Decentralized Web Service Orchestration: A Reflective Approach

R. Jiménez-Peris,
Facultad de Informática
Universidad Politécnica de Madrid (UPM), Spain
rjimenez@fi.upm.es

M. Patiño-Martínez
Facultad de Informática
Universidad Politécnica de Madrid (UPM), Spain
mpatino@fi.upm.es

E. Martel-Jordán
IUMA-Universidad de Las Palmas de Gran Canaria, Spain
emartel@iuma.ulpgc.es

ABSTRACT

Web service orchestration is widely spread for the creation of composite web services using standard specifications such as BPEL4WS. The myriad of specifications and aspects that should be considered in orchestrated web services are resulting in increasing complexity. This complexity leads to software infrastructures difficult to maintain with interwoven code involving different aspects such as security, fault tolerance, distribution, etc. In this paper, we present ZenFlow a reflective BPEL engine that enables to separate the implementation of different aspects among them and from the implementation of the regular orchestration functionality of the BPEL engine. We illustrate its capabilities and performance exercising the reflective interface through a decentralized orchestration use case.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Throughput

Keywords

Web Service Orchestration, BPEL4WS, Reflection, Composite Web Services, Decentralized Orchestration

1. INTRODUCTION

Web services are the preferred technology for building Service Oriented Architectures (SOA) and Enterprise Application Integration (EAI). Web service orchestration is playing an important role to enable the creation of complex composite web services. Specifications such as BPEL4WS [19] are widespread. The myriad of specifications and aspects that should be considered in orchestrated web services result in high complex programs and sophisticated infrastructures difficult to maintain with interwoven code involving different aspects such as security, fault tolerance, distribution, etc.

Reflection [24] and aspect oriented programming (AOP) have been advocated as powerful approaches for separating the different aspects of implementations and attaining an effective separation of concerns. AOP focuses on providing programming constructs to attain low level concern separation at programming time. Reflection is an approach at the architectonic level in which a system is architected so its behavior can be observed and modified through an interface called meta-interface. This meta-interface enables to encapsulate the implementation of non-functional aspects as separated components. This makes possible the extension of the reflective system without any modification to the code. The use of reflection has been advocated for dynamic web services [33].

In this paper, we present ZenFlow, a reflective BPEL web service orchestration engine [25]. Its reflective capabilities enable to implement all non-functional aspects in a separated manner reducing the complexity and increasing the maintainability and modularity of the BPEL engine. Reflection may be used to introduce transactional semantics, decentralized orchestration, fault tolerance, debugging and monitoring capabilities, etc. In this paper, we choose as case study decentralized orchestration. Decentralized orchestration enables to cope with higher loads, increasing the scalability with respect to centralized orchestration. An exhaustive experimental evaluation has been performed to demonstrate the superior scalability of the decentralized execution implemented in a reflective manner. The evaluation shows that the reflection overhead is very low and does not impact the performance of the system.

The rest of the paper is organized as follows. Section 2 reviews computational reflection. The implementation of the reflective BPEL engine of ZenFlow is presented in Section 3. Section 4 presents a throughout evaluation of both the performance benefits of the decentralized process execution using reflection, and the cost of reflection itself. Related work is summarized in Section 5. Finally, conclusions are presented in Section 6.

2. COMPUTATIONAL REFLECTION

Reflection refers to the ability of a system to reason about and act upon itself. More specifically, a reflective system is one that provides a representation of its own behavior

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

which is amenable to inspection and adaptation, and is casually connected to the underlying behavior it describes. *Casually-connected* means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior, and vice-versa. It can therefore be said that a reflective system is one that supports an associated *casually connected self-representation* (CCSR). This definition was first used by Maes [24] in the context of programming languages.

According to the reflection paradigm, a system is structured in two layers: the *base-level* and the *meta-level*. The former executes the application components, whereas the latter runs the components devoted to the implementation of non-functional requirements (e.g. security, fault-tolerance, tracing). The base-level provides an image of the structural and behavioral features to the meta-level. This image, called *meta-model*, is casually connected to the base-level. That is, any change in one of the levels leads to the corresponding change upon the other.

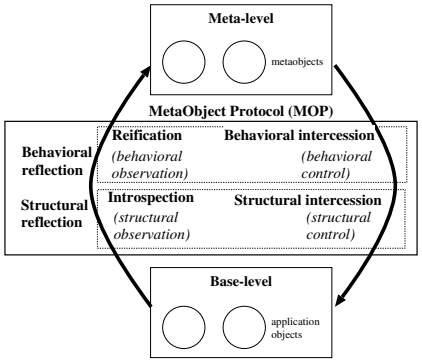


Figure 1: Architecture of a reflective system

Four methods may be used to observe and control at the meta-level the features of the system's base level (Figure 1). *Reification* corresponds to the process of exhibiting to the meta-level the occurrence of base-level events. *Introspection* allows the meta-level to retrieve the base-level structural information. *Intercession* enables the meta-level to act on the base-level behavior.

