A Scalable and Highly Available Brokering Service for SLA-Based Composite Services

Alessandro Bellucci, Valeria Cardellini, Valerio Di Valerio, and Stefano Iannucci

Università di Roma "Tor Vergata", Viale del Politecnico 1, 00133 Roma, Italy
{cardellini,iannucci}@ing.uniroma2.it

Abstract. The introduction of self-adaptation and self-management techniques in a service-oriented system can allow to meet in a changing environment the levels of service formally defined with the system users in a Service Level Agreement (SLA). However, a self-adaptive SOA system has to be carefully designed in order not to compromise the system scalability and availability. In this paper we present the design and performance evaluation of a brokering service that supports at runtime the self-adaptation of composite services offered to several concurrent users with different service levels. To evaluate the performance of the brokering service, we have carried out an extensive set of experiments on different implementations of the system architecture using workload generators that are based on open and closed system models. The experimental results demonstrate the effectiveness of the brokering service design in achieving scalability and high availability.

1 Introduction

The complexity of service-oriented systems poses an increasing emphasis on the need of introducing runtime adaptation features, so that a SOA-based system can meet its quality of service (QoS) requirements even when operating in highly changing environments. In addition, the SOA paradigm allows to build new applications by composing network-accessible services offered by loosely coupled independent providers. A service functionality, e.g., booking an hotel, may be implemented by several competing services (referred to as *concrete services*) with different QoS and cost attributes, thus allowing a prospective user to select the services that best suit his/her requirements. Hence, being able to effectively deliver and guarantee the QoS levels required by differentiated classes of users may bring competitive advantage to a composite service provider over the others.

In this paper, we present the design and performance evaluation of MOSES (MOdel-based SElf-adaptation of SOA systems), a runtime adaptation framework for a SOA system architected as a brokering service and operating in a sustained traffic scenario. MOSES offers to prospective users various composite services, each of which presents a range of service classes that differ for the QoS performance parameters and cost. Its goal is to drive the adaptation of the composite services it manages to fulfill the SLAs negotiated with its users, given

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 527–541, 2010.

the SLAs it has negotiated with the concrete services used to implement the composite services, and to optimize a utility goal (e.g., the broker revenue).

The major goals of the MOSES brokering service are: (1) its ability to manage in an adaptive manner the concrete services so that it guarantees the QoS parameters agreed in the SLAs with the composite service users; (2) its scalability and availability, being the brokering service subject to a sustained traffic of requests; therefore, its architecture should not affect the performance of the managed composite services. To achieve these goals, 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 internal and external changes that occur during its operation. Moreover, the MOSES prototype exploits the rich capabilities offered by OpenESB (an implementation of the JBI standard) and MySQL, which both provide interesting features to enhance the scalability and availability of complex systems. We have evaluated the performance and scalability of the MOSES prototype through an extensive set of experiments using workload generators that are based on open and closed system models. The results show that under every load condition the MOSES prototype based on OpenESB and MySQL achieves a significant performance improvement in terms of scalability and reliability with respect to a previously developed version of MOSES [4]. In addition, the clustered version of the prototype further enhances the performance introducing only a negligible overhead due to the load balancing.

The MOSES architecture is inspired by existing implementation of frameworks for QoS brokering of Web services (e.g., [1,2,10]). Menascé et al. have proposed a SOA-based broker for negotiating QoS goals [10] but their broker does not offer a composite service and its components are not organized as a self-adaptive system. PAWS [1] 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. Proxy-based approaches, similar to that used by MOSES for the runtime binding to concrete services, have been previously proposed, either for re-binding purposes [2] or for handling runtime failures in composite services as in the TRAP/BPEL framework [5]. The SASSY framework for self-adaptive SOA systems has been recently proposed in [11]: it self-architects at run-time a SOA system to optimize a system utility function. Nonetheless, to the best of our knowledge none of the previous works in the SOA field has evaluated the proposed prototype in terms of performance and scalability, but this kind of evaluation is needed for any prototype to be adopted and developed in an industrial environment.

