Management of Static and Dynamic Requirements in Service Composition

Claudia Canali, Michele Colajanni, Riccardo Lancellotti

Abstract  The development paradigm for complex application is evolving towards the exploitation of the so-called Service Oriented Architecture, that requires runtime composition of Web services offered by different service providers. The choice of which providers may better satisfy the end-user requirements in terms of quality of service remains an open issue in the context of Web services. The runtime service composition has to match requirements of different nature: requirements may refer to static quality characteristics of the service providers, that do not change over time or change slowly (for example related to service price or provider reputation), and to dynamic quality characteristics, that may change on a per-invocation basis (typically related to performance, such as the response time).

The main contribution of this paper is to propose a family of novel runtime algorithms that compose services on the basis of quality requirements involving both static and dynamic characteristics, as in a typical Web scenario. We implement the proposed algorithms in a prototype and compare them with the solutions commonly used in service composition, that consider all the service provider characteristics as static for the scope of the composition process. Our experiments show that a static management of quality requirements is viable only in the unrealistic case where workload remains stable over time, but it leads to very poor performance in variable environments. On the other hand, the combined management of static and dynamic quality requirements allows us to achieve better user-perceived performance over a wide range of scenarios, with the response time of the proposed algorithms that is reduced up to a 50% with respect to that of static algorithms.

Keywords: Web services, Dynamic Web, Runtime service composition, Performance evaluation

1 Introduction

In the last years a growing amount of Internet applications are provided trough Web services that are composed together [8]. Protocols and technological solutions to this aim are already available: for example, the Service Oriented Architecture explicitly supports the composition of Web services [24]. The growing popularity of service-oriented computing and the advent of new paradigms, such as cloud computing, are likely to further increase the popularity of service composition techniques. However, as different versions of the same service can be provided by multiple providers but with different quality characteristics, the task of selecting the best solution for service composition in a distributed infrastructure will be a key challenge for future systems.
When service composition is carried out in a distributed infrastructure, the execution of a composite service typically consists of multiple invocation of component services that satisfy the functional requirements of the user request. Besides functional requirements, service composition should also satisfy the user preferences in terms of non-functional requirements on the characteristics of the service provider, such as price, reputation, successful execution rate, availability, security, response time and throughput; these requirements are related to the so-called quality of service \cite{2,14}. The management of non-functional service quality requirements that should be matched by the service providers represents an open research topic. To support service composition, we consider two types of quality requirements: 1) requirements for static characteristics of services, that are constant or change very slowly over time (e.g., service provider security or service monetary cost) and that we will call static requirements for short; 2) requirements for dynamic characteristics (e.g., response time or throughput), that may change in short time even as a consequence of the service composition decisions (for example, the decision about service composition may determine overload on the servers of a provider) and that we will call dynamic requirements.

Existing solutions for Web service composition assume a static behavior for all the quality characteristics offered by the service providers, meaning that the quality characteristics are considered constant for the scope of the service composition \cite{10,30,31}. The assumption of static behavior considers that even quality characteristics related to performance, such as response time or throughput, are changing slowly with respect to the service composition process. However, the impact of workload on these characteristics is likely to be amplified or even triggered by the decisions on the service composition. Indeed, when a decision on the service composition determines an overload condition on some service provider platform, the quality requirement related to the response time may explode beyond any acceptable level. The opposite approach of focusing only on dynamic requirements is typical of literature on high-performance Internet applications \cite{20,29}. However, these solutions do not take into account other service quality characteristics and, therefore, cannot be directly applied to Web service composition.

The main contribution of this paper is to propose a new class of algorithms that support service composition by distributing requests for component services among a set of service providers with heterogeneous characteristics, taking into account both static and dynamic requirements for the composition process. The support for static and dynamic requirements is an innovative feature that is achieved through the integration of real-time decisions in models for managing the static service quality requirements. Specifically, our proposal evaluates how composition decisions may impact on the platforms of the service providers and on their capability to meet dynamic service quality requirements. To the best of our knowledge, no effort has been made up to now to manage together quality characteristics with different behaviors in composing Web services.

To demonstrate the viability of our proposal, we present a case study based on the prototype of a Web portal for mobile clients. Each user request for a composite service is mapped into a set of invocations to component services that are offered by providers in a distributed system. We compare the proposed static and dynamic algorithms against other solutions that consider just static requirements; we demonstrate that the static approach can be applied only in the case of a stable workload, where requirements and workload patterns are not subject to rapid changes over time. A sensitivity analysis with respect to multiple workloads and scenarios is carried out to identify the algorithm that achieves the most robust performance in all contexts.

The remaining part of this paper is organized as follows. Section 2 proposes a model for the service composition with static and dynamic requirements in the case of stable and variable workload scenarios. Section 3 describes the proposed class of algorithms, while Section 4 illustrates the service composition process in the considered case study. Section 5 compares the performance of the service composition algorithms for multiple scenarios. Section 6 evaluates the sensitivity of the algorithms with respect to a large set of workload parameters. Section 7 discusses the related work and Section 8 concludes the paper with some final remarks.
2 Model for Web service composition

In this section, we model the problem of composing Web services to satisfy static and dynamic service quality requirements. We first describe the distributed Web system support the service composition, then we introduce the formal notation for the problem modeling.

2.1 Distributed system for Web service composition

We consider a distributed system for Web services where a Web portal offers composite services to the users. The Web portal (represented as a box in the central part of Figure 1) is a user-facing aggregation point of component Web services. To provide users with closer access points to the system, the Web portal may be geographically replicated [5]. The component Web services are offered by remote service providers, each one operating a different platform [23] (the service providers are represented as boxes in the rightmost part of Figure 1 the provided services as circles). We assume that each service provider offers the same set of component services from a functional point of view, but that the quality characteristics are heterogeneous across different service providers. This means that a component service is offered by providers with different quality characteristics. For example, a subscription service may be offered by providers having different levels of data security, and an image adaptation service by providers characterized by different prices. The Web portal maintains a repository with the information about the characteristics of the service providers, that are communicated periodically (dashed line in Figure 1).

Let us now describe the steps to compose Web services in the described system. Upon receiving a client request for a composite service (thick arrow from the clients to the Web portal in Figure 1), the Web portal analyzes the request and translates it into multiple requests for component services, that are offered by remote service providers. It is worth to note that the component services may be carried out concurrently: the assumption of parallel execution for the component services does not reduce the scope of the considered problem, because composition techniques involving a mix of parallel and sequential execution may be implemented through multiple invocation of the proposed algorithms [31]. At this point, the Web portal has to carry out the service composition process, that assigns each request for a component service to a provider.