3. A REFLECTIVE BPEL ENGINE

ZenFlow is a programming environment programmed in Java for the development and deployment of BPEL processes. One of the components of ZenFlow is a BPEL interpreter or engine. The BPEL engine reflective architecture exhibits a metalevel called *meta-interpreter*, which enables the engine extension by means of metaobjects.

Metaobjects are objects that enable adding new features to the BPEL engine. These methods allow specifying the code to be executed before and after the interpretation of every activity of a BPEL process: the *pre method* of a metaobject includes the actions to be executed before an activity is interpreted, whereas the *post method* of a metaobject refers to the actions to be executed after the activity interpretation. The *pre method* returns true if the activity of the business process will be invoked or false otherwise. In this way, the meta-level can cancel the execution of activities in the base-level.

The metaobjects may have different scopes in ZenFlow:

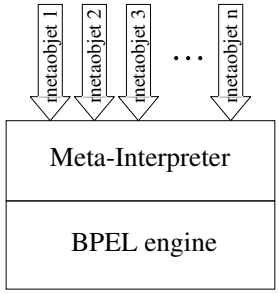


Figure 2: Reflective BPEL engine

(a) *A metaobject is associated to all the activities of a business process.* The *pre* and *post methods* are invoked for all the activities which are interpreted during the business process execution. This scope is, for instance, useful to implement monitoring facilities.

(b) *A metaobject is associated to a certain type of activity of a business process.* In this case, the metaobject is invoked when that type of activity is interpreted. For example, a metaobject may be associated to invoke activities to perform semantic mediation.

(c) *A metaobject is associated to a single activity.* The metaobject is invoked when the selected activity is interpreted. This might be used, for instance, for debugging a business process.

Another important set of meta-interfaces are those providing *introspection and intercession capabilities* over the business process structure and state. These meta-interfaces allow retrieving the current catalog of published and running business processes. They also allow navigating the structure of a published business process and change it. This can be done for the business process template and/or for running instances. The latter enables *workflow evolution*. When navigating the business process instance, the state of the process is also accessible and modifiable. An additional set of meta-interfaces deal with life-cycle activities such as the creation of new business process instances, destruction, etc. They enable associating meta-objects to this kind of activities that are not specific to concrete business processes.

Finally, there is a set of meta-interfaces that are accessible remotely (through RMI) to enable the distributed interaction of meta-objects across BPEL servers. These meta-interfaces enable intercession and introspection in a remote fashion, like for instance, injecting a business process instance at another site running an instance of ZenFlow.

An important issue in ZenFlow is that the cost of reflection is only paid when metaobjects are associated to activities. For activities without metaobjects associated there is no cost derived from reflection. The reflection provided by ZenFlow therefore effectively provides *partial behavioral reflection* [32].

3.1 Reflective Approach to Decentralized Execution

Building a reflective architecture for providing decentralized execution of a BPEL process requires controlling and adjusting at runtime the behavior of the system. Let us consider a simple execution of a business process which has

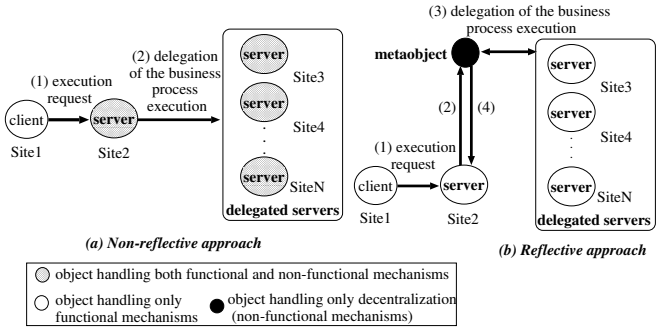


Figure 3: Non-reflective approach vs. reflective approach

been configured for decentralized execution. The client requests the execution of the BPEL process and this execution is delegated to the corresponding remote site. Using a conventional non-reflective approach (see Fig.3-(a)), the server code has to implement both the interpretation of the business process and the decentralized execution. This results in a tightly coupled implementation mixing functional (BPEL interpretation) and non-functional aspects (decentralized execution) of the BPEL server lacking separation of concerns and leading to a hardly maintainable implementation.

Using a reflective approach, metaobjects are responsible for delegating the execution to a remote site. Figure 3-(b) provides a high level view of the reflective approach. The client first starts the execution of a business process. If the process is configured to be executed at a different site, this event is intercepted at the metalevel (step 2). Then, the metaobject extracts the state of the business process, and sends it to the remote server through the remote RMI meta-interface (step 3). The remote server takes this state, recreates the process locally through the intercession meta-interface and resumes the execution of the recreated process. When the execution finishes, that site extracts once again the process state which is returned to the metaobject that installs the new state in the server to reflect the result of the remote execution (steps 4).

