Autonomic Resource Provisioning for Software Business Processes

Cesare Pautasso Thomas Heinis Gustavo Alonso

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Abstract

Software development nowadays involves several levels of abstraction: starting from the programming of single objects, to their combination into components, to their publication as services and the overall architecture linking elements at each level. As a result, software engineering is dealing with a wider range of artifacts and concepts (i.e., in the context of this paper: services and business processes) than ever before. In this paper we explore the importance of having an adequate engine for executing business processes written as compositions of Web services. The paper shows that, independently of the composition language used, the overall scalability of the system is determined by how the run time engine treats the process execution. This is particularly relevant at the service level because publishing a process through a Web service interface makes it accessible to an unpredictable and potentially very large number of clients. As a consequence, the process developer is confronted with the difficult question of resource provisioning. Determining the optimal configuration of the distributed engine that runs the process becomes sensitive both to the actual number of clients and to the kinds of processes to be executed. The main contribution of the paper is to show how resource provisioning for software business processes can be solved using autonomic computing techniques. Tthe engine separates execution in two stages (navigation and dispatching) and uses a controller to allocate the node of a cluster of computers to each one of those stages as the workload changes. The controller can be configured with different policies that define how to reconfigure the system. To prove the feasibility of the concept, we have implemented the autonomic controller and evaluated its performance with an extensive set of experiments.

Key words:
Service Oriented Architectures, Web Service Composition, Autonomic Computing, Distributed Business Process Execution Engines

Email addresses: pautasso@inf.ethz.ch (Cesare Pautasso), heinist@inf.ethz.ch (Thomas Heinis), alonso@inf.ethz.ch (Gustavo Alonso).

1 Introduction

In the pursuit of ever higher level abstractions for developing applications, software engineering has recently adopted the notion of *Service Oriented Architectures* (SOA) as the next step towards improving support for application integration and the development of large scale enterprise software. Services follow the tradition of objects and components in terms of dealing with modularity and composition through the description of entities that have a well defined interface. However, the transition from object oriented development to component based development to service oriented architectures represents not only a shift in terms of the scale of the entities manipulated but also an increasing reliance on the underlying infrastructure (e.g., middleware) to determine the properties of the resulting software artifact. In other words, what the software does and how it behaves are less determined by the language used and the way the program is constructed than by the infrastructure where the actual program is deployed for execution. This reliance on a relatively large and complex infrastructure plays a crucial role in developing reusable software to be delivered as a service [1–3].

In Service Oriented Architectures, there is growing consensus that a good way to develop software is through business process modeling languages [4–6]. Processes provide well suited abstractions for modeling business protocols, conversation rules and the information flow linking a set of services [7–9]. Processes support a form of component based software engineering where the recursive nature of software composition plays a major role [10,11]. In this context, a Web service implemented through the reuse and composition of a set of existing Web services (using a business process) is published to be further composed into larger distributed applications (Figure 1). This simple idea has a number of implications on the run-time infrastructure used to execute such language when the composite service is published on the Web.

The first such implication is that the developer has to deal somehow with the problem of resource provisioning for the execution of the business process. Services published on the Web have the potential to be concurrently invoked by a very large number of clients with surges of requests arriving at unpredictable times [12]. Whenever a new client contacts the service, a new conversation is started and a new process instance must be created to keep track of the state of the interaction. 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 [13]. A common way to provide the necessary infrastructure to support these operations is to partition the correlation of the messages and the management of their corresponding business process instances among a distributed execution environment, for example, a cluster of computers [14].

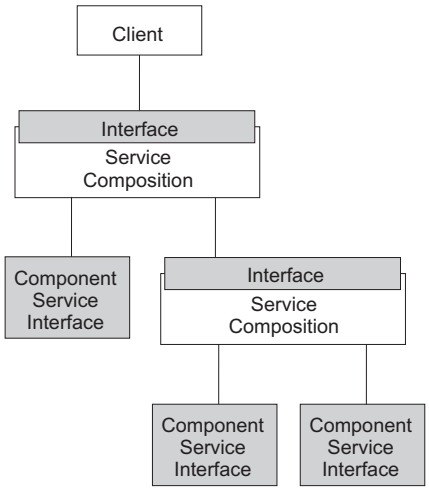


Fig. 1. Web service composition is recursive

Although such a distributed process execution environment can scale to handle large workloads, the provisioning and configuration of such cluster-based systems is difficult to determine *a priori*, especially when facing an unpredictable workload. This is an important systems management problem. Its solution requires to dynamically reconfigure the engine [15]. This way, the size and the configuration of the distributed engine can be adapted to keep the balance between servicing its workload with optimal performance and ensuring efficient resource allocation.

In this paper we address the problem of autonomic engine reconfiguration for service composition based on processes. The paper makes two main contributions. One conceptual, regarding how to organize the execution of a software business process according to a stage-based architecture. With it, the *navigation* over the structure of the process is separated from the *dispatching* of messages to perform the actual invocation of the services involved in the process. As a practical consequence, this opens up the possibility for the developer to delegate to the run-time infrastructure the decision on how many execution threads should be dedicated to each stage. Building on this result, we also show how to implement such architecture by distributing it over a cluster of computers to deal with the execution of a very large number of concurrent processes. The second contribution of this paper is the idea of applying autonomic computing techniques to automatically solve the management problem for such distributed process execution engine [16,17]. The relevance and novelty of the contribution lies in the fact that most existing systems tackle the scalability problem by statically and manually partitioning the business process across different sites (e.g., [18]). Instead, our approach uses the current workload to dynamically and autonomously determine the optimal configuration of the system at run-time.

To demonstrate the validity of these ideas, as part of the JOpera project [19],

we have designed and implemented a distributed engine for running Web service compositions that achieves scalability by replicating its key components across a cluster of computers. Additionally, the engine 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, if some reconfiguration actions have to be carried out. 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 extensive experiments show the feasibility of the approach, demonstrate that the autonomic controller can reconfigure the system automatically and compare the behavior of alternative control policies.

The paper is organized as follows: Section 2 discusses related work. Section 3 describes the basic architecture of the JOpera distributed engine, emphasizing how it can be extended with autonomic features. In Section 4 we define the requirements for autonomic resource provisioning in the context of a process execution engine and state what the assumptions regarding the applicability of our solution are. Section 5 introduces the control algorithm and its policies that drive the self-management actions of the autonomic controller, whose implementation is described in more detail in the following Section 6. Throughout the paper, we also include experimental results concerning the manual configuration of the engine (Section 3.4), the comparison of the effect of different control policies (Section 5.4) and the overall evaluation of the autonomic resource provisioning capabilities of the system (Section 6.4). We draw some conclusions in Section 7.

2 Related Work

Distributed execution of process-based Web service compositions is an important area of research (e.g., [20,6,18,21,14,22]). However, to the best of our knowledge, very little research has been published towards applying autonomic computing principles to solve the management problem opened up by the distributed design of such engines.

Distributed engines are typically motivated by the need to support business processes across companies without having to use a centralized entity [23]. This type of decentralization introduces several problems on its own such as the lack of a global view over the process [24]. It neither addresses the scalability and reliability problems per se since the problem is simply translated to each node that executes parts of the process. Moreover, given the design of a distributed engine, the management problem of how to configure it in order to guarantee that clients are serviced with a satisfactory level of performance under different workload conditions is still poorly understood.

This problem has been addressed by introducing tools (e.g., GOLIAT [25]) that use the expected characteristics of the workload to make predictions about the performance of a certain configuration of the engine. At deployment time, this kind of tools help system administrators to determine interactively on how many resources the engine should be distributed in order to achieve the desired level of performance. As a natural extension of this approach, autonomic computing [26–28] techniques can be used to replace such manual (and static) configuration steps. In this paper we argue that an autonomic controller should be applied to determine the configuration of the distributed engine automatically, taking into account measurements of the system's performance under the actual (and unpredictable) workload.

The problem of adaptively replicating functionality to achieve higher throughput has also been identified in the database community (e.g. [29–31]): the challenge consists of replicating specific functional components depending on the workload, but doing so only when this leads to performance improvements. As we are going to show, the concept of *staged architecture*, applied to databases in [32], has also influenced the design of the distributed JOpera engine. With the results presented in this paper we confirm that a staged architecture can be suitable for building a self-managing system.

