SLA-Driven Business Process Management in SOA

Vinod Muthusamy and Hans-Arno Jacobsen
{vinod,jacobsen}@eecg.toronto.edu
University of Toronto

Phil Coulthard, Allen Chan, Julie Waterhouse, Elena Litani
IBM Canada*

1 Introduction

In a Service-Oriented Architecture (SOA), distributed applications are built by orchestrating reusable services using high-level workflows or business processes. The complexity of developing and maintaining these processes is addressed by SOA development cycles that identify the roles of participants at each stage. To assist development, sophisticated tools have been developed, such as the IBM® WebSphere® suite of SOA products. However, the development, administration and maintenance of a business process still requires much manual effort that can be automated. In particular, the non-functional goals of a business process, often expressed as Service Level Agreements (SLA) need to be manually considered at each stage of the development process.

This paper presents a vision to achieve end-to-end SLA management by facilitating the various stages of business process development using formally encoded SLAs. We argue for a distributed architecture for the execution of business processes, and develop a model to control the provisioning of business processes in this architecture based on high-level goals that can be specified independently of the implementation details of a process.

*The views and opinions expressed in this paper solely reflect the personal views of the authors and do not necessarily represent the views, positions, strategies or opinions of IBM or the University of Toronto.

2 SOA Development Cycle

The SOA development cycle consists of modelling, development, execution, and monitoring stages. In the modelling stage, a business analyst defines the process abstracting from implementation details. Then, in the development stage, architects and developers break the abstract process model into development artifacts such as services, and implement the required business logic. This results in a set of components which are deployed in a runtime environment managed by an administrator responsible for provisioning sufficient resources to meet the goals of the process. Finally, runtime metrics may be monitored, and presented to the stakeholders in the preceding development stages.

An analyst may specify high-level, declarative goals such as the throughput requirements, or cost constraints of the process. These goals are formalized into SLAs, which can be represented at different levels of abstractions, and may simply be a document. SLA documents are passed down the chain of the development process, with each stakeholder ensuring conformance to the SLAs. The architects and developers interpret and ensure that the services developed conform to the SLAs. During execution, SLA conformance is often achieved by over-provisioning resources and manually tuning the system. Finally, monitoring subsystems are instantiated to verify that the SLA goals are met, and that violations are reported to the appropriate parties. These violations are manually addressed by changes to the process, redevelopment of services, or provisioning of resources.

Paper appears in *CASCON 2007*, Richmond Hill, Ontario, Canada, October, 2007.

2.1 Integration of SLAs

If SLAs are integrated into the tools and translated into execution and monitoring models, violations can be tracked more quickly and resources can be provisioned on demand in response to or in anticipation of violations. For example, SLAs at the modelling stage can be mapped to lower-level requirements on the services developed and resources provisioned, and translated to metrics that need to be monitored to observe SLA violations. Furthermore, adaptations on the process itself or its resource provisioning can be performed automatically at runtime to maintain the SLA goals.

Several runtime adaptations can be performed to maintain SLAs. With *dynamic service selection* the most appropriate services are chosen among a catalogue of available services. *Monitoring* can be optimized by only observing those metrics relevant to the SLA. The *distributed execution* of business processes becomes feasible by automatically assigning portions of a process to strategic locations in the system. Furthermore, dynamic *resource allocation* can ensure the process has sufficient resources (CPU, bandwidth, etc.) to maintain the SLA with changing load. Finally, the Enterprise Service Bus (ESB) [4] that underlies the SOA can be reconfigured to satisfy the SLA. The encoding of the SLAs in a language such as WSLA [2] into the SOA tools in a machine-understandable format makes it possible for these runtime adaptations to take place dynamically and without human intervention.

Incorporating SLAs into the SOA development tools *simplifies* the specification of SLAs since the analyst can declaratively specify high-level goals without detailed knowledge of the underlying implementation technologies. Additional *flexibility* is achieved by allowing the developers and administrators to make design decisions without having to be as concerned about SLA violations since the tools can perform some of these tasks.

