POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione

PhD course

Advanced topics in computer system performance analysis

09/11/2007

Performance evaluation of Web Services

Authors
Luca Cavallaro
Matteo Miraz
Tutor
prof. Giuseppe Serazzi

1 Introduction

The Service-oriented Architecture (SOA) paradigm foresees the creation of business applications from independently developed services. In this vision, providers offer similar competing services corresponding to a functional description of a service; these offerings can differ significantly in some Quality of Service (QoS) attributes like performance [1]. On the other side, prospective users of services dynamically choose the best offerings for their purposes. Using the SOA paradigm to build applications, services can be dynamically selected and integrated at runtime, so enabling system properties like flexibility, adaptiveness, and reusability.

In this context, the key point is to build applications through the composition of available services. The application can be specified as a process in Business Process Execution Language (BPEL) language in which the composed Web Services (WSs) are specified at an abstract level. The interfaces of individual services are specified using Web Service Description Language (WSDL), the W3C standard to model WSs interfaces, with documented quality properties. Specifically, the agreed performance attributes and levels can be specified by means of appropriate notations that augment the service specifications [2].

Applications built on services face different challenges: on one hand they should ensure that users experience the required performance, and on the other hand they have to maximize the resource utilization, so that provider incomes are maximize. Besides, due to the high dynamism of the applications, a deployment time quality prediction may lose significance during the application life time. In fact the quality of the application depends on the selected services that may be changed at run time.

In this paper we propose an approach to estimate a service oriented application performance. Our approach starts from the description of a service oriented application given using BPEL. This description is translated into a queue network and it is used to run a deployment time analysis of the application performance. This estimation can be used by application clients to monitor QoS. Since performances may change during application run time, due to changes in invoked services, the QoS needs to be monitored, in order to be kept up to date. For this task our approach suggests to annotate the BPEL process with performance indices and to monitor application QoS using WS-CoL (web service costraint language) framework [3]. Using WS-CoL it is possible to become aware of changes in application performance and to react to changes invoking services offering a better QoS.

The rest of the paper is organized as follows: section 2 describes in detail service oriented architectures, section 3 presents a case study to demonstrate our approach, section 4 presents our approach to performance estimation for service compositions, section 5 proposes the results of the approach on our case study, section 6 presents some considerations about the approach and proposes some future developments.

2 SOAs and Web Services

Service Oriented Architecture (SOA) is an evolution of distributed computing and modular programming. SOAs build applications out of software services. Services are relatively large, intrinsically unassociated units of functionality, which have no calls to each other embedded in them. They typically implement pieces of functionality most humans would recognize as a service, such as filling out an on line application for an account, viewing an on line bank statement, or placing an on line book or airline ticket order. Instead of services embedding calls to each other in their source code, protocols are defined which describe how one or more services can talk to each other. This architecture then relies on a business process expert to link and sequence services, in a process known as orchestration, to meet a new or existing business system requirement.
The goal of SOA is to allow fairly large chunks of functionality to be strung together to form ad-hoc applications which are built almost entirely from existing software services. The larger the chunks, the fewer the interface points required to implement any given set of functionality; however, very large chunks of functionality may not be granular enough to be easily reused. Each interface brings with it some amount of processing overhead, so there is a performance consideration in choosing the granularity of services. The great promise of SOA is that the marginal cost of creating the n-th application is zero, as all of the software required already exists to satisfy the requirements of other applications. Only orchestration is required to produce a new application.
A service oriented application usually has to meet the three following requirements:

- Interoperability between different systems and programming languages. The most important basis for a simple integration between applications on different platforms is a communication protocol, which is available for most systems and programming languages.

- Clear and unambiguous description language. To use a service offered by a provider, it is not only necessary to be able to access the provider system, but the syntax of the service interface must also be clearly defined in a platform-independent fashion.

- Retrieval of the service. To allow a convenient integration at design time or even system run time, a search mechanism is required to retrieve suitable services. The services should be classified as computer-accessible, hierarchical or taxonomies based on what the services in each category do and how they can be invoked.

