Design and Evaluation of an Autonomic Workflow Engine

Thomas Heinis, Cesare Pautasso, Gustavo Alonso

Department of Computer Science

Swiss Federal Institute of Technology (ETHZ)

ETH Zentrum, 8092 Zürich, Switzerland

`{heinist,pautasso,alonso}@inf.ethz.ch`

Abstract

In this paper we present the design and evaluate the performance of an autonomic workflow execution engine. Although there exist many distributed workflow engines, in practice, it remains a difficult problem to deploy such systems in an optimal configuration. Furthermore, when facing an unpredictable workload with high variability, manual reconfiguration is not an option. Thanks to its autonomic controller, the engine features self-configuration, self-tuning and self-healing properties. The engine runs on a cluster of computers using a tuple space to coordinate its various components. Its autonomic controller monitors its performance and responds to workload variations by altering the configuration. In case failures occur, the controller can recover the workflow execution state from persistent storage and migrate it to a different node of the cluster. Such interventions are carried out without any human supervision. As part of the results of our performance evaluation, we compare different autonomic control strategies and discuss how they can automatically tune the system.

1 Introduction

1.1 Motivation

Workflow management systems are increasingly being applied to domains beyond traditional business process automation. Examples include e-commerce [26, 24], virtual laboratories [1], DNA sequencing [18], scientific computing [2, 17], and Grid computing [4]. More recently, the idea of process-based Web service composition has gained widespread acceptance [5, 16].

In all of these scenarios, workflow engines are used in an open environment, where the characteristics of the workload are not known at the time the system is deployed. Thus, e.g., it may be difficult to choose between a centralized solution or a distributed implementation of the engine. Al-

though a distributed engine may solve some of the scalability issues [3, 14], it opens up the problem of configuring the system in an optimal way. Considering the number of parameters involved and the variability of the workload, having a system administrator in charge of manually monitoring and reconfiguring the system does not seem a feasible solution.

1.2 Related Work

Decentralization of workflow process execution is an important area of research. Typically this is done to support business processes across companies without having to use a centralized entity [6]. This type of process decentralization can lead to higher scalability but it also introduces several problems on its own such as the lack of a global view over the process. It also does not address the scalability and reliability problems per se since the problem is simply translated to each node that executes parts of the process.

To address the afore mentioned problem, tools have been proposed (e.g., GOLIAT [9]) which 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 an extension of this approach, autonomic computing [12] techniques can be used to replace such manual (and static) configuration steps. In [15] an approach to self-optimizing computer systems has been developed. The approach uses an online control algorithm which relies on workload prediction to optimally reconfigure a Web Server with respect to QoS goals over a limited time horizon. The problem of adaptively replicating functionality to achieve higher throughput has also been identified by the database community (e.g. [10, 27]): unbounded replication of functionality can lead to performance losses. The challenge therefore is to replicate distinct functionality depending on the workload only when required. To the

best of our knowledge very little research has been published towards applying autonomic computing principles in the context of distributed workflow engines.

1.3 Contribution

Our goal is to add self-tuning and self-configuration capabilities to a distributed workflow engine so that it can dynamically determine its configuration automatically by taking into account measurements of its performance under the actual (and unpredictable) workload. Furthermore, thanks to the self-healing capabilities of the system, 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 resources are available at the moment. The system has been implemented as an extension to the JOpera engine [20]. JOpera is a Java based service composition tool that combines a workflow engine with an open architecture to provide support for Web service composition, Grid computing and specialized workflow engines [22]. With the extensions described in this paper, JOpera can be initially deployed as a centralized engine and gradually evolve to a distributed engine as the load increases. Similarly, it can revert to a centralized configuration once the load decreases. This is possible thanks to the flexible architecture we propose: components can be deployed on a different node of a cluster and controlled independently. Furthermore, key system modules (e.g., the ones responsible for process navigation and task invocation) can be replicated to handle large workloads. Other components (e.g., the autonomic controller itself) can be paired with a backup to achieve fault tolerance. An important additional feature we propose in this paper is that the autonomic controller can be configured by selecting different reconfiguration strategies at a high level of abstraction. Each strategy defines the information to be collected while monitoring the system, the set of available reconfiguration actions and how they should be applied to adjust the system's configuration. This allows JOpera to react dynamically to load changes rather than using static analysis to reach an optimal configuration [9].

With this, the key contributions of the paper include: (1) the novel system architecture, which is generic and can be adopted by many engines operating under different models and languages (e.g., BPEL [13], XL [7]); (2) the resulting scalability and fault tolerance, which make JOpera flexible enough to support the very large loads present in computational applications and large scale Web service composition; (3) the independence of the underlying workflow model, which make JOpera easily extensible to support many different kinds of services [21], thereby becoming an ideal vehicle for building service oriented middleware platforms.

The remainder of the paper is organized as follows. In Section 2 we discuss the requirements for the design of an autonomic workflow engine. In Section 3 we outline the architecture of the system and show how it fulfills the requirements of self-configuration, self-tuning and self-healing (Section 4). In Section 5 we present an extensive performance evaluation. We conclude in Section 6.

2 System Background

In this section we describe the potential self-management capabilities of an autonomic workflow engine. We also describe the type of processes and deployment environments we are targeting.

2.1 Requirements

To support autonomic behavior, the workflow execution engine must feature self-configuration, self-tuning and self-healing capabilities.

Self-configuration entails switching the system's configuration on the fly without manual intervention and, most importantly, without disrupting the system. This requires the workflow execution engine to provide mechanisms to expose the state of its configuration as well as to support means to dynamically and efficiently change the configuration.

