BPM in Cloud Architectures: Business Process Management with SLAs and Events

Vinod Muthusamy and Hans-Arno Jacobsen

University of Toronto

1 Introduction

Applications are becoming increasingly distributed and loosely coupled in terms of their development processes, software architectures, deployment platforms, and other aspects. For example, in Web mashups [12, 13], utility or cloud computing environments [2, 3], and service-oriented architectures (SOA) [4, 10] applications are developed by orchestrating reusable services using high level workflows or business processes. Application developers, however, must navigate a complex ecosystem that includes the services they depend on, the execution platforms of their applications and services, and the users of these applications, none of which they have much control over.

Business process management (BPM) practices address complexity in such environments with systematic development processes [1]. However, the development, administration and maintenance of a business process still requires much manual effort that can be automated. For example, non-functional goals, often expressed as Service Level Agreements (SLA) [7, 11] defining a contract between a service provider and consumer, are specified during an early design stage but may need to be manually considered at each subsequent stage of development.

We present an SLA-driven approach to BPM for service-oriented applications in environments such as cloud computing platforms. The approach employs two key ideas: formally specified SLAs of applications, and event processing technologies. The SLAs are used to simplify tasks such as process deployment and monitoring, and an event paradigm is used to develop components, such as an event-based distributed process execution engine, that can exploit the characteristics of such distributed environments. These ideas complement one another in that the goals specified in the SLA can be used to automatically monitor the relevant parts of a process's execution in a loosely-coupled manner, or optimize the deployment of the process at runtime.

2 System Model and Architecture

Consider a typical SOA development cycle illustrated in Fig. 1 consisting of modelling, development, execution, and monitoring stages. Each stage differs in the level of abstraction considered and is performed by the indicated roles, each of whom have varying expertise and concerns.

SLA Model: We make no major changes to the common development process outlined in Fig. 1 other than requiring a precise definition of the SLA requirements during the modelling stage. The details of our SLA model, which is designed to simplify the authoring of complex SLAs by composing and configuring existing SLA artifacts, is detailed in [5]. Capturing the SLAs early in a formal model has a number of benefits. 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 SLAs violations. Furthermore, to ensure SLAs are satisfied, several runtime adaptations can be performed such as those discussed in Sec. 3.

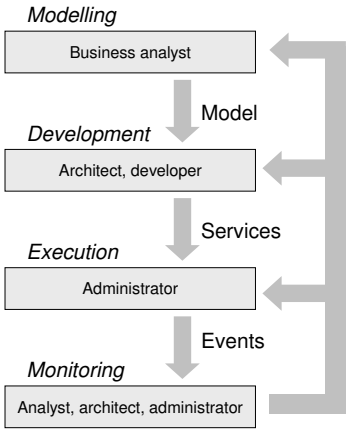


Fig. 1. Development cycle

Cost Model: Portions of the SLA are mapped to a cost model that captures various relevant factors. Some of these cost factors are shown in Table 1 grouped into cost components, such as the distribution cost which represents the overhead of distributing a process into fine-grained activities, the engine cost which captures the resource usage of an activity on the engine it is executing on, and the service cost which relates to the expense of calling external services.

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

Table 1. Cost model components

A cost function based on a combination of cost components can flexibly express a variety of goals. The cost function specifies that an arbitrary weighting of the cost components 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, a cost function can constrain process response times to three seconds, or minimize the network traffic overhead of a process.

Distributed Process Execution: 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. We take a fundamentally different architectural approach whereby even individual process instances are executed in a distributed manner.

Our distributed execution engine decomposes a process, such as a BPEL process, into its individual activities and deploys these activities to any set of execution engines in the system. These activities then coordinate by emitting and

consuming events over the PADRES[1] distributed publish/subscribe platform. For example, an activity that only executes after two other activities finish would subscribe to the composite event that indicates that both dependent activities have completed. Further details on how a BPEL process is mapped into this event-driven engine is presented in [8].

