

# Enhancing BPEL Processes with Self-tuning Behavior

Adina Mosincat

Faculty of Informatics

University of Lugano, Switzerland

*adina.diana.mosincat@usi.ch*

Walter Binder

Faculty of Informatics

University of Lugano, Switzerland

*walter.binder@usi.ch*

**Abstract**—The performance of BPEL processes depends on the composing web services. Monitoring web service performance and adapting to changes in service performance are essential for creating self-tuning BPEL processes. BPEL lacks constructs for performance monitoring and offers little support for runtime adaptability. In this paper, we present a monitoring and dynamic binding infrastructure for BPEL that transparently enhances existing BPEL processes with self-tuning behavior. The portable infrastructure relies on an automated transformation of BPEL processes and is compatible with any BPEL engine. Its flexible monitoring mechanism supports both complete and more efficient sampling-based monitoring. The infrastructure allows for pluggable service selection strategies. We present a probabilistic strategy, where the service selection probability depends on the previously monitored service performance. A thorough evaluation confirms that our infrastructure significantly increases BPEL throughput in settings where web service performance fluctuates.

**Keywords**-BPEL processes; self-tuning; dynamic service binding; performance monitoring

## I. INTRODUCTION

Robustness and high performance are important features of any software. For complex, service-oriented systems, achieving the two aforementioned features is particularly challenging, because these systems typically depend on several distributed services. Web service compositions, usually defined as processes in the Business Process Execution Language (BPEL<sup>1</sup>), are good examples of such service-oriented systems. In a dynamic environment, such as the web, the ability of a process to adapt to changes is critical [1]. For BPEL processes, this translates into the ability of every process instance<sup>2</sup> to dynamically select the services to bind at runtime. Moreover, with the boost of web services, being competitive is a strong requirement. Processes need not only recover from failures of bound services, but they also have to maintain and improve performance, acting on changes such as the availability of new services.

BPEL faces two shortcomings: it offers limited runtime adaptability and it does not specify any means of performance monitoring. While it is possible to use the dynamic

partner link assignment to change a partner link that represents a service in a process at runtime, this feature results in coupling the dynamic binding with process business logic and does not allow adding new available services after the process has been deployed. The BPEL specification does not support monitoring of bound services, and the monitoring of the execution of process instances is left to the BPEL engine implementor which may or may not provide monitoring functionality in a proprietary way. The issue of monitoring is vital when performance is considered. The inability of gathering information on a running process instance cuts down any chance of self-tuning and leaves the process without any control over its performance.

The infrastructure presented in this paper addresses these shortcomings, providing mechanisms to select the service to bind at runtime and to monitor the execution of the bound service, learning about the changes in the environment and reacting on the gathered information. Our approach separates *service selection* from the process business logic, providing runtime adaptability for the process through dynamic binding, by automatically transforming the process at deployment time. In this way, we achieve transparency for the process developer, who does not have to provide any special constructs, and full compatibility with standard BPEL engines.

Self-tuning behavior is crucial in the lifetime of a process because it allows the process to gain independence from changes in the composing web services. Our infrastructure enhances the process with the ability to observe changes in service performance through a monitoring mechanism and to adapt itself to these changes through dynamic service selection and binding mechanisms. Thanks to dynamic binding, each process instance may use different service bindings.

Our monitoring mechanism supports both complete monitoring and sampling, allowing to balance between monitoring overhead and the amount of gathered service execution statistics that enable self-tuning. The service selection mechanism maintains pools of functionally equivalent services and periodically (re-)assigns service selection probabilities according to the monitored service performance; if service *A* has recently outperformed service *B*, *A* will be selected with higher probability than *B* in the near future. Hence, the service requests originating from the BPEL processes are

<sup>1</sup><http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>

<sup>2</sup>A process instance is created by the BPEL engine when one of the receive activities (a start activity) in the process is triggered, and is ended after completion of the corresponding reply activity.

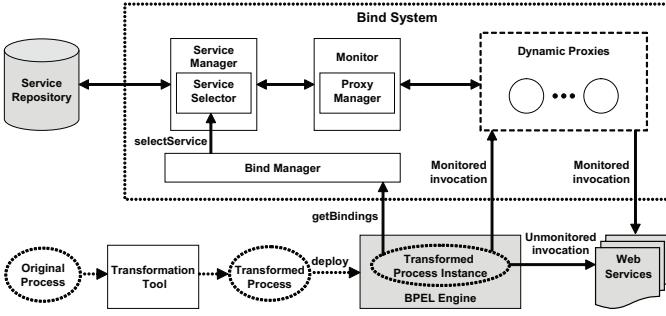


Figure 1. System architecture

probabilistically distributed between functionally equivalent services, taking the recently observed service performance into account.

