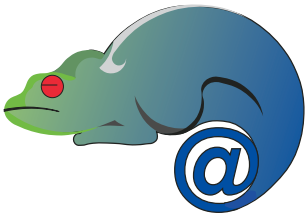
ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

CS Availability Container



Deliverable Identifier: D10
Delivery Date: August, 19, 2004
Classification: Public Circulation
Authors: Gustavo Alonso, Cesare Pautasso, Biörn Biörnstad
Document version: FINAL – AUGUST 2004

Contract Start Date: 1 September 2002
Duration: 36 months
Project coordinator: Universidad Politécnica de Madrid (Spain)
Partners: Universitá di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Universitá degli Studi di Trieste (Italy), University of Newcastle (UK), Hewlett-Packard (UK)

Project funded by the European Commission under the Information Society Technology Programme of the 5th Framework (1998-2002)

Dependencies with other deliverables

This deliverable is a description and evaluation of parts of the ADAPT composition engine (deliverable D14, CS Middleware) which is currently at the prototype stage (deliverable due on month 29). The deliverable builds upon deliverables D9 (CS Middleware Architecture, due on month 5) and D7 (Composition Language, due on month 11). The aspects of the evaluation plan that pertain to this module are covered by this deliverable in the form of extensive performance measurements.
As indicated by the reviewers on the first review of the project, we have opted for a format in the report that matches scientific papers. Part of this report has been accepted for publication in: C. Pautasso, G. Alonso, JOpera: a Toolkit for Efficient Visual Composition of Web Services, International Journal of Electronic Commerce (IJEC).

1 Motivation for the deliverable

Although they may not solve all interoperability problems, Web service technologies show great promise in reducing the complexity of integrating heterogeneous software components over the Internet. They provide standard protocols for invoking (SOAP [26]), describing (WSDL [27]), and discovering services (UDDI [20]) published on the Internet in a platform, programming language and vendor independent manner. A most natural evolution of these technologies concerns the ability to compose complex Web services from basic ones [5]. Especially in E-Business scenarios, the standardization efforts to integrate Web services into business processes have produced many proposals [6, 13, 14, 18, 23, 28, 29]. However, none of these is yet well established in practice [24], although the Business Process Execution Language for Web Services (BPEL4WS [13]) specification seems to be ahead at the moment.

The standardization efforts behind Web services, and the increasing number of aspects that are being formalized, open up the possibility of reducing the development cost and complexity of large distributed information systems. In particular, a visual approach to Web service composition may very well be a suitable complement to such existing XML-based composition standards. Using a visual programming language may help to bridge the different standards and will certainly help to make Web services much more designer-friendly. In such a system, the order of invocations of services, their data exchanges and the necessary failure handling behavior could all be specified with a simple visual syntax. However, having a visual programming language, such as the ADAPT composition language, for Web service composition is not enough. The visual programming environment needs also a set of tools for efficient, scalable and reliable execution of such composite applications.

Regarding the efficient and scalable execution of composite Web services, we describe a novel architecture for a process support system, in which a flexible container for process execution can be tailored to provide different availability guarantees. This is achieved through a set of well defined abstractions for storing the state of a process, propagating events and executing tasks. By switching between different implementations of these abstractions, different quality of service levels can be achieved in terms of both scalability and reliability. Within this flexible architecture, we show that replication can be applied to any system component to increase the overall system's throughput. In particular, we focus on parallel navigation, as it is an important scalability bottleneck not well understood in practice. Finally, we validate our approach with an extensive set of experiments, which systematically compare the performance of different system configurations.

In this report, we concentrate on the ADAPT Availability Container: in Section 2 we present its architecture, describing how the system executes the processes written in the ACL language. In Section 3, we list some of the configurations into which the ADAPT container can be deployed. To compare them, we conducted a systematic performance evaluation and in Section 4 we present the most important results.

2 ADAPT Availability Container

In this section we present the architecture of the ADAPT Availability Container. The main purpose of such container is to provide an execution platform for the ADAPT Composition Language which can be tailored to different levels of performance.
The relationship between the ADAPT Visual Development Environment and the Process Execution Container is defined as follows. The processes defined in the ACL language are created and edited using the development environment. As we will explain in the following section, once the processes are complete and ready to be executed, they are compiled into Java and the resulting process template plugins are then dynamically loaded into the container for execution. Once a new process instance has been started, its execution is managed by the container, which may be run independently of the development environment. However, users may connect the development environment to an existing container to monitor the activity and the progress of their processes.

Before describing in detail the architecture of the ADAPT Container, we would like to give some background on how it is possible to execute the description of a process written in ACL using the so-called navigation algorithm.

2.1 Process Navigation

Navigation is the procedure whereby the system determines the set of tasks to be executed next, given the current state of the process and its control flow graph, specifying the partial order of execution of the tasks. To do so, the navigation procedure interprets the information of a directed graph, where the nodes represent the tasks and are labeled with their current state, and the edges represent control flow dependencies between the tasks. When a state change of a task occurs, the algorithm proceeds in two steps. First, in order to determine the set of tasks affected by the state change, it follows all outgoing control flow dependencies.
Then, it evaluates the starting conditions of these tasks to check if they are ready to be started. This way, after every task state change it is possible to determine the set of tasks to be started next. This approach is very similar to mapping the process description to a set of Event, Condition, Action (ECA) rules. State changes of tasks trigger events, which will cause the evaluation of the conditions associated with the set of dependent tasks and, when these rules fire, the actions required to start the tasks can be carried out. More specifically, during navigation the system mainly performs two types of actions. The first type concerns the actual task execution, that is packing all the necessary information into a job that can be submitted to the scheduler responsible for finding a suitable machine for running the task or the correct provider to which a request message should be sent in order to invoke the service. The second type groups operations that access or modify the state information of a process. These include, for example, copying the data from the parameters of one task to another as specified in the data flow of the process, as well as setting metadata values, such as the starting time of a task, or accessing the state of a set of tasks to determine whether they have failed.