The methodology at the basis of MOSES has been presented in [3]; its distinguishing features are the *per-flow* approach to adaptation and the combination of *service selection* and *coordination pattern selection*. The per-flow approach means that MOSES jointly considers the aggregate flow of requests, generated by multiple classes of users; to the contrary, most of the proposed adaptation methodologies (e.g., [1,2,14]) deal with single requests to the composite service, which are managed independently one from another. The second feature regards

the adaptation mechanisms used by MOSES, that combine service selection with coordination pattern selection. The first mechanism aims at identifying for each abstract functionality in the composite service one corresponding concrete service, selecting it from a set of candidates (e.g., [1,2,14]), The coordination pattern selection allows to increase the offered QoS by binding at runtime each functionality to a properly chosen subset of concrete services, coordinating them according to some redundancy pattern.

The paper is organized as follows. In Sect. 2 we present an overview of the MOSES architecture. The MOSES design and implementation are discussed in Sect. 3. We present the testing environment and analyze the experimental results assessing the effectiveness of MOSES design in Sect. 4. Finally, we draw some conclusions and give hints for future work in Sect. 5.

2 Overview of the MOSES Architecture

The MOSES architecture represents an instantiation for the SOA environment of a self-adaptive software system [8], focused on the fulfillment of QoS requirements. The architecture of an autonomic system comprises a set of managed resources and managers, that operate as part of the IBM's MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) reference model [9]. This autonomic loop collects information from the system, makes decisions and then organizes the adaptation actions needed to achieve goals and objectives, and controls the execution. Figure 1 shows the MOSES architecture, whose core components are organized in parts according to the MAPE-K cycle. In the following we provide a functional overview of the tasks carried out by the MOSES components, while in Sect. 3 we discuss in details their design and implementation.

The Execute part comprises the *Composition Manager*, *BPEL Engine*, and *Adaptation Manager*. The first component receives from the brokering service

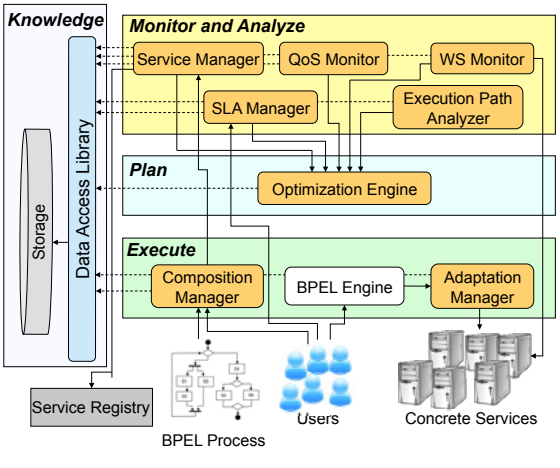


Fig. 1. MOSES high-level architecture

administrator a new BPEL process to be deployed inside MOSES and builds its corresponding behavioral model. To this end, it interacts with the Service Manager to identify the concrete services that implement the functionalities required by the service composition. Once created, the behavioral model, which also includes information about the discovered concrete services, is stored in the Knowledge part to make it accessible to the other system components.

While the Composition Manager is invoked rarely, the BPEL Engine and Adaptation Manager are the core modules for the execution and runtime adaptation of the composite service. The first is the software platform that actually executes the business process and represents the user front-end for the composite service provisioning. It interacts with the Adaptation Manager to invoke the proper component services: for each abstract functionality required during the process execution (i.e., `invoke` BPEL activity), the Adaptation Manager dynamically binds the request to the real endpoint that represents the service. The latter is identified by the solution of a linear programming (LP) optimization problem [3] and can be either a single service instance or a subset of service instances coordinated through some pattern. The MOSES methodology currently supports as coordination patterns the 1-out-of-n parallel redundancy and the alternate service [3]. With the former, the Adaptation Manager invokes the concurrent execution of the concrete services in the subset identified by the solution of the LP problem, waiting for the first successful completion. With the latter, the Adaptation Manager sequentially invokes the concrete services in the subset, until either one of them successfully completes, or the list is exhausted.

The *Optimization Engine* realizes the planning aspect of the autonomic loop. It solves the LP 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 the concrete services. The model is kept up to date by the monitoring activity carried out by the components in the Monitor-and-Analyze part. Since the optimization problem is formulated as an LP problem, it is suitable to be solved at runtime because of its efficiency [3] and does not represent a bottleneck for MOSES scalability. The problem solution provides indications about the adaptation actions that must be performed to optimize the use of the concrete services with respect to the utility goal of the brokering service and within the SLA constraints.