To assign the component services to the providers, the service composition considers static and dynamic quality requirements. Static requirements refer to characteristics of the service provider that can be considered constant for the scope of service composition, such as the provider security, reputation, availability. A static requirement is satisfied if it matches with the corresponding characteristic offered by the service provider. To the aim of service composition, we consider that static requirements may include mandatory and not mandatory requirements: mandatory requirements must be satisfied, not mandatory requirements should be satisfied whenever it is possible. As for dynamic requirements, they refer to characteristics that may change for every service invocation, such as performance-related metrics like service response time and throughput. We should consider that these characteristics may be affected by the decision of the service composition process: indeed, they would be significantly degraded by an overload condition on some provider platform, that may be determined in consequence of the decision about the service composition. For this reason, we consider that the dynamic requirements are satisfied by avoiding overload on the service provider platforms; dynamic requirements are considered mandatory to prevent unacceptable platforms overload.

The problem of satisfying static and dynamic service quality requirements is not guaranteed to have always a solution: if the amount of requests with a given static requirement exceeds the capacity of the platforms with the appropriate level of static characteristics, the constraints about the dynamic and the static requirements cannot be satisfied together. In this case, the service composition needs to relax some static not mandatory requirement, causing what we call a mismatch. The final goal of the service compo-
2.2 Model description

We model the problem of service composition by considering that the workload is subject to changes even in a short time range, that is a common scenario in real Internet-based services [26, 29]. In the case of a workload where the mix of requested services changes over time, we must consider sudden variations of the request flows, hence we cannot use average request inter-arrival times in our model. To overcome this problem, we model the service composition at the level of a single client request. We consider the client request for a composite service as a set \( R \), with \( r \in R \) being a request for a component service that may be offered by \( N \) service providers distributed over a geographical area.

It is worth to note that in case of variable workload, it is hardly feasible to maintain a global knowledge of the updated information on the dynamic characteristics of every service provider in a geographically distributed system [15]. For this reason, we consider a decentralized approach where a Web portal receiving a client request for a composite service \( R \) distributes the component service requests \( r \) among a restricted subset \( \mathcal{N} \subset N \) of neighbor service providers. The service providers belonging to the \( \mathcal{N} \) set are selected to minimize the communication latency. This choice represents a commonly adopted solution to reduce the network-related delays in geographically distributed systems [15], that is also motivated by the need of reducing the overhead due to the exchange of load status information among the entities of the system. The service composition is translated into a problem of distributing requests for component services, assigning each request \( r \in R \) to a service provider \( n_r \in \mathcal{N} \).

Let us now consider the static characteristics of the service providers. Each service provider \( n \in \mathcal{N} \) is characterized by a multidimensional service level for static characteristics, that we represent as a vector \( l_n \in \mathbb{H} \). The vector in general consists of \( d \) characteristics. Each service level \( l_{n,i} \) for the service provider \( n \in \mathcal{N} = [1, N] \) and the characteristic \( i \in [1, d] \) may range between 0 and \( \bar{l} \), that is \( 0 < l_{n,i} < \bar{l} \). The assumption of having the same top-level value \( \bar{l} \) for all the static characteristics does not limit the
generality of the problem formulation, because we consider to normalize the values that may be assumed by the elements of the service level vector.

From the point of view of static service requirements, we consider that a request for a component service \( r \) is characterized by a static requirement vector, that we represent as \( \vec{q}^r \in H^+ \supset H \), where \( H^+ = H \cup \{ \vec{q}^+ \} \). The value \( \vec{q}^+ \) has been introduced to distinguish the class of requests having a static mandatory requirement for the highest service level. In other words, a service request \( r \) with a static requirement \( \vec{q}^r \in H \) indicates the service level of the service provider that should be used to process the request, while the case \( \vec{q}^r = \vec{q}^+ \) represents requests that must be assigned to the service providers offering the highest service level (that is, \( t_{n,i} = \bar{l}, \forall i \in [1, d] \)).

The dynamic characteristics of the service providers concern performance-related metrics, such as user response time and throughput, that depend on the load of the platforms executing the services. The composition process must evaluate how its decisions may impact on the service providers platforms to avoid overload conditions that would degrade the dynamic characteristics of the providers. To this aim, we should consider that the component services may be heterogeneous from the computational point of view, with service times that may span over multiple orders of magnitude. For this reason, we consider each request for a component service \( r \) as belonging to a class of computational cost \( c_r \in C = [1, C] \) that we think convenient to classify according to the service time. The extraction of computational costs from service is a well known problem that can be addressed both via benchmarking of the Web services or through analysis of logs [22].

To satisfy the service dynamic requirements, the service composition process must avoid overload conditions on the service provider platforms, meaning that the time-dependent utilization of each service provider platform must not exceed 1. The utilization of a provider platform \( n \) before service composition \( \rho_n(t) \) is defined as the fraction of time the service provider platform \( n \) is busy during the time interval \([t−T, t]\), where \( t \) is the instant of distribution assignment, and \( T \) is the observation period for the evaluation of the platform utilization.

We assume that the decision about the service composition impacts on the platform utilization with a contribution \( \rho^R_n \), due to the services \( r \in R \) that are assigned to the platform of the service provider \( n_r \) at time \( t \). If we consider that each service \( r \) is characterized by a computational class \( c_r \), we have:

\[
\rho^R_n = \sum_{r \in R : n_r = n} \frac{1}{\mu_{c_r} T}
\]  

(1)

where \( \frac{1}{\mu_{c_r}} \) is the service time for the request \( r \) belonging to the computational class \( c_r \).