Distributing the execution of a workload over a cluster of computers to achieve scalability inevitably overlaps with the problem of load sharing, load balancing, resource management and scheduling [33,34]. As we are going to show in Section 5.3, as part of the algorithm carried out by the autonomic controller, several results from the resource management literature can be applied to optimally choose which resource should be allocated to the autonomic engine. Furthermore, it is important that an architecture which undergoes automatic reconfigurations supports the rebalancing of the workload in the newly configured system. While it is a necessary condition to have such kind of adaptability in the processing of the workload, we believe that an autonomic system goes beyond that. Whereas a dynamic 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 service the workload better.

3 Background

In this section we first present how business process modeling languages satisfy the requirements of recursive Web service composition. This requirement also affects the architecture of the corresponding run-time infrastructure, which should be capable of publishing processes as Web services and, as a consequence, should feature good scalability when such Web services are invoked by a large number of clients. The design presented in this section has been

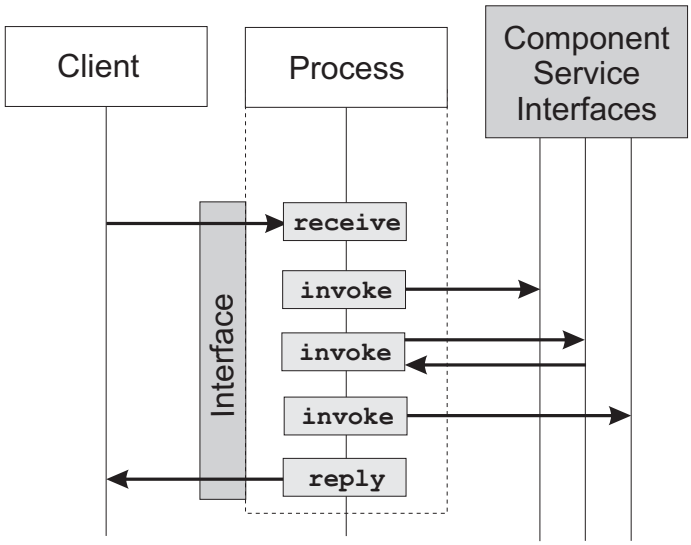


Fig. 2. Publishing a process as a Web service

implemented in the JOpera distributed engine [22]. A brief evaluation of its scalability is included at the end of this section.

3.1 Recursive Web Service Composition with Business Process Models

Business processes model the interactions between different tasks by defining their data flow and control flow dependencies. The data flow defines data exchanges between tasks, whereas the control flow constrains the order of task execution [35]. In the case of processes applied to Web service composition, the tasks of a process are typically bound to service invocations, i.e., they represent the exchange of messages between the process and a remote service.

Given the existing literature on the subject (e.g., see [7–9,36,37] as starting point) and the current standardization efforts (e.g., WS-BPEL [38]) in this paper we do not further elaborate on details of such process-driven composition languages. Instead, we focus on how they satisfy the requirement of recursive composition. Processes interact with external services in two ways (Figure 2). They play an active role with tasks sending messages to a component service and also a passive role with tasks receiving and replying to messages sent by clients.

In the active role, the process orchestrates the invocation of one or more external services. As part of the execution of each task, a message is sent from the process to the service and – in case of synchronous invocation – the task will block until the corresponding response message is received.

To publish a process as a reusable Web service involves defining a mapping

between the process and a Web service interface [39]. This way, the process becomes accessible to clients that do not necessarily have to know that they are dealing with a process, or, in general, a composite service. Clients simply invoke the operations provided by the service interface and their messages will be routed to the appropriate process instance.

In the simplest of such mappings, such an interface provides only one operation. This is used by clients to initiate the execution of a new process instance with a request message. Once the process completes its execution, a response is returned to the client. A more complex and flexible mapping is prescribed in the WS-BPEL standard, where messages sent to a process are received by a specific task that can trigger the execution of a new process instance (*instantiating receive*) or simply deliver the content of the message to an existing process instance which is waiting for it.

3.2 Running Web Service Compositions

The execution of processes is controlled through the engine's API, so that processes can be started by clients that send messages to the engine, by users – through their service composition tools – as well as from other processes. As shown in Figure 3, the API of the engine queues such requests into the *process space*. These requests are consumed by the *navigator*, which 1) creates a new process instance and 2) begins with the actual execution of the process. To do so, the navigator uses the current state of the execution of a process to determine which tasks should be invoked, based on the control and data flow dependencies that are triggered by the completion of the previous tasks. A navigator can execute multiple process instances as their state is stored separately and will be retrieved to navigate each process found in the process space. Once the navigator determines that a certain task is ready to be invoked, the corresponding requests are stored in the *task space*.

The invocation of the tasks is managed by the *dispatcher* component. The name of this component is derived from its function of executing tasks by dispatching messages to and from different kinds of service providers through the corresponding adapters. These include, e.g., worklist handlers for tasks that should be carried out by human operators, but also adapters to invoke standard compliant Web services, as well as many other kinds of services [40]. In this design, each dispatcher supports the execution of 64 tasks in parallel. Still, this is not a scalability limitation, because multiple dispatchers can be run in parallel, e.g., using a pool of threads, as we are going to present in the next section. After the execution of the task has been finished, the dispatcher notifies the navigator through the *process space*. More precisely, the dispatcher packages the results of the invocation into a task completion tuple, which is

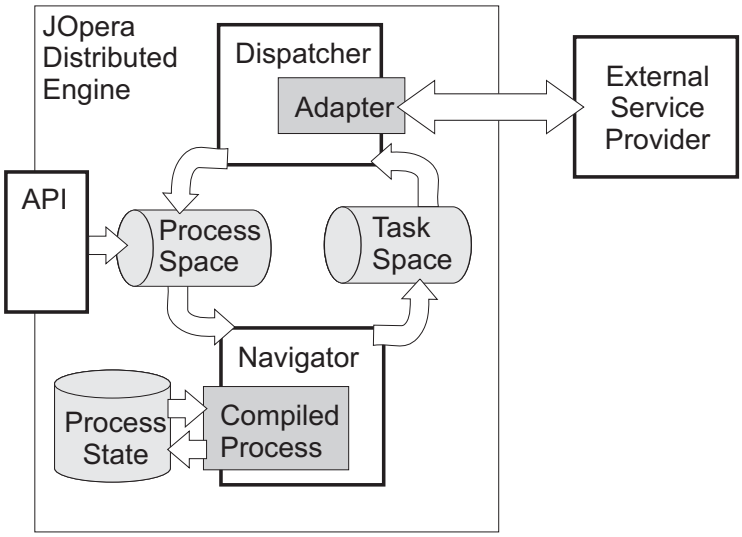


Fig. 3. Logical architecture of the JOpera distributed composition engine

put into the process space. Such tuples are then consumed by the navigator in order to update the state of the execution of the corresponding process and to carry on with its execution.

The main reason for separating the navigation over the process from the dispatching of its tasks lies in the observation that these operations have a different granularity. It is to be expected that the execution of a task performed by the dispatcher may last significantly longer than the time taken by the navigator for scheduling it [22]. With our approach, a slow task does not affect the execution of other concurrently running processes because these two operations are handled by different threads. Likewise, the engine supports the parallel invocation of multiple tasks belonging to the same process. The asynchronous interaction between process execution and task invocation is an important departure from the design of existing engines where navigation and dispatching are serially executed by a single thread.

3.3 Distributed Execution with Tuple Spaces

Decoupling process navigation from task invocation enables the system to scale along two orthogonal directions. In case a large task invocation capacity is required, the dispatcher thread can be replicated to manage the concurrent invocation of multiple tasks. Likewise, if many processes have to be executed concurrently, the navigator can also be replicated.

By using tuple spaces, the navigator and dispatcher threads are loosely coupled into two distributed pools. This makes it possible to scale the system to run on a cluster of computers, as navigators and dispatchers can be physically

located on different nodes.

The flexibility provided by tuple spaces makes it also feasible to dynamically reconfigure the system, as the number of navigators and dispatchers can be increased or decreased without having to interrupt the entire system. The system offers a reconfiguration API to control which thread is running on each node of the cluster. Tuple spaces also offer a convenient mechanism for instrumenting the system in order to gather performance information that can be fed back into the self-tuning algorithm of the autonomic controller.