The Monitor-and-Analyze part comprises all the components that capture changes in the MOSES environment and, if they are relevant, modify at runtime the behavioral model and trigger a new adaptation plan. Specifically, the *QoS Monitor* collects and analyzes information about the QoS levels perceived by the composite service users and offered by the concrete services providers. The *WS Monitor* checks periodically the concrete services availability. The *Execution Path Analyzer* monitors variations in the usage profile of the composite service functionalities by examining the business process executed by the BPEL Engine; it determines the expected number of times that each functionality is invoked by each service class. The *Service Manager* and the *SLA Manager* are responsible for the SLA negotiation processes in which the brokering service is involved.

Specifically, the first negotiates the SLAs with the concrete services, while the latter is in charge to add, modify, and delete users SLAs and profiles. The SLA negotiation process towards the user side includes the admission control of new users; to this end, it involves the use of the Optimization Engine to evaluate MOSES capability to accept the incoming user, given the associated SLA and without violating already existing SLAs. Since the Service and SLA Managers can determine the need to modify the behavioral model and solve a new instance of the LP problem, we have included them within the Monitor-and-Analyze part.

In the current MOSES prototype, each component in the Monitor-and-Analyze part, independently from the others, senses the composite service environment, checks whether some relevant change has occured on the basis of event-condition-action rules and, if certain conditions are met, triggers the solution of a new LP problem instance. Tracked changes include the arrival/departure of a user with the associated SLA (SLA Manager), observed variations in the SLA parameters of the concrete services (QoS Monitor), addition/removal of concrete services corresponding to functionalities of the abstract composition (WS Monitor and Service Manager), variations in the usage profile of the functionalities in the abstract composition (Execution Path Analyzer).

Finally, the Knowledge part is accessed through the *Data Access Library*, which allows to access the parameters of the composite service operations and environment, among which the solution of the optimization problem and the monitored model parameters.

3 MOSES Design

We have designed the MOSES architecture on the basis of the Java Business Integration (JBI) specification. JBI is a messaging-based pluggable architecture, whose components describe their capabilities through WSDL. Its major goal is to provide an architecture and an enabling framework that facilitates the dynamic composition and deployment of loosely coupled participating applications and service-oriented integration components. The key components of the JBI environment are: (1) the Service Engines (SEs) that enable pluggable business logic; (2) the Binding Components (BCs) that enable 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 chose *OpenESB*[1], 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 binding components, service engines, and the 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.

[1] ESB stands for Enterprise Service Bus.

3.1 MOSES within the JBI Environment

Each MOSES 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 MOSES components except the BPEL Engine. Developing the components with J2EE Engine improves the flexibility, because they can be accessed either as standard Web services or as EJB modules through the NMR.

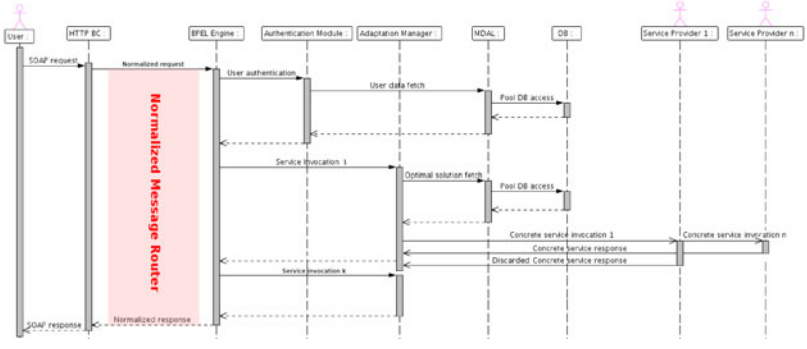


Fig. 2. Typical execution flow in the ESB-based MOSES prototype

The typical execution flow of a request to the composite service is illustrated by the sequence diagram in Fig. 2. As first step, the registered user issues a SOAP request to the MOSES front-end, that is the HTTP BC; the request format follows what expected by the BPEL process to whom the request is addressed. The HTTP BC normalizes the HTTP request and sends it to the BPEL Engine through the NMR. Upon receipt of the message, the BPEL Engine denormalizes the message and starts to serve the request. The first task performed within the process is the invocation of the authentication module (not shown in the high-level architecture of MOSES) to verify that the user issuing the request is properly registred. If not, an exception is forwarded to the user. Otherwise, for each `invoke` activity within the BPEL process, the Adaptation Manager reads the solution of the LP problem from the storage layer and for that abstract functionality invokes the subset of concrete services using the coordination pattern as determined by the solution (Fig. 2 shows the use of the 1-out-of-n parallel redundancy pattern for one service invocation). Finally, when the response is ready for the user (these steps are not shown in Fig. 2), the BPEL Engine puts the response message on the NMR, the HTTP BC de-normalizes it, obtaining a plain SOAP response message that is finally forwarded to the user.