It is important to note that \( \rho_n(t) \) does not take into account the contribution to platform utilization derived from the service composition carried out at time \( t \). Hence, we should add the contribution \( \rho^R_n \) in the definition of the not overloading constraint:

\[
\rho_n(t) + \rho^R_n < 1, \quad \forall n \in N
\]  

(2)

The overall system is assumed to be not under-provisioned, hence the global computational demand of incoming requests does not exceed the whole available computational power. However, the problem of satisfying together dynamic and static requirements is not guaranteed to have always a solution. We recall that this can happen if the amount of requests with given static requirements exceeds the capacity of the platform with the appropriate level of static characteristics. We now define how to measure the mismatch occurring when the constraints about the dynamic and the static requirements cannot be satisfied together. We should consider that dynamic requirements are considered mandatory because relaxing these bounds would result in in an unacceptable platform overload. Hence, the service composition can only to relax the constraints on the static not mandatory requirements, assigning some request with a given static not mandatory requirement to a service provider with a lower static service level. To measure the mismatch, we introduce a penalty for the case where service composition does not satisfy all static not mandatory requirements. We model the mismatch through the function \( M(\vec{q}, \bar{l}) \) that represents the
penalty associated to the assignment of a request to a service provider \( n \) having an unsatisfactory service level, that is \( \overrightarrow{l_n} \) does not match \( \overrightarrow{q} \). The mismatch function returns 0 if all the static requirements are satisfied, that means \( l_n,i \geq q_i, \forall i \in [1,d] \). Otherwise, the penalty function returns a positive value that is a function of the vectors \( \overrightarrow{q} \) and \( \overrightarrow{l_n} \). For example, for \( \overrightarrow{q} \in H \) the mismatch function can be defined as \( M(\overrightarrow{q}, \overrightarrow{l_n}) = \sum_{i=1}^{d} \alpha_i \times \max(0, q_i - l_n,i), \) with \( \alpha_i > 0, \forall i \in [1,d] \). The coefficients \( \alpha_i \) for \( i \in [1,d] \) represent the weights of the static characteristics that may be specified by the Web portal providing the composite service to the clients.

We can now define the service composition problem as an optimization problem that aims to minimize the mismatch penalty \( M_R \) defined through a the function \( M(\overrightarrow{q}, \overrightarrow{l_n}) \), that is,

\[
\text{minimize } M_R = \sum_{r \in R} M(\overrightarrow{q_r}, \overrightarrow{l_{nr}}) \tag{3}
\]

subject to:

\[
\rho_n^R < 1 - \rho_n(t), \quad \forall n \in \mathbb{N}
\]

### 3 Static and dynamic algorithms for service composition

In this section, we present a new class of static and dynamic algorithms for service composition that solve the optimization problem defined in Section 2 under the assumption of a variable workload; in particular, we propose three algorithms: \textit{Capacity-based}, \textit{Cost-blind Balancing-based} and \textit{Cost-aware Balancing-based}. These algorithms share the same model for achieving static requirements goals, but follow different schemes to satisfy the dynamic requirements. To this aim, all of them exploit information about the utilization of the service provider platforms, but the Capacity-based algorithm just aims to avoid overload, while the two Balancing-based algorithms aim to balance the utilization of the service provider platforms involved in the service composition. The Capacity-based algorithm relies on accurate information about the computational costs of the component services to achieve good performance, while the Balancing-based algorithms may be blind or aware with respect to the information about the computational costs.

#### 3.1 Capacity-based algorithm

The static and dynamic Capacity-based algorithm (S&D-Cap) distributes requests according to the static quality requirements and by considering the computational costs of the component services, with the aim of avoiding overload conditions. The algorithm solves a multiple knapsack optimization problem, where each knapsack is a service provider platform in the \( \mathbb{N} \) set. The capacity of each knapsack is the residual capacity of the platform, which is a function of its CPU utilization: \( 1 - \rho_n(t) \). Each object is a component service with the same unitary value and a volume equal to the contribution to the platform utilization \( \frac{1}{\rho_n(t)} \). We want to maximize the number of requests assigned according to their static quality requirements, which means that we aim to minimize the mismatch over all the component services:

\[
\text{minimize } M_R = \sum_{r \in R} M(\overrightarrow{q_r}, \overrightarrow{l_{nr}}), \quad \text{subject to the bounds of not exceeding the residual capacity of each service provider platform:}
\]

\[
\sum_{r: s_r = n} \frac{1}{\rho_n(t)} \leq 1 - \rho_n(t), \forall n \in \mathbb{N}.
\]

A greedy algorithm solves the multiple knapsack problem: the knapsacks are filled starting with the smallest objects (that is, requests with the lowest service time) until their residual capacity is exhausted. The algorithm uses a set of additional variables \( \alpha_n \) that represent the residual capacity of the platforms. First, requests for services with static quality requirements equal to \( \overrightarrow{q}^+ \) are assigned to the service provider with the highest quality service level (line 3 of Algorithm 1) discarding any concern due
to dynamic requirements. The remaining requests for component services are grouped according to their static requirements, starting with the highest values. To compare the values of different static requirements, we introduce a new metric based on the Simple Additive Weighting technique (SAW), also known as the weighted sum method, that has been used by [9]. Basically, for a static service level requirement vector \( \overrightarrow{q} \), we define \( \| \overrightarrow{q} \| = \sum_{i=1}^{d} \alpha_i q_i \), where \( \alpha_i \geq 0 \) represents the weight coefficient of the \( i^{th} \) quality characteristic. The algorithm, hence, starts to consider the requests with the highest values of \( \| \overrightarrow{q} \| \).

Requests with the same static quality requirements are scanned in increasing order of their computational costs (due to the sorting in line 4 of the algorithm) to minimize the amount of mismatch [17]. For each request, the algorithm scans the available service providers trying to allocate the request without exceeding the computational capacity of their platforms. To this purpose, the algorithm divides the service providers in two groups: the set \( N_{\overrightarrow{q}} \) contains service providers that satisfy the static requirements of the request and are sorted by increasing \( \| I_{r_{n}} \| \), while \( N_M \) includes the service providers that cause a mismatch and are sorted by increasing values of \( M(\overrightarrow{q}_r, I_{r_{n}}) \). The algorithm first examines the service providers in \( N_{\overrightarrow{q}} \), starting from the providers with the minimum required static service level \( \| I_{r_{n}} \| \) up to those with higher level. Then, if everything else fails, the static requirement is relaxed and service providers in \( N_M \) are considered for the request assignment, starting from those that cause a lower mismatch. Finally, the algorithm distributes the requests for the remaining component services following a round robin scheme among the service provider platforms with the largest residual capacity, discarding static quality requirements. The computational complexity of the algorithm is \( O(n \log n) \) with \( n = |R| \), due to sorting in line 4.

