Six Strategies for Building High Performance SOA Applications

Uwe Breitenbücher, Oliver Kopp, Frank Leymann,
Michael Reiter, Dieter Roller, and Tobias Unger

University of Stuttgart, Institute of Architecture of Application Systems (IAAS)
{uwe.breitenbuecher, firstname.lastname}@iaas.uni-stuttgart.de

Abstract. The service-oriented architecture (SOA) concepts such as loose coupling may have negative impact on the overall execution performance of a single request. There are ways to facilitate high performance applications which benefit from this kind of architectural style compensating the caused overhead significantly. This paper gives an overview on six high level strategies to improve the performance of SOAs with a central service bus and presents how to apply them to build high performance service-oriented applications without corrupting the SOA paradigm and concepts on the technical level.

Keywords: Service-oriented architecture, High Performance, Strategies

1 Introduction

The key concepts of service-oriented architectures (SOAs) such as loose coupling, interoperability, or abstraction may have negative impact on the overall performance of applications. The reasons are additional costs for time-consuming operations such as message format transformations, dynamic service discovery, etc. [10]. In this paper we present six different improvement strategies which may increase the performance and show how to apply them to build high performance SOA applications. As the presented strategies are applied on a higher level than the operations causing the overhead, the strategies compensate these time-consumptions and additionally increase the overall performance.

In this paper we use two metrics for assessing the performance: Throughput and response time. Throughput denotes the maximum number of requests a SOA application can process in a certain period. Response time measures the time an application needs to respond to a request [14].

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 presents six strategies to improve the performance and how to apply them in an abstract service-oriented architecture with a central service bus. Finally, Section 4 concludes and provides an outlook on future work.

2 Related Work

This paper is a first attempt to show how a set of high level strategies can be applied to improve the performance of a SOA application without corrupting the underlying SOA concepts. Other work in the area of SOA performance improvements are focusing on the technical level. One example for technical improvements are performance best practices considering optimization strategies focusing on message processing, message structure, and message design of XML based protocols [11]. These optimizations are different from the presented strategies in this paper in the level of abstraction: The six presented strategies in this paper are applied on a high abstract level while the best practices propose optimizations for concrete technologies. FastSOA [12] is an architecture and software coding practice which considers optimization through native XML environments, a mid-tier service cache, and the use of native XML persistence. It combines the cache strategy presented in this paper with best practices by Endrei et al. [11], but lacks applying the other high level strategies in order to gain a higher overall performance.

3 High Performance Strategies

In this section we present six strategies which enable high performance service-oriented applications without corrupting the underlying SOA concepts [4]. We do not claim that these strategies are complete: They are inspired by the experiences in our research projects – mainly SimTech[1] – and have to be seen as a first attempt to design high performance SOA applications which has to be extended.

We present Parallel Processing, Caching, Dynamic Service Discovery, Dynamic Service Migration, Multiple Service Instantiation, and Multiple Service Invocation. Each strategy targets performance issues on an architectural level.

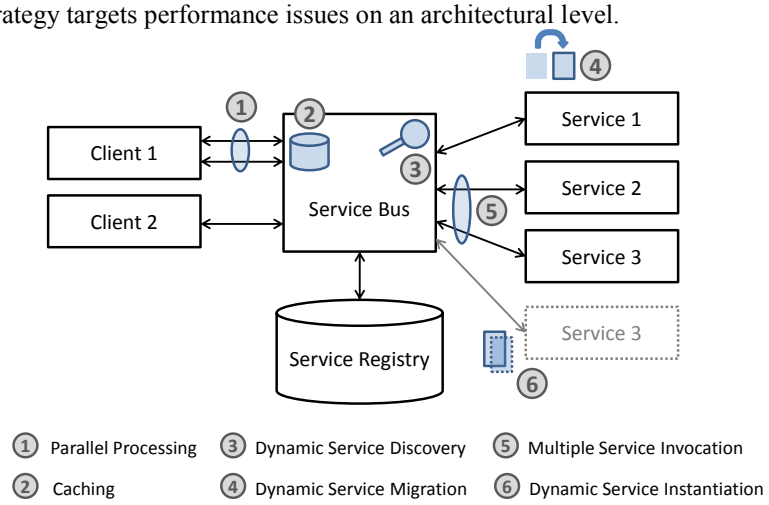

Fig. 1. Six high performance strategies applied to a SOA with central service bus (based on [3])