The above decentralized execution illustrates the use of reflection made in the decentralized execution implementation:

- *Reification.* Whenever the meta-interpreter detects a decentralized execution during the interpretation of a business process, the meta-interpreter activates the metaobject designed for decentralized execution. This metaobject is responsible for delegating the execution to the adequate remote site.

- *Introspection.* The metaobject needs the structure of the business process, the state of the execution flow and the current state of variables. This information constitutes the state of the business process which is retrieved from the base-level.

- *Intercession.* As a result of the execution delegation, a process instance is created at the remote site and the received state is injected into this instance.

ZenFlow checks whether the delegated activities can be coherently executed in a decentralized fashion. Basically, it is checked whether parallel flows to be executed in a decentralized manner do not share state. That is, if a branch writes a variable no other branch reads or writes it. In that way, during decentralized execution no inconsistencies arise, and the resulting state from the different branches can be merged consistently.

BPEL provides *links* to express synchronization dependencies between activities that are nested directly or indirectly within a flow activity. The *meta-interpreter* checks the relation of the current activity with the links within the flow activity. In a decentralized setting, the source and target activities of a link can be at different servers. The metaobjects for decentralized execution take care of notifying link conditions across servers if necessary.

Since the decentralized execution is transparent to clients of a BPEL process, a client can send a message to the process but the process now is executing at a different site. A metaobject checks whether the message corresponds to a receive activity that is executed locally or remotely. In the latter case, the message is forwarded to the corresponding delegated remote ZenFlow instance. Reply activities (synchronous invocation) are dealt with similarly.

Exceptions are also dealt with in decentralized execution. If an exception is thrown and not handled by the piece of code whose execution is delegated to a remote site, the exception is propagated to the server site. If there is no code for handling the exception at the server side, that site will throw it back to the client, as in a centralized execution of the BPEL process.

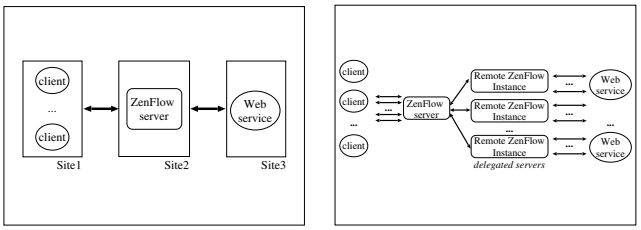
4. EVALUATION

In this section we evaluate the cost of the reflective features of ZenFlow. Decentralized execution is an adequate way to demonstrate the effectiveness and power of our reflective approach due to: 1) It can show that the potential scalability to be gained through decentralized execution is not lost due to the use of reflection; 2) It shows that sophisticated non-functional aspects can be implemented relying exclusively on the meta-interface achieving a full separation of concerns. For this purpose we have conducted a set of experiments that measure the benefits of decentralized execution. The centralized execution of the equivalent experiment is used for comparison purposes. In the centralized execution, clients run on one site, a single ZenFlow server runs the BPEL process (Fig.4-(a)), which runs on a different site. The BPEL process invokes one or more web services that run on different sites. In the *decentralized scenario* the whole process or part of it is delegated to another ZenFlow instance (*delegated servers*), which runs on different sites (Fig.4-(b)). The BPEL processes have only one synchronous client invocation. The web service invoked by the BPEL process in most of the experiments is a dummy service which receives a string and returns it back.

All experiments were run on a cluster of PCs equipped with two AMD Athlon(tm) MP 2000+ processors, 1GB of RAM, running Linux (Kernel version 2.6.9) and Sun's JDK version 1.5.0. Sites are connected by a 100Mb/s LAN. The total number of requests submitted is 3,200. Only the 3,000 central requests, corresponding to the steady state, are considered for the measurements.

4.1 Reflection Overhead

In this experiment, the cost of the reflective BPEL engine



(a) Centralized (b) Decentralized

Figure 4: Scenarios for ZenFlow evaluation

is evaluated in terms of response time. For this purpose we use a BPEL process with a while activity that contains an assign activity. There is one meta-object that is executed before and after the assign activity. The meta-object methods are empty, so its invocation reflects exclusively the cost due to reflection. The number of iterations enables to measure the reflection overhead. The same experiment is run without any meta-object to quantify the incurred overhead.

Figure 5 shows that the reflection overhead is negligible. For 200 iterations the overhead is 14.7 ms. For 10,000 iterations, this overhead is just 652 ms. The overall response time for the latter case without meta-objects is 50,100 ms. So, relative overhead is just 1.3%.