Algorithm 1 Capacity-based

```markdown
Require: \( R, \overrightarrow{q}_r, \frac{1}{\mu_{n_r}} \forall r \in R, N, \rho_n(t) \forall n \in N, T \\
Ensure: \{n_r\} (S&D-Cap)
1: \text{for all } n \in N \text{ do}
2: \quad a_n \leftarrow 1 - \rho_n(t)
3: \quad \text{assign services with } \overrightarrow{q}_r = \overrightarrow{q}^+ \text{ to service providers with the highest static service level, then update } a_n
4: \quad \text{sort } R \text{ by increasing } \frac{1}{\mu_{n_r}}
5: \text{for all } \overrightarrow{q} \in H, \text{ sorted by decreasing } \| \overrightarrow{q} \| \text{ do}
6: \quad \text{for all } r \in R: \overrightarrow{q}_r = \overrightarrow{q} \text{ do}
7: \quad \quad \text{select } n \in N: \text{ if } M(\overrightarrow{q}_r, I_{r_{n}}) = 0
8: \quad \quad \text{select } n \in N: \text{ if } M(\overrightarrow{q}_r, I_{r_{n}}) > 0
9: \quad \quad \text{sort } N_{\overrightarrow{q}} \text{ by increasing } \| I_{r_{n}} \|
10: \quad \quad \text{sort } N_M \text{ by increasing } M(\overrightarrow{q}_r, I_{r_{n}})
11: \quad \text{for all } n \in N_{\overrightarrow{q}} \cup N_M \text{ do}
12: \quad \quad \text{if } a_n \geq \frac{1}{\mu_{n_r}} \text{ and } n_r = "\text{not assigned}" \text{ then}
13: \quad \quad \quad M_R \leftarrow M_R + M(\overrightarrow{q}_r, I_{r_{n}})
14: \quad \quad \quad n_r \leftarrow n
15: \quad \quad \quad a_n \leftarrow a_n - \frac{1}{\mu_{n_r}}
16: \quad \text{assign remaining services to least loaded service provider platforms}
17: \text{return } \{n_r\}
```

3.2 Cost-blind Balancing-based algorithm

The Cost-blind Balancing-based algorithm (S&D-CB-Bal) aims to satisfy static and dynamic requirements by following a load balancing strategy without considering the computational costs of the com-
ponent services. This algorithm computes for each composite service the number of component service requests that have to be assigned to each provider \( n(k_n) \) in Algorithm 2 in order to satisfy the condition \( \rho_n(t) + \rho_n^R = \rho(t), \forall n \in \bar{N} \).

The algorithm is cost-blind because it considers that all component services have the same computational cost. Hence, the \(|R|\) service requests are partitioned proportionally to the residual capacity \( 1 - \rho_n(t) \) of the service provider platforms (line 2 of Algorithm 2). The function used to determine \( k_n \) is chosen for its simplicity, because preliminary experiments suggest that more complex functions do not provide significant performance benefits. The service requests are distributed among the \( \bar{N} \) service providers starting from the services with the highest static quality requirements. As for the other algorithms, the assignment of services with \( \bar{q}^+ \) static quality requirements is the first operation and is carried out bypassing performance concerns (line 3 of the algorithm). For each other service request, the algorithm first considers the service providers in \( \bar{N}_{MT} \) that satisfy the static quality requirements, starting from the service providers with lower static service level \( ||\bar{r}_n|| \). If the services cannot be assigned to these service providers (because \( k_n = 0 \)), the algorithm considers service providers with higher static service level in \( \bar{N}_{MT} \), and only as the last alternative the algorithm relaxes the static quality requirements, and considers the service providers in \( \bar{N}_M \) for the request assignment, thus introducing a mismatch.

The algorithm has a computational complexity \( O(n) \) with \( n = |R| \), that is determined by the array scans to assign the requests for component services to the service providers.

### Algorithm 2 Cost-blind Balancing-based

Require: \( R, \bar{q}^r \forall r \in \bar{N}, \rho_n(t)\forall n \in \bar{N} \)

Ensure: \( \{n_r\} \) (S&D-CB-Bal)

1: for all \( n \in \bar{N} \) do
2: \( k_n := \lfloor |R| \sum_{r \in \bar{q}^r} \frac{1 - \rho_n^R(t)}{\rho_n(t)} \rfloor \)
3: assign services with \( \bar{q}^r = \bar{q}^+ \) to service providers with the highest static service level, then update \( k_n \)
4: for all \( \bar{q} \in H \), sorted by decreasing \( ||\bar{q}|| \) do
5: for all \( r \in R : \bar{q}^r = \bar{q} \) do
6: \( \bar{N}^{MT} = \{ n \in \bar{N} : M(\bar{q}^r, \bar{l}_n^\rightarrow) = 0 \} \)
7: \( \bar{N}_M = \{ n \in \bar{N} : M(\bar{q}^r, \bar{l}_n^\rightarrow) > 0 \} \)
8: sort \( \bar{N}^{MT} \) by increasing \( ||\bar{l}_n|| \)
9: sort \( \bar{N}_M \) by increasing \( M(\bar{q}^r, \bar{l}_n^\rightarrow) \)
10: for all \( n \in \bar{N}^{MT} \cup \bar{N}_M \) do
11: if \( k_n > 0 \) and \( n_r = "\text{not assigned}" \) then
12: \( M_R := M_R + M(\bar{q}^r, \bar{l}_n^\rightarrow) \)
13: \( n_r := n \)
14: \( k_n := k_n - 1 \)
15: return \( \{n_r\} \)

#### 3.3 Cost-aware Balancing-based algorithm

The Cost-aware Balancing-based (S&D-CA-Bal) algorithm is an improved version of the balancing-based strategy that considers also the computational class of the component service requests. Unlike the Capacity-based solution, this algorithm does not assume to know the exact computational cost of each component service request \( \frac{1}{\mu_r} \), but just the order of magnitude of the service time that is related to each computational cost class. This information is used to balance the load of the \( \bar{N} \) service provider platforms. The Cost-aware Balancing-based algorithm assigns to each component service \( r \) a computational cost \( \sigma_r \)
that depends on the order of magnitude of the service time $\frac{1}{\mu_c}$. For example, we associate the cost $1$ to the service times in the order of milliseconds, the cost $10$ for service times in the order of tens of milliseconds, and a cost $100$ to the services requiring hundreds of milliseconds. This rough assignment allows a straightforward implementation of a cost-aware distribution, because it avoids all the issues that are related to an accurate estimation of the computational costs as in the Capacity-based algorithm [22].