The execution engines in this architecture can be light-weight as they only execute fine-grained tasks, as opposed to complete processes. Another benefit of such an architecture is the ability to deploy *portions* of processes close to the data they operate on, thereby minimizing bandwidth and latency costs of a process. For example, for data intensive business processes, it is possible to deploy only those portions of the process that require access to large data sets close to their respective data sources. Different parts of the process that operate on different data sets can be independently deployed near their respective data sources. This is not possible in a clustered architecture since the entire process instance must be executed by a single engine.

Execution Engine: The internal components of an execution engine are presented in Fig. 2. Each engine is autonomous in deciding whether an activity should be redeployed to another engine. These decisions are based on the cost function associated with the process. Notably, there is no centralized component that is used to gather statistics, or to make activity placement decisions.

The distributed execution engine shown in Fig. 2 consists of a core Activity Manager that provides support services for the activities to collaborate among one another to execute a particular business process [8]. A Candidate Engine Discovery component is used to find other execution engines in the system [14], and these candidates are periodically probed by the Engine Profiler to gather various statistics. The Redeployment Manager computes the cost function for each activity executing in the engine, and determines if a more optimal placement of the activity is available among the known candidate engines. Finally, activities that are to be moved are redeployed using the Atomic Redeployer component which is responsible for ensuring that the movement of the activity does not affect the execution of the process [6].

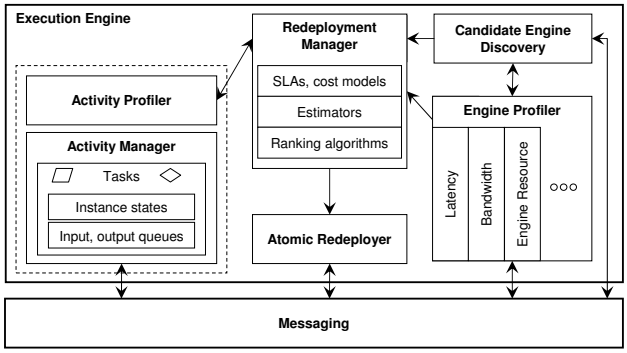


Fig. 2. Distributed execution engine

Redeployment Manager: The Redeployment Manager maintains for each activity a_i the engine is currently hosting, the cost function $f(a_i)$ associated with the

[1] Available for download at at `http://padres.msrg.org`.

activity, a running average of the cost $c(a_i, e_j)$ imposed by the activity were it hosted by engine e_j, and the engines where activity a_i's predecessors and successors are hosted. For convenience, the cost of deploying a_i at the current engine is denoted as $c(a_i)$.

The running average of the cost $c(a_i, e_j)$ of an activity is computed and maintained based on information from various profilers. An update of the cost $c(a_i, e_j)$ may reveal a better placement for activity a_i. For example, if the cost function is to be minimized, the algorithm finds the engine $e_{min} \in E$ such that $c(a_i, e_i)$ is minimized across all $e_i \in E$ where E is the set of known candidate engines. Activity a_i is then moved to engine e_{min}. On the other hand, if the cost function associated with activity a_i is a threshold function, a check is made to see if the accumulated cost $c(a_i)$ exceeds the threshold. If the cost is still within the threshold, nothing further is done. Otherwise, the system finds the engine e_{min} that results in $\min_{e_i \in E} c(a_i, e_i)$, and redeploys activity a_i to engine e_{min}. Now it may be that $c(a_i, e_{min})$ still exceeds the cost function threshold, in which case the predecessors of activity a_i are asked to redeploy themselves. This "back pressure" by activities to force a redeployment of their predecessors will occur repeatedly as long as the optimal placement of the activity is not sufficient to satisfy the cost function threshold. The redeployment procedure is further discussed in [9].

3 Benefits

Runtime Redeployment: We present a sample of the benefits of the SLA-driven distributed process execution architecture. The process consisting of nine activities whose dependencies are depicted in Fig. 3 is deployed in a network of nine execution engines, with each engine running on a machine with a 1.6 GHz Xeon processor and 4 GB of memory. The execution engines utilize an overlay of PADRES publish/subscribe brokers communicating over a 1 Gbps switch.