In previous work [2], we presented an automated solution to transparent fault handling of BPEL processes, supporting immediate rebinding for failed stateless services, as well as process restart and rebinding for failed stateful services. In this paper, we introduce an infrastructure that extends the generic architecture of our previous self-healing system with novel monitoring and service selection mechanisms in order to enable self-tuning of processes. This paper improves, refines and evaluates the approach outlined in our poster [3].

The original, scientific contributions of this paper are twofold:

1) We introduce a new approach to transparent BPEL process self-tuning. We provide a flexible process monitoring mechanism that allows balancing the amount of data collected and the monitoring overhead. Furthermore, we present a probabilistic service selection algorithm that takes the monitored service performance into account.

2) We implemented our architecture and a testbed using state-of-the-art technologies. The testbed includes web services, BPEL processes, workload generators, as well as performance measurement tools. It allows simulating different web service performance models using discrete time Markov chains. With the aid of the testbed, we thoroughly evaluate the performance of our system in different scenarios.

This paper is structured as follows. Section II introduces the concepts used in this paper and presents the architecture of our infrastructure. In Section III we describe the monitoring mechanism, and Section IV explains the details of the service selection algorithm. We evaluate our approach in Section V and discuss related work in Section VI. Section VII concludes this paper.

## II. ARCHITECTURE

The following definitions will be used throughout the paper:

- Service type – Unique identifier of a group of functionally equivalent services that can substitute for each other.

- Selection probabilities – For each service type, a tuple containing the probabilities of the corresponding services to be selected. The selection probabilities are periodically recomputed according to the monitored service performance.
- Monitoring probability – Probability of a service to be monitored.
- Observing interval – Period of time in which service performance is monitored, while the selection probabilities tuples remain unchanged.

We refer to our infrastructure as the *Bind System*. Figure 1 shows the system architecture.

### A. Components Introduced in Prior Work

The *Bind System* extends the generic architecture introduced in [2], which transparently supports dynamic binding of services and failure recovery for BPEL processes. This subsection presents important background for understanding the *Bind System*.

The *Transformation Tool* automatically transforms the process at deployment time to use our infrastructure, assuring complete transparency to the process developer. The logic inserted in the transformed process handles the dynamic binding of services for each process instance. The transformed process also supports failure recovery and automatic process restart in case of failure of a stateful service.

The *Bind Manager* constitutes the interface to the process. When a process instance is created, it first requests the services to be used from the *Bind Manager*. The *Bind Manager* keeps track of the service bindings for each process instance. The main component of the *Bind Manager* is a web service. All interactions of the process with the *Bind System* are web service invocations, thus assuring compatibility with any BPEL engine. For performance reasons, the *Bind System* is typically deployed on the same machine hosting the BPEL engine.

A detailed description of the *Transformation Tool* and the *Bind Manager* can be found in [2].

### B. Bind System

In this section we describe the novel components of the *Bind System* that are important for the monitoring capabilities of the system and for achieving process self-tuning behavior. (see Figure 1)

The *Service Manager* has the following responsibilities:

- Service classification. The *Service Manager* builds the service types by matching functionally equivalent services that can substitute for each other. All services of the same service type must offer the same interface. Our current *Service Manager* implementation assumes the service type information to be specified by the service provider.
- Dynamic selection of services. Services are selected according to ranking criteria implemented in a *Service*

*Selector*. The concrete *Service Selector* presented in this paper implements an algorithm that favors the selection of services with good performance, distributing the service requests originating from different process instances to those services that have recently performed best.

- Creation and periodic update of the selection probabilities for each service type.
- Service quarantine. A service is quarantined if it fails or if its performance is continuously poor. A quarantined service is excluded from selection for at least one observing interval.

The *Monitor* provides statistics on service performance. It has the following responsibilities:

- Creation of *Dynamic Proxies*. A *Dynamic Proxy* can measure the response time of any service of a given service type, if the service is invoked through the *Dynamic Proxy*.
- Collection of measurements from the *Dynamic Proxies*.
- Aggregation of measurements and provision of service response time statistics to other parts of the system.

The dotted arrows in Figure 1 illustrate the transformation of a process. At deployment time the process is transformed by the *Transformation Tool*, which replaces every service used in the original process with the service type provided by the *Service Manager*; the resulting transformed process is deployed in the BPEL engine.