2.1.1 Executing Navigation

In practice, it is not necessary to compile a process description into a generic ECA-like representation to be interpreted by the process engine. Instead, to implement our process navigation algorithm we build on the idea of mapping the process description to a program embodying the specific rules corresponding to the process description, generated using an ordinary programming language. This way, we can use the language's compiler to produce executable code which then can be dynamically loaded and linked into the container's runtime environment to be executed. This approach has the potential to provide better performance. First of all, the executable code is generated in a standard programming language, in our case Java, which then is compiled one more time. This way we can map the process model to standard language constructs, which can be efficiently executed. Moreover, during code generation it is possible to analyze the structure of the process and perform optimizations.

In addition to this, the generated program is completely stateless, as it only contains a mapping of the process structure. To perform navigation over a particular process instance, the program reads its state as input. Therefore, it is possible to perform navigation over many instances of the same process using the same program code, which only needs to be loaded once. In many existing systems, this clear separation between the state of a process and its structure is missing and before each invocation of the navigation procedure both types of information need to be loaded from the persistent repository, incurring in unnecessary overhead.

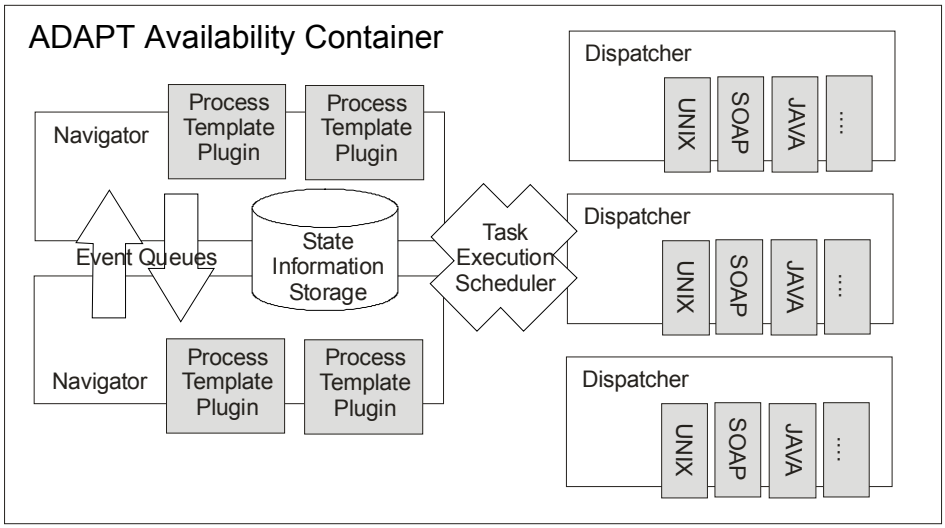


Figure 1: Architecture of a Distributed Container

2.2 Architecture

The core infrastructure necessary to run the processes written in ACL is depicted in Figure 1. The ADAPT process execution container includes mechanisms to 1) run the navigation algorithm, 2) schedule and 3) dispatch tasks for execution in the correct environment, 4) access and modify state information about tasks and processes, and 5) exchange event notifications triggering the execution of the navigation algorithm itself. It should be noted that our navigation algorithm is independent of the actual implementation of these basic facilities, which are described in the rest of this section.

2.2.1 Navigator

The navigator is the container component responsible for handling incoming process events, which are generally triggered by changes in the state of tasks or represent user requests. When such events occur, for example when the dispatcher has finished executing a task, the navigator runs the algorithm for deciding what task should be executed next. The navigator acts as a container for the process plugins, which embody a process specific version of the navigation algorithm. Upon receipt of events concerning a particular process, if necessary, the navigator dynamically loads the appropriate plugin.

2.2.2 Task Execution Scheduler

This component couples the navigator, generating task execution requests, with the dispatcher, which manages the actual task execution. In a distributed container (Figure 1), the scheduler receives task execution requests from a number of navigators and forwards them to a set of dispatchers. This is a key component concerning the scalability of the system, as its throughput limits the rate at which tasks can be executed.

2.2.3 Dispatcher

If the navigator is in charge of deciding what tasks should be started next, the dispatcher is the component which actually starts executing the tasks by dispatching them to the appropriate execution subsystem. In order to increase the navigator's throughput, the actual task startup operation has been decoupled from the navigation step which triggers it. This way, the navigator may asynchronously issue multiple task startup requests to the task execution scheduler, which queues and forwards them to one or more dispatcher components. Once the dispatcher receives a job it checks what the job's characteristics are and sends it to a matching execution subsystem. The current prototype contains mechanisms to execute jobs containing Unix programs, SOAP [26] requests, Java method calls, XML transformations, as well as sub-process invocations. Once the job's execution has completed, the dispatcher sends an event encapsulating its results to the navigator. [Figure 2 about here.]

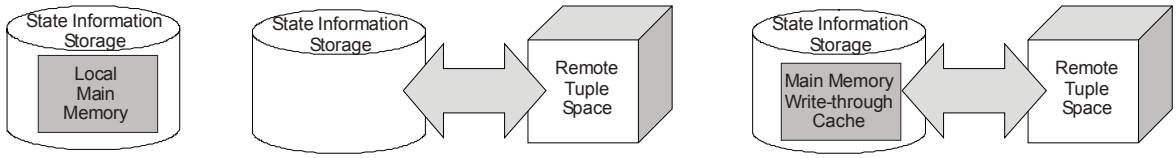
2.2.4 State Information Storage

This is the component responsible for storing the state information about the process instances. Its design has been influenced by many requirements, such as performance, reliability, and portability across different data repositories. The component's interface supports only a simple, key-value based data model, where the key has been structured as the following tuple `(Process, Task, Instance, Box, Parameter)` and is used to uniquely identify a certain data `Value` across the system. The definition of the key reflects the structure of the information to be stored: a process is composed of a set of tasks, of which there can be many instances. Each process/task instance has multiple parameters which are grouped into three boxes (or logical namespaces): system, input, and output.