The *self-tuning* capabilities should ensure that system reconfiguration leads to a configuration which is optimal given the current workload. In order to enable self-tuning capabilities the workflow 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 in response to the current workload. 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 [25]. 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 workflow execution state in order to determine how the running processes have been affected.

2.2 Workload Assumptions

The workload is assumed to be a collection of concurrent workflow processes. In general, users may define an arbitrary process and initiate its execution at any time. In our evaluation we focus on a worst case scenario where many processes are submitted for execution simultaneously. However, neither the size nor the structure of the processes is taken into account when designing the self-tuning algorithm, which should be able to deal with the execution of any process that can be normally submitted to the engine.

Furthermore, in this paper we do not deal with workload prediction issues. Our autonomic engine is purely reactive in that it observes the current load and reacts to it. A pro-active system would try to anticipate workload changes before they occur [23]. Since JOpera collects a detailed history of past executions, a data mining algorithm could analyze it and use it to predict future workflow arrival times. We will pursue that option as part of future work.

2.3 Deployment Environment

We assume that JOpera runs on a dedicated cluster of computers and can use these resources exclusively. With this assumption, the main goal of the autonomic features is to ensure the optimal configuration of the cluster. This requires both efficient resource utilization as well as a good allocation of the available nodes to the different system components. The assumption of a dedicated cluster does not imply that the cluster configuration is static. Clearly, the pool of nodes assigned to the engine can change over time and, as in every cluster-based system, nodes may fail. In practice, the system could be extended to use shared nodes that are also used for other purposes. However, this is a resource management problem orthogonal to the autonomic issues we want to explore in this paper.

3 System Architecture

In this section we describe the main features of JOpera's architecture that are used to implement autonomic behavior (JOpera is publicly available [20]).

3.1 Workflow Execution

Workflow processes (or workflows) model the interactions between different tasks by defining the data flow and control flow between them. The data flow defines data exchanges between tasks whereas the control flow constrains the order of task invocations [8].

The execution of a process begins with a request sent through the corresponding API of the engine. Processes can be started by users or invoked from other processes.
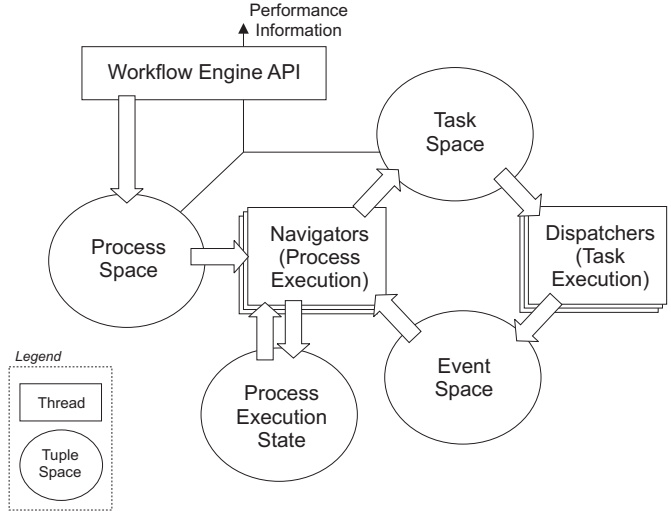

Figure 1. Logical architecture of the JOpera distributed workflow execution engine

The engine API queues such requests into the *process execution requests* space (or process space). As shown in Figure 1, these requests are handled by the navigator, which 1) creates a new workflow instance into the *process execution state* space and 2) begins with the actual enactment of the workflow. To do so, the navigator uses the current state of the execution of a process to determine which tasks should be invoked next based on the control and data flow dependencies that are triggered by the completion of the previous tasks. Once the navigator determines that a certain task is ready to be invoked, the corresponding tuple is stored in the *task execution request space* (or 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 the corresponding service providers. These include, e.g., worklist handlers for tasks that should be carried out by human operators, but also standard compliant Web services, as well as many other kinds of services [21]. After the execution of the task has been finished, the dispatcher notifies the navigator through the *event space*. More precisely, the dispatcher packages the results of the invocation into a task completion tuple, which is posted into the event space. Such tuples are then consumed by the navigator in order to update the state of the execution of the corresponding process and carry on with its execution.

The main reason for separating the execution of the workflows from the execution of their tasks lies in the observation that these operations have a different level of granularity. It is to be expected that the execution of task performed by the dispatcher may last significantly longer than

the time taken by the navigator for scheduling it. With our approach, the platform supports the parallel invocation of multiple tasks belonging to the same process. Furthermore, a slow task does not affect the execution of other processes running concurrently because these two operations are handled asynchronously by different threads. This is already an important departure from existing workflow engines where navigation and dispatching are serially executed by a single thread.

3.2 Distributed Workflow Execution

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 across multiple nodes to manage the concurrent invocation of multiple tasks. Likewise, if many processes have to be executed concurrently, the navigator can also be replicated. The resulting pool of navigator and dispatcher threads are loosely coupled by using tuple spaces as depicted in Figure 1. We have chosen to use tuple spaces primarily for their persistent data storage capabilities as well as for the simple API provided. This way, navigators generate tuples containing task execution requests which are consumed by the dispatchers. Similarly, dispatchers send tuples back to the navigators to notify them of the results of the invocations. Thus, 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. However, at most one thread (dispatcher or navigator) is running on a node at a given time.

Thanks to the flexibility provided by tuple spaces, it becomes feasible to dynamically reconfigure the system, as the number of navigators and dispatchers can be increased or decreased without having to stop the whole system. To do so, the system offers a reconfiguration API that makes it possible 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.