When a process instance starts, it requests the services (*getBindings* in Figure 1) to bind from the *Bind Manager*. To this end, the process instance sends the list of all service types it may invoke to the *Bind Manager* and receives back the concrete service bindings. The *Service Selector* determines the bindings to be used by a process instance according to the current selection probabilities. While a process instance will use the same service bindings throughout its execution (unless it fails, in which case alternative bindings are requested from the *Bind Manager* [2]), different process instances may receive distinct bindings for each service type.

Each service binding returned by the *Bind Manager* may either directly refer to a service of the requested service type, or it may refer to a *Dynamic Proxy*. In the former case, the invocations of the service by the process instance will not be monitored (*Unmonitored invocation* in Figure 1), whereas in the latter case, the service invocations will be monitored by the *Dynamic Proxy* (*Monitored invocation* in Figure 1). For each service, the *Bind Manager* decides whether monitoring is desired based on the service's monitoring probability.

### III. MONITORING

In order to be able to adapt to changes in the environment, a system must be aware of these changes. The role of the *Monitor* is to observe the performance of services and

to provide aggregated performance statistics to the *Service Selector*, which leverages these statistics when recomputing the selection probabilities.

The *Proxy Manager* handles the creation of *Dynamic Proxies*, which measure and store the response time for each service invocation. While each *Dynamic Proxy* is specific to a service type<sup>3</sup>, it can handle invocation of any service of that service type. When a new service type is registered in the system, the *Proxy Manager* creates a pool of *Dynamic Proxies* (at least one) for the service type. The number of *Dynamic Proxies* can be dynamically changed for each service type so as to ensure that they do not become performance bottlenecks.

Upon the start of a new process instance, the *Bind Manager* communicates the selected services to the process instance by sending the corresponding service endpoint references<sup>4</sup>. If a selected service is to be monitored, the *Proxy Manager* chooses a *Dynamic Proxy* from the pool of the corresponding service type. The *Bind Manager* then returns the endpoint reference of that *Dynamic Proxy* to the process instance. The endpoint reference of the selected service to be invoked by the chosen *Dynamic Proxy* is embedded within the returned endpoint reference as *ReferenceParameters*. Hence, the transformed BPEL process is not aware whether invocations of a service will go through a *Dynamic Proxy* or not; this approach simplifies process transformation, because it is not necessary to integrate any logic related to monitoring into the transformed process.

When a *Dynamic Proxy* receives a request, it first inspects the header of the SOAP message, building the endpoint reference of the service to invoke from the *ReferenceParameters*. Then the *Dynamic Proxy* invokes the service while measuring the response time. Each *Dynamic Proxy* maintains a mapping from service endpoint references to lists of measured service response times. The *Proxy Manager* collects these measurements after each observing interval; afterwards, the measurement data in the *Dynamic Proxy* is discarded.

*Dynamic Proxies* introduce monitoring overhead. Hence, it is important to choose a monitoring strategy that allows gathering enough information to accurately assess service performance while causing as little overhead as possible. We consider two distinct strategies: complete monitoring and sampling-based monitoring. For each service, the monitoring probability determines the percentage of bindings returned by the *Bind Manager* that refer to a *Dynamic Proxy* as endpoint reference. A monitoring probability of 100% for each service implies complete monitoring, whereas a monitoring probability below 100% for at least one service relates to sampling.

<sup>3</sup>*Dynamic Proxies* can be automatically generated from the interface corresponding to a service type, as provided in a WSDL file.

<sup>4</sup><http://www.w3.org/Submission/ws-addressing/>

At system startup, in the first observing interval, we always resort to complete monitoring so as to gather as much measurement data as possible. In subsequent observing intervals, different monitoring strategies are possible:

- All services are monitored with the same given monitoring probability.
- For each service, the monitoring probability is computed as a function of the service's selection probability, reflecting the service's recent performance (fast services will be assigned higher selection probabilities than slow services). For example, the function

$\text{monitoring probability} = \max\{3/10, (1 - \text{selection probability})\}$  ensures that services with low selection probability are monitored with higher probability than services with high selection probability and that each service is monitored with a probability of at least 30%.

After each observing interval, the *Proxy Manager* collects and aggregates the measurements from the *Dynamic Proxies*. The aggregation function is configurable. For example, the arithmetic mean, the geometric mean, or the median of the measurements in the last observing interval may be used. Data from previous observing intervals may be included as well, e.g., using some form of weighted average. If a service was not monitored in the last observing interval, the service's previous performance statistics are preserved.

#### IV. SERVICE SELECTION

In this section, we describe the selection algorithm implemented by our *Service Selector* and explain the mechanism of quarantining failing services.

Dynamic service selection is based on ranking criteria implemented in *Service Selectors*. *Service Selectors* are deployed in the *Service Manager* and use the provided service type classification. The ranking criteria implemented by a *Service Selector* typically relate to non-functional properties, such as Quality-of-Service (QoS) parameters (e.g., response time). Our infrastructure supports pluggable *Service Selectors*.