The Cost-aware Balancing-based algorithm starts by computing the amount of computational demand, that is $\sum_{r \in R} o_r$, and assigns an amount of computational costs $o_n$ to each service provider $n \in N$.

To determine the request distribution, the algorithm scans the service requests starting with the services having the highest static quality requirements and assigns them to the service providers without exceeding the pre-determined value of computational demands $o_n$ (with a separate handling of components with $\rightarrow q$ static requirements). Resources with the same static quality requirement are assigned starting with the lowest values of $o_r$ to minimize the mismatch, using the same greedy approach described for the Capacity-based algorithm.

The algorithm has a computational complexity $O(n \log(n))$ with $n = |R|$, that is determined by the sorting operations.

Algorithm 3 Cost-aware Balancing-based

Require: $R, \rightarrow q_r, o_r \forall r \in R, N, \rho_n(t) \forall n \in N$

Ensure: $\{n_r\}$ (S&D-CA-Bal)

1: for all $n \in N$ do
2: $o_n \leftarrow (\sum_{r \in R} o_r) \cdot \frac{1}{\sum_{k \in N} (1 - \rho_k(t))}$
3: assign services with $q_r = \frac{1}{\mu_c}$ to service providers with the highest static service level, then update $o_n$
4: sort $R$ by increasing $o_r$
5: for all $q_r \in H$, sorted by decreasing $\|q_r\|$ do
6: for all $r \in R: \rightarrow q_r = q_r$ do
7: $N_{M_r} = \{n \in N : M(q_r, l_{nr}) = 0\}$
8: $N_{M_r} = \{n \in N : M(q_r, l_{nr}) > 0\}$
9: sort $N_{M_r}$ by increasing $\|l_{nr}\|$  
10: sort $N_{M_r}$ by increasing $M(q_r, l_{nr})$
11: for all $n \in N_{M_r} \cup N_{M_r}$ do
12: if $o_n \geq o_r$ and $n_r =$"not assigned" then
13: $M_R \leftarrow M_R + M(q_r, l_{nr})$
14: $n_r \leftarrow n$
15: $o_n \leftarrow o_n - o_r$
16: return $\{n_r\}$

4 Case study

We now present the case study used for the evaluation of the static and dynamic algorithms for service composition. The case study represents a typical scenario where Web services are used to support the Mobile Web; in particular, we consider a Web system offering services for the generation of personalized content for mobile clients. This choice is motivated by the importance of the Mobile Web in future network scenarios, testified by several authors outlining the issues of supporting mobile users [13][32] and providing personalization services [3][11]. In the rest of this section, we first present the considered Web system and the service composition process, then we describe the characteristics of personalization services and workload models used in the experimental evaluation.
4.1 Service composition process

The Web system for personalization services is shown in Figure 4.1. Architecturally, our Web system consists of a Web portal receiving the client requests and aggregating Web services from multiple service providers dislocated in remote geographic locations. Each service provider publishes the offered services in the Service registry, where the name of the service and the static characteristics of the provider are stored in the Service list and Static service characteristics repository, respectively. The service registry also contains a repository for Dynamic service characteristics, with the information sent by the providers about the time-varying status of their platforms. Differently from the static characteristics, the dynamic information needs to be updated frequently due to its rapidly changing nature, for example through periodic notifications to the Service monitoring, or by piggybacking the information in each communication between the service providers and the Web portal. All the information contained in the Service registry are pushed in the Service registry cache of the Web portal to be rapidly accessed during the service composition process.

Let us now describe the sequence of steps necessary to satisfying a client request. The client request for a composite service \( R \) is received by the Web server, that is the front-end of component of the Web portal. The Web server analyzes the client request and generates the list of the component services, that is passed to the composition engine along with the static and dynamic requirements for each component service request. The composition engine carries out the service composition algorithm to identify a provider for each component service on the basis of its requirements and of the provider static and dynamic characteristics, that are stored in the service registry cache. The service composition specifies the service provider \( n_r \) selected for the assignment of each request for component service \( r \in R \). In our case study, we consider that the composition engine assigns requests for component services to \( N \) service providers, with \( N = 8 \). In our experiments we consider as the main static characteristic of the service providers the level of reputation of the provider. We assume that the level of reputation is determined on the basis of the user feedbacks and of some specific feature of the provider platform. For example, service providers that received positive feedbacks and can guarantee high data protection thanks to a complete control over their platform have the highest reputation level, while other service providers, with negative feedbacks and a looser control on their platforms due to servers replicated, housed or hosted in other data centers, are characterized by lower reputation levels. In our system we consider 4 different levels of reputation (ranging from 0 to 3) equally distributed among the 8 providers, that means we have 2 providers for each level of reputation. Furthermore, the computational power of the system is equally distributed among all the service providers. Once the decision on the service composition has been taken, the service broker invokes the component services of the selected providers through remote portlets, implemented following the Web Services for Remote Portlets (WSRP) specification.

4.2 Services and workload

Let us now describe the personalization services offered by our prototype to users connected through mobile devices.

The Web portal offers the following services for the Mobile Web:

- **Subscription and profile editing.** Service that allows the users to subscribe to the Web portal by paying a subscription fee; the service also offers the possibility to visualize and edit every field of the user profile.
- **Context-sensitive banner insertion.** Service that selects banners from a database according to user interests and location, and inserts them into the generated content.
- **Aggregation of RSS feeds.** Service that dynamically aggregates and converts RSS-XML code to HTML depending on the user preferences.
- **Adaptation to user device and context.** Service that tailors HTML code and embedded images according to the capabilities of the mobile device or to the user context. For example, the case of a driving user may require services, such as text-to-speech conversion [19], to access Web contents without the need to read the information.

From the point of view of the dynamic requirements, we consider that the four offered services are characterized by heterogeneous computational costs, which may involve service times of different orders of magnitude. Banner insertions typically require few milliseconds, subscriptions and operations of profile editing may be in the order of tens of milliseconds as well as the feed aggregation tasks, while the adaptations of embedded images may take from hundreds of milliseconds up to one second [6]. In our experiments, we consider that requests for component services are distributed among the four offered services in the following way: 10% for subscription and profile editing, 30% for banner insertion, 30% for RSS feeds aggregation and 30% for content adaptation (we will refer to this service mix as to WL1). However, in the sensitivity analysis, we will also introduce another service mix, namely WL2, that is more skewed from the point of view of the computational costs: in WL2 the majority of the requests is evenly distributed among the two services with lowest and highest computational requirements (40% for Banner insertion and 40% for Content adaptation), and a low percentage of requests for the other two services (10% for subscription and profile editing and 10% for RSS feeds aggregation).