3.3 Scalable Workflow Execution

Although tuple spaces offer good abstractions for decoupling and replicating the navigator and dispatcher threads, they pose a potential scalability bottleneck [19]. To address this problem, we use several layers of caching between the tuple space and the threads producing and consuming tuples. As this optimization affects the self-configuration mechanisms, in this section we describe it in more detail.

Instead of running only one tuple space server on a dedicated node, the distributed workflow engine replicates such
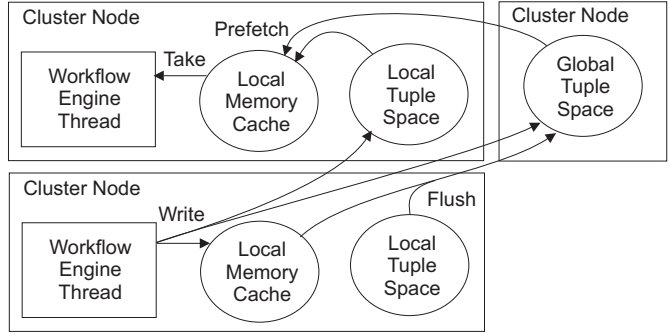

Figure 2. Layers of caching between each thread and the global tuple space

tuple space server on each node of the cluster. One of the replicas is then configured to act as the global space, while all of the others are considered to be local with respect to the thread that is running on a particular node (Figure 2).

When a tuple is written by a thread, its destination is chosen in order to place it as closely as possible to the consumer. Thus, if a navigator posts an event to itself (e.g., when a workflow calls another workflow), the corresponding tuple is written in the local memory cache. However, if a dispatcher should notify a navigator of a completed task execution, the dispatcher will write a tuple in the space which is nearest to the receiving navigator, i.e., its local one. In all other cases the tuple is written in the global space. With the added complexity of maintaining a routing table which defines in which space the tuples should be written into, these optimizations help to reduce the load on the global tuple space [22].

This routing table is also used when reading tuples. In order to increase the throughput of the threads, each thread pre-fetches into its memory cache all relevant tuples, which are located both in the local and global spaces. This way, the tuples are immediately available when a thread is ready to take one.

The routing table contains a mapping between the ID of a process instance and the address of the node on which the navigator thread executing this process instance is located. This mapping is created when a navigator thread first begins running a particular process and is removed when the process is finished. However, if a system reconfiguration occurs and a navigator thread should be stopped, such mapping is also temporarily removed so that tuples are routed to the global space until the process execution is migrated to a different node. Also in this case, all cached tuples are flushed to the global space.

4 Autonomic Capabilities

In this section we describe the design of the autonomic controller of the workflow engine. Figure 3 shows its main components (self-tuning, self-configuration, and self-healing) and the interactions between them.

4.1 Self-Tuning

The self-tuning component is responsible for determining whether the current system configuration is optimal. 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. To do so, the self-tuning component acts upon three different strategies. The information strategy describes which performance indicators should be monitored in order to enhance the overall performance of the system. The optimization strategy defines how to achieve an optimal configuration, i.e., an optimal partitioning of the cluster between navigator and dispatcher threads. The selection strategy describes how to map the reconfiguration decision onto the cluster.

4.1.1 Information Strategy

By considering the architecture of the workflow engine (Figure 1), 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 strategy 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.

4.1.2 Optimization Strategy

The goal of the optimization strategy is to establish a configuration such that the number of navigator and dispatcher threads is balanced. Since navigator threads produce task invocation requests and dispatcher threads consume them, the task tuple space is solely influenced by the internal system's configuration defined in terms of the number of dispatchers and navigators. This does not hold for the process tuple space where processes are submitted by the API and is therefore subject to external influences which are independent of the configuration of the system.

The optimization strategy thus defines that the number of dispatcher threads needs to be increased when the rate of growth of the task space exceeds a certain threshold. Similarly, if the size of the process space increases, additional navigator threads should be started in order to execute the newly started processes. Given the limited number of available resources, the optimization strategy must determine how many nodes should be allocated to run navigators and how many should run dispatchers threads. Therefore, in case there are no idle nodes left, a navigator (or dispatcher) thread needs to be stopped in order to free a node for starting a dispatcher (or navigator). Stopping a navigator implies less task production capacity and starting a dispatcher means more task consumption capacity. Thus, switching from a navigator to a dispatcher thread effectively reduces the growth of the task space. Conversely, if all navigators are busy handling task completion notifications, the size of the process space will grow and additional navigators are required to execute the newly started processes.

4.1.3 Selection Strategy

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

Arriving at a concrete configuration that is to be submitted to the self-configuration component from an abstract configuration plan is done by prioritizing nodes according to how well suited they are for a configuration change. 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. Similarly, if there are more threads than necessary, idle threads should be the ones stopped. However, 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 strategy therefore needs to take this reconfiguration cost into account when deciding which thread should be stopped. The simplest selection strategy chooses the threads randomly, regardless of the resulting rescheduling overhead. However, we also experimented with a smart selection strategy that chooses threads with the goal of minimizing the overhead caused by rescheduled tasks and 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.
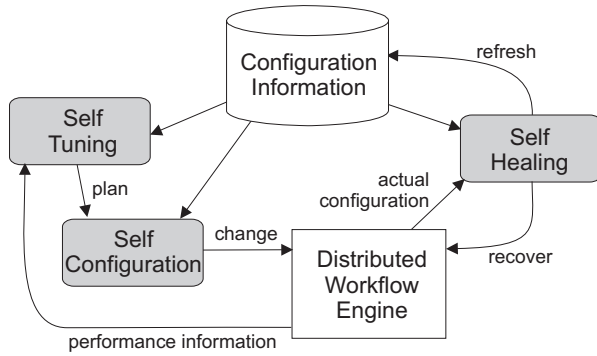
Figure 3. Interaction between the components of the autonomic controller

4.2 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 cluster and applies changes to it. Implementing the new configuration requires time and the result may not be available immediately.