2.1 Web service

A service orienteds system is not tied to a specific technology. SOA can be implemented using one or more of these protocols and, for example, might use

a file system mechanism to communicate data conforming to a defined interface specification between processes conforming to the SOA concept. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks. For this reasons SOA have been implemented using many technologies.
The most well known SOAs are *web services*. Web services are SOAs that make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages, just like HTTP or SMTP. Web services applications meet the three SOA requirements using SOAP, WSDL and UDDI. SOAP (Simple Object Access Protocol) is a protocol similar to RPC that allows to call methods on remote objects residing on different tiers of a network. WSDL (Web Service Description Language) is an interface description language that allows a service to expose information about its interface that may be used by a remote client to invoke a method on the service. UDDI (Universal Description Discovery and Integration) is a registry that allows a remote client to run queries to discover and use web services.

2.2 Service workflow and BPEL

One of the most interesting possibilities of SOA is the possibility for a service to invoke other services using their interfaces. In this way fine-grained services can be composed in more coarse-grained services. This mechanism makes possible incorporating services into business processes and workflows, specified using high level languages.
The most spread workflow language is BPEL. This is an XML language, developed by Microsoft to describe web service compositions. Using BPEL it is possible to define the workflow of a service oriented application. A BPEL workflow can be invoked by a client, just like a remote application, since it exposes to the client information about parameters to provide. BPEL does not provide the possibility to annotate performance indices into the workflow. For this reason an extension of the language is needed. Since BPEL syntax allows annotations the extension can be easily performed.

3 Case Study

A travel agency manager wants to move its company business on the web. He wants to offer a travel booking service. Users accesses to the service through a front end web application. After accessing the web application users can query a map service to plan their travel, then they can book train, plane or taxi to reach the selected destinations. Their tickets can be stored in a cart and, when they are done with their booking, they can pay for their order using a payment service.
The travel agency manager plans that all the customers, in order to access his

on line trip planner, must fill a registration form. Moreover he thinks that there will be two user classes that will use the trip planner: a class that uses the planner to plan trip from a start point to an arrival point, without intermediate stops, and a class that will plan trips featuring intermediate stops.
This process is built using web services. Each step that an user can perform is implemented by a service. Services are then composed in a BPEL workflow, following the schema reported in figure 1. Users access the planner through the front end. This service invokes a map service, called *mappe*, that returns the starting and destination point of the trip. The map service is invoked once by users planning punctual trips and at least twice by users planning trips with intermediate stops. The data retrieved by the map service are returned to the front end that uses them to book tickets on train, plane or tax rides, using three services called respectively *treno aereo taxi*. Tickets data are returned to the front end that uses them for concluding the order. Users booking a punctual itinerary are sent directly to the payment service, called *pagamento*, by the front end, when their ticket is chosen. Users booking an itinerary with intermediate stops are supposed to store their tickets using a shopping cart service, called *carrello*. Finally the reservation is returned to the users.
The travel agency manager chose some existing services to build his composition, and he wants to know which quality of service he can offer to his customers. Since he knows that service oriented applications can be dynamically changed in a cost-effective way he also wants to know how many customers he can serve without make them experiencing a low quality of service and, if any problem arises during the application life time, which services he should change in order to speed up the composition.

4 Our Approach

Our approach is articulated in four main steps:

- Build a service composition, choosing services to invoke
- Build a queue network to model the composition
- Assign performance indices to the invoked services
- Use the *JMT* suite to study system performances [4] [5].

The first step can be performed using BPEL to describe the composition. BPEL is an XML syntax language that allows to specify which services have to be invoked in a composition and how data have to be passed between invocations. The second step consists in modeling the composition as a queue network. Each service in the composition can be modeled as a station with queue. It is necessary to model the composition itself as a station with queue, since the composition can be considered as a service. This station will be called *frontend* and will have service time equals to the time the server hosting composition is busy, excluding the time of other services invocation. If the number of users

