Autonomic Execution of Web Service Compositions[*]

Cesare Pautasso Thomas Heinis Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{pautasso,heinist,alonso}@inf.ethz.ch

Abstract

An increasing amount of Web services are being implemented using process management tools and languages (BPML, BPEL, etc.). The main advantage of processes is that designers can express complex business conversations at a high level of abstraction, even reusing standardized business protocols. The downside is that the infrastructure behind the Web service becomes more complex. This is particularly critical for Web services that may be subjected to high variability in demand and suffer from unpredictable peaks of heavy load. In this paper we present a flexible architecture for process execution that has been designed to support autonomic scalability. The system runs on a cluster of computers and reacts to workload variations by altering its configuration in order to optimally use the available resources. Such changes happen automatically and without any human intervention. This feature completely removes the need for the manual monitoring and reconfiguration of the system, which in practice is a difficult and time-consuming operation. In the paper we describe the architecture of the system and present an extensive performance evaluation of its autonomic capabilities.

1 Introduction

Open service oriented architectures face an important scalability problem when the services published on the Web become successful. Successful services have the potential to be concurrently invoked by a very large number of clients [12]. In the past few years, process management tools and languages have gained widespread acceptance to model and enforce the business conversations and protocols followed by the Web services (e.g., [1, 10, 18]). Thus, scalability is an important issue that imposes high demands on the underlying process management infrastructure. Whenever a new conversation is started, a new process instance has to be created. Then, for every message exchanged with the service, the state of the underlying process has to be updated to reflect the progress of the conversation.

Typically, to achieve the required level of scalability, the correlation of the messages and the management of their corresponding business processes are partitioned among a replicated execution environment, for example, a cluster of computers [9]. This has the advantage that such distributed process execution environments can be scaled to handle large workloads. However, the configuration of such cluster-based systems is not easily determined *a priori*, especially when facing an unpredictable workload. In order to deal with such a highly variable volume of messages, dynamic reconfigurability is a very important requirement [17]. With it, the size of a running system can be adapted to keep the balance between servicing its workload with optimal performance and ensuring efficient resource allocation.

In this paper we present the architecture of an autonomic process execution platform for Web service composition which can dynamically and autonomously determine its optimal configuration based on the current workload. This is an important contribution as most existing systems tackle the scalability problem by statically and manually partitioning the workload across different sites [4].

As part of the JOpera project [13], we have designed and implemented a flexible platform for process execution that achieves scalability by replicating its key components across a cluster of computers. Additionally, the system employs an autonomic controller that monitors the current workload and state of the system. It uses this information to determine whether the system is running in the optimal configuration or, alternatively, whether reconfiguration actions have to be carried out. For example, if a peak of request messages is detected, more nodes of the cluster are allocated to process them. To do so, the autonomic controller uses different policies which can be chosen according to different goals (e.g., minimize resource allocation or minimize response time). Our experiments show the feasibility of the approach and demonstrate that the autonomic controller can reconfigure the system automatically.

[*]Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820) and the *Swiss Federal Office for Education and Science* (ADAPT, BBW Project No. 02.0254 / EU IST-2001-37126).

The paper is organized as follows: in Section 2 we present the architecture of the JOpera Web service composition platform. Our approach of extending a reconfigurable system with autonomic features is outlined in Section 3. A performance evaluation of several different control policies is presented in Section 4. In Section 5 we briefly discuss related work before drawing some conclusions in Section 6.

2 Background

In this section we give a brief overview of the distributed architecture of the JOpera Web service composition platform (Figure 1). More details can be found in [14].

Processes model the interactions between different Web services in terms of their data exchanges and constraints in the order of invocation [3]. The execution of a process begins with a request message sent by a client. Such requests are added to the *process queue* and handled by the navigator, which executes the processes by scheduling the invocation of the next services based on the service invocations that have already been completed.

Once the navigator determines that a certain service is ready to be invoked, the corresponding service invocation requests are stored in the *task queue*. The interaction with the service provider using the appropriate mechanisms and protocols (e.g., by exchanging messages encoded in SOAP [16]) is managed by the *dispatcher* component. The name of this component reflects its function of dispatching messages to and from the corresponding service providers. After a response message has been received, the dispatcher notifies the navigator through the *event queue* by posting the results of the invocation so that the process may resume its execution.

We have chosen to decouple the execution of the processes from the execution of their tasks because these operations have a different granularity. It is to be expected that the invocation of a remote service performed by the dispatcher may last significantly longer than the time taken by the navigator for scheduling it.