3 System Architecture

Business process execution engines are typically centralized systems in which one node is provisioned to execute and manage all instances of one or more business processes. To address scalability, the centralized engine can be replicated and the process instances balanced among the replicas. However, process *instances* are still executed in a centralized manner, and control and data is still concentrated in the cluster.

In this work, we take a fundamentally different architectural approach whereby even individual process instances are executed in a distributed manner. A process is first decomposed into tasks, encapsulated within an *agent*—the software entity that carries out the task—and assigned to various execution engines in the system. Benefits of such an architecture include scalability, in-network processing, fine-grained use of IT resources, and the ability to deploy *portions* or processes close to the data they operate on, thereby minimizing bandwidth and latency costs of a process. This is not possible in a clustered architecture since the entire process instance must be executed by a single engine.

We use the PADRES distributed publish/subscribe messaging infrastructure [1], which supports decoupled interaction patterns, to facilitate the movement of agents in a distributed execution engine in a transparent manner [3]. The benefits of the agent-based execution engine architecture, however, are only achieved if the agents are deployed in a strategically. This can be a labour-intensive procedure that requires knowledge of the changing system resources, and process characteristics. To help automatically determine an optimal placement of agents, we now develop a cost model to compare different placement possibilities.

3.1 Cost Model

The cost model is a framework that captures various factors that can influence the agent placement decisions. Some cost factors are shown in Figure 1 grouped into *cost components*.

The first component is the *distribution cost*, which represents the overhead of distributing a process into small, fine-grained agents. This overhead can be expressed in terms of the bandwidth or latency of the inter-agent communication, depending on the desired goal. An-

Component	Notation
Distribution cost	C_{dist} (distribution overhead)
Message rate	C_{d1}
Message size	C_{d2}
Message latency	C_{d3}
Engine cost	C_{eng} (execution overhead)
Load	C_{e1}
Resources	C_{e2}
Task complexity	C_{e3}
Service cost	C_{serv} (service overhead)
Service latency	C_{s1}
Service execution	C_{s2}
Marshalling	C_{s3}

Table 1: Cost model components

Criteria	Cost function mapping
3s response time	$C_{d1}+C_{d3}+C_{e3}+C_{serv} < 3$
Optimize bandwidth	$min(C_{d2})$

Table 2: Examples of Cost Functions

other important cost component captures the resource usage of an agent on the engine it is executing on. Factors here include the number of concurrent instances an agent is executing, the resource utilization (in terms of processor or memory) of an agent, and the complexity of the task the agent is executing. The third cost component in Figure 1 is the *service cost*, which represents the cost of calling external services. This includes the time to call the service (which is a function of the network conditions between the agent and service), and the execution time of the service (which depends on the particular service provider used to execute the desired service).

A *cost function* based on the various cost components is used to flexibly and easily specify different goals. The cost function specifies a weighting of the various cost components that should either meet a *threshold* or be *minimized*. In the former case, the process is adapted only when the threshold is violated, while in the latter, process adaptation occurs whenever a more optimal placement is found. For example, Table 2 shows cost functions that ensure that the response time of a process is within three seconds, and that minimize the network overhead of a process.

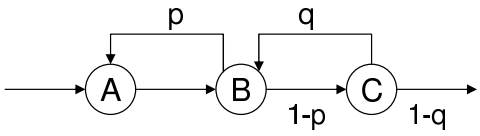

Figure 1: Business Process with Loops

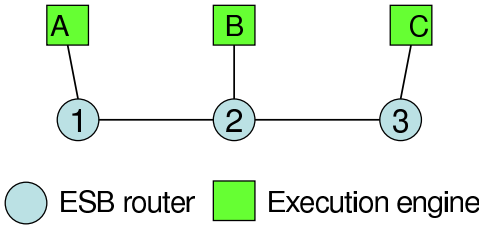

Figure 2: Experimental Topology

4 Evaluation

In this experiment, we use the business process shown in Figure 1 initially deployed to a set of execution engines in Figure 2. The process is designed to model a business process with time varying branch probabilities.

The SLA associated with the process seeks to minimize the bandwidth used by the process (see Table 2). Consequently, we observe the number of messages sent between the execution engines. Note that messages between agents deployed on the same engine are not counted in this metric.