The main advantages of this approach are summarized in the following arguments. First of all, since the information in the key is neutral with respect to the physical locations of the data, it becomes possible to transparently move the data to exploit locality and even replicate it among different physical repositories to improve its availability. Furthermore, the hierarchical nature of the key, suggests a natural data partitioning strategy. Another advantage is that changes and extensions to the data model of the processes' state information do not affect the

storage component, since this low level data representation is mostly independent from the data and metadata that needs to be stored [1].

Finally, as shown in Figure 2 the data layer can be implemented with a wide variety of mechanisms. These range from centralized memory based data structures (such as a hash map), to traditional forms of persistent storage (such as network file systems, or relational databases), or distributed storage systems (such as Linda-like tuple spaces [4] like TSpaces [17] or JavaSpaces [7])



(a) Volatile Storage (b) Remote Persistent Storage (c) Remote Persistent Storage with Cache

Figure 2: State Information Storage Implementations for the monolithic container

2.2.5 Event Queues

The various container components communicate by exchanging event notifications managed by the event queues. Sources for the events consumed by the navigator components are the user interface, other navigators and the dispatchers. Events are sent by the user interface in order to start, stop, and, in general, interact with a process instance. The dispatcher notifies the navigator with an event every time a task has finished its execution. Navigators also exchange events, for example, when a sub-process has completed its execution and navigation over the calling process, managed by a different navigator, needs to be triggered. The priority of these three classes of events can be adjusted. In a distributed container, event communication is also quite important concerning the system's scalability. We have been experimenting with several implementations of the event queues, each having different scalability properties.

First, as a reference, we used a single tuple space server to which all container components connect and exchange events by writing and taking tuples. As expected, this centralized event queue quickly becomes a bottleneck if all events sent by multiple dispatchers to a set of replicated navigators need to go through it. Therefore, in a second design, we chose to use a multi-layered approach, by distributing the event queue across all navigator components, with the following heuristic in mind: the navigator responsible for handling the incoming events should be kept as close as possible to the events themselves. This way, the dispatchers may directly send the "task-finished" events to the appropriate navigator. Events which are not sent to a specific navigator still go through the central queue, which, in this configuration, needs to handle relatively less traffic. For example, user generated process startup requests are queued centrally and the corresponding "start-process" events may be retrieved by idle navigators. To further reduce the communication overhead, events generated by a navigator which can be processed by the same navigator are kept locally and do not need to be sent over the network.

2.3 Parallel Navigation

We chose to parallelize the navigation algorithm based on the observation that each process instance is a fully independent entity. In particular, changes to the state of one instance do not affect other process instances. Therefore, it is possible to partition the system's workload at

the granularity level of the process instance and perform navigation on different, independent process instances in parallel. As a consequence, the navigation algorithm presented here doesn't need to be changed, since it can be implemented in a thread-safe manner. However, the underlying infrastructure needs to support the concurrent execution of the algorithm, triggered by events concerning independent process instances.
Once the system is capable of performing parallel navigation, issues such as load balancing and fault tolerance should be dealt with. In our current prototype we support two different load balancing strategies: either process instances can be statically partitioned among different parallel navigators (load sharing), or events and state information can be dynamically moved between different navigators in order to keep the system balanced. Moreover, since each navigational step is executed atomically within a transaction, and if the state information can be stored remotely and persistently, recovery from failures occurring in the navigator becomes completely transparent. In fact, when a dynamic load balancing strategy is used, upon detection of the failure, the process instances belonging to a failed navigator can be immediately assigned to another one.

	Event Queues	State Information Storage	Task Execution Scheduler	Dispatcher	Navigator
(a)	Local	Volatile	Local	Single	Single
(b)	Local	Persistent	Local	Single	Single
(c)	Centralized	Volatile	Remote	Multiple	Single
(d)	Centralized	Persistent	Remote	Multiple	Single
(e)	Distributed	Volatile	Remote	Multiple	Multiple
(f)	Distributed	Persistent	Remote	Multiple	Multiple

Table 1: Deployment Scenarios

3 Deployment Scenarios

In this section we present some of the configurations, listed in Table 1, in which the flexible architecture of the ADAPT container can be deployed and discuss the main advantages and disadvantages regarding their performance. Not only flexibility is an important aspect for performance reasons, but also it allows the system to be adapted to different requirements, so that it can be deployed into several environments and configurations to match a specific workload target. For example, our architecture can be deployed as a light-weight process simulation engine, attached to a process development tool. Similarly, it can be embedded into standalone Java applications that require process enactment capabilities to coordinate the invocation of different components and services. This way, the coordination logic specified as a process can be directly executed within the context of the application.
Alternatively, the ADAPT container can be used as a reliable service orchestration platform running inside an application server, which can also scale to handle very large workloads, using a cluster based configuration.
Furthermore, flexibility is one of the basic requirement of an infrastructure capable of exhibiting autonomic behavior. To this end, it is important that the system can be reconfigured dynamically following the decisions of an autonomic system controller, which monitors the current workload conditions and determines the optimal system configuration [3].

The simplest configuration (a) is a so-called *monolithic container*, where one navigator and one dispatcher run on the same machine. The state information storage, the event queues and the task execution services are implemented using the appropriate main memory data structures. Since all data is kept in main memory, this configuration trades recoverability from failures with very fast access to the state information. Given its centralized nature, such an architecture does not scale well with large workloads. In addition to the ease of deployment, its main benefit lies in its very low overhead with small workloads.