Decoupling process navigation from service invocation also enables the system to scale along two different and orthogonal directions. In case a large service invocation capacity is required, the dispatcher thread can be replicated to manage the concurrent invocation of multiple services. Likewise, if many processes have to be executed concurrently, the navigator can be replicated as well. The resulting pool of navigator and dispatcher threads are linked by event queues. This way, navigators generate service invocation requests which are consumed by the dispatchers. Vice versa, dispatchers send event notifications back to the navigators with the results of the invocations.

By providing a distributed implementation of such queues, it is possible to scale the system to run on a cluster of computers, as navigators and dispatchers can be physically located on different nodes. Furthermore it is possible to dynamically grow and shrink the size of the system without disrupting its normal operation.

3 Autonomic Execution of Compositions

Given the reconfigurable architecture of JOpera [6], in this section we present the design of the autonomic controller, the component responsible for automatically configuring the system.

3.1 Control Algorithm

The control algorithm implemented by the autonomic controller loops over the following three phases [11]. First, during the monitoring phase, a snapshot of the current state of the system configuration is taken. The information policy defines what information is collected at this stage. Second, during the planning phase, the controller uses the collected information to determine whether the system is balanced or whether a configuration change is necessary. The optimization policy determines the criteria (e.g., thresholds) and the outcome (e.g., a certain number of idle nodes of the cluster can be released). The actual configuration changes are carried out during the third phase, which is simply skipped if no such actions are required. The selection policy is used to convert the general reconfiguration plan to a concrete change in the configuration, using additional information collected during the monitoring phase.

3.2 Information Policy

The information policy defines which performance indicators and which part of the configuration information are fed back into the autonomic controller.

By considering the architecture of the distributed process execution platform (Figure 1), there are several points that can be instrumented to provide performance indicators. Since the navigator and dispatcher threads communicate asynchronously through event queues, it is possible to sample the current *queue length* in order to detect whether the system is balanced. In case the queue grows, it is likely that there are not enough consumers processing its events. Conversely, if the length of a queue drops, there may be too many consumers (or too few producers).

In order to compare the performance of different optimization policies, it may also be useful to measure their corresponding *resource allocation*. To do so, the system can track for how long it has been using a certain node of the cluster. These allocation logs are kept as part of the configuration information.

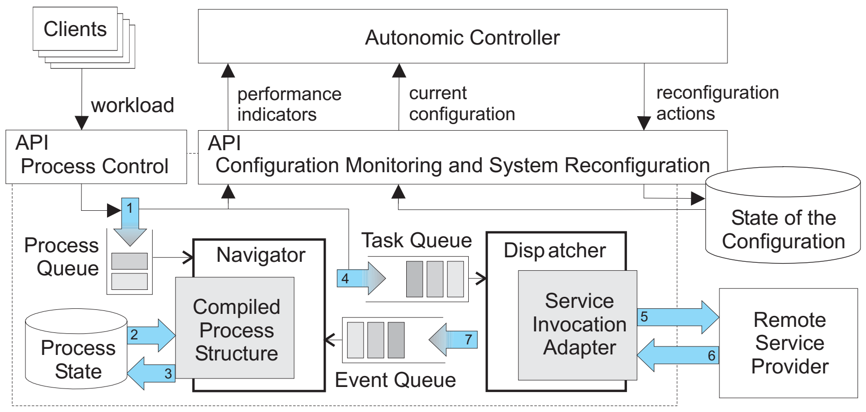
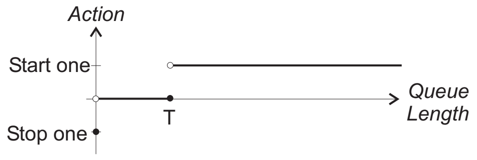


Figure 1. Architecture of the JOpera Autonomic Service Composition Platform

3.3 Optimization Policy

The optimization policy specifies how to achieve certain goals in terms of mapping a combination of the previously defined performance indicators onto a set of reconfiguration actions. In general, the controller addresses multiple (and contradictory) goals. First of all, it should ensure that the system reacts with reasonable performance under a given workload. The simplest way to achieve this points to a strategy that configures the system to always provide excess capacity so that unpredictable peaks in the workload can be absorbed. Although this approach maximizes the performance of the system measured in term of its process execution capacity, it turns out to be wasteful in terms of resource allocation. Thus, the optimization policy must provide support for both of these goals: maximizing the system's throughput and minimizing the resource allocation.