As the main static requirements of the services we consider the level of reputation $q_r \in H^+$ associated to a component service $r$. To this aim, we consider five classes of reputation requirements: $H^+ = \{(0), \{1\}, \{2\}, \{3\}, \mathbb{N}^+\}$ In the set of services offered by our prototype, the Subscription and
profile editing is the only service with a mandatory requirement requirement for the highest level of reputation, that is $\overrightarrow{q} = \overrightarrow{l}^+$, because it requires access to sensible information (e.g., credit card data or health information). For the other three services, the users may choose between the four not mandatory levels of reputation, considering that the reputation level of the service provider is directly proportional to the economical cost of the offered service.

For the algorithm experimental evaluation we consider two workload scenarios, namely stable and variable workload, that differ for the reputation requirements. The stable workload is characterized by a distribution of reputation requirements that does not change during the experiment, as shown in Figure 3(a). For the whole duration of the experiment we have 5% of requests with mandatory reputation requirements $\overrightarrow{q} = \overrightarrow{l}^+$, and 20%, 25%, 25% and 25% of requests with reputation requirements $\overrightarrow{q} = \{3\}$, $\{2\}$, $\{1\}$, and $\{0\}$ respectively. The variable workload is characterized by an average amount of reputation requirements that corresponds to that of the stable workload, but the reputation requirements in the workload changes over time throughout the experiment. This workload aims to capture the Web variability, where the workload is subject to changes even in short periods (e.g., in the order of few minutes) [26,29]. Specifically, in Figure 3(b) we show that the components with $\overrightarrow{q} = \{3\}$ grow throughout the experiment from 5% to 35%. In a similar way the amount of components with $\overrightarrow{q} = \{2\}$ grows from 10% to 40%, while the amount of components with $\overrightarrow{q} = \{1\}$ and $\{0\}$ decrease following similar patterns.

The mismatch penalty $R_M$ may assume different values depending on the penalty function $M(q_r, s_{nr})$. In our testbed we use the simple values in Table 1. As a consequence, all the algorithms force component services with reputation requirements $\overrightarrow{q} = \overrightarrow{l}^+$ to be assigned to the providers owning the most trusted platforms that have $l_n = \{3\}$; minimizing the mismatch penalty is equivalent to limiting assignments of requests with a given reputation requirements to a service provider platform with a lower reputation level. Other information necessary to the service composition algorithms is related to dynamic requirements, such as the utilization $\rho(t)$ of the platforms. Some algorithms require also information on the computational cost class $c_r$ of each component service $r$.

<table>
<thead>
<tr>
<th>Provider trust level</th>
<th>$\overrightarrow{q} = {0}$</th>
<th>$\overrightarrow{q} = {1}$</th>
<th>$\overrightarrow{q} = {2}$</th>
<th>$\overrightarrow{q} = {3}$</th>
<th>$\overrightarrow{q} = \overrightarrow{l}^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_n = {0}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$l_n = {1}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$l_n = {2}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$l_n = {3}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
To exercise our prototype we use synthetically generated traces, since none of the existing benchmarking models (e.g., Spec-Web, TPC-W) includes the considered features for the evaluation of the Web services composition with heterogeneous requirements. The requests in the traces are divided into sessions, each related to a different user. Sessions are initiated at the rate of 8 sessions per second and sent to the Web portal. Each session is modeled by a bi-modal distribution, where 80% of the sessions are composed by only one composite service request, while the other sessions consist of an average (of an exponential distribution) of 10 composite service requests; each composite service involves 10 component services on average. It is worth to note that the traces used in our experiments generate a workload intensity that is lower than the overall system capacity, because the goal of our experiments is to evaluate the effectiveness of the service composition algorithms in a system which is correctly dimensioned with respect to the offered load.

5 Algorithm evaluation

In this section we evaluate the proposed static and dynamic algorithms for service composition in terms of satisfaction of static and dynamic requirements. To evaluate the satisfaction of static requirements, we consider the mismatch penalty as the main metric, while for the evaluation concerning the dynamic requirements we consider the user perceived performance, measured as the response time experienced by the client requests. As our mismatch penalty function returns only two possible values (that is 0 if there is no mismatch and 1 if there is a mismatch), our mismatch penalty represents the percentage of requests where a mismatch occurs.

For the algorithm evaluation, we consider the stable and variable workload scenarios described in Section 4.2. Moreover, we use other two algorithms as term of comparison, namely Static-oriented and Dynamic-oriented: the Static-oriented algorithm does not allow any mismatch and distributes component services considering only static requirements; on the contrary, the Dynamic-oriented algorithm does not consider the static requirements and distributes services just on the basis of dynamic requirements.

5.1 Stable workload

We initially evaluate the service composition algorithms in the unrealistic case of stable workload. Table 2 presents the 90-percentile of response time and the mismatch penalty achieved throughout the experiment. We observe that almost all algorithms achieve similar performance, as testified by the 90-percentile of the response time very close to each other. The Static-oriented algorithm obtains a response time slightly higher if compared to the alternatives, but the performance loss with respect to the Dynamic-oriented algorithm is limited to 15%. This result suggests that, in the case of a stable workload, a correct provisioning of the platform resources of the underlying Web system guarantees that every algorithm can successfully satisfy the dynamic requirements.

If we consider the results about static requirements, shown in the third column of Table 2, we observe a clear difference among the algorithms. Obviously the Static-oriented algorithm obtains a 0% mismatch penalty for every workload. On the other hand, the Dynamic-oriented algorithm, which applies a static-blind service composition, causes a high mismatch that is close to 30%. The class of static and dynamic algorithms obtains mismatch penalty percentages that, although not reaching the optimal value of the Static-oriented algorithm, are almost 7 times lower than that of the Dynamic-oriented algorithm. We can conclude that, in the (unrealistic) case of a stable workload, a Static-oriented approach is a simple and viable solution, because it completely satisfies the request static requirements and achieves a good level of performance. Even S&D algorithms are a viable solution because each algorithm of this class achieves performance close to the best response time at the cost of a little mismatch penalty, that is never higher than 4.5%.
Table 2 Static and Dynamic evaluation for the stable scenario

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>90-percentile of response time [s]</th>
<th>Mismatch penalty [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&amp;D-Cap</td>
<td>4.94</td>
<td>3.8%</td>
</tr>
<tr>
<td>S&amp;D-CB-Bal</td>
<td>4.85</td>
<td>4.5%</td>
</tr>
<tr>
<td>S&amp;D-CA-Bal</td>
<td>4.81</td>
<td>4.1%</td>
</tr>
<tr>
<td>Dynamic-oriented</td>
<td>4.32</td>
<td>27.6%</td>
</tr>
<tr>
<td>Static-oriented</td>
<td>5.14</td>
<td>0%</td>
</tr>
</tbody>
</table>

5.2 Variable workload

The variable workload is characterized by requests with a distribution of static requirements that changes throughout the experiment. This is a more realistic and more interesting case with respect to the stable scenario, because it captures the inherent variability of Web workload patterns and represents a more challenging scenario for all the service composition algorithms. The results of static and dynamic evaluation are reported in Table 3.