In order to execute the reconfiguration plan, the self-configuration component uses a closed feedback-loop controller that 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.

4.2.1 Reconfiguration Actions

The self-configuration component can alter the configuration in the following ways:

Starting Threads: In order to start a thread on a particular node, the JOpera API first needs to be started. The API waits for start and stop commands sent to it. Starting dispatcher and navigator threads can be done as long as the node is idle. The self-configuration component only needs to issue the start command on the node and the according thread will start working immediately.

Stopping Navigator Threads: Stopping a navigator thread entails migrating the state of the processes the navigator thread is working on and redirecting associated events. Mi-

grating the state of a process is done by flushing the locally cached state into the global tuple space so that a next navigator can pick it up and resume working on it. All cached events which the navigator has not yet processed will also be transferred into the global tuple space. Furthermore, events that may be triggered by dispatcher threads executing task invocations that belong to a process that is about to be migrated, are redirected to the global event tuple space.

Stopping Dispatcher Threads: In contrast to stopping navigators, stopping a dispatcher thread is more difficult. Dispatcher threads are executing tasks that may involve the invocation of a local application or the interaction with a remote service provider on the Web. In some cases, it may not be possible to transparently interrupt such executions. Processes can contain metadata that define whether a task is repeatable, which can be used under these circumstances to choose the appropriate method.

More concretely, we take this into account by providing different methods of stopping a dispatcher thread. The *kill method* immediately stops all active task executions in progress on a particular dispatcher thread and ensures that all task invocations will be repeated on a different dispatcher thread by placing the corresponding tuple back in the task space. Repeating all tasks which have been executing introduces some overhead as the process execution is delayed. Clearly, this method can only be applied to repeatable or resumable tasks which are more likely to be found in scientific computing applications.

The *stop method* immediately ceases to take tuples from the task space. As a consequence, no new tasks will be started, but the dispatcher will wait for all task executions to finish before stopping. This method has the disadvantage that – as long as all tasks have not finished their execution – the node is not immediately available for starting a different thread. A dispatcher thread executing tasks which both require stop semantic and allow kill semantic, may only be interrupted using the stop method. The engine therefore schedules tasks with kill semantics on specific dispatcher threads which can then be stopped immediately allowing faster reconfiguration of the system.

4.3 Self-Healing

The task of the self-healing component is to ensure that the workflow 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 thus 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 workflow engine. More precisely, failures are handled differently, depending on what kind of thread has failed.

Handling Dispatcher Thread Failures: In case a dispatcher fails, the tasks that were managed by it are lost and have to be restarted. The self-healing component queries the state of the execution of the process to determine which were the tasks currently assigned to the failed thread. These tasks are automatically restarted by resubmitting the corresponding task into the task execution request space. This recovery procedure is very similar to the one carried out when the self-configuration component kills a dispatcher in order to reconfigure the system. Also in that case, some tasks may have to be re-executed.

Handling Navigator Thread Failures: Should a navigator thread fail, the state of the execution of the process is still available in the global process execution state space because the navigators perform work only on a cached copy of the state. The self-healing component can recover the processes by simply removing their entries in the tuple routing table which point to the failed navigator. This way, all pending events can be routed through the global space until another navigator becomes available to process them.

5 System evaluation

The goal of the system evaluation is to analyze the autonomic capabilities of the workflow engine. In particular, we want to explore how the system adapts to different workload conditions automatically and how it reacts to failures. In the first part we study how, given a workload as described in Section 2.2, the autonomic controller reconfigures the system optimally by using the self-configuration as well as the self-tuning component. Then we present an interesting self-healing result where the engine not only recovers the execution of its tasks, but can also re-balance its configuration to optimally use the nodes which remain available after a failure.

5.1 Experimental Setup

For the experiments, JOpera has been deployed on a cluster of up to 20 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. One additional node was allocated to the global tuple space server, running IBM's T-Spaces v2.1.3 [11].
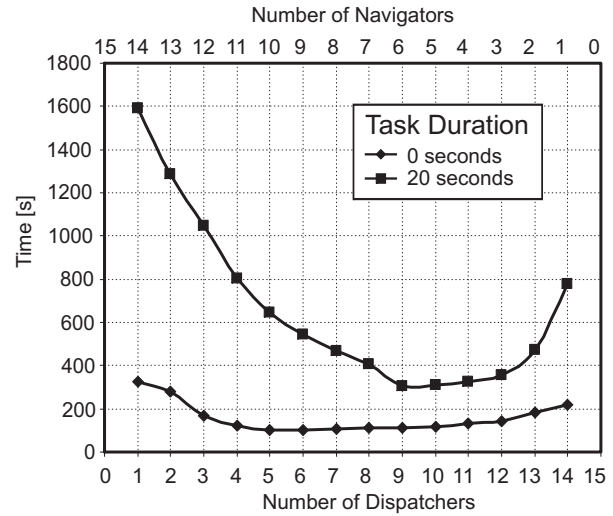

Figure 4. Time required to execute two different workloads of 1000 processes using all possible static configurations

5.2 Base line

In this section we motivate the need for an autonomic controller by showing that the optimal configuration of the engine is highly sensitive to the workload. Furthermore, we use these results to validate the self-tuning component, which should dynamically find the optimal configuration so that the self-configuration component can correctly configure the system.