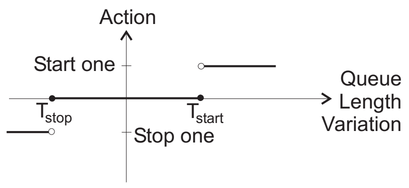
The simplest optimization policy we have considered uses a single threshold T compared to a certain non-negative controlled variable v. Whenever $v > T$ the controller decides to grow the size of the system by one thread. This ensures that peaks in the workload causing the controlled variable to increase will be detected and taken care of by growing the system. If $v = 0$, the outcome is to shrink the size of the system by one thread. No reconfiguration action is planned if $0 < v \leq T$.



Simple Threshold Policy

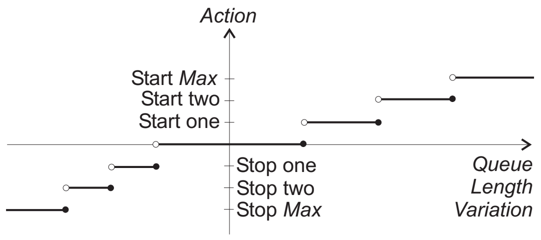
We have applied this *simple* control policy by binding the controlled variable v to the length q of the queue of events consumed by the navigators and the dispatchers and by introducing different thresholds (T_d, T_n) for each kind of thread. To tune their values, the thresholds can be interpreted as the number of events which is expected to be handled by each kind of thread. Typically $T_n > T_d$, as navigators can handle a larger volume of events than dispatchers.

As opposed to reading the current length of the event queue, the *differential* control policy uses the first order variation ($\Delta q = q(t) - q(t-1)$) of the queue length to make its decisions.



Differential Policy

Still, the possible outcomes and the decision strategy are the same as in the simple threshold policy. We introduced this policy because the length of the event queues is a good indicator of the internal activity of the system. Its variations can be used to detect whether the system is lagging behind (when $\Delta q > 0$) or the number of events to be processed is diminishing ($\Delta q < 0$). Thus, two different thresholds are used to determine whether a new thread should be started ($\Delta q > T_{start} > 0$) or stopped ($\Delta q < T_{stop} < 0$).



Proportional Policy

The *proportional* control policy uses a set of thresholds to determine whether one or more threads should be started or stopped, proportionally to Δq. To avoid instability problems, we set a limit to the maximum number of threads that can be started or stopped at once. This policy also uses the previously described Δq as controlled variable, since it provides both positive and negative values that can be used as input into the control decisions. Compared to the simple and differential policies, we expect this policy to be more reactive, as it can plan to start many threads at once if a large variation in the workload is detected.

3.4 Selection Policy

The selection policy defines how to map abstract reconfiguration decisions to concrete actions affecting the current system configuration. One of the reasons for separating the planning of the configuration change from the actual modification actions is that, depending on the current state of the configuration, it may not always be possible to follow the plan. For example, when the system is overloaded, there may not be any spare capacity available to start new threads. A simple model of the system representing the most current state of the configuration is stored in the form of a list containing the IP address of each node and the set of threads the node is running. This model is stored centrally so that it is easy to access and modify. It is updated by the nodes which register and unregister themselves.

With this information the controller may already find out which nodes can be used to grow the system (the nodes which are currently not running a particular kind of thread). Conversely, only the nodes that are already in use are the candidates for being released, if the controller decides to shrink the system size.

4 Measurements

The goal of the measurements is to show the autonomic process execution platform in action, whereby the configuration of the system is adapted automatically to different workload conditions. We begin with a brief description of the characteristics of the workload we have used to benchmark the Web service composition platform and its autonomic reconfiguration capabilities. In the following, we show the performance of a static configuration to demonstrate JOpera's basic scalability properties before we move on to the evaluation of the three autonomic control policies introduced in Section 3.