In this section, we present a specific *Service Selector* that leverages our monitoring mechanism and implements a probabilistic selection algorithm that takes the monitored service response time into account. The selection algorithm presented below aims at balancing the service requests between equivalent services, favoring those services that recently performed better.

We consider a service type  $T$  and assume there are  $n^T$  services  $S_i^T$  of type  $T$  ( $1 \leq i \leq n^T$ ). The selection probabilities corresponding to service type  $T$  are defined in the tuple  $sp^T$ , where  $sp_i^T$  is the probability that service  $S_i^T$  is selected upon a binding request for service type  $T$ . For all  $i$  ( $1 \leq i \leq n^T$ ),  $0 < sp_i^T \leq 1$ , and  $\sum_{i=1}^{n^T} sp_i^T = 1$ . In the initial state, at system startup,  $sp_i^T = 1/n^T$ .

The *Monitor* provides the average response time  $rt_i^T$  for each service  $S_i^T$ . The configurable function  $util(rt_i^T)$

computes an utility value for service  $S_i^T$  depending on  $rt_i^T$ . In this paper, we use a simple utility function of the form  $util(rt_i^T) \equiv 1/rt_i^T$ . That is, the utility of a service is indirectly proportional to its average response time.

The *Service Selector* executes the following algorithm upon each binding request for service type  $T$ :

- 1) Let  $r$  be the next pseudo-random value from a uniform distribution,  $0 \leq r \leq 1$ .
- 2) Select the service  $S_m^T$ , where  $m = \min\{i : (1 \leq i \leq n^T) \wedge (r \leq \sum_{j=1}^i sp_j^T)\}$ .

After each observing interval, the tuple  $sp^T$  is updated according to the utility values  $u_i^T = util(rt_i^T)$  computed from the most recent average response times provided by the *Monitor*:  $sp_i^T = u_i^T / (\sum_{j=1}^{n^T} u_j^T)$ .

A newly available service  $S_{new}^T$  ( $new = n^T + 1$ ) is included in the selection probabilities tuple  $sp^T$  upon the next update of  $sp^T$ . Initially,  $rt_{new}^T = (\sum_{i=1}^{n^T} rt_i^T) / n^T$ ; afterwards,  $n^T$  is incremented.

The success of our service selection strategy depends on whether the selection probabilities properly reflect the current state of the environment. Thus, the observing interval must be chosen such that on the one hand, it gives sufficient time for the *Monitor* to gather new measurements, and on the other hand, the selection probabilities are updated frequently enough.

On service failure, the selection probabilities of the corresponding service type are immediately updated by removing the failed service from the tuple and recomputing the selection probabilities for the remaining services. The failed service is put in quarantine. All services that fall below a given performance threshold, which is a configurable value, are also put in quarantine, which means that they are taken out of selection for at least one observing interval. The quarantining period is configurable.

After the quarantining period has elapsed, a service becomes again eligible for selection upon the next update of the corresponding selection probabilities. Its average response time is assumed to correspond to the slowest service of the same service type in the previous observing interval. If all services of a service type get quarantined, the quarantine of all services is immediately canceled and the system starts from the initial state.

#### V. EVALUATION

In this section, we evaluate the *Bind System* in five different settings in order to explore (1) the response time of processes using the *Bind System* when service performance fluctuates, compared to processes that statically bind services; (2) BPEL throughput in the presence of functionally equivalent services with different performance; (3) BPEL throughput when service performance fluctuates; (4) the decrease of BPEL throughput due to increasing service performance fluctuations; (5) BPEL throughput depending

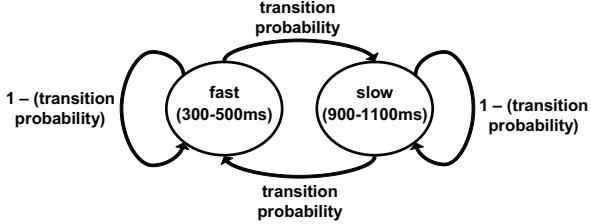


Figure 2. Discrete time Markov chain modeling web service response time

Table I  
SETTING PARAMETERS. THE MONITORING PROBABILITY APPLIES FOR ALL SERVICES.

Setting #	Transition probability	Monitoring probability	Reference
1	30%	40%	Original process
2	0%	40%, 100%	Round-robin process
3	30%	40%, 100%	Round-robin process
4	0–100%, step 10%	100%	Round-robin process
5	30%	10–100%, step 10%	—

on the monitoring probability (i.e., the trade-off between the overhead incurred by monitoring and the amount of collected monitoring information).