JOpera can also be configured statically. In this case, the number of dispatchers and navigators is fixed and no dynamic changes will be applied at runtime as the autonomic controller is disabled. In order to determine the optimal static configuration for a given workload, we have carried out a series of experiments using different configurations in order to determine the configuration which minimizes the response time of the system. Figure 4 depicts the total execution times of two different workloads: 1000 concurrent processes containing 10 parallel tasks of duration of 0 seconds (workload 0) and 1000 processes containing 10 parallel tasks of duration of 20 seconds (workload 20). A total of 15 nodes were used in the experiments and all possible configurations starting with 14 navigators and 1 dispatcher up to 14 dispatchers and 1 navigator were tested.

The results of experiments in Figure 4 and 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, Figure 5 shows that –
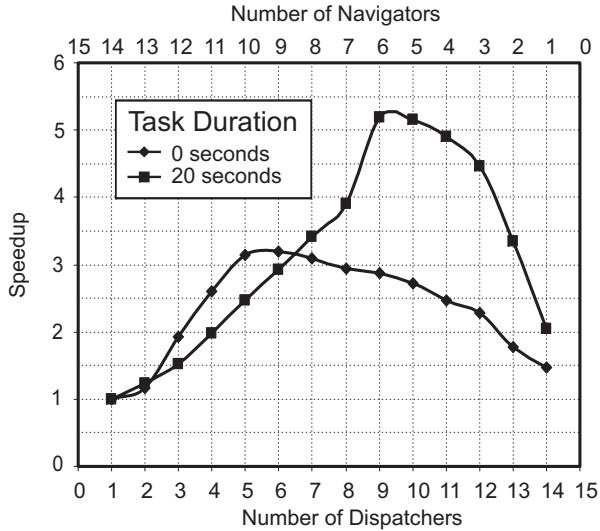
Figure 5. Speedup relative to the slowest configuration (D=1, N=14) of Figure 4 achieved using all possible configurations

Figure 6. Comparison of the execution time when using different strategies to run the combined workloads

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.

This is shown in Figure 6 (right two columns), where we take both of the optimal configurations and use them to run a combined workload of two peaks separated by 120 seconds, the first consisting of workload 0 and the second of workload 20. With the configuration optimal for running workload 0, the time is 826 seconds, while the other configuration is faster with only 758 seconds.

5.3 Autonomic Behavior

Figure 7 shows the behavior of the system as it automatically adapts its configuration to different workloads. We first describe the trace of one experiment, obtained by sampling various performance indicators and logging their values at regular intervals (every second). Then, we compare different selection strategies combined with a different choice of reconfiguration actions to determine the corresponding reconfiguration overheads.

5.3.1 Self-Configuration

Figure 7a shows the size of the *process execution requests* tuple space over time. This gives a good overview of the rate at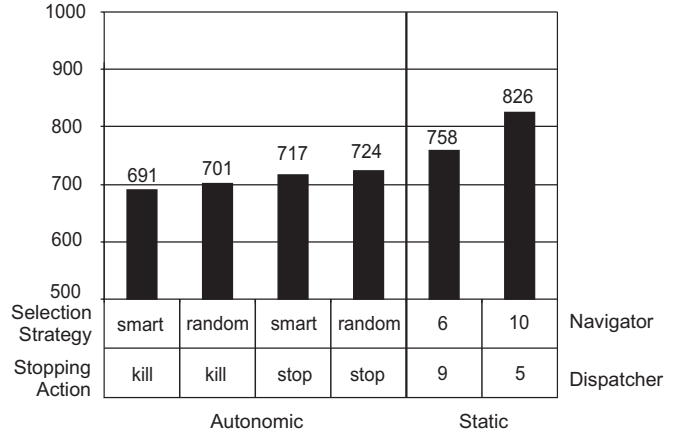 which processes are queued to be started (the curve grows) and instantiated and executed by navigators (the curve drops). It also directly reflects the workload which is applied to the system, which – in this experiment – consists of two peaks with varying characteristics.

The first peak occurring at t=0 consists of 1000 processes which execute 10 parallel tasks having a duration of 0 seconds. The characteristic of the processes requires more navigators than dispatchers to be started: since the tasks can be executed in virtually no time, the dispatchers can execute many tasks in a given period of time. As the dispatchers can handle a lot of tasks, there is a need for a significant number of navigators handling their completion notifications as well as issuing new ones.

As can be seen in Figure 7c, the controller configures the system accordingly by allocating only up to 5 dispatchers, while using the rest of the nodes to run navigators. This configuration will change as the second peak hits the system at t=120s when the number of processes that still wait to be executed is already declining. As can be seen in Figure 7a, in response to this peak, the number of process execution requests waiting in the tuple space grows as the new processes are fed into the system.