4.1 Workload and testbed description

Since there are no standardized benchmarks for autonomic Web service composition execution platforms, we have defined a simple workload to evaluate the system under extreme conditions. The workload imposed on the system can be described as a peak of concurrent client requests to start the execution of a certain number of new processes. Thus, the *size* of the workload can be characterized by the number of processes to be executed concurrently. Although the number of tasks and the structure of the processes also influence the performance of the system, for these experiments we have focused on a homogeneous workload consisting of processes composed by 10 parallel tasks whose invocation time has been set to 8 seconds. We limited our experiments to this kind of workload because this simplifies the analysis of the results of our experiments and due to space limitations. We plan to continue the evaluation the system with heterogeneous and continuous workloads as part of future work. For the experiments, JOpera has been deployed on a cluster of up to 32 nodes. Each node is a 1.0GHz dual P-III, with 1 GB of RAM, running Linux (Kernel version 2.4.22) and Sun's Java Development Kit version 1.4.2.

4.2 Static configuration

At the very beginning of this paper we have argued that scalability is an important issue for Web service composition execution tools and we have also been describing JOpera's ability to execute processes across a cluster of computers. Figures 3a and 3b demonstrate JOpera's scalability: while it takes 973.22s to execute 800 concurrent processes using only 1 dispatcher and 1 navigator thread, it requires only 73.13s to process the same workload with a system statically configured to use 10 navigator and 22 dispatcher threads. This *static 22/10* configuration is suitable for this kind of workload: at the time when the processes are started, the system is able to cope with it. Enough threads are ready to handle the execution of both processes and performance indicators.

However, configuring the system statically reveals two main problems. First, static configuration potentially leads to a waste of resources since the cluster remains fully allocated to JOpera although it would be possible to reduce the resource allocation after processing the surge. Second, the configuration may not be optimal to deal with all kinds of workloads, hence reconfiguration is still required. Manual reconfiguration is not a trivial task because misconfiguring the system may lead to a loss in performance, as can be seen when comparing the batch execution time achieved with the *static 22/10* and the *static 10/22* configurations for the same workload (Figure 2).

4.3 Autonomic configuration

In order to compare the autonomic controller with a statically configured system, we have implemented and eval-

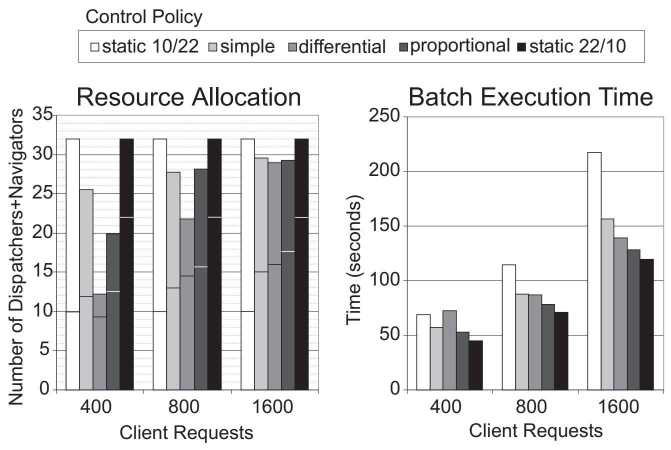


Figure 2. Comparison of the policies

uated the policies described in Section 3 using different workload sizes and configuring the control algorithm to run every second.

Simple Policy: The *simple* policy configures the system by adding one navigator thread at a time to the configuration as long as the size of the process queue exceeds the threshold $T_n = 50$. The same holds with $T_d = 10$ for the dispatcher threads servicing the task queue. Figure 3c shows the *simple* policy responding to a peak of 800 processes. Although the process queue is filled up quickly, the configuration adapts only slowly. The size of the task queue grows comparatively high because of the large number of navigator threads that are active as soon as the configuration has grown to its maximum size. The *simple* policy attempts to grow the configuration as long as the queue sizes are bigger than T_n and T_d. Given the values of the thresholds, this happens during most of the experiment's duration.

Differential Policy: Instead of only considering the queue length, the *differential* policy takes the growth of the queue into consideration. Once the growth of the task queue has overstepped the T^d_{start} threshold (set to 10 in this experiment), one dispatcher thread is added to the configuration. Vice versa, if the variation of the queue is below $T^d_{stop} = -10$, a dispatcher is removed from the configuration. The same mechanism applies to navigator threads, except that $T^n_{start} = 50$, $T^n_{stop} = -50$. As it can be seen in Figure 3d, this policy adapts to the current workload without letting the system constantly grow until saturation is reached. Instead, the growth of the system is coupled with the growth of the workload. The thresholds chosen allows the controller to follow small variations in the workload. Small increases of the task queue length result in an increase in the number of dispatcher threads.