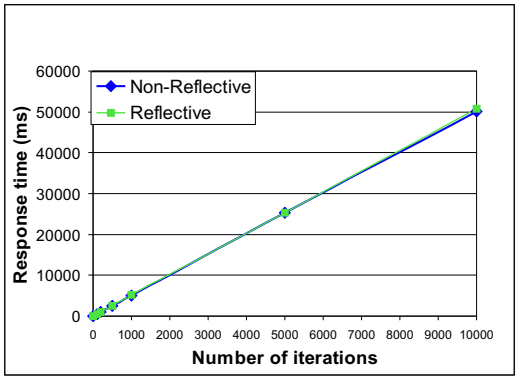


Figure 5: Reflection cost

4.2 Scalability

The goal of this experiment is to measure the *scalability* of ZenFlow with decentralized execution. The scalability is measured by looking at the evolution of performance for an increasing load (# of clients). For this purpose we use a simple BPEL process that calls 10 times a web service. One ZenFlow instance receives client requests (ZenFlow server), creates an instance of the process and delegates the execution to one of the other ZenFlow instances using a round-robin algorithm to select the site (Fig.6). As baseline we use a centralized ZenFlow.

Three sites (client, BPEL server, invoked web service) are used for the centralized execution (Fig.4-(a)). Three (resp. six) more sites are used when the execution is delegated to three (resp. six) ZenFlow instances (Fig.6).

Figure 7-(a) shows the response time for an increasing load. For low loads (5 clients) the response time is lower for the centralized execution than for the decentralized execution. Since decentralized execution involves some communication cost with the site where execution is delegated to, the response time is slightly higher. When the load increases (till 15 clients) all configurations show a similar response time what means that the latency overhead of decentralization becomes negligible for medium-high loads. When the load increases in the centralized configuration, the response time also increases smoothly till 25 clients and then deteriorates very fast (200 ms for 25-30 clients and 400 ms for 45-50 clients). A single ZenFlow instance is not able to process more than 50 concurrent clients, although response time deteriorates with 25 clients. On the other hand, the response time increases smoothly with three sites till 50 clients (the maximum load a single ZenFlow server can cope with). Then, the response time increases but at a lower pace than in the centralized execution (100ms increase from 80 to 90 clients). The reason for this lower increase is that the load is shared among three sites, so the system saturates more slowly. The same situation happens when the load is shared among six ZenFlow instances. The response time increases smoothly till 80 clients, and then it increases at a lower pace than with the other two configurations.

Figure 7-(b) shows the throughput as the number of client requests served per minute. All configurations increase their throughput with an increasing load. They do at the same pace till they saturate, then the throughput increases more slowly till no more load can be handled and the ZenFlow server(s) collapses. The different configurations saturate with 25, 60 and 120 clients, respectively, which corresponds to a high increase in the response time. The centralized configuration is able to reach a maximum of 600 requests per minute. 3 servers reach a throughput of 1,500 requests per minute and 2,700 with 6 servers. This shows a reasonable scalability for small cluster sizes.

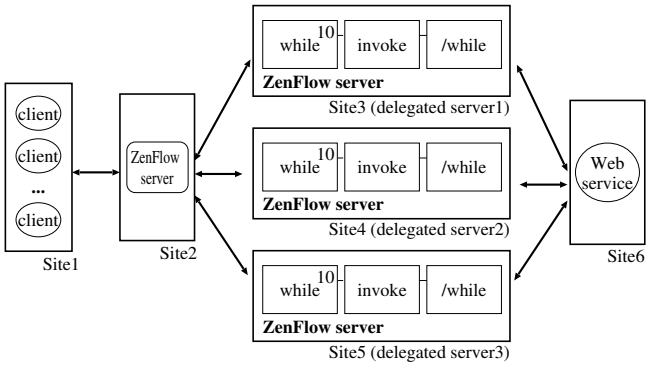
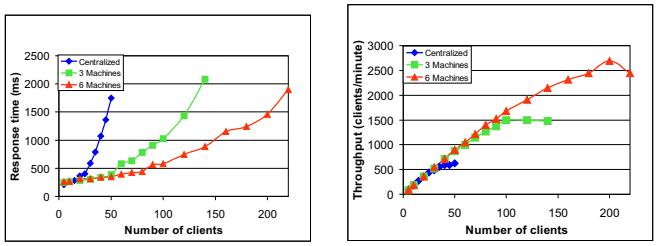


Figure 6: 3 delegated servers with round-robin



(a) Response time (b) Throughput

Figure 7: Scalability

4.3 Delegation Cost

In this section, we study the cost of delegating the execution of a process in order to get a deeper understanding on the scalability of the system.

4.3.1 Communication and Instantiation Cost

In this experiment we evaluate the cost of sending the state of a process to be executed at a different site. Once the information is received (transmission time in Fig.9), an instance of the process must be created and then, the execution of the process can start. We also evaluate the time needed to recreate and start the execution for an increasing process size (processing time in the graph). In this experiment, there is a single client that sends one request to ZenFlow, which delegates the process execution to a remote ZenFlow instance. The state size ranges from 20 KBytes till 2.2 MBytes, at the same time the size of the instantiated process increases. The process has a sequence with 50 assign activities followed by an increasing number of flow activities (0, 20, 40, 60, ...) with five branches each one (Fig.8).