We implemented a testbed including web services, BPEL processes of configurable complexity, workload generators, as well as performance measurement tools. The testbed models web services' performance with discrete time Markov chains [4]. We are modeling the performance of *external* web services, which is fluctuating depending on the load due to third-party service requests. We assume that the load created by our BPEL processes on the web services does not significantly affect their performance. That is, in this evaluation we are not considering highly concurrent process execution, where the performance of web services is strongly impacted by the concurrent requests from our BPEL processes.

For the results reported in this paper, we use a simple service model with the two states *fast* (corresponding to a lightly loaded service) and *slow* (representing a heavily loaded service) as shown in Figure 2. In the state *fast*, the service response time is 300–500ms (randomly chosen upon each invocation); in the state *slow*, the response time is 900–1100ms. The *transition probability* is the probability that the service changes state. We configure the two-state service model with a time slot of 30 seconds, i.e., after each elapsed 30 seconds, it is randomly decided whether the state is kept or changed based on the transition probability.

Table I summarizes the parameters of the different evaluation settings. We are using 10 functionally equivalent services  $S_1$ – $S_{10}$ , each implementing the two-state service model. Initially,  $S_1$ – $S_5$  are in the state *fast*, whereas  $S_6$ – $S_{10}$  are in the state *slow*. We base our evaluation on a BPEL process with three service invocations. As reference, we

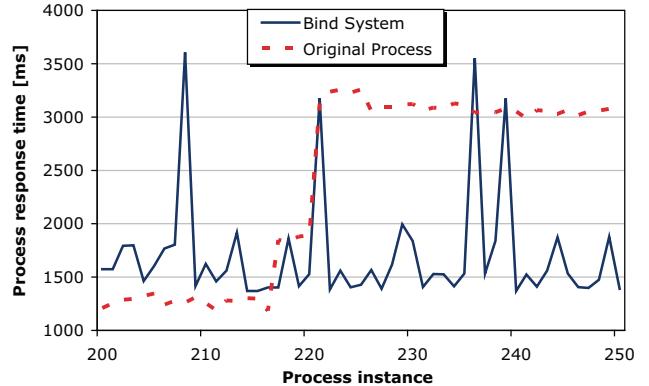


Figure 3. *Bind System* versus original process: Response time for process instances 200–250. Transition probability 30% and monitoring probability 40%. The original process always binds service  $S_1$ . Parameters correspond to setting #1 in Table I.

use the *original process* that statically binds service  $S_1$ , respectively a process with a hard-coded *round-robin* service selection mechanism (without using our *Bind System*). The value of the observing interval is 30 seconds. The response time reported through monitoring represents the arithmetic mean of the response times for a service measured within one observing interval.

In the first setting, we investigate process response time, whereas in the other settings we explore BPEL throughput. For the measurements in settings 2–3, BPEL throughput is computed for each 200 completed process instances. In settings 4–5, BPEL throughput is computed for 2000 process instances. The process instances are started by 5 concurrent threads. All measurements were repeated 15 times and we report the median of these measurements.

Our implementation uses Java 5, Apache Axis 1.4, and BPEL 2.0; as BPEL engine we use ActiveBPEL<sup>5</sup> 4. Both the *Bind System* and the BPEL engine are deployed in an Apache Tomcat 4.1.24 installation. Our measurement machine is an Intel Core 2 Duo (2.4GHz, 2GB RAM) running Mac OS X v10.4.

Figure 3 illustrates the adaptive behavior of the transformed process.<sup>6</sup> We compare the response time of the original process that statically binds service  $S_1$  with the response time of the transformed process, while the services are changing state with a probability of 30%. Figure 3 presents the response time of 50 process instances during the transition period of service  $S_1$  from state *fast* to state *slow*. The response time of the original process only depends on the performance of service  $S_1$ , degrading when  $S_1$  changes from state *fast* to state *slow*. In contrast, the transformed process maintains stable performance by adapting to the

<sup>5</sup><http://www.activevos.com/>

<sup>6</sup>We refer to the process that uses the *Bind System* as the *transformed process*.

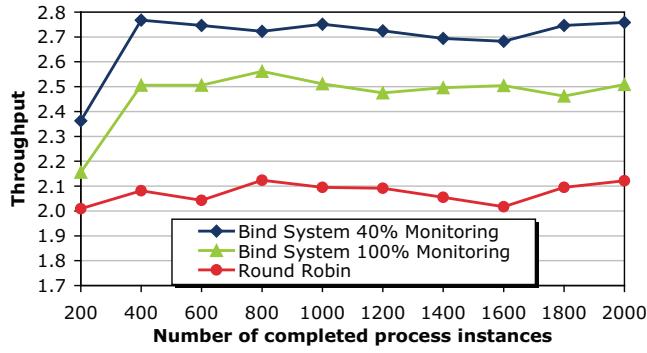


Figure 4. *Bind System* versus Round Robin: Throughput (measured for each 200 process instances) for **transition probability 0%** and monitoring probability 40%, respectively 100%. Parameters correspond to setting #2 in Table I.