The next configuration (b) is called monolithic *persistent* container. Again, one navigator and one dispatcher run on the same machine, but the storage of the state information is implemented using a remote, persistent, data repository. This makes the container recoverable, at the cost of a larger overhead, as the results already shown in Figure 3 indicate. The limitations of these centralized configurations concern all five main system components: the Navigator, the Dispatcher, the Task Execution Scheduler, the State Information Storage and the Event Queues. If one is replicated in order to improve its throughput, very soon another component becomes a bottleneck. For example, if a set of navigators send task execution requests to the dispatchers through a centralized scheduler, the throughput of the scheduler limits the rate at which tasks can be executed. Similarly, if the performance of the state information storage improves, the navigator will be able to produce and consume events at a higher rate, putting a higher burden on the event queues. Thus, while scaling up the system and configuring it with replicated components (Figure 1), care must be taken to keep the system well balanced.

The first replicated configuration we present concerns the dispatcher component. In this case, a single navigator (with (d) or without (c) persistent storage) manages the processes, whose tasks are executed by an increasingly large number of dispatchers. As the task execution capacity of the system increases, it is to be expected that the system may be capable of handling a larger workload. As the measurements show, this is only true when the task duration is long enough, e.g. longer than 10 seconds. For tasks lasting a shorter time, the actual bottleneck lies in the navigator component.

This problem is addressed by the configurations (e) and (f), where also the navigator component is replicated, keeping the number of dispatchers and the corresponding task execution capacity constant. As our measurements indicate, the system's scalability is now bound by the persistent storage service. In fact, using a centralized data repository with an increasingly large number of clients (the navigators) only scales up to a certain limit. Therefore, we also tested a configuration (e) having the storage of the state information localized at the navigator.

Because of the improved performance of the storage service, the limiting factor shifted to the event communication service, which also had to be partitioned in order to keep the system functioning.

4 Measurements

The goal of the experiments is to analyze the performance of a significant subset of the deployment and configuration options described above. First of all we attempt to point out the scalability limits of a centralized system, where all the data storage, event and job scheduling services are implemented using the local main memory. Then we add external persistent

storage for the state information to determine what is the cost of adding persistence to the system. Then we replicate the dispatcher and navigator components, and observe the changes to the system's throughput.

4.1 Hardware Setup

The hardware and software setup for the experiments is as follows: the navigator and dispatcher container components were running on a cluster of dual Pentium-III 1000Mhz PCs with 1024 MB of RAM using Java 1.4.1 running on Linux v2.4.17. The three tuple space servers dedicated to state information storage, task execution scheduling and event communication were running each on separate dual Athlon 1.5Ghz with 1024 MB of RAM, Java 1.4.1 on Linux v2.4.19 and used the IBM's TSpaces implementation version 2.1.2 [12].

Variable	Values
Number of concurrent processes	1, 64, 128, 256, 512, 1024, 2048
Number of tasks	1, 10, 100
Task duration (seconds)	0, 1, 10, 30
Control flow topology	Sequential, Parallel, Matrix

Table 2: Workload Control Variables

4.2 Workload description

The behavior of the system is affected by the properties of the workload, which are defined by the control variables listed in Table 2. The number of processes indicates how large is the batch of concurrent processes to be executed. The size of a process is the number of tasks composing it. We used three different process sizes: 1, 10 and 100 tasks. Larger processes require more storage space and generate a higher number of jobs and events. The duration of the tasks affects the navigator's throughput, since the longer a task runs, the longer the delay between a job startup request and the corresponding termination event. During this time the navigator(s) may be free to process other events or have to remain idle. Finally, different topologies of the control flow of the processes generate different patterns of event exchanges. In the case of a process composed of a single task there are no degrees of freedom concerning the control flow, but as soon as the size of the process increases it is possible to connect the tasks in different ways. We have been testing our system with a variety of control flow graphs. In the case of ten tasks we used two topologies, one sequential, where the tasks are executed sequentially and a parallel one, where all tasks are executed concurrently. The same parallel topology has also been used with the larger process, composed of 100 tasks. In the case of a large process, we also tested a more complex control flow graph modeling a matrix-like computation.

4.3 Measured variables

First of all, we are interested in measuring the user perceived effect of the different configurations. This effect is measured by how long a process takes to complete. More precisely, we computed the average wall-clock time over all the concurrent processes of a certain batch.

Second, for every experiment, we recorded the batch execution time, this is how long it took to run an entire batch of concurrent processes. In the case of tasks running for 0 seconds, the execution time of the batch of processes can be used to compute the average throughput of the system, defined as the number of processed tasks per second. This value indicates the overall speed of the system in performing the operations (navigation, scheduling, running and results gathering) required to execute the tasks.
Third, in order to observe the system's internal behavior we instrumented the state information storage services to measure the time necessary to create the image of a new process instance. In our experience, this critical step is a potential performance bottleneck, since it is not possible to perform navigation until an instance has been created. We expected process instantiation to be expensive since, depending on the size of the process, (a lot of) information about the process, its tasks and their parameters needs to be written out to the state information storage service.

4.4 Results

4.4.1 Reliability Overhead

The limitations of centralized architectures can be illustrated by analyzing the performance of a centralized process support system. Such a system is built with a single component dedicated to process navigation, which uses a centralized repository to keep track of the state of the execution of the processes. As it has been often observed [16, 22], both centralization and persistence generate a significant overhead in process support systems under heavy workload.