We can observe in Fig.9 that for the smallest size (20 KBytes), the transmission time is almost negligible (4.8 ms) and all the time is consumed in the instantiation of the process (26.5 ms). The situation changes with a large state size, 2.2 MBytes, in this case the transmission time grows to 450 ms. Regarding the time needed to instantiate the process and start execution (processing time) it takes 80 ms for a process with 20 flow activities (size 400 KBytes) and 330 ms for a process with 100 flow activities (2,150 Kbytes).

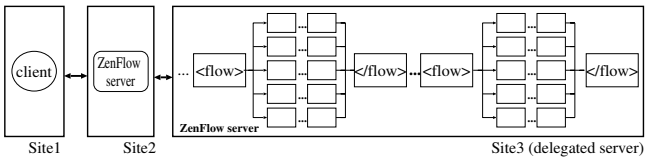


Figure 8: Decentralized setup for communication and instantiation costs

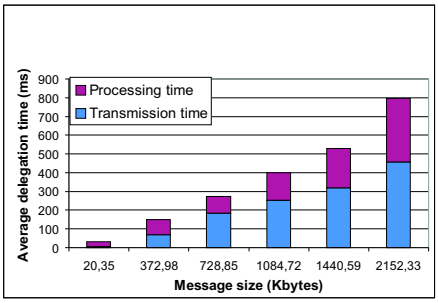


Figure 9: Communication and instantiation cost

4.3.2 Process Execution Cost

In this experiment we measure the trade-off between the process execution time and the cost of delegating the process to a another site. The BPEL process consists of a flow activity with two identical branches that invoke a web service (Fig.10). Each of the branches executes on a different site. The number of times each branch (while activity) invokes the web service ranges from 1 to 20 yielding an increasing process execution time. The experiment runs either on four sites, when the execution is delegated, or in three sites, when there is a single ZenFlow instance. The test has been run for an increasing load in order to study the effect on concurrency.

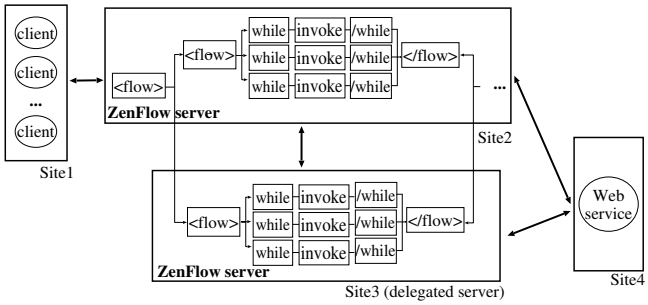


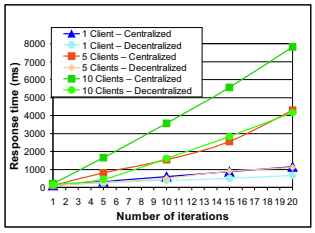
Figure 10: Decentralized scenario. Execution cost

Figure 11-(a) shows that with one client, the response time for the centralized execution is better than the decentralized one, when the web service invocation in the while activity is executed once. If the number of iterations in the while activity increases, even for a single client, the decentralized version shows a better response time (e.g., the response time for centralized is 1,200 ms and 660 ms for the decentralized with 20 iterations). This difference starts earlier when the number of clients increases (10 iterations for 5 clients and 5 iterations for 10 clients). This means that with a low load and light processes a single centralized ZenFlow instance executes faster than with delegation. As soon as the execution time of the process increases, delegation offers a better response time. This situation happens earlier with higher loads.

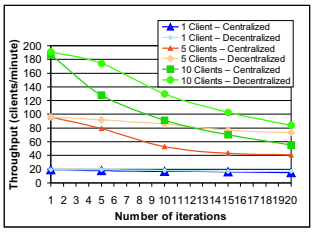
For one client the throughput of both executions is almost the same, even with a high number of iterations (Fig.11-(b)). Therefore, both configurations process the same number of requests per minute although, the response time (what a single client observes) is quite different. Therefore, we can say that the decentralized ZenFlow performs better than the centralized one. When the number of clients increases, the difference in terms of throughput is noticeable even for light processes (5 iterations), and increases for heavier delegated processes.

4.4 Parallel Execution

In this experiment, we evaluate the performance benefits of executing each branch of a flow activity at a different site. The process used in the experiment has a flow activity with



(a) Response time



(b) Throughput

Figure 11: Process execution cost

3 parallel activities: a while activity that invokes 10 times a web service (Fig.12). 6 sites are used in the evaluation. Each activity (branch) of the flow activity is executed by a different instance of ZenFlow at a different site. Centralized execution at a single site is used for comparison purposes. The response time and throughput have been measured for an increasing load. Figure 13-(a) shows the response time in both scenarios. Initially, a single instance of ZenFlow can handle the injected load and therefore, the response time is better than in the parallel execution. However, as the load increases, the centralized server saturates, and the response time increases. The parallel server shows a more graceful degradation for high loads.