As these processes begin their execution, the size of the task tuple space also starts to increase as a result (Figure 7b). This can be explained by the different characteristics of the second peak. Although it still comprises of 1000 processes, executing 10 parallel tasks, the task duration has now been set to 20 seconds. Thus, more dispatchers are required, as tasks now take longer to run. Between t=150 and t=200, the controller attempts to balance a system which lags behind both in the execution of processes (at t=200, the number of waiting processes peaks at almost 1000) and in
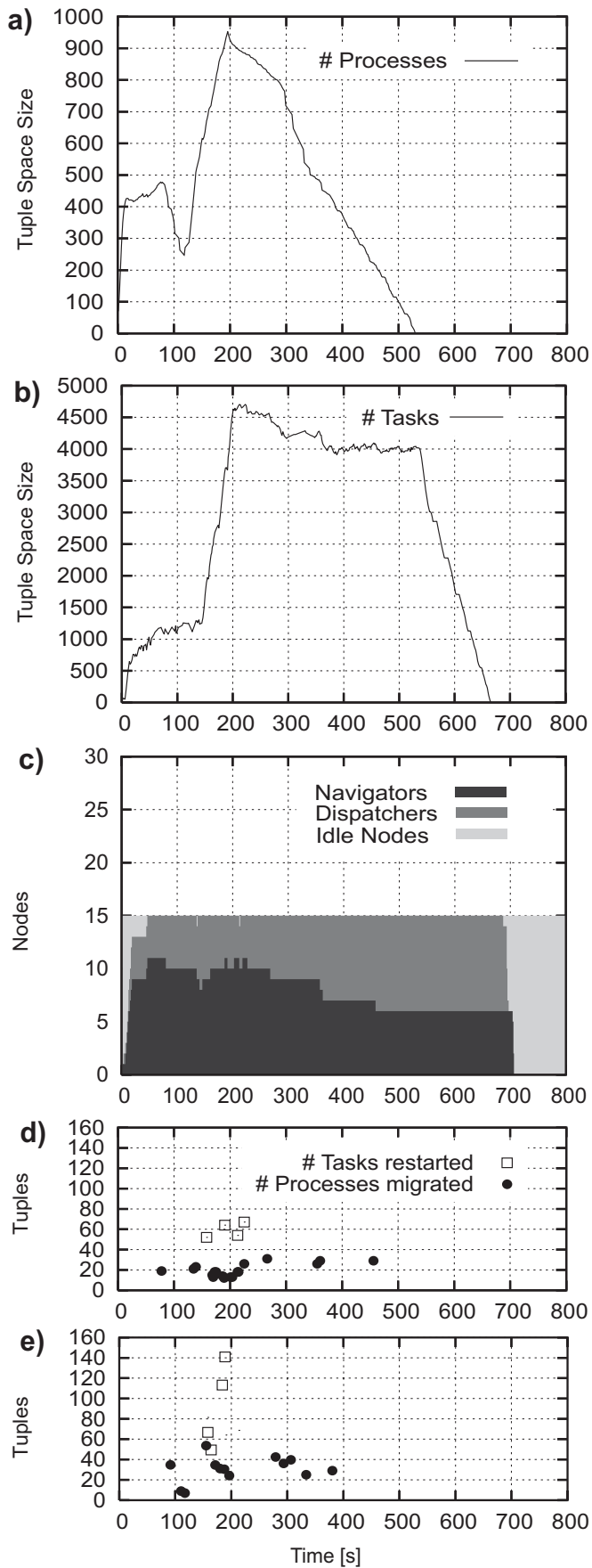
Figure 7. Autonomic Controller reconfiguring the system as workload changes
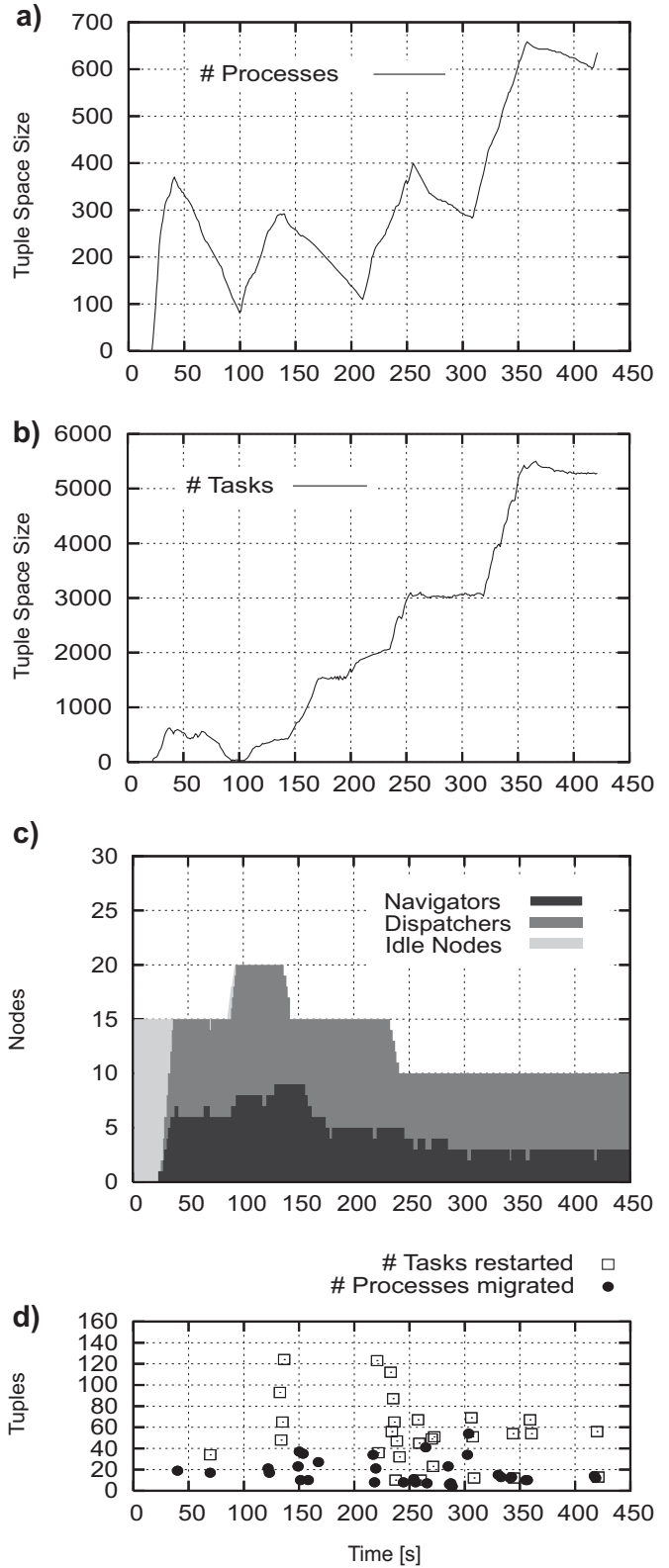