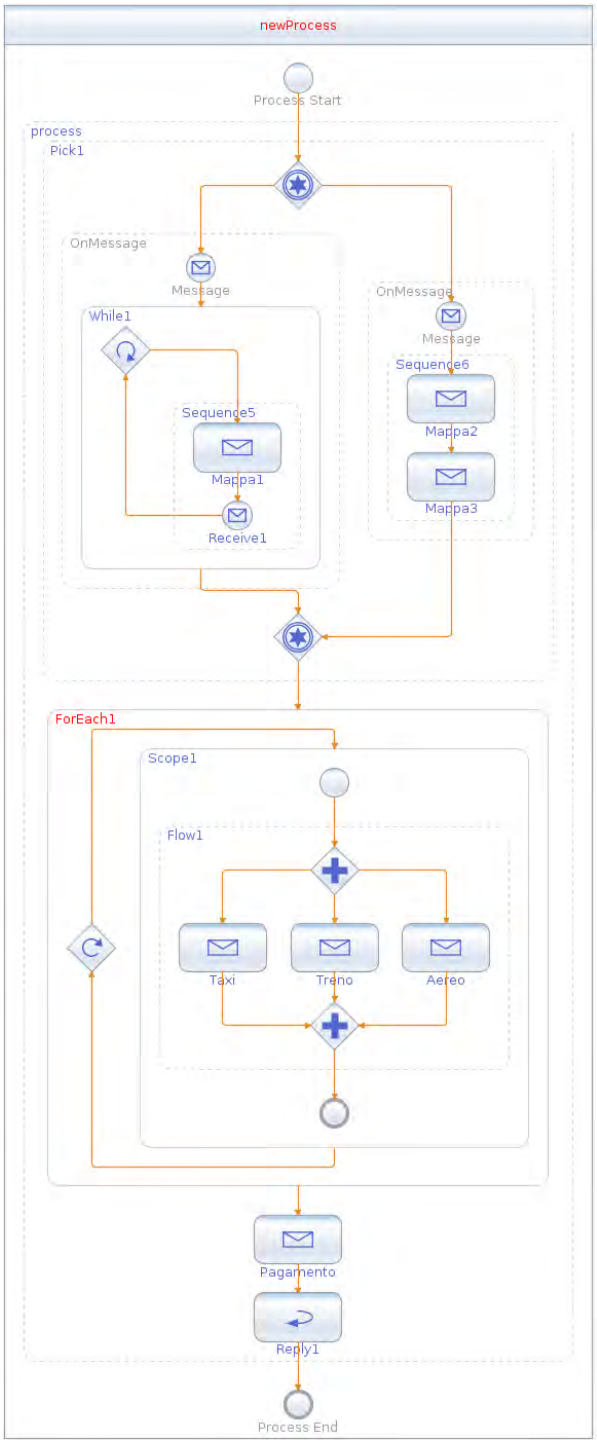

Figure 1: BPEL process

that can use the system is fixed (i.e. users need registration in order to use the composition) then it is necessary to model also users, using a station without queue. The service time of this station will be the think time.
The third step can be performed in two different ways, depending on the goal the performance study has:

- estimating the performances of a web service composition before deploying it
- estimating a web service composition QoS when it is running and deciding if it is necessary to replace a service in the composition.

If we want to estimate performances of a service composition at deploy time the data can be evinced from services logs, if we use existing services, or can be estimated by the system administrator, in case the services are built ad-hoc. In this case performance analysis can be used to estimate system run time performances with a prefixed load and to estimate when the system will become overloaded, using a what-if analysis.
If we want to estimate the composition QoS we need an historical log of service composition performance. This log can be obtained using a service monitor [6] to log residence times and visits for each service.
These data can be used to obtain service demands for each invoked service in the composition. Service demand lattice can be used to estimate bottlenecks in the system and to decide if any invoked service needs to be substituted. The approach is exemplified in section 5 using the case study of section 3.

5 Performance Evaluation

The approach we presented in section 4 was demonstrated on the case study reported in section 3.
The travel agency server composition was translated into a queue nework. The network was reported in figure 2. Each service in the composition is modeled as a station with queue, with the same of the service it models. The queue network presents, in addiction, two more stations. The station with queue called *frontend* models the BPEL composition itself, as the composition is a service. The station without queue called *users* models the registered users of the system. Since the system is closed we can assume that the number of users is fixed.