[1] http://simtech.uni-stuttgart.de

The strategies focus on SOAs having a service bus as central component (see Fig. 1): A s*ervice* is an application processing request messages and may returning response messages. A *client* is any application that sends request messages which have to be processed by services to a central component called service bus (a client can be a service, too). The *service bus* is a middleware component providing an integration platform to connect clients with services [6], [7]. It uses a *service registry* which stores all available services combined with a description of their functionality to look up appropriate services [7]. All messages sent by a client are routed through the service bus, which looks up an appropriate endpoint and sends the message to the selected service. After the service finishes the processing, response messages may be routed back to the requesting client.

The following subsections describe the six strategies. Each strategy has a *goal* describing the strategy's impact on the performance in one sentence. The *description* explains in more detail how to apply the strategy and why the performance is improved. The *assumptions* paragraph describes preconditions which have to be met to apply the strategy successfully. *Benefits* of applying the strategies are summarized as well as *downsides and problems* in a separate paragraph.

3.1 Parallel Processing

Goal. The goal of this strategy is increasing the internal throughput of requests in the application to improve the overall application performance.

Description. An application implemented as SOA consists of different services orchestrated together to provide new functionality: The application receives a request, invokes several services and thereby delegates tasks to them needed to provide the overall application functionality. This concept is called "Programming in the large" [9]. If the invocations are independent from each other they can be done in parallel at the same time which increases throughput and therefore the overall processing performance. The distributed computing paradigm of service-oriented architectures enables this feature. The strategy has to be implemented in clients (Fig. 1, Point 1).

Assumptions. It is assumed that each service is hosted on its own physical environment and therefore isolated from each other regarding performance. Thus, multiple concurrent service executions do not influence the performance of each other.

Benefits. The application of this strategy does not need a special performance optimized service bus to achieve high throughput.

Downsides and Problems. The client has to be able to send multiple requests at the same time to the service bus and has to wait for multiple responses which may arrive in various orders. This needs special programming effort as this kind of requesting has to be done asynchronously. There are technologies enabling this kind of service orchestration. One example is BPEL [8]. Another difficulty is identifying which requests can be done in parallel and which requests have to be processed sequentially. For applying this strategy to existing applications, the application flow may have to be changed which can lead to modifications of the overall application architecture.

Application Example. Examples for applying this strategy are all scenarios where requests can be processed independently from each other. For instance, in simulations there are often multiple matrix equations which may be solved at the same time. These equations are independent from each other as they are self-contained in a way that no external information is needed for solving.

3.2 Caching

Goal. The goal of this strategy is avoiding multiple processing of identical requests to speed up the application's performance.

Description. One opportunity to improve the performance of an application's request processing is to avoid the actual request processing at all by exploiting caching. The service bus is the central component which is responsible for any primary service request message consumption: Clients send request messages to the bus which routes the messages to selected services and the responses back to the corresponding requestors [3]. For certain requests the responses are always the same, e.g. a matrix equation solving service returns always the same solution for the same requested equation. These requests can be cached by the service bus to decrease the response time [1]. The strategy has to be implemented in the service bus (Fig. 1, Point 2).

Assumptions. The requests have to be comparable in a way that identical requests can be recognized.

Benefits. The application of this strategy is transparent for the requesting client. Thus, this strategy can be applied without the need for modifying already existing components (of course they have to send all requests to the service bus). If a request is served by the cache the whole service system is discharged.

Downsides and Problems. The identification of cacheable request-response pairs is difficult and causes overhead at the design time of the application. A request which cannot be served by the cache causes additional overhead for cache lookup and management tasks and even decreases the performance for processing this request.

Application Example. In the scientific domain, experiments are executed many times with only little modified input values and therefore internal simulation data within the simulation is often identical [2]. Thus, requests depending on this internal simulation data are also identical and can be cached for further experiment executions.

3.3 Dynamic Service Discovery

Goal. The goal of this strategy is to choose the fastest service for a certain request at runtime to decrease the response time.