Unlike the case of stable workload, the performance of the service composition schemes presents a significant difference, with the response time of the Static-oriented algorithm that is more than doubled with respect to that of the Dynamic-oriented algorithm. The results for the static and dynamic algorithms basically confirm those of the stable workload: the S&D-CA-Bal achieves the lowest response times with a 90-percentile very close to that of the Dynamic-oriented algorithm, while the S&D-Cap algorithm obtains the worst performance within the S&D class. However, it is important to note that the response times achieved by the static and dynamic algorithms are very similar (differences within 0.3 seconds on the 90-percentile). We can conclude that high performance can be achieved through both the approaches followed by the proposed static and dynamic algorithms, that are to avoid overload on each service provider platform (as in Capacity-based algorithm), and to balance the load among the platforms (as in Balancing-based algorithms).

Let us now consider the static evaluation results shown in the third column of the Table 3. The Static-oriented algorithm completely satisfies the static requirements at the expense of extremely high response times; the opposite is true for the Dynamic-oriented algorithm that pays its good performance with a high percentage (45%) of mismatch penalties. The static and dynamic algorithms achieve intermediate results, with an amount of mismatch penalty less than one third of that of the Dynamic-oriented algorithm.

The experimental results allow us to draw multiple conclusions. In a Web context where the workload is subject to dynamic changes, service composition algorithms must take into account both static and dynamic requirements. When the workload is variable, indeed, both the Dynamic-oriented and the Static-oriented classes are not viable solutions due to the excessive mismatch penalty and poor response time, respectively. On the other hand, the new class of static and dynamic algorithms may successfully combine both the types of requirements.

Table 3 Static and Dynamic evaluation for the variable scenario

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>90-percentile of response time [s]</th>
<th>Mismatch penalty [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&amp;D-Cap</td>
<td>4.98</td>
<td>13.9%</td>
</tr>
<tr>
<td>S&amp;D-CB-Bal</td>
<td>4.77</td>
<td>15.1%</td>
</tr>
<tr>
<td>S&amp;D-CA-Bal</td>
<td>4.68</td>
<td>14.6%</td>
</tr>
<tr>
<td>Dynamic-oriented</td>
<td>4.47</td>
<td>45.2%</td>
</tr>
<tr>
<td>Static-oriented</td>
<td>9.12</td>
<td>0%</td>
</tr>
</tbody>
</table>
6 Sensitivity analysis

In this section we present a sensitivity analysis to the parameters that may affect the stability of evaluation results of the proposed static and dynamic algorithms in the variable workload scenario. Specifically, we investigate the results sensitivity to the number of component services for client request, the service time estimation error, and to the workload computational skewness. We may anticipate that, while response time may be highly variable with respect to some parameters, the mismatch penalty is surprisingly very stable (variation in the order of few percent points) for every considered scenario. For this reason, we focus on the response time as the main comparison metric. The sensitive analysis aims to identify which algorithm belonging to the static and dynamic class is able to provide the most stable performance for a wide range of workload and system scenarios.

6.1 Overhead of the service composition algorithms

The user-perceived response time is influenced by two contributions that may hinder the scalability of the system: the network-related delays and the time spent to take decisions on service composition. A preliminary study by the authors [7] suggests that network-related delays do not affect the results of the algorithms, also thanks to the choice of a limited subset of service providers characterized by low latency to offer the requested services. We can thus conclude that the main limit for the system scalability is the overhead introduced by the service composition process. In this experiment, we evaluate if the time required for the service composition becomes unacceptable as the number of component services grows. We compare the proposed algorithms against a baseline that is represented by a stateless round-robin algorithm. Figure 4 shows the 90-percentile of the service composition time as a function of the number of required component services. As expected, the round-robin service composition time remains constant and below all the others. The service composition time of the static and dynamic algorithms is sensitive to the number of component services, but for all these algorithms it remains in the order of few milliseconds even in the unlikely case of 100 composed services. Hence, we can conclude that the service composition overhead is not an issue for any proposed algorithm.

![Fig. 4 Overhead of the service composition algorithms](image-url)
6.2 Sensitivity to service time estimation

The estimated service time for a component service represents an important input information for two of the proposed static and dynamic algorithms, the S&D-Cap and the S&D-CA-Bal. Thanks to the experimental setting that guarantees a complete control over workload and system, so far we have evaluated the S&D-Cap algorithm in an ideal case where the service time for each component service and the platform load are assumed to be known a-priori and estimated every second, respectively. However, we should consider that a perfect knowledge of the service time is unfeasible because it can only be approximated through complex statistics \[22\]. We now want to evaluate the impact that the removal of ideal conditions has on these algorithms.

To evaluate the impact of an inaccurate estimation of service times, we compare the ideal case, where the algorithms have an exact knowledge of the service time of the component services, with scenarios where we introduce an error on the estimated service time. Figure 6.2 shows the 90-percentile of the response time for all static and dynamic algorithms as a function of the error in estimating the service time. As expected, the S&D-CB-Bal algorithm is insensitive to an inaccurate estimation of the service time, while the performance of the S&D-CA-Bal is slightly sensitive to the service time inaccuracy. On the other hand, the 90-percentile of the response time for the P&Q-Cap algorithm significantly increases as the value of the error grows. This increment is much higher than that initially expected: the performance degradation is over 100% when the error is equal to 10%. From this analysis we conclude that the S&D-Cap algorithm is actually a theoretical algorithm that can achieve good performance just in a controlled and unrealistic environment, but it is hardly applicable to a real system.