Alternative execution flows can be split in monitoring 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, QoS Monitor, and Execution Path Analyzer are periodically invoked by the Scheduler BC, and each of them can trigger the

Optimization Engine when a new adaptation plan is needed. The Service Manager can be invoked either by the Scheduler BC or by the Composition Manager when new concrete services are needed. The SLA Manager is invoked by users when they register or establish new SLAs with MOSES; the Composition Manager is invoked by the MOSES administrator to manage new BPEL processes.

We observe that MOSES requires that only the BPEL Engine, the Adaptation Manager and the storage layer must be up and running to complete the request-response cycle. When only these components work, the broker can orchestrate the composite service (although in a sub-optimal way, being not able to solve a new instance of the optimization problem), but it still succeeds in providing a response to the users.

3.2 MOSES Components

We analyze in detail only the Adaptation Manager and storage layer design, because these are the components that mostly influence the MOSES performance and scalability. We have designed and implemented all the other components, except the Service Manager; their detailed description can be found in [4]. We note that all inter-module communications exploit the NMR presence: message exchanges are faster than those based on SOAP communication, because they are "in-process", thus avoiding to pass through the network protocol stack. However, thanks to OpenESB we can expose every MOSES component as a Web service.

The tasks of the Adaptation Manager are to modify the request payload in order to make it compatible with the subset of invoked concrete services and to invoke these services according to the coordination pattern determined by the solution of the optimization problem.

Being the Adaptation Manager the MOSES component that receives the highest request rate, its design is crucial for scalability and availability. We have investigated three alternative solutions for its implementation. The first realizes the component directly in BPEL, but we discarded it because the Sun BPEL Service Engine does not currently support the `forEach` BPEL structured activity with the attribute `parallel` set to 'yes'. We needed this activity to realize in BPEL the 1-out-of-n coordination pattern. With the second alternative we investigated how to realize the Adaptation Manager as a Java EE Web service. We found a feasible solution (based on the *Provider* interface offered by the JAX-WS API) but we discarded it because it causes a non negligible and useless performance overhead for the service invocation itself. The solution we finally implemented realizes the Adaptation Manager as a Java class which is directly invoked inside the BPEL process. The advantage is the higher communication efficiency and the consequent reduction of the response time perceived by the users of the composite service, as shown in Sect. 4.

The storage layer represents a critical component of a multi-tier distributed system, because the right tradeoff between responsiveness and other performance indexes (like availability and scalability) has to be found.

We have investigated various alternatives to implement the MOSES storage layer and decided to rely on the well-known relational database MySQL, which

offers reliability and supports clustering and replication. However, to free the MOSES future developers from knowing the storage layer internals, we have developed a data access library, named MOSES Data Access Library (MDAL), that completely hides the data backend. This library currently implements a specific logic for MySQL, but its interfaces can be enhanced with other logics.

3.3 MOSES Clustered Architecture

In designing the clustered architecture of MOSES we made a tradeoff between flexibility and performance. By flexibility we mean the ability to distribute the MOSES components at the finest level of granularity (i.e., each component on a different machine); however, we have found that having a high degree of flexibility impacts negatively on the overall MOSES performance [4]. Therefore, we have carefully distributed the MOSES components in order to minimize the network overheads for inter-module communications and storage access. Following this guideline, we have collocated the BPEL Engine and the Adaptation Manager on the same machine; in such a way, for each invoked external service whose binding is executed at runtime by the Adaptation Manager, the BPEL Engine does not need to communicate through the network. In addition, being these two components executed by the same JVM, the Adaptation Manager is called as a Java class rather than as a Web service, with consequent performance speedup.

Figure 3 illustrates the MOSES clustered architecture composed by three clusters, where each one owns two replicas of the components placed in that cluster. The BPEL Engine and the Adaptation Manager constitute the *core* cluster, while the other two clusters provide additional features that are not mandatory for the basic execution. The *front-end* cluster provides the broker with the ability to receive new BPEL processes and negotiate SLAs with users. The *back-end* cluster comprises the components to monitor and analyze the environment and to determine a new adaptation plan. In front of those clusters that are accessed by the composite service users, there is an HTTP load balancer that distributes the requests among the replicas.