3.4 Performance Evaluation of a Statically Configured Engine

As a motivation for introducing the autonomic features of the distributed engine presented in the rest of this paper, in this section we evaluate the performance of the engine running over a cluster of computers with a static configuration. Our goal is to show that the optimal configuration of the distributed engine (in terms of the number of navigator and dispatcher threads that are used) is highly sensitive to its workload. Thus, it is important to be able to dynamically change the configuration of the engine in order to optimally service workloads with different characteristics.

3.4.1 Experimental setup

Since there are not yet standardized benchmarks for autonomic process execution engines, we have defined a simple workload to evaluate the system under extreme conditions. The workload imposed on the system is 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 the experiments included in this paper we have focused on a homogeneous workload, consisting of processes composed by 10 parallel tasks with fixed (and controllable) invocation time. This simplifies the analysis of the results of our experiments. We plan to continue evaluating the system with heterogeneous and continuous workloads as part of future work.

For all of the experiments presented in this paper, JOpera has been deployed on a 32 node cluster. 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.

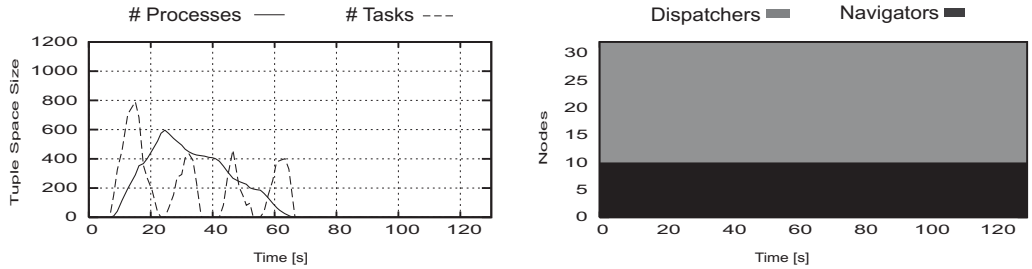
The traces of the engine's configuration (number of dispatchers and navigators) and performance indicators (size of the process and task queue) have

been sampled at regular intervals of 1 second so that it is possible to closely follow the dynamics of the system.

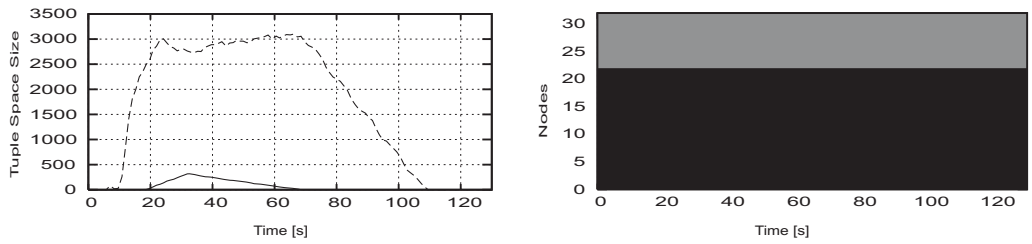
3.4.2 Static Resource Allocation

The execution traces shown in Figure 4 demonstrate that the engine's behavior depends on how the 32 nodes of the cluster are allocated between navigators and dispatchers. In Figure 4a, the workload (800 processes of 10 parallel tasks lasting 8 seconds) runs in 68 seconds using 10 navigators and 22 dispatchers. However, with the symmetric configuration (22n, 10d, Figure 4b) the execution time significantly increases by 42.9 seconds (or 60.2%). This discrepancy can be explained by observing how the size of the process and task spaces evolves over time (Figure 4 left). In Figure 4a, the task execution capacity is balanced with the amount of resources allocated to process execution. In other words, 22 dispatchers can keep up with the execution of the tasks that are produced by 10 navigators that are handling the workload peak. This is not the case for Figure 4b. 22 navigators quickly consume the processes queued in the process space but overwhelm the 10 dispatchers with task tuples. With this configuration, the task spaces reaches a plateau of 3000 tuples (as opposed to the first configuration, where it oscillates up to 800 tuples).

These simple results already give an intuition of the difficulty of manually configuring the distributed engine. A misconfiguration can lead to longer process execution times and suboptimal resource allocation.



a) Static Configuration with 10 Navigators, 22 Dispatchers (Cluster of 32 nodes)



b) Static Configuration with 22 Navigators, 10 Dispatchers (Cluster of 32 nodes)

Fig. 4. Execution traces of the size of the process and task spaces (left) for a workload of 800 processes with two different static configurations (right).

3.4.3 Finding the Optimal Configuration

In order to find the configuration which minimizes the response time of the system for a given workload, we have carried out an exhaustive search of the configuration space. In this experiment, we used a cluster of 15 nodes and all possible configurations starting with 14 navigators and 1 dispatcher up to 14 dispatchers and 1 navigator were tested. For each configuration, Figure 5 depicts the total execution time and the speedup of two different workloads: 1000 concurrent processes containing 10 parallel tasks of the duration of 0 seconds (workload 0) and 1000 processes containing 10 parallel tasks of the duration of 20 seconds (workload 20). Similar results can be obtained with processes having different control flow structures.

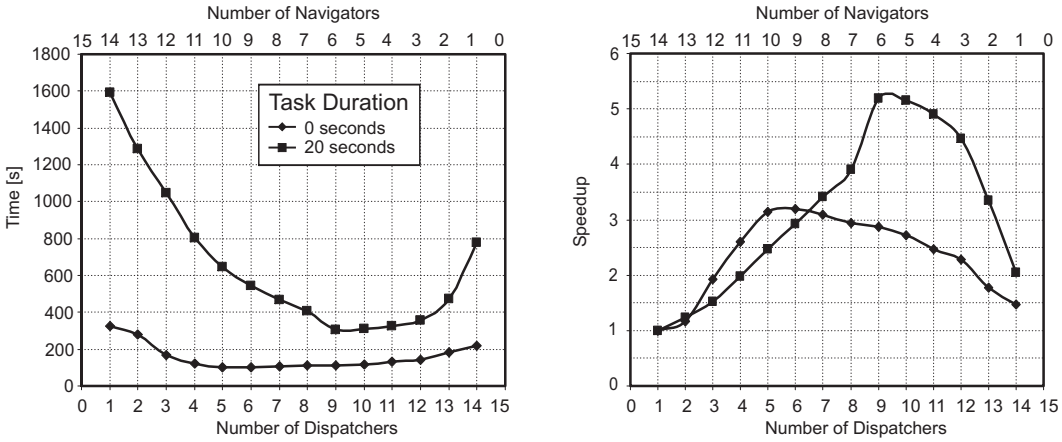


Fig. 5. Time required to execute two different workloads of 1000 processes using all possible static configurations (left) and speedup achieved relative to the slowest configuration (right)

The speedup profiles shown on the right side of Figure 5 clearly illustrate that the optimal configuration for the two workloads is not the same. For workload 20 the optimal configuration is the one using 9 dispatchers and 6 navigators while for workload 0 the best configuration is the one using 5 dispatchers and 10 navigators. On the one hand, in the worst case the penalty of a misconfigured system is a factor of 5 in performance. On the other hand, if the system is optimally configured to handle one workload, its performance will suffer when it is subjected to a different one.

Configuring the system statically therefore has two main problems. First, static configuration potentially leads to inefficient resource allocation, since the engine could release part of the cluster after processing a surge of requests. Second, a given configuration may not be optimal to deal with all kinds of workloads, hence reconfiguration is still required. In practice, such reconfiguration is quite difficult to perform manually. With the approach based on autonomic computing techniques presented in the rest of the paper, we show how it can be done automatically.

4 Requirements for an Autonomic Engine

After introducing the problem of resource provisioning in a distributed engine for executing software business processes, in this section we describe the requirements and the assumptions to satisfy in order for the engine to be considered "autonomic".

4.1 Self-Management Capabilities

An autonomic system must feature self-configuration, self-tuning and self-healing capabilities [41].

Self-configuration entails changing the system's configuration on the fly without manual intervention and without disrupting normal system operation. This requires the engine to provide mechanisms to expose the current state of its configuration as well as to support means to dynamically and efficiently change the configuration.