Regarding throughput (Fig.13-(b)), the parallel server produces a better throughput than the centralized execution event with very low loads. When the load increases (30 clients), the throughput with delegation doubles the one of the centralized execution.

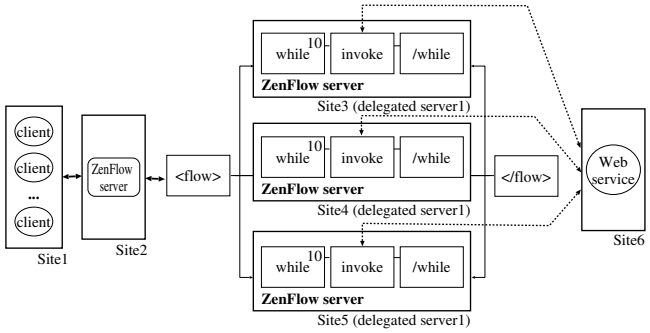


Figure 12: Parallel execution

4.5 Locality

If a web service (or set of web services published by a particular site) is invoked several times from a process, several messages are sent between the site running the process and the site where the web service is deployed. This can affect the performance of the whole process, even if the exchanged data is not large. In this experiment, we execute the BPEL code that accesses a given web service at the same site where the web service resides. That is, the process is moved to the site where the web service it invokes runs. This might not be feasible in general but, in the context of a single organization (e.g. when using web services for Enterprise Application Integration, EAI) it may be possible. With this, the network is traversed only once and can reduce the network hops and consumed bandwidth between the process and the invoked service.

The process used in the experiment has a flow activity with 3 parallel activities: a while activity that invokes 10 times a web service. Each activity (branch) of the flow activity is executed by a different instance of ZenFlow at a different site (Fig.14). So, the top while activity runs at site 3, the middle one runs at site 4, and the bottom one at site 5.

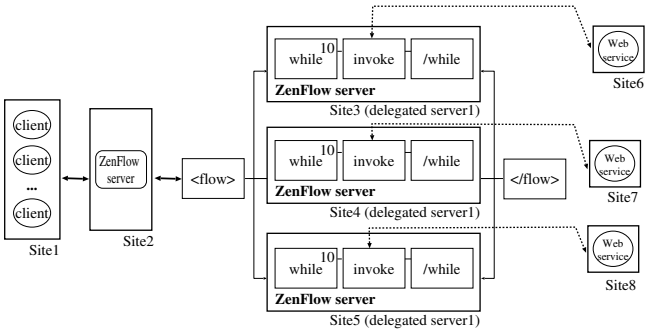


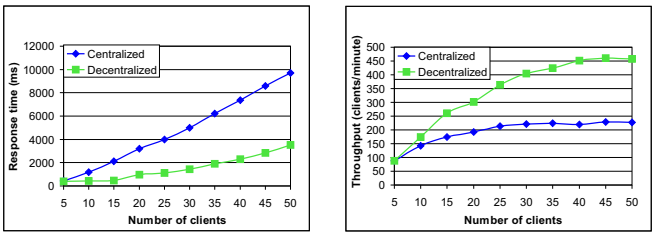
Figure 14: Collocated orchestration

Figures 15-(a) and 15-(b) show the benefits of collocating the fragments of a composite web service with the invoked web services. The response time of the decentralized execution is less than half than the one of centralized execution. The state to be transferred for each fragment was set to 86 KBytes (a medium state size). With a smaller state the performance benefits would have been substantially bigger.

The reasons for the performance improvement are on one hand that the execution is parallelized. On the other hand, when the fragment performs several invocations to web services deployed at the same site, many network hops are saved. In fact, this can be seen as a form of batching [4]. This situation can be common in EAI and data grids, in which segments of a workflow perform a series of interactions with web services located at a particular site of the enterprise.

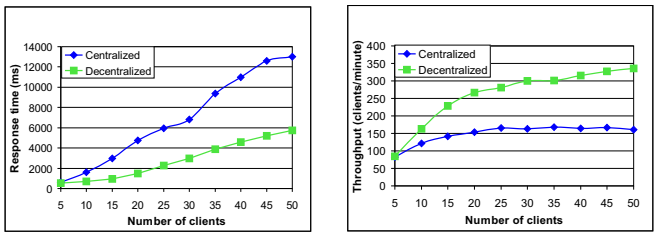
5. RELATED WORK

Papers [28], [22] and [9] suggest the use of Aspect-Oriented Programming (AOP) in order to improve the evolution of web service technologies. [28] focuses on the conjunction of AOP and WS-Policy to decouple the non-functional capabilities at description and implementation level. [22] uses AOP to achieve web service adaptation. [9] presents an aspect-oriented extension to BPEL in order to make web service composition more modular, flexible and adaptable. AOP can be provided on top of reflective systems [30]. The discussion is made in the context of object oriented pro-



(a) Response time **(b) Throughput**

Figure 13: Parallel execution of flow activities



(a) Response time **(b) Throughput**

Figure 15: Locality

gramming languages such as Java. In ZenFlow reflective facilities also enable dynamic aspect weaving, by selecting which metaobjects to activate and when.