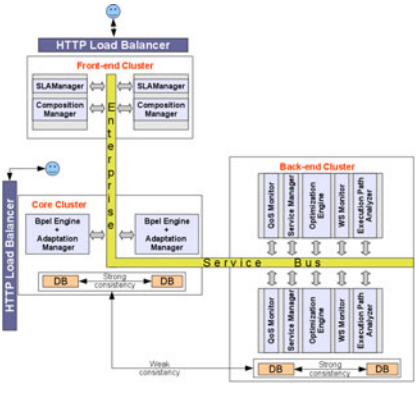


Fig. 3. MOSES clustered architecture

As regards the distribution of the storage layer, the core cluster hosts its own high available DB server with strong consistency to execute the DB queries as fastest 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 directed to it 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.

4 Experimental Results

In this section we present the results of the experiments we have conducted on the MOSES prototype based on OpenESB. We compare its performance to that of a previous version of the MOSES prototype, whose components have been developed in Java as Web services. We refer to the latter as **MOSES WS**, while the current version is referred to as **MOSES ESB**. We also analyze the performance of the clustered MOSES ESB. Prior to present the experimental environment and the tools supporting the performance testing, we briefly review the main features of MOSES WS, whose detailed discussion can be found in [4].

4.1 MOSES WS

MOSES WS was entirely designed and implemented using the Web services stack as core technology. It included each component of the high-level MOSES architecture in Sect. 2; we also realized its replicated version.

Some choices we made during the MOSES WS design have turned out not to be appropriate, especially from the performance point of view. First of all, the adoption of *Apache ZooKeeper* [15] for the storage layer. ZooKeeper is a distributed coordination system for distributed applications, that provides synchronization primitives as well as a shared tree data structure. We relied on it to have an uniform data view from every application instance and to build mechanisms such as distributed counters and distributed locks. However, the penalty for this choice is a significant performance overhead, caused by a large amount of disk I/O operations. Secondly, we used *ActiveBPEL* from Active Endpoints as BPEL engine. Although we chose it for its better performance with respect to Apache ODE and for its usage in many research works on SOA systems, it turned not to be sufficiently stable; moreover, it was also suddenly retired. Finally, the adoption of SOAP as the core application protocol for the components inter-communications gave us a great flexibility, because we could place the components everywhere, even in a geographically distributed fashion. However, the cost paid for such flexibility is the overhead for managing the SOAP messages.

4.2 Experimental Setup

The testing environment is composed by 3 Intel Xeon quad-core servers (2 Ghz/core) with 8 GB RAM each (referred to as nodes 1, 2, and 3); 2

dual-processor Athlon MP servers with 1 GB RAM each (nodes 4 and 5); a Gb Ethernet connection for the quad-core machines, 100Mbps for the others.

We have analyzed the performance of MOSES ESB in the non-clustered and clustered versions: for each of these configurations we have executed the experiments using two different workload generators that are based on closed and open system models. Using the closed system model, we have identified the maximum system throughput. The open system model has been useful to find the effective response time in a real world SOA environment, where the generation of new requests does not depend on the completion of previous ones, and to establish how MOSES response time changes according to a controlled variation in the request rate. Closed and open system models can lead to different system behaviors, as discussed in [13]: therefore, using both we can analyze MOSES performance in a more complete way. The closed-model experiments have been performed with *The Grinder* [6], while *httperf* was used for the open-model load testing [7]. The first is an open source powerful load testing framework, that allows to test every application accessible through a Java API. For our testing purposes, we have used the Grinder plugin HTTPlugin, therefore encapsulating the SOAP request message to the composite service inside the HTTP request message. The latter is an open-source tool largely used for measuring the performance of Web servers: therefore, it can be also used to measure the performance of Web services when they can be accessed through HTTP.

Differently from traditional Web workload, SOA workload characterization has been not deeply investigated up to now (some results have been published in [12]). Therefore, to evaluate the performance of our SOA system, we have defined a BPEL process that mimics a "trip planner" SOA application, with 6 `invoke` activities (corresponding to 6 functionalities of the abstract composition). The tasks are orchestrated using most of the BPEL structured activities, including `flow`, which is used for the concurrent execution of the activities. Two concrete services can be selected for each functionality in the abstract composition and the binding is carried out at runtime by the Adaptation Manager; the used service(s) and the coordination pattern depend on the solution of the LP optimization problem managed by the Optimization Engine. For the experiments, we disabled the monitoring activities executed by the QoS and WS Monitors. The invoked Web services are simple stubs with no internal logic, being the analysis of MOSES scalability the goal of our performance study.