Description. One can distinguish between two different binding techniques: Static binding and dynamic binding [10]. The first one enables the client to explicitly define

which service should be used while the latter one sends the request to the service bus which discovers a service matching the functional requirements of the request and then sends the request to this selected service [3]. This service discovery can be enriched by taking non-functional requirements expressing capabilities into account, too [6]: If there are functionally equivalent services, non-functional capabilities of the service are analyzed to select the service guaranteeing the fastest response time. This enables optimized load balancing, too. The Dynamic Service migration strategy (see Section 3.4) may be applied to optimize the services before comparing them. The strategy has to be implemented in the service bus to enrich the service discovery (Fig. 1, Point 3).

Assumptions. To select the fastest service, all available suitable services have to be comparable in their performance for processing a certain request. This performance values have to be predictable automatically (either by the service bus or by the respective services).

Benefits. The application of this strategy is transparent for the requesting client. Thus, this strategy can be applied without the need for modifying already existing legacy components (of course they have to send all requests to a service bus).

Downsides and Problems. This strategy only improves the performance if the overhead caused by the discovery is below the time saved by the faster service. Otherwise dynamic service discovery even slows down the performance.

Application Example. An example is a simulation orchestrating services for complex calculations whose response time depends on a specified requested quality of the output data. Some algorithms offer only low quality of data but guarantee a fast calculation. Other algorithms are designed to achieve high quality of data but are more time-consuming. Depending on the required quality of data (non-functional requirement) the fastest service can be chosen.

3.4 Dynamic Service Migration

Goal. The goal of this strategy is to achieve the fastest response time for processing a request regarding the environment and location conditions a service is hosted on.

Description. There are services whose response time to process a request depends on the power of the environment they are hosted on. The Dynamic Service Migration strategy moves services from less powerful machines to more powerful ones to scale up [13, 15]. Other scenarios increasing the performance are the migration of one service collocated to another service to cut down network costs or the migration of other services hosted on the same environment to other environments to free resources. This component-based migration is possible because of the loose coupling concept of SOA. The strategy may be implemented in the service bus which triggers and manages the migration (Fig. 1, Point 4).

Assumptions. To apply this strategy, the migration of services has to be feasible.

Benefits. The application of this strategy is transparent for the requesting client. Thus, this strategy can be applied without the need for modifying already existing components. Recall that we assume that the services send all requests to a service bus.

Downsides and Problems. The migration of a running service from one machine to another machine is complex and needs management operations which have to be implemented. Especially handling of local data is difficult, because even if a service can be migrated to another machine, a huge amount of data which also has to be transferred can lead to problems: the time savings achieved by the more powerful environment may be too small and the overall processing time (including the time needed for migration) for a single request even increases. To avoid this, the design of the services and the overall architecture of the application have to be aware that this strategy may be applied. This causes additional overhead at the development time and is generally difficult. To find out whether a migration of a service on runtime leads to a faster response time to process a certain request is difficult and depends on many factors. The component managing this migration has to calculate predictions in which scenarios and constellations a migration makes sense.

Application Example. One example taken from our experiences with bone remodeling simulation workflows is the processing of big data sets. The simulations typically process a huge amount of data by sequentially invoking services and passing the data from one service to another service. Because the services are used by multiple simulations, it is not possible to host all services and store all needed data on a single environment. Thus, the migration of services co-located to the data to be processed on runtime may improve the performance in terms of response time because network costs are cut down.

3.5 Multiple Service Invocation

Goal. The goal of this strategy is to choose the fastest service for a certain request at runtime to decrease the response time.

Description. The selection of the fastest service can be difficult, especially if there are completely different ways to process a single request. There are situations where the Dynamic Service Discovery strategy cannot be applied to discover the fastest service because the required values to compare the different services are not calculable. SOA offers a solution to achieve maximum request processing performance by sending the request to all available appropriate services concurrently and taking the response returned by the first responding service. This decreases the response time to an ideal value as the fastest available service is implicitly chosen. To make this work, the different services have to be isolated in a way that they do not affect each other's response time. The strategy has to be implemented in the service bus (Fig. 1, Point 5).

Assumptions. It is assumed that the multiple service invocations have no negative impact on the performance of other request processing services.

Benefits. The application of this strategy is transparent for the requesting client. Thus, this strategy can be applied without the need for modifying already existing components. Recall that we assume that the services send all requests to a service bus.

Downsides and Problems. The concurrent invocation of multiple functional identical services basically produces unnecessary workload for the whole service-oriented environment. To avoid negative impact on other services in terms of performance cloud technology may be used for the provisioning of new services discharging the system (i.e., applying the Multiple Service Instantiation strategy, see Section 3.6).