We started our analysis assuming that there is a set of services whose service time is known and we have already designed a BPEL workflow that uses those services. In this case we should instrument our application in order to retrieve the services' time requests. Afterwards we should model the service time requested by each service with an exponential process whose lambda is compatible with the measured mean.
Since the goal of this study is to evaluate the validity of the overall approach, we used our experience to propose a set of plausible values for the service time

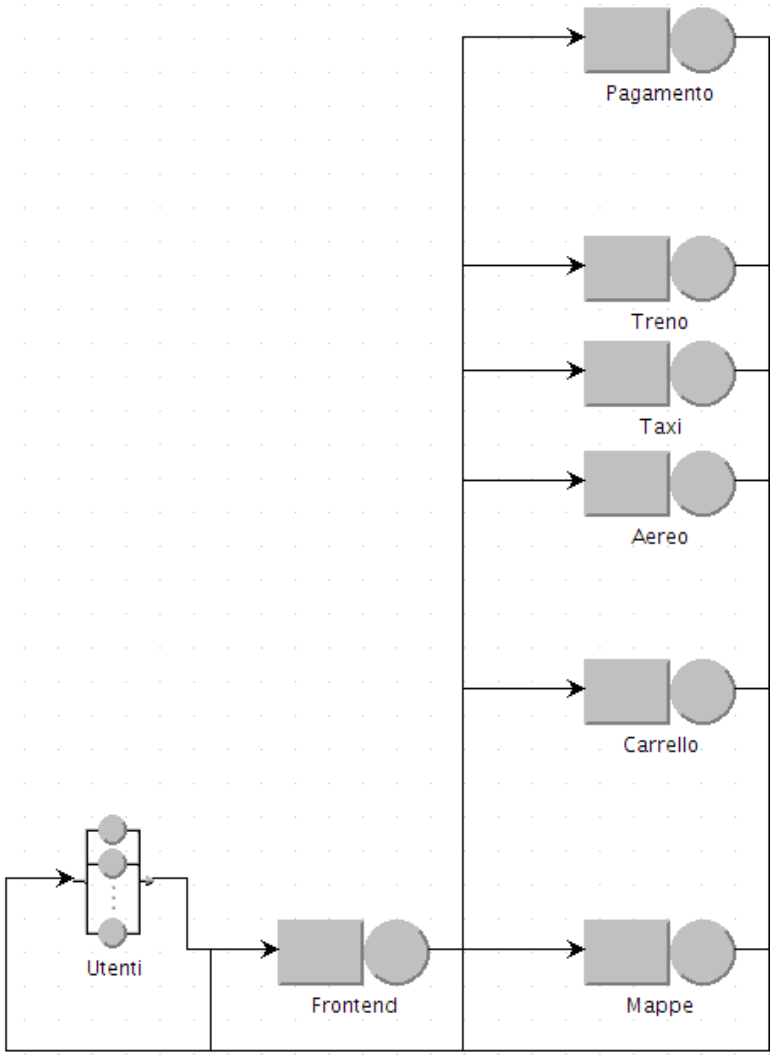

Figure 2: Qn Model of the travel agency service composition

of each web-service. The table 1 reports the mean values we selected for the various services.

	"*Puntuale*"	"*Itinerario*"
Frontend	0.0020	0.0066
Carrello		0.0050
Mappe	0.0025	0.0005
Treno	0.0050	0.0010
Aereo	0.0030	0.0008
Taxi	0.0030	0.0008
Pagamento	0.0010	0.0010

Table 1: Service request: mean values.

After selecting the service request, we had to analyze the work-flow and calculate the likelihood of visiting a service. Considering the performance model reported in figure 2, we had to decide the routing of the *fronted* service, and we proposed to use the probabilities listed in table 2.

	"*Puntuale*"	"*Itinerario*"
Carrello	0.00	0.20
Mappe	0.37	0.24
Treno	0.09	0.08
Aereo	0.09	0.08
Taxi	0.07	0.12
Pagamento	0.12	0.04

Table 2: Routing probabilities.