In the next sections we present the performance results, first considering the non-clustered version of MOSES ESB and MOSES WS under closed and open system models. Then, we analyze the performance of the MOSES ESB clustered architecture. We anticipate that the experimental results show that MOSES ESB outperforms MOSES WS for every load condition. The choice of MySQL and the optimization of some components (e.g., the Adaptation Manager) allows to remove most performance problems of MOSES WS; furthermore, from the stability point of view GlassFish proved to have a high availability: even after many stress tests no response error was received.

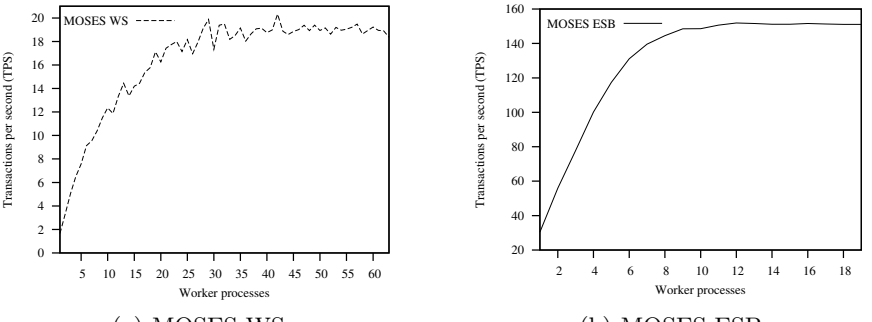
4.3 Closed-Model Experiments

In a closed system model, there is a fixed number of users who use the system forever. Each user repeats two steps: (a) submits a job; (b) receives the response and "thinks" for some amount of time. In a closed system, a new request is only triggered by the completion of a previous one. We set a think time equal to 0, because our aim is to perform a stress testing of the system to determine its effective throughput. Each closed-model test was performed on a three-machine environment, where node 1 hosted a full MOSES instance without data backend, node 2 the data backend together with the concrete services, and node 3 The Grinder. The latter generates an increasing number of client processes named "worker processes", each of which behaves like a user above described.

Figure 4(a) shows the MOSES WS throughput in terms of Transactions Per Second (TPS), which represents the mean number of transactions per second for a given number of worker processes. MOSES WS does not achieve a high TPS: the maximum value is around 21 TPS, which is definitively too low to cope with a relatively sustained incoming request rate. Furthermore, the maximum TPS value is reached with a relatively high number of worker processes. The motivation is that we get high response times (on average equal to the number of worker processes divided by the TPS value) and a non-optimal CPU utilization. By analyzing the components of the response time, we found that a large fraction of the response time is spent in waiting for the data storage layer, which is based on Apache ZooKeeper. Figure 4(a) illustrates the performance reason that lead us to design and develop the second version of our prototype, i.e., MOSES ESB.

Figure 4(b) shows the MOSES ESB performance in terms of TPS within the same testing environment. MOSES ESB achieves a significant performance improvement with respect to MOSES WS: the maximum TPS value is around 140 and this maximum is achieved with only 9 worker processes.

As regards the availability of the two prototypes, MOSES ESB is again the winner: MOSES WS reported an error percentage equal to 1.87 (2593 errors on a total of 139030 requests), while MOSES ESB never returned an error message for the entire experiment duration.



(a) MOSES WS (b) MOSES ESB

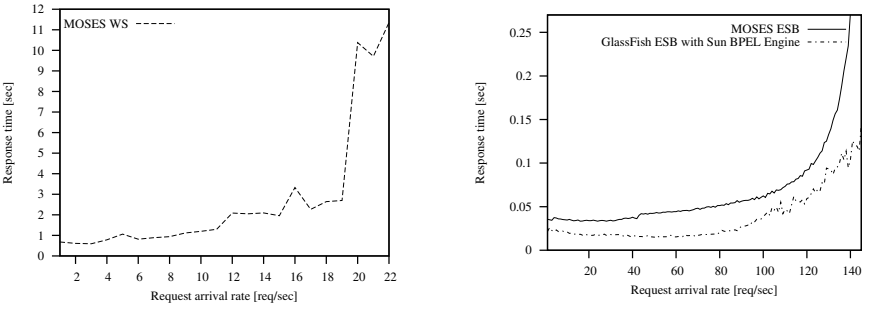
Fig. 4. Throughput in the closed system model

4.4 Open-Model Experiments

In an open system model there is a stream of arriving users with an average arrival rate. Each user submits one job to the system under test, waits for the response, and then leaves. The main difference between open and closed systems is that the arrival of a new request is only determined by a new user arrival and does not depend on the completion of a previously issued request. We believe that the real world SOA environment in which MOSES can operate is closer to an open system model, because users with already established SLAs can generate new requests independently of the completion of previously issued requests.