Proportional Policy: The *proportional* policy tries to improve the reaction time of the system. In contrast to the *simple* and *differential* policies, the *proportional* policy adds or removes a variable (but limited) number of threads to the current configuration proportionally to the variation of the queues. The magnitude of the reconfiguration actions has been limited to 3 navigators and 10 dispatchers. By adjusting the number of threads in larger increments, the system is able to adapt faster to the workload peak (Figure 3e). Given the initial surge of the process queue, the system reaches a stable configuration much faster by quickly increasing the number of navigators and in turn also the number of dispatchers. Similar to the *simple* policy, the size of the task queue remains quite high in this case as well, as the configuration uses a large number of navigators. However, the controller reacts to this by adding more dispatchers, causing a drop in the task queue (Figure 3e, seconds 6, 12 and 24).

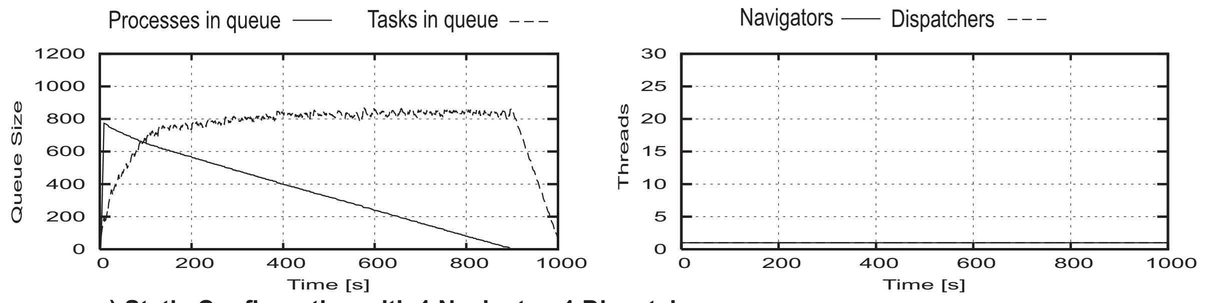
4.4 Comparison of the policies

In order to develop an idea about how well the different policies perform, we have evaluated the policies regarding the time used to execute batches of 400, 800 and 1600 processes and the average resource allocation over this time. Figure 2 gives an overview of the results.

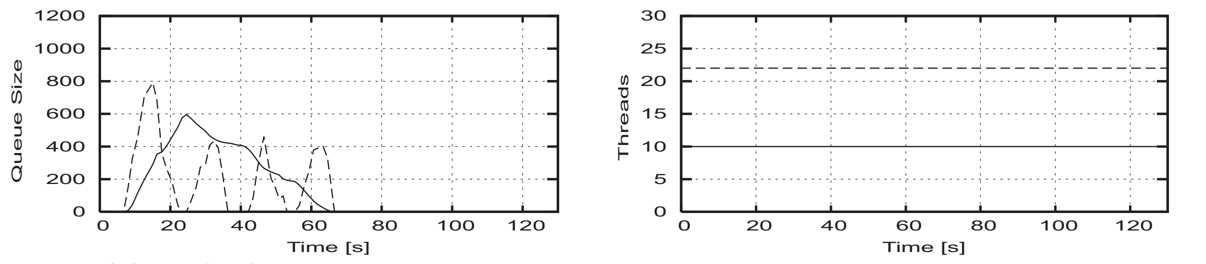
The time for executing a batch was measured as the time between the arrival of the first client request in the queue and the time the execution of the last process was completed. The average resource allocation was measured as the sum of the time each of the nodes was running a JOpera thread divided by the duration of the batch.

Not surprisingly, the average resource allocation for the two static configurations, with 22 dispatchers and 10 navigators and vice versa, is 32. A more interesting result is that, although the same number of nodes is used, the time to execute the same batch is between 50% (batch size 400 processes) and 82% (batch size 1600 processes) bigger. This behavior implies that the static configuration using 10 navigators and 22 dispatchers is more suitable to run the workload. Thus, configuring the system manually and statically potentially leads to a suboptimal configuration both in terms of performance and resource allocation. The *static 22/10* configuration serves as a good example for this behavior: while it is between 10% and 62% faster than the autonomic policies tested, it also uses the most resources (between 108% and 262% more).