The *self-tuning* capabilities should ensure that such changes lead to a configuration which is close to the optimal, given the current workload. In order to provide self-tuning capabilities, the composition engine must give access to its internal state, such that control algorithms can analyze current and past performance information in order to plan configuration changes. Our assumption is that the characteristics of the workload affect the system's performance and that the self-tuning algorithm can optimally adapt the system to the workload by monitoring key performance indicators.

Finally, the system also needs to provide *self-healing* capabilities [42]. This means that it should be able to detect configuration changes due to external events, such as failures of nodes. If a discrepancy between the model of the configuration and the actual configuration is detected, the self-healing functionality should perform the necessary recovery actions. From this, we identify the requirement to support mechanisms for detecting failures and configuration changes of the cluster and to query the composition execution state in order to determine how the running processes have been affected.

4.2 Architectural Constraints

To describe how to address the above requirements, we begin by defining the relationship between the autonomic controller and the rest of the engine (Figure 6). The autonomic controller component encapsulates all control decisions

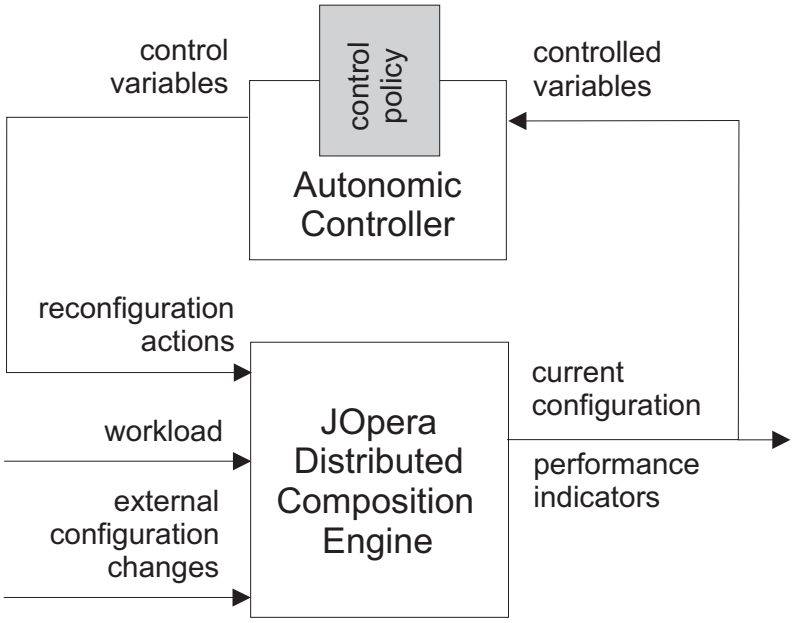


Fig. 6. Relationship between the Autonomic Controller and the rest of the engine

affecting the configuration of the system that are automatically carried out in response to changes in the workload conditions. A clear understanding of this relationship is very important, as the interface between the controller and the controlled system affects both the performance and the autonomic capabilities of the whole system [43].

The autonomic controller acts as a closed feedback-loop controller [44]. It periodically monitors the state of the engine and, based on different policies, it decides to apply control actions to adjust the configuration of the system. More precisely, the controller periodically reads the values of *controlled variables*, which include both the current system configuration and a set of performance indicators.These provide information about the current level of performance of the system (e.g., in terms of throughput and response time).

The system also contains and provides access to a model of its current configuration. This information is mainly used by the controller to plan reconfiguration actions, as it describes the available resources and their current utilization state.

After gathering this information, the controller runs the self-tuning algorithm to determine whether the current configuration needs some adjustment. This algorithm can be configured to use different optimization policies, which determine the goals that drive the controller to determine the reconfiguration actions. In order to implement its decisions (self-configuration), the controller acts upon a set of *control variables*. Changes in the control variables will result in reconfiguration actions applied to the engine.

In addition to the information flowing between the controller and the engine, Figure 6 also shows that the engine is influenced by two different external factors. On the one hand, a certain *workload* can be applied to it, i.e., whenever clients send messages to the processes run by the engine. On the other hand, the configuration may change independently of the will of the controller, e.g., if a failure occurs. The controller detects these conditions indirectly as they affect the values of the controlled variables (self-healing).

4.3 Workload Assumptions

The workload consists of a collection of concurrent process instances which represent running Web Service compositions. In general, users may define an arbitrary composition and initiate its execution at any time. In our evaluation, we focus on a worst case scenario where a large number of processes is submitted for execution simultaneously. However, neither the size nor the structure of the composition is taken into account when designing the self-tuning algorithm, which should be able to deal with any process that can be normally executed by the engine. Furthermore, in this paper we do not deal with workload prediction issues [45]. The autonomic engine observes the current load and reacts to it. A pro-active system would try to anticipate workload changes before they occur [46]. Since JOpera collects a detailed history of past executions, a data mining algorithm could analyze it and use it to predict future composition arrival times. We will pursue that option as part of future work.

5 Control Algorithm and Policies

In this section we present the control algorithm of the autonomic controller and discuss several control policies that can be employed to adjust its behaviour. At the end of the section we study the effect of different policies by presenting the results of an experimental comparison.

The control algorithm can be seen as an infinite loop along the following three stages (Figure 7). We will discuss a concrete implementation in the following section.

(1) 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 and what the strategy is for keeping it up-to-date (i.e., how often it should be sampled).
(2) During the planning phase, the controller uses the collected information to determine whether the system is balanced or whether a configuration

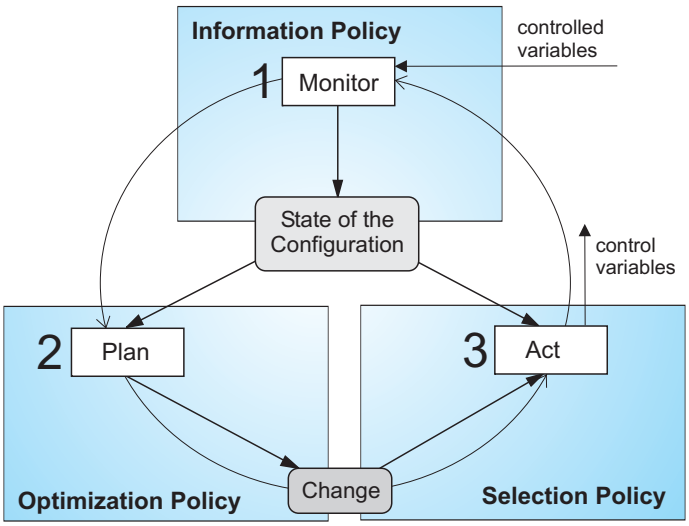


Fig. 7. Control algorithm executed by the autonomic controller

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).

(3) 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. For example, if a certain number of cluster nodes must be released, the selection policy chooses which ones are the most likely candidates according to a specific criteria (e.g., they are the slowest ones which are currently idle).

5.1 Information Policy

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

5.1.1 Modeling the system configuration

As we have previously discussed, one of the requirements for autonomic architectures consists of managing a model of the system configuration. This opens up different issues concerning the model itself, i.e., what is the minimal information to be captured in order to describe the configuration of the system, as well as where this information should be stored and how often it should be updated.

We begin by introducing a simple model of the system configuration, which describes the cluster of computers to be used (listing the IP addresses of each node) and the set of the threads which are currently running on each of the nodes. For each kind of thread (dispatcher or navigator), at most one can be running on the same node. This simple model can be augmented with information describing the characteristics of each node (e.g., its speed, its current load or the amount of available memory) which can then be used as a basis for a more sophisticated node selection policy. Additional statistics can be collected for each thread, allowing – for example – to detect whether a thread is idle or not. More precisely, each navigator reports the number of processes that is currently running and each dispatcher counts the number of tasks that have been assigned to it.

Depending on the information to be collected for the specific policy, it is worth noting that it may be expensive to perform some kinds of measurements as well as to ensure that all information is kept up-to-date. Thus, we identify an important trade-off between the cost of maintaining fresh and detailed configuration information and the potential benefits that may derive from it in terms of more accurate control decisions.

In our approach, we employ a mix of push and pull strategies for transferring and storing the configuration information. The list of cluster nodes assigned to the engine is stored into the configuration space, so that it is easy to access and to modify. The navigator and dispatcher threads register themselves with this registry as soon as they are started or stopped on a particular node. However, the rest of the state information describing, e.g., the load of each node or whether a thread is idle, is measured and stored locally on each node. During the execution of the control algorithm, the autonomic controller collects such information by directly querying each node as specified by the information policy in use. This way, it is possible to reduce network traffic as the information is only transferred at the time it is needed by the controller.