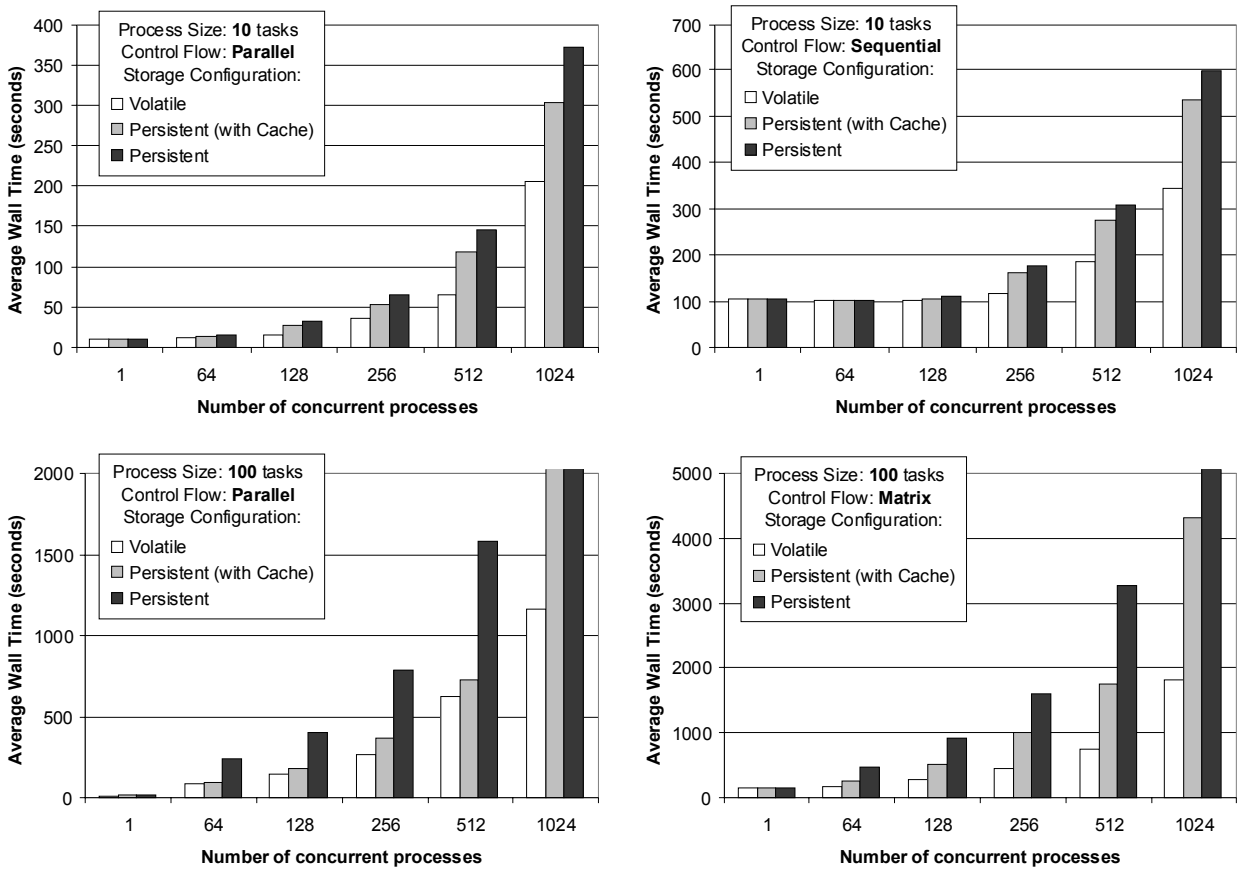


Figure 3: Performance degradation of a centralized process support system under increasingly large workloads

In Figure 3 we quantify the user perceived behavior of a centralized system while running four different types of processes. As the results show, the system's response time, i.e., the average wall-clock duration of a process, grows as a function of the system's workload defined as the number of processes running concurrently within the system. Relative to an unloaded system, where only one process at a time is executed, in the worst case the response time grows about 200 times when the workload size is increased thousand-fold.

The actual performance degradation depends both on the type and size of the processes and on the specific properties of the system's configuration (Figure 2). First of all, it can be observed that a relative performance improvement can be obtained by sacrificing the reliability of the system. In fact, using the local, volatile, memory of the process navigation component to store the processes' state information, can lead to response times up to 50% shorter than the time required to perform navigation over persistent state. As an attempt to combine the benefits of both configurations, we added a write-through cache located between the navigation component and the persistent storage. As the results indicate, a cache significantly reduces the penalty of using a remote storage service but still has limited scalability.

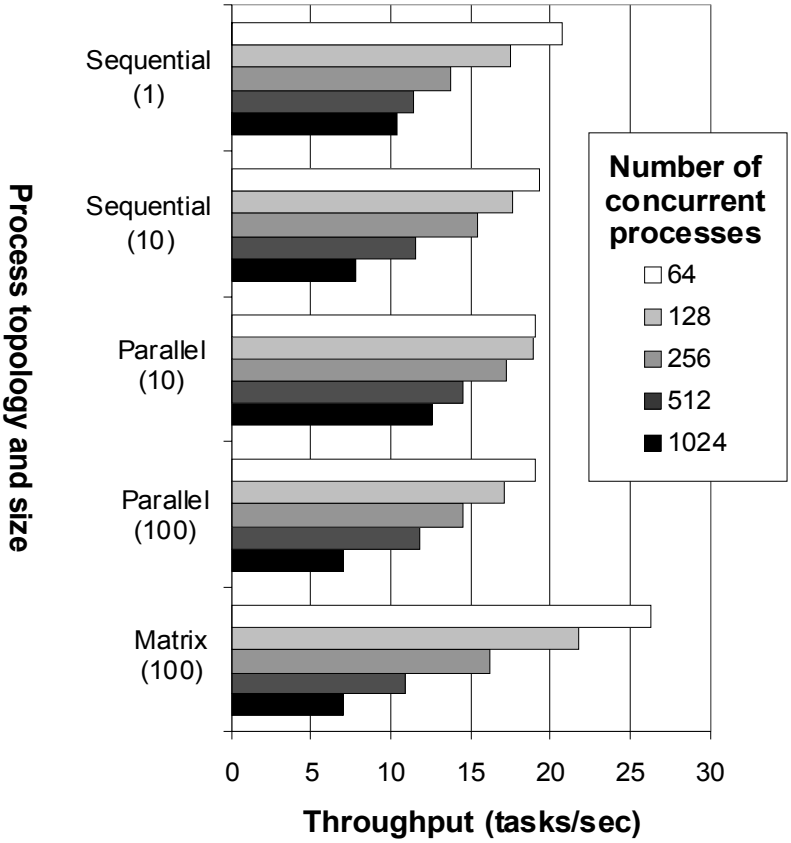


Figure 4: Throughput degradation of a centralized process support system under increasingly large workloads