In order to retrieve the **D** matrix of the system, we run the simulation using JMT (Java Modeling Toolkit [5]) with only one user. In this mode the service requests of the different web-services represent the **D** matrix of the system. Table 3 reports the result of that simulation. Looking at the results it is possible to oversee eventual bottlenecks, analyzing the D for each service. The service with the highest D value (usually that value is referred with D_{max}) has the greater demand, thus will require more time than other services and will represent the bottleneck of the system. Since the case study has two classes, we can have two different bottleneck, one for each class. In this case we can notice that the *frontend* service represent the bottleneck for both classes, requiring respectively 0.704 s for the "Puntuale" class and and 2.148 s for the "Itinerario" class.

In order to improve the system, we propose to enhance the server that host the *pagamento*, doubling its speed. The time required for each visit on that server now is distributed as an exponential process with a lambda factor of 0.0040 for the "Puntuale" class and 0.0110 for the "Itinerario" class. We re-run the simulation with the new values, and we got the new values of the **D** matrix of the system, that are reported in 4. Analyzing this table we can notice that the *pagamento* is no longer the bottleneck of the system, so enhancing the server

	"*Puntuale*"	"*Itinerario*"
Frontend	**0.704**	**2.148**
Carrello	0.000	0.165
Mappe	0.610	1.951
Treno	0.079	0.252
Aereo	0.122	0.396
Taxi	0.074	0.593
Pagamento	0.510	0.161

Table 3: Initial service residence time.

that hosts it we are able to remove the original bottleneck.

	"*Puntuale*"	"*Itinerario*"
Frontend	0.350	1.027
Carrello	0.000	0.161
Mappe	**0.599**	**1.741**
Treno	0.075	0.261
Aereo	0.112	0.404
Taxi	0.074	0.605
Pagamento	0.496	0.167

Table 4: Simulated service residence with enhanced front-end service.

Continuing the analysis on the new system, we can detect a new bottleneck, that is the *mappe* service. That service requires 0.599s for processing a user of the "puntuale" class and 1.741s for a user of the "itinerario" class, that is more than any other service. Again, in order to improve the overall performance, we can enhance the speed of the *mappe* server. For this reason, we model the time requested by the *mappe* service to process each visit as an exponential process with a lambda factor of 0.0011 for the "itinerario" class and of 0.0050 for the "puntuale" class. We re-run the simulation and we got the new **D** matrix, that is reported in table 5.

Analyzing the result of the simulations, we can detect that the enhancement is useful since the bottleneck on the *mappe* service is resolved. The new system has a different bottleneck for each class: the one for the "puntuale" class is the *pagamento* service (imposing the D^p_{max} to 0.5 s), while for the "itinerario" class the bottleneck is the *Frontend* service (imposing the D^i_{max} to 1.041 s). In order to better understand the bottlenecks of the system, we plot the D values in a convex hull diagram, that is reported in figure 3.

Analyzing the convex hull diagram, it is possible to visualize better the bottlenecks of the system. The *frontend* and *pagamento* services are two potential bottlenecks, while all the other services are dominated by this two services. For

	"*Puntuale*"	"*Itinerario*"
Frontend	0.352	**1.041**
Carrello	0.000	0.166
Mappe	0.300	0.900
Treno	0.075	0.266
Aereo	0.112	0.400
Taxi	0.075	0.600
Pagamento	**0.500**	0.166

Table 5: Simulated service residence with enhanced front-end and map services..