Minimizing bandwidth would result in all agents being deployed on the same execution engine, so we fix the agents associated with tasks A and C to their initially deployed engines in Figure 2 (perhaps necessary for administrative or security reasons), and allow agent B to move freely.

Figure 4 shows the results of an experiment in which the branch probabilities of the process are fixed at $p = 0.9$ and $q = 0.1$, and 100 instances of the process in Figure 1 are invoked. Being biased toward the loop involving tasks A and B, we observe that the system reconfigures itself to one where agents A and B run on the same execution engine. In Figure 4, this results in the dynamic algorithm having about 10% of the message cost of the static algorithm in which the agents remain in the initial deployment. The point here is that it is not necessary to manually deploy agents in a strategic man-

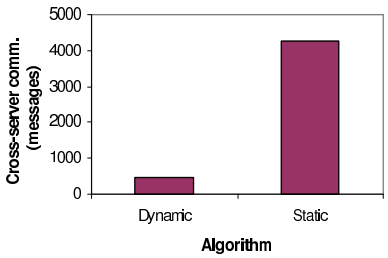

Figure 3: Performance of Static Workload

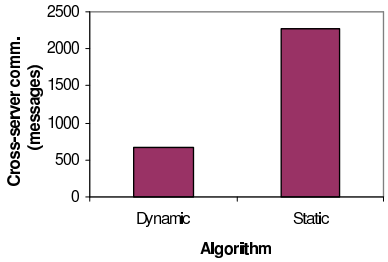

Figure 4: Performance of Dynamic Workload

ner — a possibly complex task — but to allow the system to configure itself.

In another experiment, the branch probabilities are varied, starting out with $p = 0.9$ and $q = 0.1$ for the first half of the experiment, and changing to $p = 0.1$ and $q = 0.9$ for the second half. This time, the deployment starts with the optimal placement for the initial branch probabilities (i.e., agents A and B at the same engine). Again, we observe that the dynamic algorithm keeps agent B in the initial optimal engine, and only when the branch probabilities change, the system moves agent B first to the middle execution engine and then to the one with agent C. The results in Figure 4 confirm that the dynamic reconfiguration algorithms adapt to the changing workload to achieve the desired goal with less than 30% of the message cost of the static case. A notable point about this experiment is that because the conditions of the system change over time, there is no optimal static deployment. Dynamic reconfiguration is required to achieve the best results.

5 Conclusions

In this paper we present a vision where business process goals are formally represented in an SLA specification in the development tools, and used to simplify each stage of the development cycle from the modelling of the process, through the development of it, to the deployment, execution and runtime monitoring of the process.

A distributed architecture is proposed for the execution of business processes in which lightweight agents collaborate to execute a larger process. This architecture affords scalability by allowing fine-grained resource allocation, and the strategic placement of computation. A cost model is developed to allow goals to be specified on the executing process, and algorithms are devised to redeploy the executing process to satisfy the specified goals.

Evaluations support the ability of the system to repeatedly adapt to changing runtime conditions to achieve a declaratively specified goal. In one workload, the system was able to save about 70% of the bandwidth by adapting a process compared to an initially optimal, but static deployment.

There is much ongoing and future research planned for this work, including more evaluations, and an investigation of the optimality of distributed reconfiguration. We also plan to apply some of these concepts in products such as IBM WebSphere Modeler and WebSphere Integration Developer.

References

[1] E. Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The PADRES distributed publish/subscribe system. In *ICFI*, pages 12–30, 2005.

[2] IBM. Web service level agreements (WSLA) project. `http://www.research.ibm.com/wsla/`.

[3] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. NIÑOS: A distributed service oriented architecture for business process execution. Technical report, Middleware Systems Research Group, July 2007. `http://padres.msrg.utoronto.ca`.

[4] Chris Nott, Peter Edwards, Andrew Humphreys, and Martin Keen. Using message sets in WebSphere business integration message broker to implement an ESB in an SOA, 2005. `http://www.redbooks.ibm.com/abstracts/redp3978.html`.