Figure 8. Autonomic Controller healing the system as nodes are added and removed

9

the execution of tasks. Thus, the configuration does not change significantly. Once all processes have been queued, Figure 7c shows that actual reconfiguration starts after t=200. More precisely, the self-tuning algorithm detects the imbalance and begins to steadily increase the number of nodes allocated to the dispatcher threads, while reducing the number of navigator threads. The configuration eventually stabilizes after t=400s. The system is balanced again, as Figure 7b shows: the number of tasks remains stable (4000) indicating that the number of consumers (the dispatchers) is balanced with the number of producers (the navigators).

At t=521, all processes have been started and thereby all the contained tasks have been put into the task execution request space. The number of tasks in the space steadily decreases thereafter. At second 691 all tasks have been executed and the controller stops the dispatchers as they become idle. Shortly afterward, the number of navigators reaches zero, because the self-configuration component also stops these threads as they become idle.

5.3.2 Reconfiguration Overhead

Figure 7d shows the reconfiguration overhead. Whenever a navigator is stopped, the cached state of its processes is transferred into a global tuple space waiting for a next navigator to pick it up. More significant is the overhead introduced by stopping dispatchers. If a dispatcher is stopped, all tasks it has been executing are stopped and need to be repeated leading to a delay in the overall execution of their process.

In this experiment, we compare different selection strategies for choosing which thread running on what node should be stopped. The goal is to determine which strategy minimizes the reconfiguration overhead. First of all, we logged the number of tasks and processes that were rescheduled and migrated as the corresponding thread was stopped. From this, it can be seen that the *random* selection strategy (Figure 7e) appears to reschedule more tasks and migrate more processes than the *smart* selection strategy (Figure 7d). When running the same workload, the number of reconfiguration actions is approximately the same, but the height of the peaks is much lower, as the smart selection strategy chooses the nodes with the least amount of work to be repeated. This leads to an decrease of the overall execution time of 10.6 seconds (Figure 6).

In this figure we also combine the selection strategies with a different choice of reconfiguration actions (*kill* vs. *stop*). As the results indicate, the dominant factor regarding execution time is the reconfiguration action. Both selection strategies perform better by using the kill method instead of the stop method for stopping dispatcher threads. The reason for this is that when using the stop method, reconfiguration

does not happen immediately. Instead, the dispatcher must wait until the longest task has been executed. In case of our experiments with tasks lasting up to 20 seconds, in the worst case reconfiguration was delayed by 20 seconds.

5.4 Self-Healing

The goal of the self-healing experiment is to demonstrate the ability of the system to react to external changes affecting the configuration of the cluster. In this experiment the system is initially configured to use 15 nodes. Then, in order to replace 5 of the nodes assigned to it, 5 additional nodes are added and a bit later a different group of 5 nodes is removed. Towards the end of the experiment, the newly added nodes fail.

This time the workload consists of four peaks of 500 processes occurring every 100 seconds. Each of the processes consist of 10 parallel tasks of 10 seconds duration. Starting with 15 nodes, the cluster has been grown to 20 nodes at t=90 and has then been reduced by 5 nodes at t=140 and again by 5 nodes at t=230. When the cluster grows by 5 nodes at t=90, the system instantly uses the additionally available nodes by increasing both the number of dispatcher as well as the number of navigator threads. The increase of the number of dispatchers leads to a the task space being empty temporarily at t=100 as can be seen in Figure 8b. The task space is filled again soon because the process space starts to grow when the second peak of the workload is fed into the system.

At t=140 5 nodes running dispatchers are removed from the cluster. Because there is still the same number of navigators producing tasks but a smaller number of dispatchers consuming them, the task space starts to accumulate tuples at t=150 due to this imbalance. The system subsequently adapts to this situation by stopping navigators and starting dispatchers again between t=155 and t=170 as can be seen in Figure 8c. The growth of the task space slows down shortly after the system has readjusted the configuration.

Figure 8d illustrates the recovery actions performed by the self-healing component: at t=140 five dispatchers are stopped and therefore the tasks that were currently running are automatically rescheduled. Navigators were stopped when the system adapts to the new conditions and their processes were rescheduled shortly after t=150.

The third configuration change at t=230 also involves the loss of 5 dispatchers. The system reacts consistently. Reducing the dispatchers while leaving the number of navigators leads to a growth of the task space which in turn triggers a reconfiguration. The system will subsequently reduce the number of navigators and increase the number of dispatchers. This change in configuration will again contain the growth of the task space. The change of the configuration can again be observed in Figure 8d where after the

degradation by 5 dispatchers tasks are rescheduled at t=230. At t=250 processes are rescheduled due to the configuration change which entails stopping navigators and starting dispatchers.

The different configurations are also reflected in Figure 8d: because the number of navigators changes, the slope of the process space also changes. When for instance comparing the number of navigators between t=40–100, t=130–210 and t=260–320, one can observe that the slope of the process space size curve becomes flatter. This is a result of the number of navigators gradually being smaller.