5.1.2 Performance Indicators

An indication of the performance of the system can be gathered by reading the process execution logs which contain the starting and finishing time of each process. From this information, it is possible to compute the *average process turnaround time*, i.e., the wall-clock time required by the system in order to execute a certain type of process. This value is influenced both by the invocation time of the services composing the process as well as by the overhead imposed by the system. If the system is overloaded, processes take longer to run.

By considering the architecture of the distributed composition engine (Figure

3), there are several points that can be instrumented to provide performance indicators. For example, since the navigator and dispatcher threads communicate asynchronously through tuple spaces, it is possible to sample the current *space size* in order to detect whether the system is balanced. In case the size of the space grows, it is likely that there are not enough consumers processing its tuples and too many producers of tuples. Conversely, if the size of a space decreases, there may be too many consumers (or too few producers). The information policy therefore defines that the variation in the size of the tuple spaces of tasks and processes should be monitored to detect imbalances in the system's configuration.

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 utilization logs are kept as part of the configuration information. They also form the basis for more advanced information policies, which take into account estimates about the economic cost of using one more node of a cluster, compared to the potential performance boost [47].

5.2 Optimization Policy

The optimization policy specifies how to achieve certain goals in terms of mapping a combination of the previously defined controlled variables 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 policy that configures the system to always provide excess capacity so that unpredictable peaks in the workload can be absorbed promptly. Although this approach maximizes the performance of the system measured in terms 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 utilization.

5.2.1 Simple Optimization Policy

The simplest optimization policy we have considered uses a single threshold T compared to a certain non-negative controlled variable v. Whenever the sampled value of the variable is higher than the threshold ($v > T$) 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 extending the system. If $v = 0$, the outcome is to shrink

the size of the system by one thread. This action follows from the second goal, whereby the resource allocation is reduced if the controlled variable indicates a reduction in the workload. No reconfiguration action is planned if $0 < v \leq T$. The symmetry in the actions that can be taken (start one, stop one, or do nothing) ensures that the system can reach any configuration. More concretely, we have applied this *simple* control policy by binding the controlled variable v to the size q of the space consumed by the navigators and the dispatchers and by introducing different thresholds (T_d, T_n) for each kind of thread. Thus, the same policy can be used to control both navigator and dispatcher threads, as long as their characteristics are taken into account. To tune their values, the thresholds can be interpreted as the number of tuples that is expected to be handled by each kind of thread. Typically $T_n > T_d$, as navigators can handle a larger volume of tuples than dispatchers.

This policy can be made more complex along several directions [48]. On the one hand, it is possible to increase the set of reconfiguration actions. On the other hand, different controlled variables can be used. We have experimented with both of these possibilities by extending the previously described simple control policy.

5.2.2 Differential Optimization Policy

As opposed to reading the current size of the process space, the *differential* control policy uses the first order variation ($\Delta q = q(t) - q(t-1)$) of the space size to make its decisions. Still, the possible outcomes and the decision policy are the same as in the simple threshold policy. We introduced this policy because the size of the process spaces 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 $\Delta q > 0$ the space is accumulating tuples) or whether 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 ($\Delta q < T_{stop} < 0$).

5.2.3 Proportional Optimization 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 to 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.

5.3 Selection Policy

Once the optimization policy has determined the new configuration of the system, the selection policy compares the new configuration with the current one in order to establish what nodes should be affected by the planned configuration change.

The goal of this policy is to define how to map abstract reconfiguration decisions to concrete actions affecting the current system configuration. Given a planned reconfiguration action (e.g., stop one thread) and a pool of candidate nodes, one node should be chosen so that the action can be applied to it (e.g., the thread running on the node is stopped).

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. Similarly, when the outcome of the optimization policy consists of a decision to shrink the size of the system, it may not always be possible to do so. For example, the selection policy may not yet find idle threads that can actually be stopped.

Deriving a concrete configuration to be fed into the self-configuration component from an abstract configuration plan is done by prioritizing nodes according to different criteria. In general, these criteria quantify how well a node is suited for a certain configuration change. For example, if there are idle nodes available and threads need to be started, the idle ones get the highest priority and are selected for the configuration change.

The simplest selection policy is non-deterministic: candidate nodes are chosen randomly. This reduces the information that is associated with each node and works quite well assuming that the available resources are homogeneous. Otherwise, the nodes should be ranked according to some criteria (e.g., their speed, their current load or the amount of free memory) so that the controller is able to use a more sophisticated selection policy.

If all threads are busy and there are no more idle nodes, some need to be selected in order to apply the configuration change. For example, if an additional navigator thread needs to be started, a dispatcher thread will have to be stopped and vice versa. Stopping non-idle threads may be expensive and the selection policy therefore needs to take this reconfiguration cost into account when deciding which thread should be stopped. Again, the simplest selection

policy chooses the threads randomly, regardless of the resulting rescheduling overhead. We also experimented with a smart selection policy that chooses threads with the goal of minimizing the overhead caused by rescheduled tasks and migrated processes. In this case, threads are further prioritized by the number of tasks (or processes) that they are currently executing. With this heuristic, threads which are running many processes (or many tasks) are less likely to be interrupted, thus less work will have to be migrated to a different node.

5.4 Comparison of the optimization policies

In this section we compare the three optimization policies with the baseline of the two static configurations presented in Section 3.4. The setup described in Section 3.4.1 also holds for these experiments.

5.4.1 Results

We have measured both the total execution time as well as the average resource allocation for three different workload sizes: 400, 800, 1600 processes (Figure 8).

The total execution time has been measured as the time between the arrival of the first client request in the process space and the time the execution of the last process has been completed. The average resource allocation has been measured as the sum of the time each of the nodes has been running a thread, divided by the total execution time.

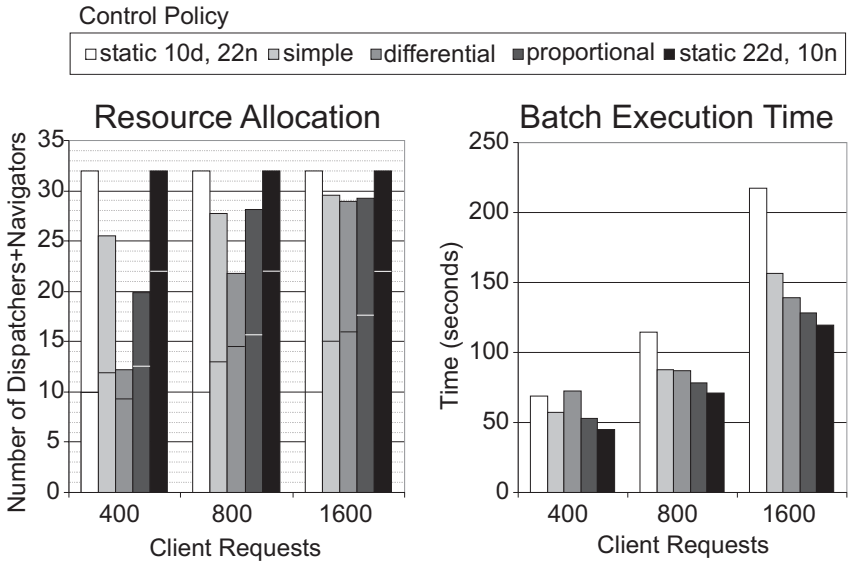


Fig. 8. Comparison of the policies: resource allocation and process execution time

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 workload is between 50% (workload size 400) and 82% (workload size 1600) larger. This difference implies that the static configuration using 22 dispatchers and 10 navigators 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 22d, 10n* 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).

The effect of the policies on the system configuration can also be compared by observing Figure 9. This graph traces the evolution of the configuration in terms of the number of nodes allocated to run navigators and dispatchers. For each policy, the traces begin with an "empty" system, where neither dispatchers nor navigators are running. As soon as the workload arrives, the controller allocates nodes to the engine's components. Once the peak has been processed, the engine shrinks back to the original configuration. Whereas the simple policy allocates an equal number of nodes to each component until it reaches saturation, the differential and proportional policies tend towards a configuration closer to the *static 10n, 22d* configuration. As shown earlier in Figure 4a, this configuration balances the number of dispatchers and navigators to provide good performance (Figure 8). This representation also shows an advantage of using the proportional policy, which modifies the configuration in leaps of several nodes, and thus can react faster than the other policies which can only grow or shrink the system's configuration by one node at a time.