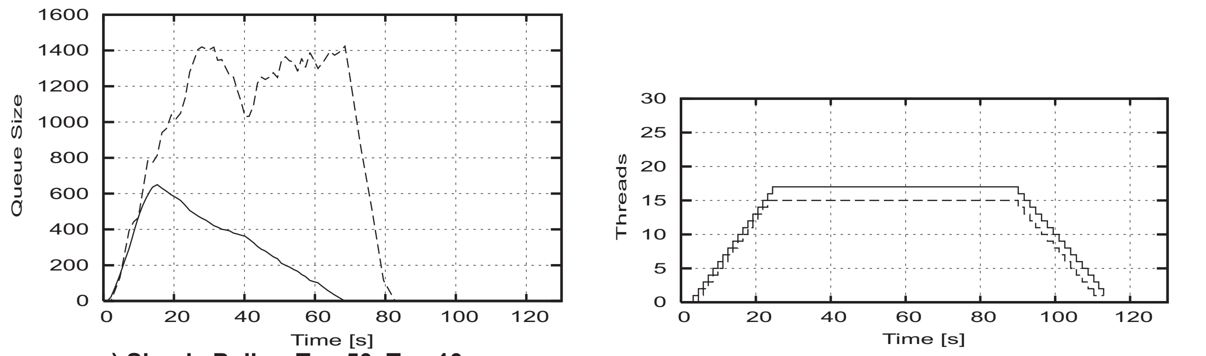
As shown in Figure 3c, the *simple* controller slowly grows the system to its maximum size keeping the number of dispatchers and navigators balanced, which turns out to be a sub-optimal configuration. This leads to an excessive use of the resources, although the high allocation does not enhance the performance. The time required to execute the 800 processes batch using the simple policy is 22% higher than in case of the *static 22/10* configuration. The two main reasons for this behavior are the following. First, the simple policy adapts slowly to the workload imposed on the system. In the case of 800 processes it requires 24.49 seconds



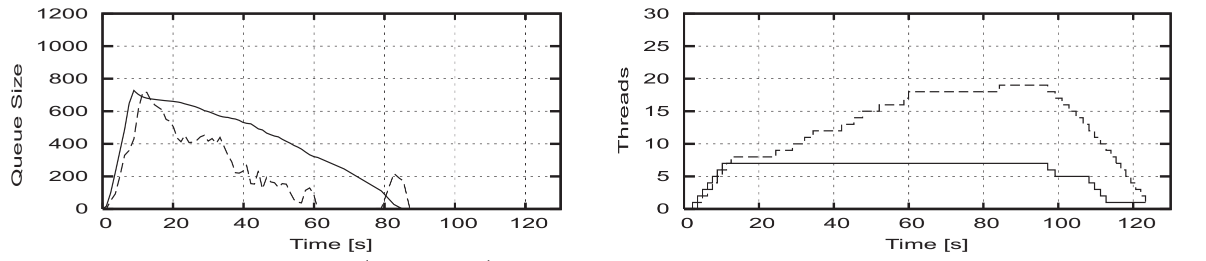
a) Static Configuration with 1 Navigator, 1 Dispatcher



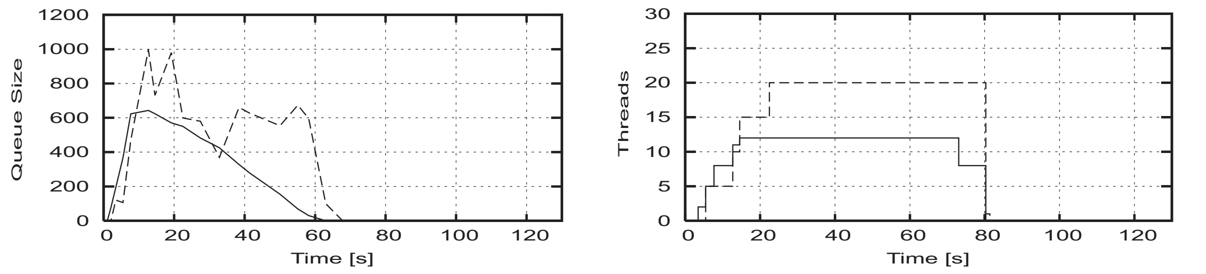
b) Static Configuration with 10 Navigators, 22 Dispatchers



c) Simple Policy, $T_n = 50$, $T_d = 10$



d) Differential Policy, $T^d_{stop} = -10$, $T^d_{start} = 10$, $T^n_{stop} = -50$, $T^n_{start} = 50$



e) Proportional Policy

Figure 3. Traces of the length of queues (left) and the state of the configuration (right) with different control policies reacting to a workload peak of 800 concurrent processes