The overall experiment is composed by a maximum of 140 runs, each one lasting 180 seconds, during which httperf generates HTTP requests at a constant rate. We note that there is a 1-to-1 mapping between an HTTP request and a request to the composite service provided by MOSES. The main performance metric we collected for each run is the mean response time, i.e., the time spent on average for the entire request-response cycle. The deployment environment for the open-model experiment is the same of the closed one.

Figure 5(a) shows the response time achieved by MOSES WS in the open model testing environment. When the system is stable (corresponding to a rate ranging from 1 to 19 requests per second), the response time varies between 600 ms and 3 sec. When the request rate reaches 20, the system becomes unstable and we observe an uncontrolled grow of the queues length. We have found that the high response times of MOSES WS is due to I/O waits. In preliminary experiments, we have also compared the response time of the composite service managed by MOSES with that of the same service offered by a standalone BPEL engine [4]. When the system is stable, we found that the response time of MOSES WS is on average 266% higher than that achieved by the ActiveBPEL engine. This overhead is very similar to that reported in [5] for the TRAP/BPEL framework, which has a simpler architecture and provides less adaptation functionalities than MOSES WS. Although the SOA system manager expects to pay some performance penalty for the system self-adaptiveness, our effort in designing MOSES ESB has been to reduce such overhead.



(a) MOSES WS (b) MOSES ESB

Fig. 5. Response time in the open system model

Figure 5(b) shows the results for MOSES ESB. First, we observe that in this case the overall experiment is composed by almost 140 runs against 22 runs for MOSES WS. The system is stable up to 130 requests per second, which represents the saturation point. The I/O waits are now reduced to less than 1% of the overall CPU execution time and this positively impacts on the smoothness of the curve with respect to that of MOSES WS. Figure 5(b) also shows the response time obtained by the standard GlassFish ESB with Sun BPEL Engine when no self-adaptive capability is provided. When the composite service is managed by MOSES ESB, the response time is on average 108% higher than that served by GlassFish ESB (the percentage increase ranges from a minimum of 30% to a maximum of 209%). Therefore, the careful design of MOSES ESB allows us to substantially reduce the overhead introduced by the self-adaptiveness.

Figure 6 compares the performance achieved by MOSES ESB and MOSES WS, using a logarithmic axes scale (base 2 and 10 for x and y axes, respectively). The performance improvement achieved by MOSES ESB is clearly evident. As regards the availability of the two prototypes, MOSES ESB again returned no error message, while MOSES WS reported 21 errors in 3600 seconds.

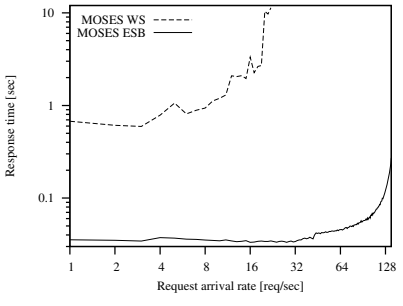
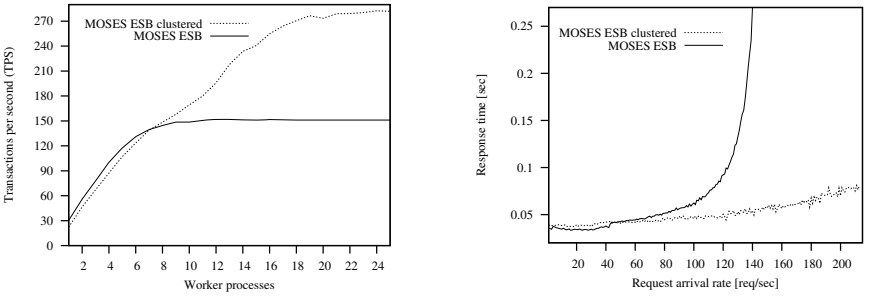