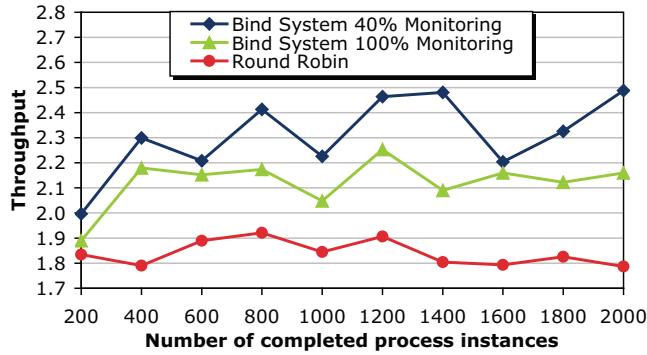


Figure 5. *Bind System* versus Round Robin: Throughput (measured for each 200 process instances) for **transition probability 30%** and monitoring probability 40%, respectively 100%. Parameters correspond to setting #3 in Table I.

change in service  $S_1$  and switching to services that are in the *fast* state.

The original process outperforms the transformed process when  $S_1$  is in the state *fast*, because the *Bind System* incurs some overhead (monitoring, as well as interactions of the transformed process with the *Bind Manager* upon process instance creation). The variations of the transformed process' response time are caused by the different services that are bound. The selection algorithm assures every service has a chance to be selected, which enables the adaptation of the process when services change state. This explains the peaks in Figure 3, where a few process instances have response times of about 3500ms when a slow service is selected. Overall, the overhead due to the *Bind System* is outweighed by the performance gains when service performance fluctuates.

Figure 4 shows the throughput for the round-robin process and the transformed process over a period when the environment remains stable (i.e., transition probability 0%). For the transformed process, the lowest throughput is measured in the beginning, because it takes some time for the *Bind*

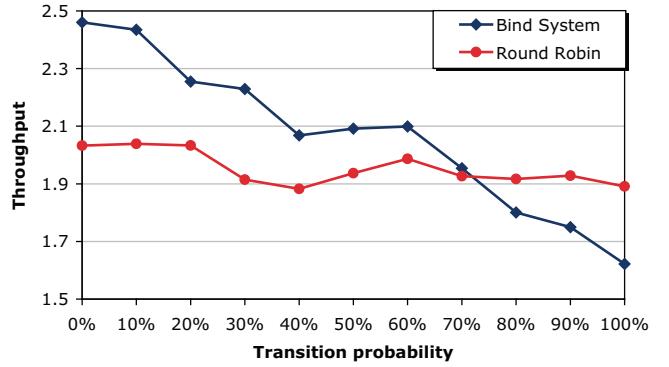


Figure 6. Throughput for different transition probabilities. Monitoring probability 100%. Parameters correspond to setting #4 in Table I.

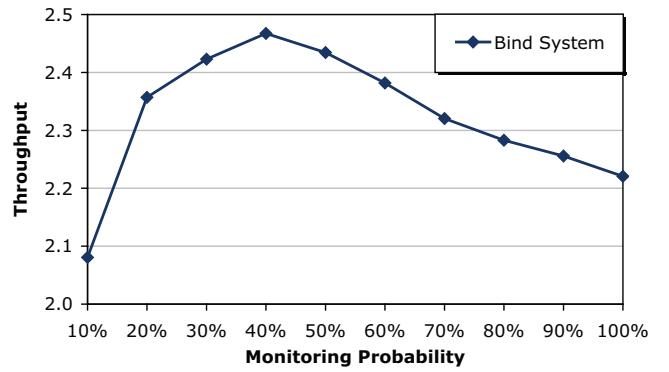


Figure 7. Throughput for different monitoring probabilities. Transition probability 30%. Parameters correspond to setting #5 in Table I.

*System* to gather monitoring data and to build the tuple of selection probabilities. On average, when using complete monitoring, the throughput improvement over the round-robin process is 19%, while when using sampling with a monitoring probability of 40%, the throughput improvement is 30%.