4.4.2 Monolithic container

In addition to the results already presented in Figure 3 concerning the degradation of the response time of a centralized system under increasingly large workloads, we would like to display the corresponding throughput's degradation in Figure 4. This set of measurements has been performed with a monolithic container configured to use volatile storage and up to 64 threads for local task execution, i.e. its execution capacity is limited to 64 concurrent tasks.
For all process types, the maximum throughput is achieved when running the smallest workload. As the number of concurrent processes increases, the throughput decreases to a minimum. The actual degradation rate depends on the process topology, as the overhead of navigation is more important for larger and more complex processes.

4.4.3 Process Instantiation

Figure 5 displays the average process instantiation time as a function of the number of navigators, the size of the process and the configuration of the state information storage. As we expected, the instantiation time grows linearly with the process size: the higher number of tasks, the more information about them needs to be written to the data repository. The Figure also contains two interesting results. Not only is the instantiation time using persistent storage more than one order of magnitude longer than the time with volatile storage, but also, the volatile storage scales well with the number of navigators, since the process instantiation time remains constant. On the other hand, the performance of the centralized repository degrades as more and more navigators store in it data about their new processes. As it has been often suggested [16, 22], replicating the persistent storage would alleviate this problem. In all cases the instantiation time remains well below the 1 second boundary.

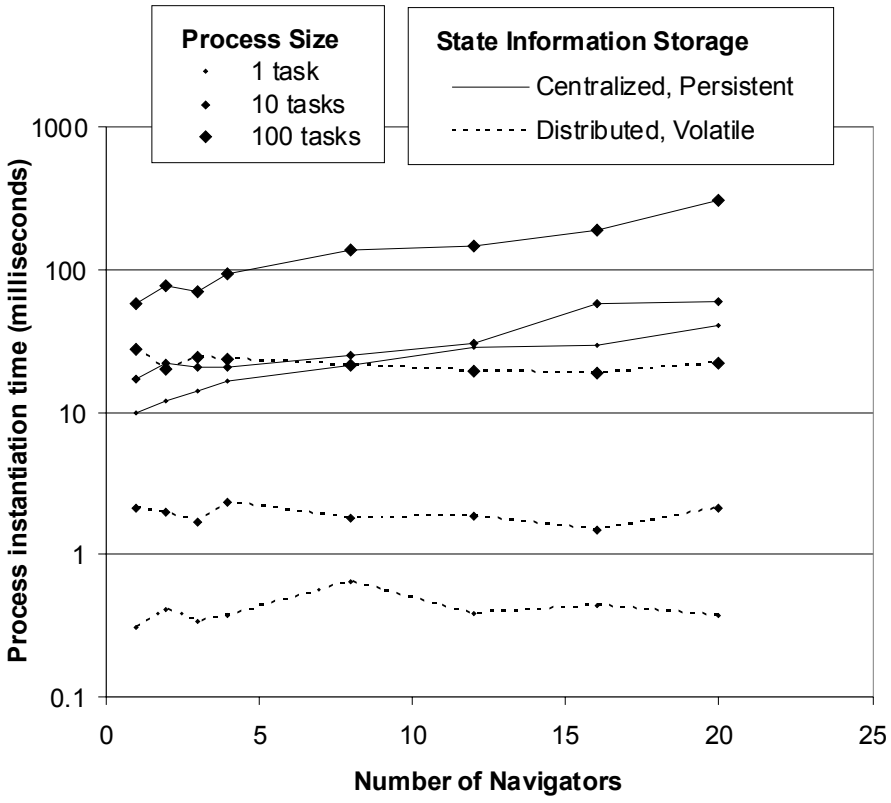
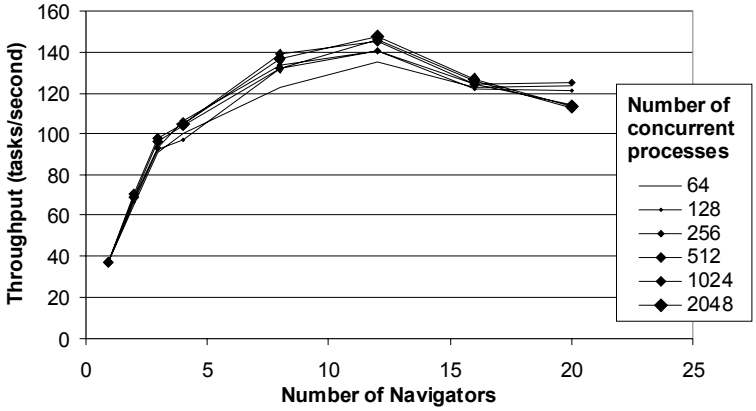