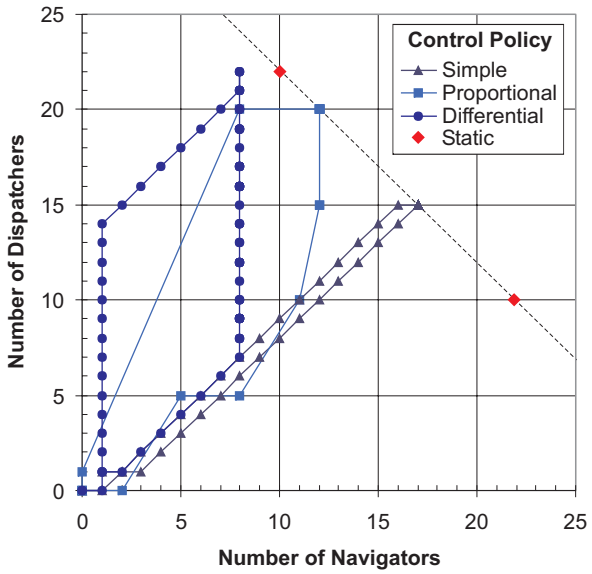


Fig. 9. Configurations reached by different control policies

5.4.2 Discussion

Using the simple strategy of monitoring the size of process and task space 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. In order to control this, 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 (i.e., we observed that a navigator can handle 5 times more tuples than a dispatcher).

In general, setting these thresholds appropriately tends to be difficult and misconfiguring them may also result in a performance penalty [49]. The problem is partly related to the low level of abstraction of these policies which need to understand the internal behavior of the engine in order to properly configure them. It would be desirable to specify goals in more abstract terms [50] (e.g., in terms of the required level of performance of the engine) and let the controller take care of mapping these to the appropriate optimization policies. As part of future work we plan to explore this issue in more detail.

6 Autonomic Controller

After describing the control algorithm and comparing the performance of its different policies, in this section we present the most important design decisions regarding the architecture of the autonomic controller.

Although it would be possible to implement the algorithm described in Section 5 directly, where the main control loop (measure, plan and act) is followed sequentially by a single component, the dynamics of the distributed composition engine make it impractical to do so. Instead, the various stages of the algorithm have been implemented by different components (self-healing, self-tuning and self-configuration), which interact asynchronously (Figure 10). As we are going to discuss, decoupling the gathering of new information about the state of the engine from the planning and reconfiguration ensures that the controller does not block when the interaction with the rest of the engine becomes problematic (e.g., when one of the nodes of the cluster does not respond due to failure or overload).

In addition to the autonomic controller we have also developed a visual monitoring tool by extending the Eclipse TPTP platform [51]. As it can be seen in

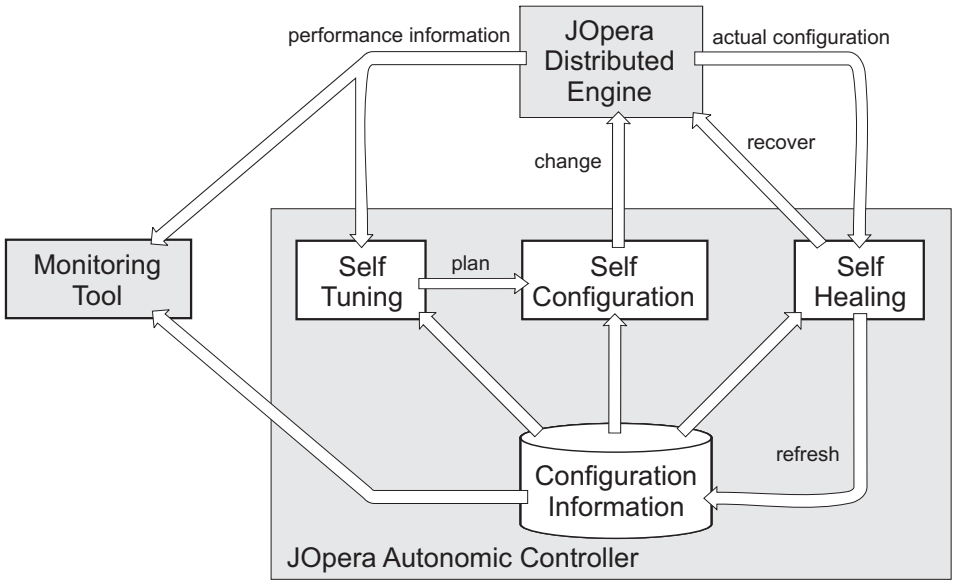


Fig. 10. Internal Architecture of the Autonomic Controller

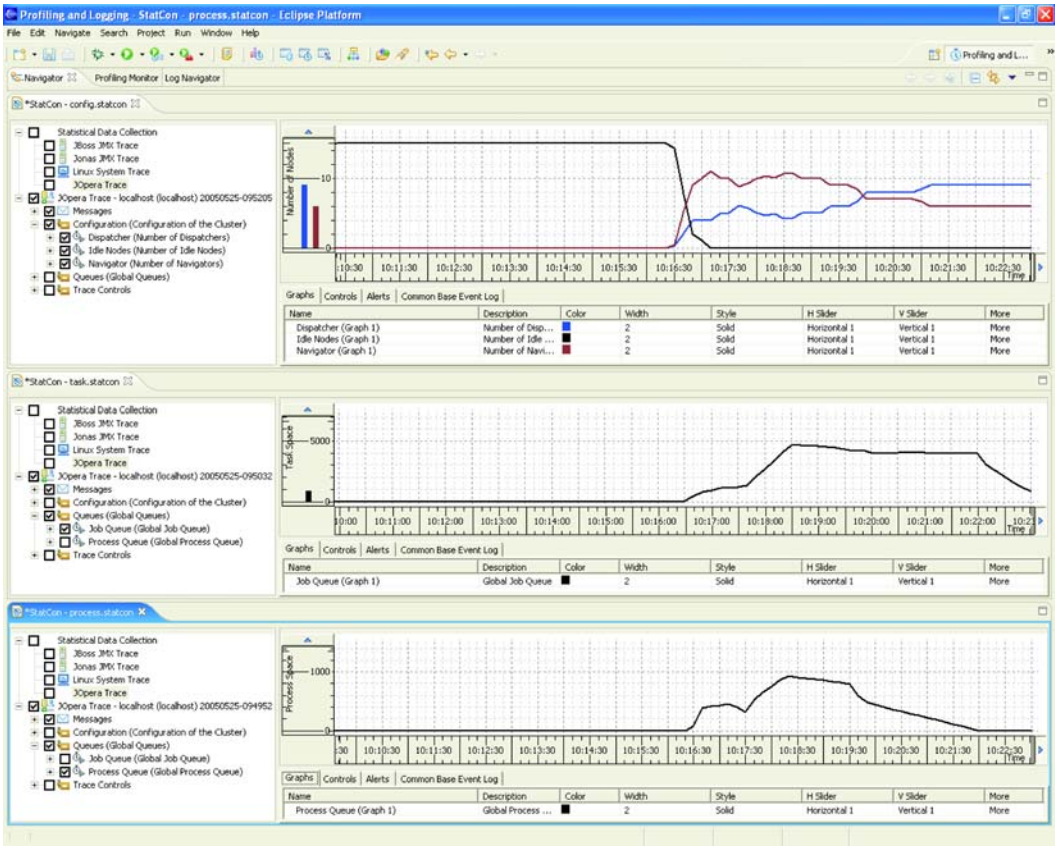


Fig. 11. Screenshot of the monitoring tool

Figure 11, the monitoring tool displays the history of the configuration of the distributed engine and the current values of the performance indicators which are updated in real-time.

6.1 Self-Healing

This component is in charge of updating the model of the system's configuration with respect to the "real" system. The primary task of the self-healing component is to ensure that the composition engine remains in a consistent state in spite of external events affecting its configuration. To do so, the component periodically monitors the nodes of the cluster, checks their availability and compares their state with the information stored in the configuration space. In addition to this pull strategy, we also keep the configuration information up-to-date by having the newly started threads register with the configuration state autonomously.