Figure 5 shows the throughput when services change their state with a probability of 30%. In this case, the *Bind System* has to adapt to changes and its performance decreases during the periods between a service state change and the subsequent update of the selection probabilities tuple, i.e., when the selection probabilities do not reflect the current state of the environment. Still, when using complete monitoring, the average throughput improvement of the transformed process over the round-robin process is 16%, while when using sampling with a monitoring probability of 40%, the average throughput improvement is 25%.

Figure 6 illustrates the decrease of BPEL throughput when service performance fluctuations increase. For a transition probability of more than 70%, the transformed process no longer outperforms the round-robin process, because the observing interval is too long to react to the frequent service

state changes. That is, if a service is fast in one observing interval, this will be reflected in the selection probabilities of the subsequent observing interval; however, chances are high that the service has already become slow in the meantime.

Figure 7 shows the dependence of throughput on the monitoring probability. We keep the transition probability at 30% and increase the monitoring probability from 10% to 100%, exploring the best trade-off between little monitoring overhead and the amount of monitoring data collected. While for a monitoring probability below 30% the amount of gathered monitoring information is not enough to compute selection probabilities tuples that accurately reflect the environment, for a monitoring probability higher than 50% the monitoring overhead results in a throughput decrease.

In summary, our evaluation confirms that the *Bind System*, using sampling-based monitoring and a probabilistic service selection strategy, can significantly improve throughput in settings where several functionally equivalent external services are available and services' performance fluctuates.

## VI. RELATED WORK

Related work on adaptation of service compositions addresses self-healing [5]–[10], but scarcely touches on self-tuning [7], [11], whereas achieving self-tuning behavior is the focus of this paper.

VieDAME [11] is an aspect-based service monitoring and selection system that intercepts SOAP messages and dynamically replaces services used in the BPEL process. It monitors process execution and gathers information on the QoS of services that is stored in a database for future use. The services are selected based on defined selectors and can be adapted to replace services that implement a different service interface. Unlike the VieDAME system that uses an engine adapter to extend engine functionality, our *Bind System* is completely transparent to the BPEL engine. In [11], the overhead caused by the VieDAME system is explored; however, performance improvements due to the monitoring and service selection mechanisms are not reported.

RobustBPEL2 [6], [7] makes use of a dynamic proxy to discover alternative services upon failure of services that have previously been marked for monitoring. In contrast, our *Bind System* is completely transparent to the user. In RobustBPEL2 all invocations of marked services are monitored, but the monitoring information is only used to react on service failure respectively on service timeout. Hence, although monitoring overhead is continuously incurred, RobustBPEL2 does not improve system performance. In contrast, our approach actively improves system performance thanks to flexible, sampling-based monitoring and to probabilistic service selection leveraging the monitoring information.

Another solution based on an aspect-oriented engine extension of ActiveBPEL is the *Dynamo* [5] system. While

providing self-healing capabilities to the process with the aid of complex recovery strategies specified using two domain-specific languages (WSCoL, the *Web Service Constraint Language* and WSReL, the *Web Service Recovery Language*), the *Dynamo* framework does not address self-tuning. An interesting related approach synthesizes different monitoring techniques, based on aspects, to provide recovery capabilities for compositions [12].

MASC [8] is a middleware extending .Net and WS-Policy. MASC adopts a policy-based approach and performs adaptation actions at the SOAP messaging layer and the process orchestration layer. MASC addresses business exceptions and runtime faults, whereas our aim is to improve system performance.

Traditional workflow systems, such as E-flow [13], address adaptability using a formal approach. E-flow focuses on modeling and formal verification. Service selection rules can be specified so as to guide dynamic service discovery and service selection.

Another approach to self-healing that acts at the service communication level is WS-Diamond [9]. WS-Diamond intercepts and monitors the SOAP messages exchanged between service providers and requesters and provides substitutions for the web services causing QoS degradation.

In [14], a BPEL process is monitored by transforming its specification into a business protocol and storing the execution traces of services into a database which is later used for querying.

A solution to adaptive compositions by ranking services according to their failure rates and popularity computed by link analysis is proposed in [10]. The developer can select the highly ranked services to use in compositions. The aim of the proposed solution is to reduce the number of service failures experienced by the consumers.

A different approach to optimize system performance is taken in [15], by taking hardware resources into consideration. The proposed solution decomposes end-to-end service response time requirements into atomic service response time requirements. Then, it collocates atomic services with similar QoS requirements on machines with similar utilization characteristics, thus reducing the number of used machines.

Automatic self-adaptation is addressed in the context of web applications mostly for failure recovery. In [16], a self-healing portal system is introduced, which leverages the ability of the service to respond at different resolutions. The portal adapts to an increasing system load by decreasing the amount of information contained in the service response.