Figure 5: Scalability of the process instantiation

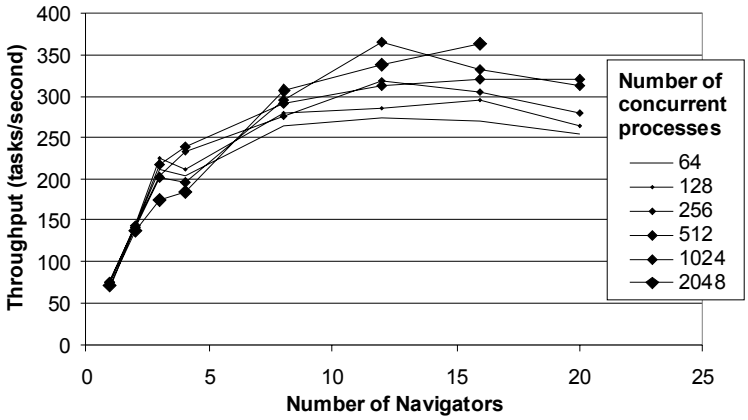
4.4.4 Scalable Process Navigation

Figure 6 shows the average system throughput with processes of 10 parallel tasks run with a variable number of navigators, 25 dispatcher components and different workload sizes. (a) In the case of persistent storage, for all workload sizes the throughput peaks at 12 navigators at about 140 tasks/second. This is a significant improvement with respect to a centralized system, especially considering that the throughput does not degrade as more and more processes run concurrently. In (b) the throughput actually improves as the workload size increases, indicating that, in the case of volatile storage, the performance of the replicated navigator does not saturate. Although the absolute throughput reaches about 350 tasks/second, this value is also obtained with 12 navigators, as the centralized task execution scheduler is the limiting factor of this configuration.

Figure 7 shows the system response time with up to 2048 concurrent processes of 10 sequential tasks run in the same settings as Figure 6. It can be observed that for small workloads, as the number of navigators increases the batch execution time approximates the time necessary to run only 1 process, which is close to 10 or 100 seconds, depending on the duration of the tasks. For larger workloads, the response time still grows linearly with the workload size, although the rate of increase can be controlled by changing the number of navigators. Using volatile storage the system scales well up to 20 navigators. Although the absolute response time is twice as high, the penalty of adding persistent storage is acceptable, as it shows good scalability up to 16 navigators accessing the same centralized data repository.



(a) Processes of 10 parallel tasks with persistent storage



(b) Processes of 10 parallel tasks with volatile storage

Figure 6: Scalable navigation: average throughput of the system using an increasingly large number of parallel navigators

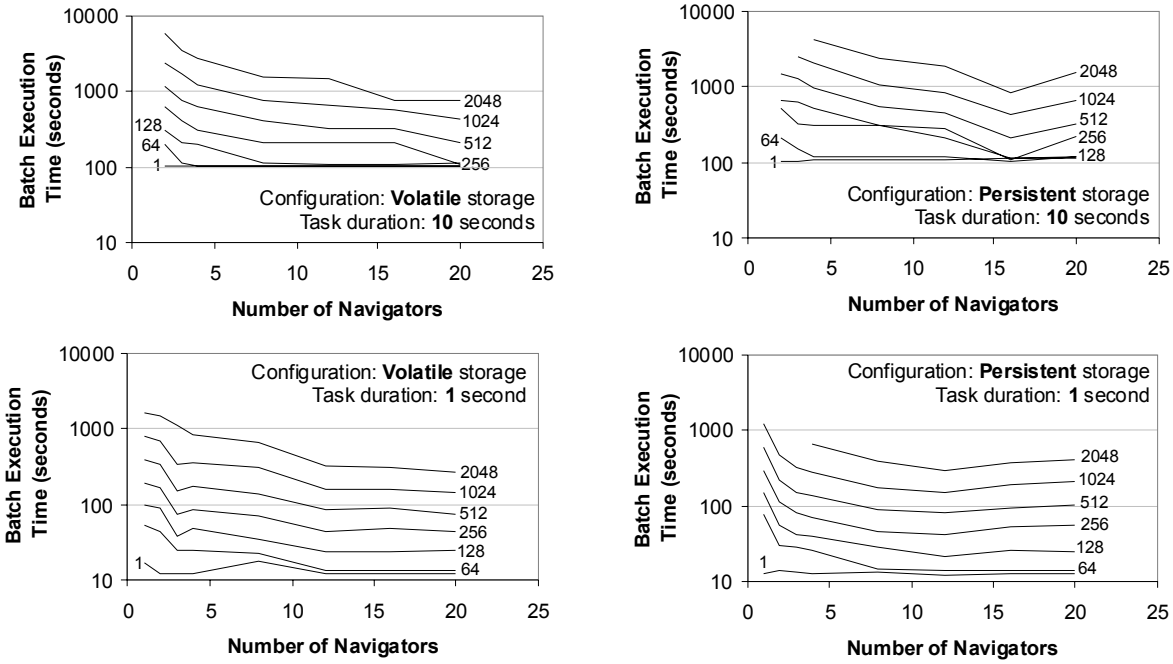


Figure 7: Scalable navigation: Execution time of batches processes having 10 tasks and sequential control flow topology as a function of 4 variables: 1) the number of navigators (X-Axis), the workload size (Z-Axis), the duration of the tasks and the system configuration (volatile or persistent storage)

5 Related Work

There is a wide range of commercial products and research projects dedicated to process management systems [8], especially regarding process modeling languages, with emphasis on flexibility [25] and transactional properties [21].
Relatively less work can be found about distributed architectures for scalable process execution. More specifically, scalability has been a common goal to be achieved through different means: replication at the database layer, distribution in the process execution engine and decoupled communication through events notification. Only rarely all of these approaches have been followed within the same project.

The idea of building a distributed workflow enactment system based on event communication and event-condition-action rules has been also proposed in the EVE project [9]. The exchange of event notifications plays an important role in our approach, however, in our experience, ECA rules are only a useful intermediate representation to bridge the gap between graph based models, which can be more readily understood by the user designing the process, and the corresponding executable code.

