated by the front-end cluster rather than having on it a new MySQL instance with its own replication strategy and related overhead.

We are evaluating the performance of MOSES 2. From preliminary experiments we have obtained quite promising results: MOSES 2 without replication halves the response time of a request-response cycle with respect to MOSES 1.

V. Lessons Learned and Conclusions

In this paper we presented the design and implementation of the MOSES broker, which provides runtime QoS-driven adaptation of SOA applications. We developed two versions of MOSES, which are both based on open source products and can be extended through replication to let MOSES scale and efficiently cope with QoS requirements coming from several concurrent users in a rapidly changing environment.

The development of MOSES 2 followed a totally different approach from MOSES 1. We re-designed the whole architecture: MOSES 1 claimed to be a completely distributed application, but the distribution of some component could be not only unnecessary but it could also introduce additional overhead. Furthermore, we carefully distributed MOSES 2 components to minimize the network overheads for inter-module communications and storage access. For example, by collocating the Workflow Engine and the Adaptation Manager on the same machine and also letting them be executed by the same JVM, we succeeded to call the Adaptation Manager as a Java class rather than as a Web service, with consequent speedup.

Another weakness of MOSES 1 was the manual integration of many software products: ActiveBPEL, Tomcat, Axis, and ZooKeeper. With MOSES 2, we decided to use JBI, which provides a standardized way to integrate components. The choice of OpenESB as JBI implementation lead us to replace Tomcat with GlassFish. The latter with OpenESB offers us the same functionalities we had in MOSES 1, plus a standard pluggable architecture and a standardized way to communicate among components.

Furthermore, OpenESB can be extended through different GlassFish instances; therefore, we can rely on GlassFish's own cluster capabilities for the replicated MOSES 2, without the need to realize any "ping and react" pattern. Again, the native support of relational DBs like MySQL offered by GlassFish lead us to conduct a more precise analysis on the storage layer, whose aim was to identify the MOSES components that execute more frequent data accesses. As a result, we identified three load sources: the BPEL Engine with the Adaptation Manager make the majority of the data accesses (and they are the components within the core cluster). Then, in decreasing load order, there are the components belonging to the back-end cluster and finally the components in the front-end cluster. Having local clusters instead of geographically distributed ones, together with the ability of MySQL to manage in-memory DBs, allowed us to relinquish the distributed storage layer in favor of the more classic DB-based solution. Specifically, we used a MySQL cluster with strong consistency and synchronous replication within the core cluster, asynchronously replicated with the MySQL cluster of the back-end cluster. With this solution it is possible to realize three types of local clusters that can be spread across geographically distributed networks (e.g., collocating the core cluster on a private cloud and the back-end and front-end clusters on Amazon EC2).

Besides completing the Service Manager component, we are planning a more comprehensive set of experiments to validate the architectural choices of MOSES 2 and prove its effectiveness in a real testing scenario. Finally, we are extending MOSES to fully support stateful services, as well as to proactively monitor its adequacy to SLAs.

Acknowledgment

Work supported by the Italian PRIN project D-ASAP.

References

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[2] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "Qos-driven runtime adaptation of service oriented architectures," in *ACM ESEC/SIGSOFT FSE*, 2009.

[3] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, June 2007.

[4] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Trans. Web*, vol. 1, no. 1, pp. 1–26, 2007.

[5] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "Paws: A framework for executing adaptive web-service processes," *IEEE Softw.*, vol. 24, no. 6, pp. 39–46, 2007.

[6] A. Erradi, P. Maheshwari, and V. Tosic, "Policy-driven middleware for self-adaptation of web services compositions," in *ACM/IFIP/USENIX Middleware 2006*, 2006, pp. 62–80.

[7] D. Menascé, H. Ruan, and H. Gomma, "Qos management in service oriented architectures," *Performance Evaluation*, vol. 7-8, no. 64, Aug. 2007.

[8] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "A framework for qos-aware binding and re-binding of composite web services," *J. of Systems and Software*, vol. 81, no. 10, pp. 1754–1769, 2008.

[9] O. Ezenwoye and S. Sadjadi, "A proxy-based approach to enhancing the autonomic behavior in composite services," *J. of Networks*, vol. 3, no. 5, pp. 42–53, 2008.

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[11] J. Cardoso, "Semantic integration of web services and peer to peer networks to achieve fault-tolerance," in *2006 IEEE Int'l Conf. on Granular Computing*, May 2006, pp. 796–799.

[12] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, no. 3, pp. 1–42, 2010.

[13] M. Comuzzi and B. Pernici, "A framework for qos-based web service contracting," *ACM Trans. Web*, vol. 3, no. 3, 2009.

[14] "Apache ZooKeeper," http://hadoop.apache.org/zookeeper/.

[15] B. Kumar, P. Narayan, and T. Ng, *Implementing SOA Using Java EE*. O'Reilly, Dec. 2009.