A failure is detected as a mismatch between the known configuration and the actual configuration of the cluster. If a failure occurs, the component ensures that the affected processes and tasks are correctly recovered by the rest of the composition engine.

6.2 Self-Tuning

The self-tuning component needs to determine whether the current system configuration is optimal. This evaluation is carried out whenever a change in the configuration is reported by the self-healing component as well as when the performance of the engine changes significantly. This information is evaluated according to one of the previously described optimization policies, which defines how to achieve an optimal configuration of the cluster based on the current performance.

In case an imbalance is detected and a change of configuration is needed, the self-tuning component submits a reconfiguration plan to the self-configuration component so that the changes can be carried out asynchronously. Given that producing a new plan may take a significantly shorter time from implementing it, we assign these concerns to two separate components. This way, it is not necessary to wait for a reconfiguration to take place in its entirety before the self-tuning component can once more evaluate the current configuration.

6.3 Self-Configuration

As outlined in the previous section, the self-tuning component suggests a new, optimal configuration for the cluster. It is up to the self-configuration component to execute the actual reconfiguration of the cluster. For this purpose the self-configuration component captures the current configuration of the clus-

ter and applies changes to it. Implementing the new configuration, however, requires time (in the order of seconds) and the result may not be available immediately.

In order to execute the reconfiguration plan, the self-configuration component takes as input the suggested configuration of the self-tuning component as well as the current configuration, as it is reported by the self-healing component. As threads are being stopped (or started) on remote nodes, this component periodically checks the progress of these reconfiguration actions and ensures that the new configuration is reached. If, in the meantime, the self-tuning component has suggested another reconfiguration plan, the execution of the current one will be interrupted.

6.4 Evaluation: Self-Healing and Self-Configuration

The goal of this final evaluation is to show the autonomic process execution engine in action, where its configuration is automatically adapted to workloads having different characteristics (self-configuration) and also to unexpected changes in the execution environment (self-healing).

This experiment demonstrates the self-configuration and self-healing capabilities of the autonomic engine. The engine is subjected to repeated peaks of workload (500 processes started every 100 seconds) and its autonomic controller reacts accordingly by allocating 15 nodes of the cluster (initially idle) to 6 navigators and 9 dispatchers (Figure 12c). The engine is also subjected to external configuration changes, where the number of nodes of the cluster grows and shrinks over time. Once 5 additional nodes become available (t=100), the controller immediately makes use of this additional resources to process the second peak of workload.

The removal of nodes from the cluster (t=140, t=230) demonstrates the self-healing capabilities of the engine. Not only is the execution of the tasks and processes recovered (Figure 12d), but the number of navigators and dispatchers is adjusted to make optimal use of the remaining resources.

This experiment reflects a common situation in the lifetime of a cluster-based system, where nodes are rotated as some of them may have to be taken off-line for maintenance. With traditional systems, such intervention would require to manually determine which parts of the engine would be affected by the reconfiguration and manually stop the components running on the nodes to be replaced. The autonomic controller was able to immediately detect the newly assigned nodes and could also recover and reconfigure the engine when some of the nodes were taken off-line.

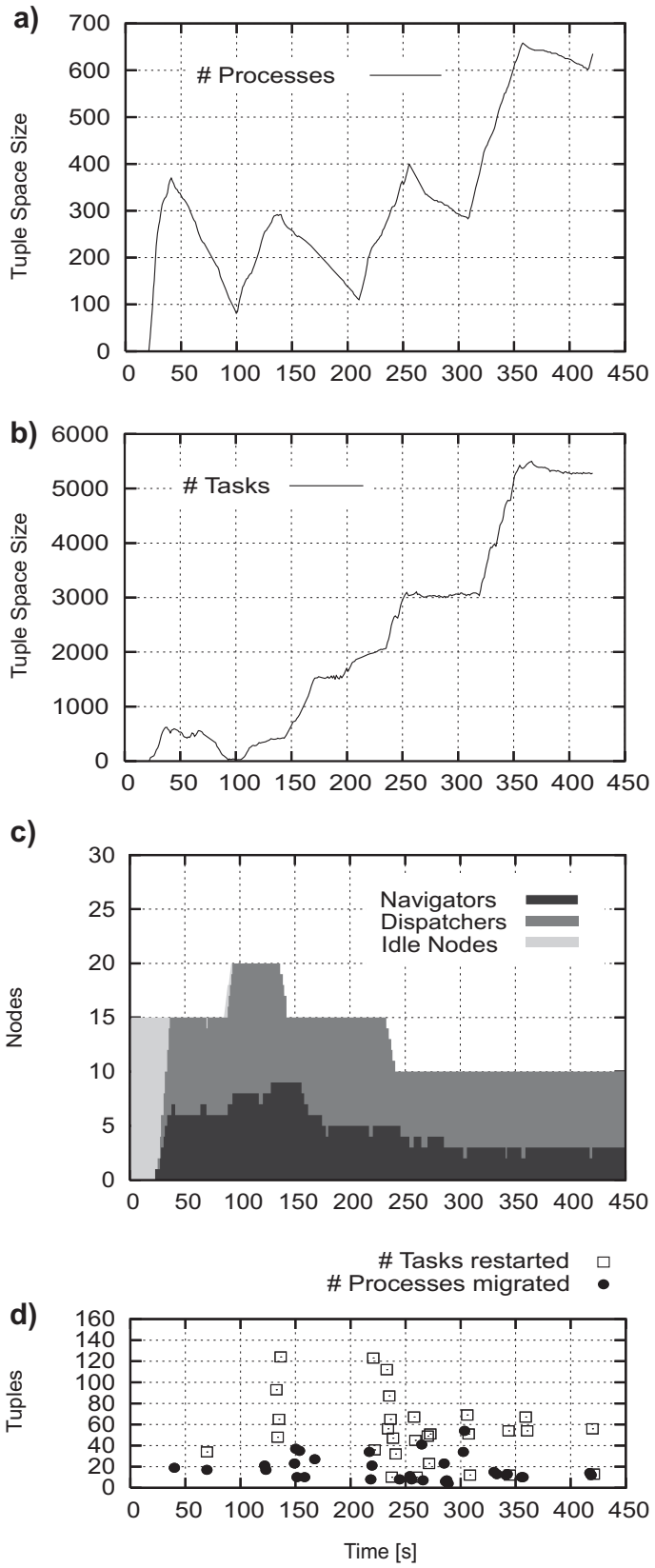


Fig. 12. Self-Configuration and Self-Healing of the autonomic engine as nodes are added and removed from the cluster

.

7 Conclusion

Service Oriented Architectures rely heavily on the underlying infrastructure to determine the actual properties of software developed in the form of a business process. In existing systems, this imposes additional constraints to the programmer who is the one made responsible for the scalability of the process and the necessary provisioning of resources. For instance, distribution is used in many composition engines to improve scalability and reliability. However, very little attention has been paid to the need for properly configuring such systems. This, in practice, remains a difficult, error-prone, manual, and time-consuming operation, especially when deploying the system to face an unpredictable workload.

To address this problem, in this paper we have presented the design of an autonomic process execution engine that can deal with resource provisioning on its own. The provisioning is also not just done statically but dynamically adjusted as conditions change. The engine runs on a cluster of computers and can automatically reconfigure itself to assign more or less computers to different tasks associated with process execution. This is done taking into account internal system failures and external modifications to the system configuration. Automatic self-configuration and self-healing of the engine greatly reduces the administrative overhead of manually monitoring and reconfiguring the system. In addition, we described the control algorithm and policies followed by the autonomic controller, the key component implementing the autonomic reconfiguration capabilities. As the performance evaluation indicates, the controller outperforms the manual, static configuration by achieving a good trade-off between two different goals: minimizing resource allocation while guaranteeing good performance. Furthermore, we have shown that the performance of the controller depends on the actual information, optimization and selection policies that are used.

These results show the feasibility of applying autonomic computing principles in order to automatically adapt the configuration of a distributed process execution engine in response to unexpected variations in its workload. An important additional contribution of the paper is to show that such autonomic engine can only be implemented by a careful analysis of the way processes are executed and the operations involved. In the case of the solution we propose, the key idea is the separation of navigation (determining what to invoke) and dispatching (actual service invocation) into two separate stages that can be distributed across a cluster of computers.