The theme of enhancing the system's fault tolerance and scalability through replication at the database layer has been pioneered by [16]. Also in the MOBILE project [11], in order to replicate the process execution layer, a scalable strategy for distributing the process data among separate databases has been proposed [22]. Although we compare the performance of a centralized, persistent repository with a distributed, volatile implementation, in this report we do not pursue replicated storage any further.

Decentralization has been pursued by the MENTOR project [30], where process definitions are analyzed and automatically partitioned among distributed execution sites in order to avoid

the bottleneck of a centralized engine [19]. This approach fits well with the requirements of workflows spanning across multiple organizations. However, it is possible for one execution site to become a hot spot, when it is involved in the execution of a large number of processes.

Once a distributed process architecture has been designed, load balancing, network congestion and quality of service guarantees become interesting options. In [15] a cluster-based workflow management system has been presented focusing on a quantitative comparison of two different load balancing strategies. In [2] simulations are used to study how different workloads influence the load of the network and thus, the scalability of the workflow engines in the context of several distributed architectures. In [10] extensive simulations are used to validate a composition model with quality of service guarantees based on service overlay networks.

6 Conclusion

In addition to our visual composition language, in this report we also presented a novel architecture for a process support system container, which can be used to execute processes written in such language. The main innovation of this architecture consists of the ability to transparently tailor the system's performance to different quality of service guarantees. By switching between different implementations of basic services, such as data storage, event communication and job execution, it is possible to deploy the same algorithm for process navigation in a variety of configurations, each with different properties regarding performance, scalability and reliability. In particular, we determine the cost of reliability by comparing navigation performed over volatile and persistent state. In this setting we also study the effect of caching. We also show that the system's throughput can be set to the desired level by performing navigation in parallel, when the container is replicated across multiple machines.

References

[1] Agrawal, R.; Somani, A.; and Xu, Y. Storage and Querying of E-Commerce Data. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*, Roma, Italy, 2001, pp. 149–158.

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

[3] Bausch, W. *OPERA-G - A Microcontainer for Computational Grids*. PhD thesis, Diss. ETH Nr. 15395, (December 2003).

[4] Carriero, N. and Gelernter, D. *How to write parallel programs*. MIT Press, 1990.

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

[6] ebXML. *ebXML Business Process Specification Schema (BPSS) 1.01*, (2001). `http://www.ebxml.org/specs/ebBPSS.pdf`.

[7] Freeman, E.; Hupfer, S.; and Arnold, K. *JavaSpaces: principles, patterns and practice*. Addison Wesley, 1999.

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

[9] Geppert, A. and Tombros, D. Event-based Distributed Workflow Execution with EVE. Technical Report 96.05, Dept. of Computer Science, University of Zurich, (1998).

[10] Gu, X.; Nahrstedt, K.; Chang, R. N.; and Ward, C. QoS-Assured Service Composition in Managed Service Overlay Networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003, pp. 194–201.

[11] Heinl, P. and Schuster, H. Towards a Highly Scaleable Architecture for Workflow Management Systems. In *Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, 1996, pp. 439–444.

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

[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] IBM, Microsoft, and BEA Systems. *Web Services Coordination (WS-Coordination)*, (2002). `http://www.ibm.com/developerworks/library/ws-coor`.

[15] jie Jin, L.; Casati, F.; Sayal, M.; and Shan, M.-C. Load Balancing in Distributed Workflow Management System. In *Proceedings of the ACM Symposium on Applied Computing*, 2001, pp. 522–530.

[16] Kamath, M.; Alonso, G.; Guenthoer, R.; and Mohan, C. Providing High Availability in Very Large Workflow Management Systems. In *Proceedings of the 5th International Conference on Advances in Database Technology (EDBT'96)*, Avignon, France, 1996, pp. 427–442.

[17] Lehman, T. J.; Cozzi, A.; Xiong, Y.; Gottschalk, J.; Vasudevan, V.; Landis, S.; Davis, P.; Khavar, B.; and Bowman, P. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35, 4, (March 2001), 457–472.

[18] Leymann, F. *Web Services Flow Language (WSFL 1.0)*. IBM, (2001).

[19] Muth, P.; Wodtke, D.; Weissenfels, J.; Dittrich, A.; and Weikum, G. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10, 2, (1998), 159–184.

[20] Oasis. *Universal Description, Discovery and Integration of Web Services (UDDI) Version 3.0*, (2002). `http://uddi.org/pubs/uddi v3.htm`.

[21] Schuldt, H.; Alonso, G.; Beeri, C.; and Schek, H.-J. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27, 1, (2002), 63–116.

[22] Schuster, H. and Heinl, P. A workflow data distribution strategy for scalable workflow management systems. In *Proceedings of the 1997 ACM symposium on Applied computing*, 1997, pp. 174–176.

[23] Thatte, S. *XLANG: Web Services for Business Process Design*. Microsoft, (May 2000).

[24] van der Aalst, W. M. P. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18, 1, (2003), 72–85. [25] van der Aalst,W. M. P. and Berens, P. J. S. Beyond workflow management: product driven case handling. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, 2001, pp. 42–51.

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

[27] W3C. *Web Services Definition Language (WSDL) 1.1*, (2001). `http://www.w3.org/TR/wsdl`.

[28] W3C. *Web Services Choreography Interface (WSCI) 1.0*, (2002). `http://www.w3.org/TR/wsci`.

[29] W3C. *Web Services Conversation Language (WSCL) 1.0*, (2002). `http://www.w3.org/TR/wscl10`.

[30] Wodtke, D.; Weissenfels, J.; Weikum, G.; and Kotz-Dittrich, A. The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In *Proceedings of the 12th International Conference on Data Engineering (ICDE 1996)*, New Orleans, Louisiana, 1996, pp. 556–565.