Since there are no additional peaks occurring after second 310 and the system will simply execute all processes and tasks until both spaces are empty with a stable configuration of 3 navigators and 7 dispatchers after t=450.

5.5 Discussion

Although it is not impossible to find an optimal static configuration for a given workload, it is very difficult to assess the workload a priori and configure the system accordingly. In our first experiments we have been able to tune the configurations in order to execute a given workload as optimally as possible. But workloads with different characteristics lead to different optimal configurations as can be seen in Figure 5. And if a statically configured system executes a workload with characteristics it has not been tuned for, its performance degrades. To overcome this, either manual reconfiguration or self-tuning plus self-configuration is required.

As the results of the self-configuration experiment indicate, the autonomic controller was able to adapt the configuration of the workflow engine according to the variable characteristics of the workload. By combining the workloads of the base line experiments, the autonomic controller shifted the system's configuration between the optimal static ones. This had an impact on the overall performance, as the comparison between different versions of the autonomic controller and the optimal static configurations indicated (Figure 6). As expected, the smart selection strategy outperformed the random selection strategy. With it, the impact of a reconfiguration is minimized, as the least number of tasks have to be restarted when stopping a dispatcher. Combining the smart selection strategy for stopping threads with the kill reconfiguration action leads to the most significant speedup compared to the static configurations. Overall, for all combinations of a selection strategy with a stopping action the autonomic engine performed better than a statically configured one.

The self-healing 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. As we have shown in the previous section, the autonomic controller was able to immediately detect the newly assigned nodes and could also transparently recover and optimally reconfigure the engine when some of the nodes were taken off-line.

6 Conclusion

In this paper we have presented the design of an autonomic workflow engine, demonstrated its self-managing behavior and evaluated its performance. The engine runs on a dedicated cluster of computers and can automatically reconfigure itself based on the current workload by using autonomic computing techniques. This is an important contribution, as workflow management systems are being more and more applied to domains that can be characterized by the unpredictability of their workloads, such as – for instance – process-based orchestration of Web services. In the past, distribution has been applied to the design of many workflow engines in order to improve their scalability and reliability. However, very little attention has been paid to the need for properly configuring such distributed systems. This, in practice, remains a difficult, error-prone, manual, and time-consuming operation, especially when deploying the system to face an unpredictable workload. In this paper we have shown how to apply the autonomic computing paradigm to greatly simplify the deployment and the maintenance of such systems. As our experiments indicate the autonomic controller of JOpera can adapt the system configuration optimally to unforeseeable, changing workload characteristics. The system furthermore takes failures into account and adapts the system's configuration accordingly. Although the results presented in this paper were obtained with relatively homogeneous workloads, we will explore the effects of workloads with more complex characteristics as part of future work as the system is deployed in realistic production settings.

References

[1] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proceedings of the 17th International Conference on Data*

Engineering (ICDE2001), pages 235–242, Heidelberg, Germany, 2001.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. M. and. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proceedings of the 16th Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, Santorini Island, Greece, June 2004.

[3] 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 IF-CIS International Conference on Cooperative Information Systems (CoopIS'97)*, pages 99–108, Kiawah Island, South Carolina, USA, 1997.

[4] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid03)*, pages 198–205, Tokyo, Japan, 2003.

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

[6] 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.

[7] D. Florescu, A. Gruenhagen, and D. Kossmann. XL: A Platform for Web services. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, 2003.

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

[9] 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.

[10] S. Harizopoulos and A. Ailamaki. A Case for Staged Database Systems. In *Proceedings of the 2003 CIDR Conference*, Asilomar, CA, 2003.

[11] IBM. *TSpaces*. http://www.almaden.ibm.com/cs/Tspaces/.

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

[13] IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web services (BPEL4WS) 1.0*, August 2002. http://www.ibm.com/developerworks/library/ws-bpel.

[14] 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.

[15] N. Kandasamy, S. Abdelwahed, and J. P. Hayes. Self-Optimization in Computer Systems via Online Control: Application to Power Management. In *Proceedings of the 1nd International Conference on Autonomic Computing (ICAC '04)*, New York, NY, 2005.

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

[17] R. McClatchey, J.-M. L. Goff, N. Baker, W. Harris, and Z. Kovacs. A Distributed Workflow and Product Data Management Application for the Construction of Large Scale Scientific Apparatus. In *Workflow Management Systems and Interoperability*, pages 18–34, 1997.

[18] J. Meidanis, G. Vossen, and M. Weske. Using Workflow Management in DNA Sequencing. In *Proceedings of the 1st International Conference on Cooperative Information Systems (CoopIS96) Brussels, Belgium*, June 1996.

[19] P. Obreiter and G. Graef. Towards scalability in tuple spaces. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002)*, pages 344–350, Madrid, Spain, March 2002.

[20] C. Pautasso. JOpera: Process Support for more than Web services. http://www.iks.ethz.ch/jopera/download.

[21] C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, Toronto, Canada, August 2004.

[22] 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.

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

[24] A. Schmidt, T. Sindt, M. Tepegoez, and G. Joeris. FlowTEC - An Information System Supporting Virtual Enterprises. In *Proceedings of the 2nd International Conference on Concurrent Multidisciplinary Engineering (CME'99)*, Bremen, 1999.

[25] M. Shaw. "Self-Healing": Softening Precision to Avoid Brittleness. In *Proceedings of the first Workshop on Self-Healing Systems (WOSS'02)*, pages 111–114, Charleston, South Carolina, November 2002.

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

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