Acknowledgements

Part of this work is funded by the European IST-FP6-004559 project SODIUM (Service Oriented Development In a Unified fraMework) and the European IST-FP6-15964 project AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

[1] F. Leymann, Web services: Distributed Applications without Limits, in: Proc. of the International Conference on Business Process Management (BPM 2003), Eindhoven, The Netherlands, 2003, pp. 123–145.

[2] L.-J. Zhang, M. Jeckle, The Next Big Thing: Web services Collaboration, in: Proc. of the International Conference on Web services (ICWS-Europe 2003), Erfurt, Germany, 2003, pp. 1–10.

[3] J. Hündling, M. Weske, Web Services: Foundation and Composition, Electronic Markets 13 (2).

[4] F. Leymann, D. Roller, M.-T. Schmidt, Web services and business process management, IBM Systems Journal 41 (2) (2002) 198–211.

[5] B. Benatallah, M. Dumas, Q. Z. Sheng, A. H. H. Ngu, Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web services, in: Proc. of the 18th International Conference on Data Engineering (ICDE 2002), San Jose, CA, 2002, pp. 297–308.

[6] G. Shegalov, M. Gillmann, G. Weikum, XML-enabled workflow management for e-services across heterogeneous platforms, VLDB Journal 10 (1) (2001) 91–103.

[7] W. M. P. van der Aalst, Process-oriented architectures for electronic commerce and interorganizational workflow, Information Systems 24 (8) (1999) 639–671.

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

[9] K. Vidyasankar, G. Vossen, A Multi-Level Model for Web Service Composition, in: Proc. of the Second International Conference on Web Services (ICWS2004), 2004, pp. 462–469.

[10] U. Assmann, Invasive Software Composition, Springer, 2003.

[11] C. Szyperski, Component technology - what, where, and how?, in: Proc. of the 25th International Conference on Software Engineering, Portland, Oregon, USA, 2003, pp. 684–693.

[12] J. Norris, K. Coleman, A. Fox, G. Candea, OnCall: Defeating Spikes with a Free-Market Application Cluster, in: Proc. of the 1st International Conference on Autonomic Computing (ICAC'04), New York, New York, 2004, pp. 198–205.

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

[14] L. jie Jin, F. Casati, M. Sayal, M.-C. Shan, Load Balancing in Distributed Workflow Management System, in: G. Lamont (Ed.), Proc. of the ACM Symposium on Applied Computing, Las Vegas, USA, 2001, pp. 522–530.

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

[16] C. Pautasso, T. Heinis, G. Alonso, Autonomic Execution of Service Compositions, in: Proc. of the 3nd International Conference on Web Services, Orlando, FL, 2005.

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

[18] G. Chafle, S. Chandra, V. Mann, M. G. Nanda, Decentralized Orchestration of Composite Web Services, in: Proc. of the 13th World Wide Web Conference, New York, NY, USA, 2004, pp. 134–143.

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

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

[21] P. Heinl, H. Schuster, Towards a Highly Scaleable Architecture for Workflow Management Systems, in: R. R. Wagner, H. Thoma (Eds.), Proc. of the 7th International Workshop on Database and Expert Systems Applications, Zurich, Switzerland, 1996, pp. 439–444.

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

[23] W. M. P. van der Aalst, M. Weske, The P2P Approach to Interorganizational Workflows, in: Proc. of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), 2001, pp. 140–156.

[24] K. A. Schulz, M. E. Orlowska, Facilitating cross-organisational workflows with a workflow view approach, Data and Knowledge Engineering 51 (1) (2004) 109–147.

[25] M. Gillmann, W. Wonner, G. Weikum, Workflow Management with Service Quality Guarantees, in: Proc. of the ACM SIGMOD Conference, Madison, Wisconsin, 2002, pp. 228–239.

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

[27] J.-P. Martin-Flatin, J. Sventek, K. Geihs, Special Issue on Self-Managed Systems, ACM Communications 49 (3).

[28] J. O. Kephart, Research Challenges of Autonomic Computing, in: Proc. 27th International Conference on Software Engineering (ICSE2005), 2005, pp. 15–22.

[29] G. Weikum, A. Moenkeberg, C. Hasse, P. Zabback, Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering, in: Proc. of the 8th International Conference on Very Large Data Bases, Hong Kong, China, 2002.

[30] M. P. Consens, D. Barbosa, A. M. Teisanu, L. Mignet, Goals and Benchmarks for Autonomic Configuration Recommenders, in: SIGMOD Conference, Baltimore, USA, 2005.

[31] S. Elnaffar, W. Powley, D. Benoit, P. Martin, Today's DBMSs: How *autonomic* are they?, in: Proc. of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03), 2003, pp. 651–655.

[32] S. Harizopoulos, A. Ailamaki, A Case for Staged Database Systems, in: Proc. of the 2003 CIDR Conference, Asilomar, CA, 2003.

[33] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, P. Wong, Theory and Practice in Parallel Job Scheduling, in: Proc. of the Workshop on Job scheduling strategies for parallel processing (IPPS'97), Vol. 1291 of Lecture Notes in Computer Science, Springer-Verlag Inc., 1997, pp. 1–34.

[34] B. A. Shirazi, A. R. Hurson, K. M. Kavi (Eds.), Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Computer Society Press, 1995.

[35] D. Georgakopoulos, M. F. Hornick, A. P. Sheth, An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, Distributed and Parallel Databases 3 (2) (1995) 119–153.

[36] C. Pautasso, G. Alonso, The JOpera Visual Composition Language, Journal of Visual Languages and Computing 16 (1–2) (2004) 119–152.

[37] R. Khalaf, A. Keller, F. Leymann, Business Processes for Web Services: Principles and Applications, IBM Systems Journal 45 (2) (2006) (to appear).

[38] OASIS, Web Services Business Process Execution Language (WSBPEL) 2.0 (2006).

[39] T. Heinis, C. Pautasso, O. Deak, G. Alonso, Publishing Persistent Grid Computations as WS Resources, in: Proc. of the 1st IEEE International Conference on e-Science and Grid Computing, 2005, pp. 328–335.

[40] C. Pautasso, G. Alonso, From Web Service Composition to Megaprogramming, in: Proc. of the 5th VLDB Workshop on Technologies for E-Services (TES-04), Toronto, Canada, 2004, pp. 39–53.

[41] J. O. Kephart, D. M. Chess, The Vision of Autonomic Computing, Computer 36 (1) (2003) 41–50.

[42] M. Shaw, "Self-Healing": Softening Precision to Avoid Brittleness, in: Proc. of the first Workshop on Self-Healing Systems (WOSS'02), Charleston, South Carolina, 2002, pp. 111–114.

[43] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, An Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems 14 (3) (1999) 54–62.

[44] N. S. Nise, Control systems engineering, 4th Edition, Wiley, 2004.

[45] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, S. M. Weiss, Predictive algorithms in the management of computer systems, IBM Systems Journal 41 (3) (2002) 461–474.

[46] L. W. Russell, S. P. Morgan, E. G. Chron, Clockwork: A new movement in autonomic systems, IBM Systems Journal 42 (1) (2003) 77–84.

[47] R. Buyya, D. Abramson, J. Giddy, H. Stockinger, Economics Paradigm for Resource Management and Scheduling in Grid Computing, Concurrency and Computation: Practice and Experience 14 (13–15) (2002) 1507–1542.

[48] J. L. Hellerstein, Y. Diao, S. Parekh, D. M. Tilbury, Feedback Control of Computing Systems, Wiley-IEEE Press, 2004.

[49] D. Breitgand, E. Henis, O. Shehory, Automated and adaptive threshold setting: Enabling technology for autonomy and self-management, in: Proc. of the 2nd International Conference on Autonomic Computing, Seattle, WA, 2005.

[50] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, S. Wasserkrug, Autonomic Self-Optimization According to Business Objectives, in: Proc. of the 1st International Conference on Autonomic Computing (ICAC'04), New York, USA, 2004, pp. 206–213.

[51] T. Thessin, H. Sluiman, M. Norman, S. Lucio, J. Saliba, M. Woods, Eclipse Test and Performance Tools Platform Project, `http://www.eclipse.org/tptp/`.