Application Example. One example from the mathematics domain is solving a matrix equation using numerical or algebraic techniques. For a numerical algorithm starting with random values trying to converge towards the solution by executing multiple iterations, the number of steps and thus the time needed to calculate the solution is not predictable. Thus, for certain equations, data sets and algorithms it is impossible to determine the fastest solving algorithm in advance.

3.6 Multiple Service Instantiation

Goal. The goal of this strategy is to increase the performance by invoking only services having free capacity.

Description. The performance of the overall system decreases if services cannot process a large number of requests any more. If there is no possibility to balance the outstanding workload differently, this strategy solves the problem by instantiating more services functionally equivalent to the overloaded ones [15]. The new instantiated services can be invoked in parallel and hence scale out. The distribution of the workload discharges overloaded services and thus increases the throughput. The strategy may be implemented in the service bus (Fig. 1, Point 6).

Assumptions. The current workload and utilization of a service has to be visible to the service bus to enable the selection of appropriate services and there have to be free resources for hosting the new instantiated services isolated in a way that the concurrent executions do not decrease the performance of each other.

Benefits. The application of this strategy is transparent for the requesting client. Thus, this strategy can be applied without the need for modifying already existing components. Recall that we assume that the services send all requests to a service bus.

Downsides and Problems. The application of this strategy only improves the performance if the additional needed time for instantiation is below the time which is saved by invoking the cloned service. Especially if local data also has to be cloned and transferred to another hosting environment, this leads to additional time-consumptions caused by network costs. A solution to solve this problem of data migration is using stateless services. Applying this strategy requires additional resources in terms of hardware or virtualized systems. Thus, it has to be ensured that this has no negative impact on the performance of other services (e.g. through cloud technology).

Application Example. In service-oriented environments where multiple SOA applications run at the same time certain services may be overloaded because their offered functionality is so general that it is used by many of these applications. In our simulation experiments matrix solving services are frequently overloaded, for example.

4 Conclusion and Outlook

We presented six high level strategies to increase the overall performance of service-oriented applications and showed how to apply them to build high performance SOA applications. As five of the six strategies can be implemented in a performance-driven service bus we plan to implement this bus and integrate our existing migration prototypes [5]. This bus enables performance optimization which is transparent to the orchestrating component and provides a basis for evaluation scenarios for showing that the applied strategies also increase the overall performance in practice.

References

1. Rao, F. Y. et al.: Message Oriented Middleware Cache Pattern – a Pattern in a SOA Environment. In: Fourth "Killer Examples" for Design Patterns and Objects First Workshop (2005)
2. Sonntag, M., Karastoyanova, D.: Next Generation Interactive Scientific Experimenting Based On The Workflow Technology. In: Proceedings of the 21st IASTED International Conference on Modelling and Simulation (2010)
3. Keen, M. et al.: Patterns: Implementing an SOA Using an Enterprise Service Bus. IBM Redbooks (2004)
4. Erl, T.: Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. Prentice Hall PTR (2004)
5. Binz, T. et al.: CMotion: A Framework for Migration of Applications into and between Clouds. In: Proceedings of SOCA (2011)
6. Leymann, F.: The (Service) Bus: Services Penetrate Everyday Life. In: ICSOC (2005)
7. Chappell, D.A.: Enterprise service bus. Theory in Practice. O'Reilly Media (2004)
8. Weerawarana, S. et al.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR (2005)
9. DeRemer, F. and Kron, H.: Programming-in-the-Large Versus Programming-in-the-Small. Software Engineering, IEEE Transactions on, SE-2, 80-86 (1976)
10. Papazoglou, M.: Web Services: Principles and Technology. Pearson Prentice Hall (2008)
11. Endrei, M. et al.: Patterns: Service-Oriented Architecture and Web Services. IBM Redbooks (2004)
12. Cohen, F.: FastSOA. Morgan Kaufmann (2007)
13. Hao, W. et al.: Dynamic Service and Data Migration in the Clouds. In: Computer Software and Applications Conference (2009)
14. Weikum, G. et al.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
15. Lee, J. Y. et al.: Software Approaches to Assuring High Scalability in Cloud Computing. In: IEEE 7th International Conference on e-Business Engineering (ICEBE) (2010)