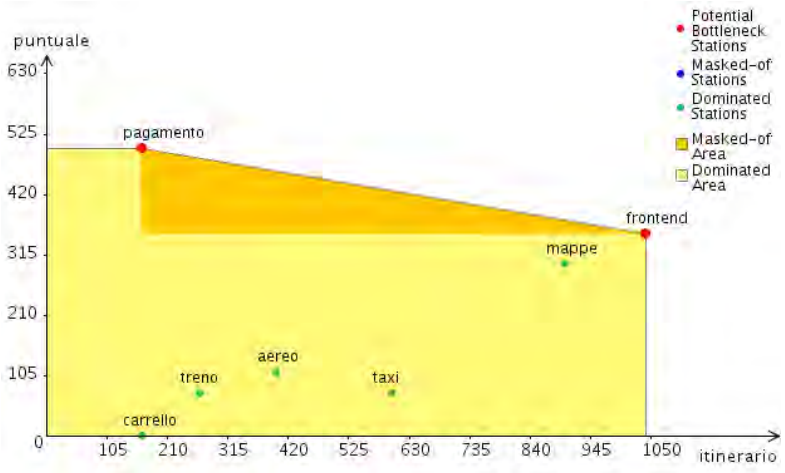

Figure 3: Convex Hull

this reason we can concentrate our analysis only on those two services. Moreover the convex hull shown that the bottlenecks services are used in a different way from the two classes; for this reason we investigate the effect of the population mix on the performance. In multiple workload mixes across multiple resources systems, changes in workload mixes can change the system bottleneck; the points in the workload mix space where the bottlenecks change are called *crossover points*, and the subspaces for which the set of bottlenecks does not change are called *saturation sectors*. The first analysis we performed is the calculation of these saturation sectors.

In figure 4 is reported the saturation sector of the system, calculated analytically with JMT. If the population has only customers of the "itinerario" class, the saturated service is *frontend*; otherwise if the population is composed only by customers of the "puntuale" class, the saturated service is *pagamento*. If the users of the system are composed by a particular mix of these two classes, both servers are saturated. If the system is in this situation, it has the maximum

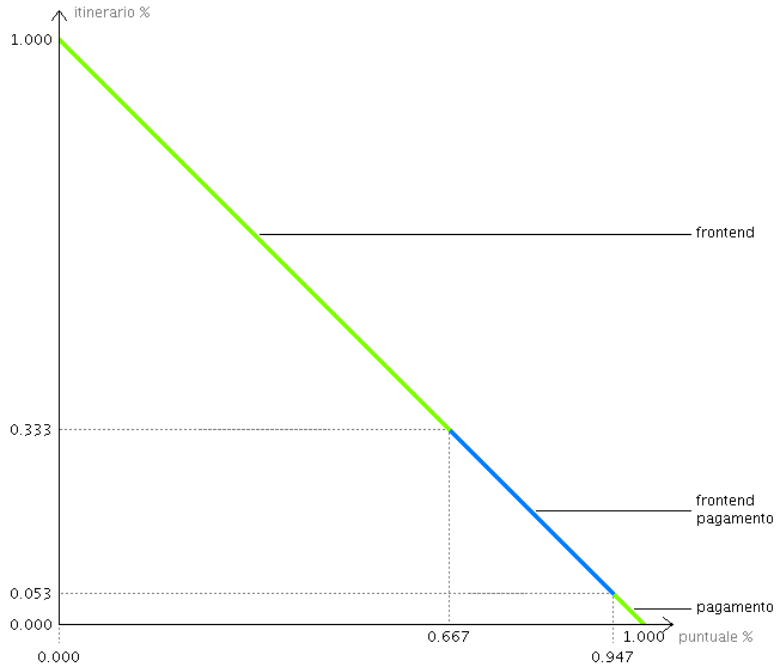

Figure 4: Saturation sectors

throughput since two servers are used at their maximum capacity. The system that we are analyzing has an optimal mix that has customers of the "puntuale" class between the 66% and 95%.
In order to validate this result, we performed a *what-if* analysis with a fixed population size and varying the mix of the two classes. In figure 5 is reported the utilization of the three more used services (*frontend*, *pagamento* and *mappe*) with respect to the population mix. The highlighted region is the common saturation sector: if there are the correct mix of users, there are two servers that are used almost completely. In this case we have the highest throughput, as measured and plotted in figure 6.

Analyzing the results of this analysis, we can state that if we are smart regarding the selection of users, we can serve more people without any server upgrade. If we are able to dynamically monitor the execution of the workflow, understanding the class that the user being served belongs to, we can detect the current population mix. At this point we can understand if the system is saturated, and in this case we should also check if the population mix is within the common saturation sector. If the current mix is not in the saturation set, we should take actions that tries to modify that mix. If the error is only a transient one, we can adopt an intelligent routing strategy that select the job belonging to the right class. Otherwise the problem is persistent, thus we can adopt ad-hoc promotions that helps in filling the gap in the population mix.