[32] proposes partial behavioral reflection to reduce the overhead of reflection and activate it only when needed on a dynamic basis. The idea is based on the notion of hooksets that enable static and dynamic reconfiguration of behavioral reflection. ZenFlow reflection is similar to partial reflection in the sense that only actual activated meta-objects do have a cost.

Reflection has been used to achieve configurability and reconfiguration in the middleware domain ([10], [11]), to guarantee platform independence ([7],[14]), and to add fault-tolerance [31, 27]. ZenFlow exploits reflection offering a similar degree of extensibility and reconfigurability but on a different domain, web service orchestration, where to the best of our knowledge there has not been previous work.

The available BPEL engines (ActiveBPEL[1], AgilaBPEL [2], CapeClear [3]) do not support mechanisms for expressing crosscutting modularity. The extension of such engines need the modification of their code, leading to tangled and scattered code involving both the functionality of the engine and different non-functional aspects. This difficults the maintenance and evolution of the engine because changes affect to several parts of the engine. In [9] this problem is solved at the programming level but a reflective variable is used whenever properties of the process are needed during execution. We have tackled the problem at the architectonic level of the BPEL engine by only using reflection. Although in this work we has focused on the decentralization non-functional aspect, the presented approach is also extendible to other non-functional aspects. The necessary actions to achieve additional aspects implies building a metaobject per non-functional aspect, and attaching it to the process definition, without altering the core code of our BPEL engine. Both AOP and reflection constitutes different approaches to solve crosscutting modularity but at different level.

Exception handling is an important issue in workflows and composition of Web Services. Papers [12, 15, 35, 18, 23] present exception management frameworks for business processes to reduce redundancies in exception processing code. They propose different techniques such as exception handling templates, assertions and *meta workflows* to reduce error handling code. In [13] exception handling is used to deal with the recovery aspect of survivability in the context of BPEL. In this paper we have not dealt with reducing redundancies of exception handling code. However, we have treated exception handling in decentralized settings guaranteeing that the original BPEL semantics is preserved despite decentralization.

Visual composition tools have been proposed such as Self-Serv [6]. An earlier paper on ZenFlow focuses on its visual capabilities [25]. Both Self-Serv and ZenFlow supports decentralized execution of composite web services but using different approaches: Self-Serv exploits peer-to-peer orchestration model [5], whereas ZenFlow uses the reflection approach for the orchestration. In [8] an Integrated Development Environment (IDE) for Composition of Web Services, Synthy, is presented. This IDE is based on two staged service composition approach that separates the functional and non-functional requirements of the service being composed. The non-functional stage exploits Semantic Web Technologies and, the functional stage uses some distributed programming techniques (like BPEL and WSDL). ZenFlow focuses in this last stage of Synthy providing decentralized execution through reflection.

The idea of partitioning business process to provide decentralized orchestration has been considered in [26, 21]. The authors focus on a technique to partition a composite web service written as a single BPEL program into an equivalent set of decentralized processes. The proposed technique gives a new code partitioning algorithm that is applicable to decentralization of composite web services. In [26] experimental results are presented to demonstrate that decentralization increases the throughput of composite web services substantially, even under high loads. [34] presents a workflow management system for distributed enactment of composite services (DECS). In DECS the specification of service composition and its enactment are separated. Processes can be deployed either for centralized or decentralized coordination. Our paper positions with respect this previous work by focusing on how to attain decentralization at the reflective level without modifying the BPEL engine. Previous work in the area results in interweaving decentralization with the regular BPEL functionality.

The system described in [17, 29] is capable of reacting to workload variations by altering its configuration in order to optimally use the available resources. Such changes happen automatically and without human intervention. In ZenFlow the developer can program the reconfiguration using metaobjects.

Some researchers have explored how to extend compositions with non-functional aspects. [20] introduces a mechanism for finding and binding partner web services at run time. This approach enables to repair long running business processes without stopping them. This kind of dynamism could also be added to ZenFlow using its reflective interface over partner web services. [16] presents facilities to enable dynamic workflow evolution adapting running instances of workflows on a per instance basis.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have demonstrated the convenience of the reflection approach for the web service orchestration. This approach enables a clear separation of functional aspects from non-functional ones, which is essential for the maintainability and update of the web service orchestration. We have exercised reflection with a complex and involved non-functional aspect that is decentralization. The evaluation shows that the overhead introduced by reflection is minimal. Scalability of the reflective decentralization has been evaluated from different perspectives showing that reflection can be effectively used without significant tradeoffs to separate concerns.