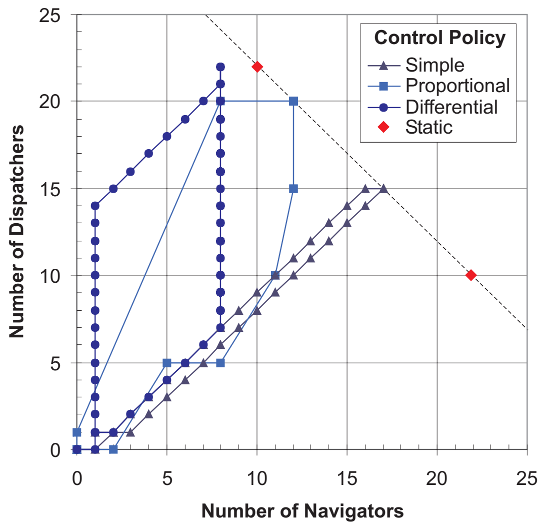


Figure 4. Configurations reached by different control policies

to grow to the full configuration because it adds only 1 node at a time. The other reason consists of the suboptimal partitioning of the nodes between navigators and dispatchers. This can also be seen in Figure 4. This figure illustrates the evolution of the configuration along its two main dimensions (the number of navigators and the number of dispatchers) when using different policies. While the *static 22/10* configuration performs better and the inverse configuration (22 navigators, 10 dispatchers) performs worse, the *simple* policy converges to an intermediate configuration (17n, 15d) because both queue sizes exceed the thresholds, letting the configuration grow symmetrically until saturation is reached. All other policies tend towards the former with (8n, 22d) for the *differential* policy and (12n, 20d) for the *proportional* policy.

The *differential* policy performs better (13% in the case of 800 processes batch) than the *simple* policy regarding time and has up to 39% (400 processes batch) smaller allocation than all other policies. The reason why the execution of the same workload takes up to 62% longer than the *static 22/10* configuration is found in its slow adaptation to the workload, similar to the *simple* policy. In contrast to the *simple* policy, the *differential* policy only increases the number of threads as long as the queue growth is bigger than the thresholds T^d_{start} or T^n_{start}: as soon as the process queue stops growing (8n, 7d in Figure 4 or after 10s in Figure 3d), the number of navigators remains the same whereas the number of dispatchers is still increasing according to the growth of the task queue. This slow and also resource saving way of growing the configuration is the main reason why the allocation is generally low: this policy does not saturate the system like the *simple* policy, but rather devotes only the strictly necessary nodes.

The *proportional* policy performs only slightly worse with regard to the batch time than the *static 22/10* configuration, but does better in terms of allocation. The difference in time can be explained with the delay required to grow the configuration: while this is none in case of the static configuration, it takes 14.56 seconds to reach the full configuration in case of the *proportional* policy, increasing the overall batch execution time by a maximum of 18%.

Regarding the resource allocation, the result of this policy compared to the static configuration is only little lower when executing the 1600 processes batch and significantly lower when executing the 400 processes batch. This result indicates that the controller adapts to the size of the workload. In case of 1600 processes, the controller grows the configuration until all nodes are used whereas in the case of 400 processes only part of the nodes will be used.

4.5 Discussion

Using the simple strategy of monitoring the length of process and task queue in order to determine reconfiguration actions, has already led to very promising results. As expected however, it is difficult to determine a globally optimal policy. The policies we evaluated offer different characteristics along the trade-off between execution time and resource allocation. Thus, a policy can be chosen to drive the automatic configuration of the system according to the overall goal within this trade-off.

Although the policies we introduced already performed satisfactorily, we intend to further extend them. For example, the random selection policy works well in homogeneous environments, but may need a more refined model of the configuration and more advanced selection strategies to deal with heterogeneous environments. Similarly, the information policy we introduced is based on the current state of the system. It would be useful to enhance it by taking into account the history of the system's configuration and past performance.

Each policy can also be tuned by setting its threshold parameters. In the experiments, we did so heuristically by observing the behavior of the system and estimating the capacity of each type of thread. In general, setting these thresholds appropriately tends to be difficult and misconfiguring them may also result in a performance penalty. As part of future work we plan to explore this issue in more detail.