A different direction in adapting web services is taken in [17] by using aspect-based templates to handle mismatches between service functionality and the published service interface. [18] introduces a solution for self-managing web services using a management proxy and intercepting incoming service requests at the middleware layer.

Service selection in our probabilistic approach is related to lottery scheduling [19]. In lottery scheduling, each operating system process receives a number of lottery tickets proportional to its importance. The scheduler randomly selects a ticket and schedules the process corresponding to the ticket. Hence, in a probabilistic manner, important processes receive more CPU than less important processes. It is trivial to change our service selection algorithm to use tickets instead of probabilities.

## VII. CONCLUSION

We have presented a novel infrastructure that transparently enhances existing BPEL processes with adaptive and self-tuning behavior. We provide a solution for monitoring and for dynamically binding web services per process instance, allowing processes to adapt to changes in the environment. Our monitoring mechanism supports different monitoring strategies allowing to optimize the trade-off between the amount of data collected and the monitoring overhead. The service selection algorithm presented in this paper assures that the system maintains high throughput by distributing the requests to available services based on the monitored average service response time. Performance evaluations conducted in various scenarios have shown that our infrastructure achieves up to 30% throughput increase when service performance fluctuates, compared to a hard-coded round-robin service selection mechanism.

Regarding ongoing research, we are working on extending the monitoring capabilities to different QoS parameters. Furthermore, we are investigating distribution and replication techniques to improve scalability and fault tolerance of our infrastructure.

## ACKNOWLEDGMENT

The work presented in this paper has been supported by the Swiss National Science Foundation.

## REFERENCES

- [1] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: A Research Roadmap. *Int. J. Cooperative Inf. Syst.* **17**(2) (2008) 223–255
- [2] Mosincat, A., Binder, W.: Transparent Runtime Adaptability for BPEL Processes. In: ICSOC. (2008) 241–255
- [3] Mosincat, A., Binder, W.: Self-tuning BPEL Processes, Poster. In: ICAC. (2009)
- [4] Meyn, S.P., Tweedie, R.L.: *Markov Chains and Stochastic Stability*. London: Springer-Verlag (1993)
- [5] Baresi, L., Ghezzi, C., Guinea, S.: Towards Self-healing Composition of Services. In: Contributions to Ubiquitous Computing. Springer Berlin / Heidelberg (2007) 27–46
- [6] Ezenwoye, O., S. Masoud Sadjadi: RobustBPEL2: Transparent Autonomization in Business Processes through Dynamic Proxies. In: Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007), Sedona, Arizona (2007) 17–24
- [7] Ezenwoye, O., Sadjadi, S.M.: A Proxy-Based Approach to Enhancing the Autonomic Behavior in Composite Services. *JNW* **3**(5) (2008) 42–53
- [8] Erradi, A., Totic, V., Maheshwari, P.: MASC-.NET-Based Middleware for Adaptive Composite Web Services. In: ICWS. (2007) 727–734
- [9] Halima, R.B., Drira, K., Jmaiel, M.: A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services. In: ICWS. (2008) 104–111
- [10] Mei, L., Chan, W.K., Tse, T.H.: An Adaptive Service Selection Approach to Service Composition. In: ICWS. (2008) 70–77
- [11] Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In: WWW '08: Proceeding of the 17th international conference on World Wide Web, New York, NY, USA, ACM (2008) 815–824
- [12] Baresi, L., Guinea, S., Pasquale, L.: Integrated and Composable Supervision of BPEL Processes. In: ICSOC. (2008) 614–619
- [13] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.C.: Adaptive and Dynamic Service Composition in *flow*. In: CAiSE. (2000) 13–31
- [14] Baazizi, M.A., Sebahi, S., Hacid, M.S., Benbernou, S., Papazoglou, M.P.: Monitoring Web Services: A Database Approach. In: ServiceWave. (2008) 98–109
- [15] Zhang, C., Chang, R.N., Perng, C.S., So, E., Tang, C., Tao, T.: QoS-Aware Optimization of Composite-Service Fulfillment Policy. In: IEEE SCC. (2007) 11–19
- [16] Naccache, H., Gannod, G.C.: A Self-Healing Framework for Web Services. In: ICWS. (2007) 398–345
- [17] Kongdenha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An Aspect-Oriented Framework for Service Adaptation. In: ICSOC. (2006) 15–26
- [18] Liu, Y., Truong, S., Chen, S., Zhu, L.: Composing Adaptive Web Services on COTS Middleware. In: ICWS. (2008) 377–384
- [19] Waldspurger, C.A., Weihs, W.E.: Lottery Scheduling: Flexible Proportional-Share Resource Management. In: OSDI. (1994) 1–11