Fig. 6. Comparison of the response time in the open system model

4.5 Performance of MOSES ESB Clustered

The experiments for the clustered version of MOSES ESB have also been based on the open and closed system models. These sets of experiments were executed with the same hardware already used for the non-clustered version, but we slightly changed the component deployment schema. We used 5 machines, where nodes 1 and 2 hosted a GlassFish instance, node 3 the data backend and the concrete services, node 4 the load balancer, and node 5 either The Grinder or httperf. GlassFish allows the system administrator to choose between two load balancers: Sun Java Web Server or Apache Web Server with a load balancing plugin. The first is a closed-source Web server; therefore, we have chosen the latter being open-source. Nevertheless, we were constrained to use a closed-source plugin in order to have an active load-balancing subsystem, which allows to react at the load-balancer level to any failure of the connected GlassFish instances, for example by re-issuing the request to an active instance.



(a) Throughput in the closed model (b) Response time in the open model

Fig. 7. Performance comparison of MOSES ESB and MOSES ESB Clustered

Figure 7(a) shows the throughput improvement achieved by adding a GlassFish instance to the MOSES cluster. The load balancer introduces a negligible overhead and the overall performance is incremented by almost a factor of 2. Figure 7(b) compares the clustered version of MOSES ESB with its non-clustered counterpart using the open system model. Similarly to what obtained in the closed-model experiment, we can see that for a low request load, the clustered version is a bit slower than the non-clustered one because of the load balancer component. However, this gap is rapidly filled starting from the request rate equal to 50. After this point, the clustered version is clearly the winner, achieving a response time that halves that of the non-clustered prototype.

5 Conclusions

In this paper we have presented an OpenESB-based prototype for a scalable and highly available brokering service that provides runtime QoS-driven adaptation of composite services. We have analyzed its performance and scalability, comparing them to those of a previous version of the prototype. The experimental results demonstrate that the key choices made during the MOSES ESB development have allowed to obtain significant performance improvements with respect to MOSES WS, which presents some similarities with other prototypes developed for service selection in SOA applications. With respect to MOSES WS, the response time achieved by MOSES ESB is two orders of magnitude lower, while the throughput is one order of magnitude higher. Furthermore, MOSES ESB clustered obtains a nearly linear performance improvement according to the number of installed GlassFish instances.

We are planning new experiments using MySQL cluster, that allows to increase the system availability and to improve further the performance through its in-memory DB feature. We will also extend MOSES to support stateful as well as asynchronous long-running services and to proactively monitor SLA violations.

Acknowledgment. Work partially supported by the Italian PRIN project D-ASAP.

References

1. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A framework for executing adaptive Web-service processes. IEEE Softw. 24(6), 39–46 (2007)
2. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81(10) (2008)
3. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: QoS-driven runtime adaptation of service oriented architectures. In: ACM ESEC/SIGSOFT FSE, pp. 131–140 (2009)
4. Cardellini, V., Iannucci, S.: Designing a broker for QoS-driven runtime adaptation of SOA applications. In: IEEE ICWS 2010 (July 2010)
5. Ezenwoye, O., Sadjadi, S.: A proxy-based approach to enhancing the autonomic behavior in composite services. J. of Networks 3(5), 42–53 (2008)
6. The Grinder, `http://sourceforge.net/projects/grinder/`
7. httperf, `http://www.hpl.hp.com/research/linux/httperf/`
8. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. ACM Comput. Surv. 40(3), 1–28 (2008)
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
10. Menascé, D.A., Ruan, H., Gomaa, H.: QoS management in service oriented architectures. Perform 7-8(64), 646–663 (2007)
11. Menascé, D.A., Ewing, J.M., Gomaa, H., Malek, S., Sousa, J.P.: A framework for utility-based service oriented design in sassy. In: WOSP/SIPEW 2010 (2010)
12. Nagpurkar, P., Horn, W., Gopalakrishnan, U., Dubey, N., Jann, J., Pattnaik, P.: Workload characterization of selected JEE-based Web 2.0 applications. In: Proc. IEEE Int'l Symposium on Workload Characterization, pp. 109–118 (September 2008)
13. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed system models: a cautionary tale. In: USENIX NSDI 2006 (2006)
14. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for Web services selection with end-to-end QoS constraints. ACM Trans. Web 1(1), 1–26 (2007)
15. Apache ZooKeeper, `http://hadoop.apache.org/zookeeper/`