5 Related Work

A large amount of research results are available in the context of scalable process execution engines (e.g., [2, 4, 7, 9, 14]). However, given the design of a distributed engine, the practical problem of how to configure it at runtime in order to achieve good performance under different workload conditions is still poorly understood. As an example, the GOLIAT [5] tool uses the expected characteristics of the workload to make predictions about the performance of a certain configuration of the Mentor-lite engine. At deployment time, the tool helps the user to determine interactively how many resources should be allocated to achieve the desired level of performance. In this paper, we show how to replace such manual configuration steps with an autonomic controller [8]. Thus, we propose to determine the configuration of the distributed engine automatically, taking into account measurements of the system's performance under the actual (and unpredictable) workload. Furthermore, with our approach it is not required to allocate resources to the engine on a permanent basis, as the autonomic controller can grow and shrink the system dynamically using whatever shared resources are available at the moment. In this context, the problem of optimally choosing which resource to use is dual to a resource management and scheduling problem (e.g., [15]): whereas a scheduler attempts to fit the workload to the available resources, the goal of the autonomic controller is to adapt the configuration of the resources to better service the workload.

6 Conclusions

This paper presents the architecture of an autonomic process execution platform for supporting service oriented architectures. It features a unique mix of distributed execution and dynamic reconfigurability that make it suitable for autonomic reconfiguration. This shows the feasibility of applying autonomic computing principles to automatically adapt the configuration of the process execution platform in response to unexpected variations in its workload. Furthermore, employing an autonomic controller greatly reduces the administrative overhead of manually reconfiguring the system, which is a difficult and time consuming task.

In particular, we have described the control algorithm and policies followed by the autonomic controller, the key component implementing the autonomic reconfiguration capabilities. As our performance evaluation with different workload sizes indicates, the controller outperformed the manual, static configuration by achieving a good trade off between two different goals: minimizing resource allocation while guaranteeing satisfying performance. Furthermore, we showed that the performance of the controller depends on the actual information, optimization and selection policies that are used.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, Architectures and Applications*. Springer, November 2003.

[2] T. Bauer and P. Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems (CoopIS'97)*, pages 99–108, Kiawah Island, South Carolina, USA, 1997.

[3] F. Casati and M.-C. Shan. Dynamic and Adaptive composition of e-services. *Information Systems*, 26:143–163, 2001.

[4] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th World Wide Web Conference*, pages 134–143, New York, NY, USA, 2004.

[5] M. Gillmann, W. Wonner, and G. Weikum. Workflow Management with Service Quality Guarantees. In *Proceedings of the ACM SIGMOD Conference*, pages 228–239, Madison, Wisconsin, 2002.

[6] T. Heinis, C. Pautasso, and G. Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *Proc. of the 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005.

[7] P. Heinl and H. Schuster. Towards a Highly Scaleable Architecture for Workflow Management Systems. In R. R. Wagner and H. Thoma, editors, *Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, pages 439–444, Zurich, Switzerland, September 1996.

[8] IBM. Autonomic Computing: Special Issue. *IBM Systems Journal*, 42(1), 2003.

[9] L. jie Jin, F. Casati, M. Sayal, and M.-C. Shan. Load Balancing in Distributed Workflow Management System. In G. Lamont, editor, *Proceedings of the ACM Symposium on Applied Computing*, pages 522–530, Las Vegas, USA, 2001.

[10] F. Leymann. Web services: Distributed Applications without Limits. In *Proceedings of the International Conference on Business Process Management (BPM 2003)*, pages 123–145, Eindhoven, The Netherlands, 2003.

[11] N. S. Nise. *Control systems engineering*. Wiley, 4th edition, 2004.

[12] J. Norris, K. Coleman, A. Fox, and G. Candea. OnCall: Defeating Spikes with a Free-Market Application Cluster. In *International Conference on Autonomic Computing (ICAC'04)*, pages 198–205, New York, New York, May 2004.

[13] C. Pautasso. JOpera: Process Support for more than Web services. `http://www.jopera.org`.

[14] C. Pautasso and G. Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce (IJEC)*, 9(2):104–141, Winter 2004/2005.

[15] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[16] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/SOAP`.

[17] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer. A system model for dynamically reconfigurable software. *IBM Systems Journal*, 42(1):45–59, 2003.

[18] L.-J. Zhang and M. Jeckle. The Next Big Thing: Web services Collaboration. In *Proceedings of the International Conference on Web services (ICWS-Europe 2003)*, pages 1–10, Erfurt, Germany, 2003.