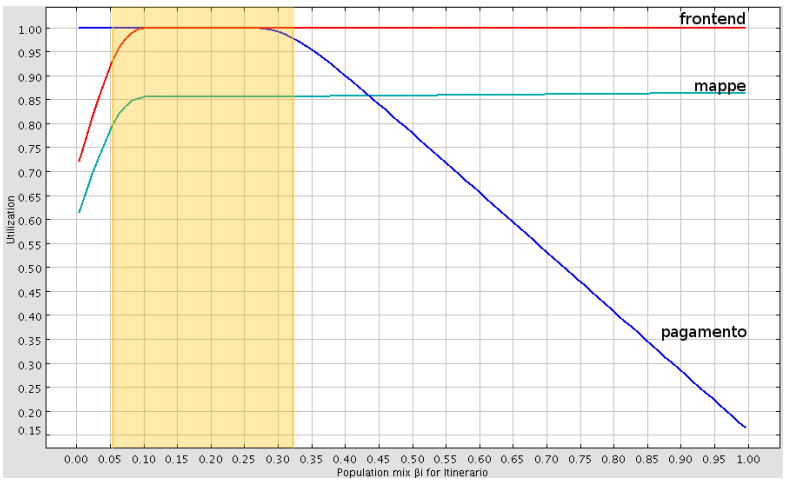

Figure 5: What-if analysis: utilization w.r.t. population mix

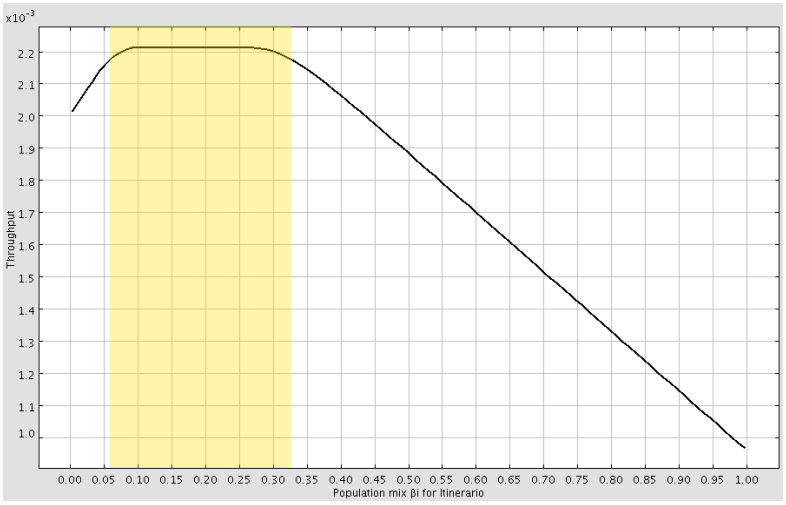

Figure 6: What-if analysis: throughput w.r.t. population mix

6 Conclusion and Future Works

In this paper we described an approach for the performance prediction of service-based applications. The approach can be used both to estimate performances of a service composition at design time and to analyze composition QoS at run time.
We demonstrated our approach on an example composition describing a travel agency application. In our demonstration we could estimate the application performances at deployment time, manually creating a QN model of our composition using as performance indices distributions of the service times of the services invoked in the composition. We also used these data to estimate bottleneck of the system, to study saturation sectors and to plan a speed up of some invoked services.
The approach was run manually, but it can be automatized. As a first step towards the realization of this approach an automatic translation from the BPEL specification to a queue network model is needed. This translation can take as input a BPEL file and give as output a JSymGraph model file. This model allows analyzing the queue network derivated from the BPEL composition using JMT. To make possible this translation we need to extend the BPEL language to make it possible to annotate QoS indices directly into the composition description.

References

[1] Daniel A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[2] Andrea DAmbrogio. A model-driven wsdl extension for describing the qos ofweb services. *icws*, 0:789–796, 2006.

[3] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In *ICSOC05, 3rd International Conference On Service Oriented Computing*, 2005.

[4] Serazzi G Bertoli M., Casale G. The jmt simulator for performance evaluation of non-product-form queueing networks. In *ANSS07 Spring Simulation Multiconference*, 2007.

[5] G. Serazzi. Java modeling tools. http://jmt.sourceforge.net/.

[6] Luciano Baresi and Sam Guinea. Dynamo and self-healing bpel compositions. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 69–70, Washington, DC, USA, 2007. IEEE Computer Society.