As a future work we plan to exercise the reflection to add new non-functional aspects to the BPEL engine, like debugging and fault tolerance. For the debugging, our BPEL engine needs to complete its current XPath evaluator feature.

7. REFERENCES

[1] *ActiveBPEL*. http://www.active-endpoints.com/.

[2] *AgilaBPEL*. http://swik.net/Agila-BPEL.

[3] *CapeClear*. http://www.capeclear.com/.

[4] F. J. Ballesteros, R. Jiménez-Peris, M. Patiño-Martínez, F. Kon, S. Arévalo, and R. H. Campbell. Using interpreted Composite Calls to improve Operating System Services. *Softw., Pract. Exper.*, 30(6), 2000.

[5] B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. *Distr. and Parall. Datab.*, 17(1), 2005.

[6] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 07(1), 2003.

[7] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *Soft. Eng., IEEE Trans. on*, 29(10), 2003.

[8] G. Chafle, G. Das, K. Dasgupta, A. Kumar, S. Mittal, S. Mukherjea, and B. Srivastava. An Integrated Development Environment for Web Service Composition. *icws*, 0:839–847, 2007.

[9] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *WWW*, 10:309–344, 2007.

[10] G. Coulson, P. Grace, G. Blair, W. Cai, C. Cooper, D. Duce, L. Mathy, W. K. Yeung, B. Porter, M. Sagar, and W. Li. A Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing. *Concurrency Computat.: Pract. Exper.*, 18(8), 2006.

[11] F. Eliassen, E. Gjørven, V. S. W. Eide, and J. A. Michaelsen. Evolving Self-Adaptive Services using Planning-Based Reflective Middleware. In *ARM*. ACM Press, 2006.

[12] M. F. Fabio Casati and I. Mirbek. An Environment for Designing Exceptions in Workflow Systems. *Information Systems*, 24(3):255–273, 1999.

[13] C. K. Fung, P. C. K. Hung, and D. H. Folger. Achieving Survivability in Business Process Execution Language for Web Services (BPEL) with Exception-Flows. In *EEE*, 2005.

[14] P. Grace, G. S. Blair, and S. Samuel. A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1), 2005.

[15] D. Grigori, F. Casati, U. Dayal, and M.-C. Shan. Improving Business Process quality through Exception Understanding, Prediction, and Prevention. In *The VLDB Journal*, 2001.

[16] J. Halliday, S. Shrivastava, and S. Wheater. Flexible Workflow Management in the OPENflow System. *EDOC*, 00, 2001.

[17] T. Heinis, C. Pautasso, and G. Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *ICAC*, 2005.

[18] P. C. K. Hung and D. K. W. Chiu. Developing Workflow-Based Information Integration (WII) with Exception Support in a Web Services Environment. In *HICSS*, 2004.

[19] IBM, Microsoft, and BEA. *Business Process Execution Language for Web Services*. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[20] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending BPEL for Run Time Adaptability. In *IEEE EDOC*, 2005.

[21] R. Khalaf and F. Leymann. E Role-based Decomposition of Business Processes using BPEL. In *ICWS*, 2006.

[22] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An Aspect-Oriented Framework for Service Adaptation. In *ICSOC*, 2006.

[23] A. Kumar and J. Wainer. Meta Workflows as a Control and Coordination Mechanism for Exception Handling in Workflow Systems. *Decision Support Systems*, 40(1), 2005.

[24] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA*, 1987.

[25] A. Martinez, M. Patino-Martinez, R. Jimenez-Peris, and F. Perez-Sorrosal. Zenflow: A Visual Web Service Composition Tool for BPEL4WS. In *VLHCC*, 2005.

[26] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In *OOPSLA*, 2004.

[27] A. Nguyen-Tuong and A. S. Grimshaw. Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications. *Parallel Processing Letters*, 9(2), 1999.

[28] G. Ortiz and F. Leymann. Combining WS-Policy and Aspect-Oriented Programming. In *AICT-ICIW*, 2006.

[29] C. Pautasso, T. Heinis, and G. Alonso. Autonomic Execution of Web Service Compositions. In *ICWS*, 2005.

[30] G. T. Sullivan. Aspect-Oriented Programming using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10), 2001.

[31] F. Taïani and J. C. Fabre. A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures. In *DSN*, 2005.

[32] E. Tanter, J. Noye, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *ACM OOPSLA*, 2003.

[33] S. Vinoski. A Time for Reflection. *IEEE Internet Computing*, 09(1), 2005.

[34] S. Woodman, D. Palmer, S. Shrivastava, and S. Wheater. Distributed Enactment of Composite Web Services . Technical Report CS-TR 848, School of Computing Science, Newcastle University, 2004.

[35] L. Zeng, H. Lei, J.-J. Jeng, J.-Y. Chung, and B. Benatallah. Policy-Driven Exception-Management for Composite Web Services. In *CEC*, 2005.