![Fig. 5 Sensitivity to service time error](image)

6.3 Sensitivity to workload computational cost skewness

The two Balancing-based algorithms appear so far the most stable solutions for the service composition. However, we recall that the Cost-blind Balancing-based algorithm assumes that every component service has the same computational cost. This is a possible drawback when the skewness of the workload from a computational point of view is high. We first evaluate the impact of the workload skewness on the performance of the static and dynamic algorithms by considering a workload model characterized by a variance
in the distribution of the service times higher by nearly 50% with respect to the workload used in the previous algorithm comparison. We compare the performance of the Balancing-based algorithms with that of the Dynamic-oriented algorithm and, for the sake of comparison, we consider also the (ideal) S&D-Cap algorithm that takes into account the exact computational costs for each service. Figure 6 reports the cumulative distribution of the response times achieved by the service composition algorithms. It is interesting to note that, in this more skewed workload, the S&D-Cap algorithm outperforms the S&D-CB-Bal algorithm, although the former is an ideal version. This result confirms that the knowledge of the computational cost of every component service allows the S&D-Cap algorithm to provide an efficient service composition, while the S&D-CB-Bal algorithm is unable to balance the load on the service provider platforms. On the other hand, the S&D-CA-Bal algorithm achieves a significant improvement with respect to the S&D-CB-Bal, with a performance gain of more than 20% on the 90-percentile of the response time. The Cost-aware Balancing-based algorithm is close to the Dynamic-oriented algorithm, but its mismatch penalty does not exceed 15% even for this workload with high variance in service time. The conclusion is that in high variable and skewed workload models some knowledge about the computational cost of the services is important to improve performance. The S&D-CA-Bal algorithm is the algorithm to choose for supporting services for the Mobile Web in a geographically distributed system, because it outperforms all other service composition algorithms and it is insensitive to several system and workload parameters. This last result is very important for its applicability in a real context.

![Cumulative distribution of response times](image)

**Fig. 6** Sensitivity to workload computational skewness

7 Related work

The deployment of algorithms for the composition of Web services is a critical problem that attracted the interest of many researchers in the last years. The issue of composing Web services on the basis of their functional requirements has been addressed by several studies [28]. Moreover, standards have recently been proposed to compose services through workflow languages, such as the Business Process Execution Language (BPEL) [21] and the Web Ontology Language (OWL-S) [18], that have emerged as widely adopted solutions for the management of complex Web services obtained through composition of elementary building blocks. On the other hand, the problem of composing Web services on the ba-
sis of non-functional requirements received less attention and presents open issues not yet addressed in literature.

Most studies on Web service composition based on non-functional requirements model the problem of satisfying quality requirement as an optimization problem, where multiple characteristics are combined in a single objective function. Interesting surveys on the state-of-the-art solutions for Web service composition are provided in [1][8][12][16]. These studies provide useful classifications of typical quality characteristics and demonstrates that most available solutions for managing quality in Web Services aim to optimize some objective function that depends on the quality offered by the Web service providers. The proposed solutions for Web services composition may be divided in two general approaches: centralized approaches using global optimization and distributed solutions relying on local selection.

Many solutions rely on global optimization to handle quality constraints [4][14][30][31]. The work of Zeng. et al. [31] focuses on a quality-driven selection of services exploiting a global planning to find the best service components; the authors use linear programming techniques to find the optimal selection of component services. Similarly to this approach, Ardagna et al. [4] extends the linear programming model to include some local constraints. Linear programming methods are very effective when the size of the problem is small, but they suffer of poor scalability due to the exponential time complexity of the applied search algorithms. In [30] Yu et al. propose heuristic algorithms that can be used to find a near-to-optimal solution. The authors propose two models for the quality-based service composition problem: a combinatorial model and a graph model. However, both algorithms do not scale with respect to an increasing number of Web services and any distributed implementation of the algorithms would raise very high communication costs. The lack of scalability of all these solutions based on global optimizations tend to make them inappropriate for service composition in contexts characterized by dynamic and real-time requirements.

Several methods based on distributed approaches have been proposed in the last few years to address the issue of performance/scalability in composing Web services [2][13][14][25]. The study in [2][25] identifies different providers for each component service applying local selection, then exploit mixed integer programming (MIP) to choose the best composition to solve the given problem. Other approaches to Web service composition rely on semantic information or on specific features of the service discovery broker, as in [13][14]. All these studies share the common assumption that the quality of a Web service is not affected by the actual decision about the composition of the required Web services. This assumption is correct for static quality characteristics, such as provider reputation or server security, but it cannot be applied to dynamic quality characteristics, such as performance-related ones, because the algorithm determining the service composition has to consider the impact on the system, like nodes overload or herding effects. Management of dynamic quality of service constraints requires a more sophisticated approach where the impact of select composition of services over the distributed system is taken into account.

A complementary problem for the composition of the Web services is the problem of placing service replicas in a distributed environment [26][27]. This problem is conceptually similar to the problem of replicating Web applications. In particular, an interesting solution to this problem is proposed in [27] where multiple Web applications are replicated in a local context with bounds on the computational demand on the CPU and on the amount of available memory. Another example of solutions for service replication and placement is provided in [26], that proposes a suite of algorithms which jointly optimize the performance of dynamic content applications by reducing the client access times while also minimizing the resource utilization. Our approach for Web service composition can be integrated with algorithms for service replication and placement without any modification because Web service composition decisions are taken in real time and our algorithms operate on a very fine grained time scale (in the order of tens or hundreds of milliseconds), while the service placement algorithms operate on a time scale of tens of minutes.
8 Conclusions

Modern Internet applications are provided through the composition of Web services offered by different service providers. In this paper we focus on the issue of composing Web service to satisfy static and dynamic requirements in terms of quality of service. We demonstrate that in a realistic scenario, where the workload is subject to rapid changes over time, traditional algorithms for service composition that consider only static requirements lead to very poor performance. For this reason, we propose a class of novel algorithms for runtime service composition that aim to satisfy together static and dynamic requirements. Our experimental evaluation demonstrates that the proposed algorithms substantially outperform traditional solutions, with the response time of the static and dynamic algorithms that is reduced up to a 50% with respect to that of static algorithms. Furthermore, we carried out a sensitivity analysis to evaluate the stability of the results with respect to multiple system and workload parameters; this analysis allows us to identify the algorithm that achieves the most robust performance in every context.

References