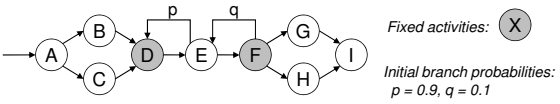
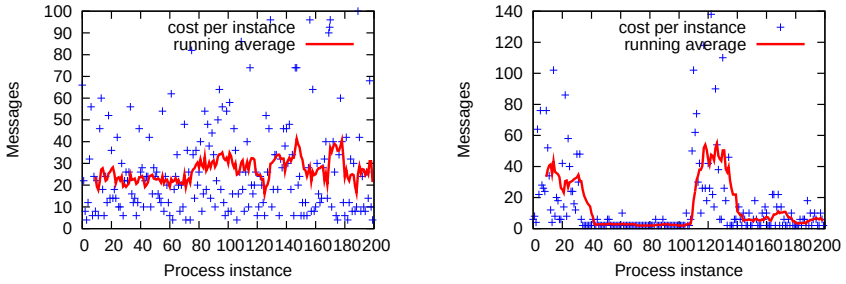


Fig. 3. Evaluated process

To simplify the experiment, activities D and F, as indicated in Fig. 3, are fixed to the initial engine they are deployed on, but the remaining activities can move. The process is invoked every second, and each process instance traverses the process graph according to the branch probabilities p and q as indicated in the figure. Notice that the 90% probability with which activity E transitions to activity D (as opposed to activity F) results in a process hotspot at activities D and E. About halfway through the experiment, the transition probabilities of p and q are reversed, so that a hotspot now occurs at activities E and F.

Fig. 4(a) presents the message overhead of executing the process in Fig. 3 under the case where activities remain in their initial deployment with each activity assigned to a different execution engine. The graph plots the number of messages the engines exchange in order to coordinate the execution of each process instance. The average overhead computed over a sliding window of ten



(a) Static deployment (b) Dynamic deployment

Fig. 4. Message traffic for process in Fig. 3

process instances shows that per-instance message traffic remains relatively stable at about thirty messages.

We expect, however, that due to the tight loop around activities D and E, the message overhead can be reduced if these activities were deployed on the same engine. Indeed, when a cost function to minimize the message traffic is applied to the process, the system reconfigures itself by moving activities A, B, C, and E to the same engine as activity D, and by redeploying activities G, H, and I to the engine where F resides. Fig. 4(b) shows that it takes about thirty seconds for the respective execution engines to determine the more optimal deployment and complete the movement of activities. Under the new deployment, the message traffic is only about 10% of the static case. After about 100 instances, the process traversal patterns are changed by swapping the initial branch probabilities p and q. This causes a momentary increase in the traffic overhead before the system again stabilizes in about thirty seconds by this time placing activity E together with activity F.

In more complex systems with multiple large processes and continuously changing process execution patterns and environmental conditions, it becomes increasingly infeasible to find an optimal static deployment, and the dynamic SLA-driven execution engine becomes more critical to ensure that the process SLA targets are achieved.

Automated Monitoring: A precisely specified SLA can be used to automatically instrument the process and generate monitoring code to detect SLA violations [5]. The monitoring subsystem is also event-driven and consumes the events emitted by the activities in our distributed process execution engine. This loose coupling allows the monitoring components to be deployed independently of the process, and can take advantage of the event processing optimizations of the underlying publish/subscribe system [8].

Furthermore, it turns out that the monitoring of a process can itself be modelled as a distributed process, thereby enabling the runtime process deployment optimizations above to be applied to the monitoring process as well [9]. For example, an SLA can be applied to the monitoring of the SLA to minimize the

network overhead of monitoring, or to optimize the monitoring latency in order to detect SLA violations quickly.

Service Selection: The external services composed by an application can have significant effect on the execution of the application. In cases where there are a number of interchangeable external services that a process may use, such as a credit verification or weather reporting service, resource discovery techniques can be used to find the service instance that helps achieve the SLA, whether it be a low latency, high throughput, or cheap service instance [14].

4 Summary and Conclusions

BPM in cloud architectures give rise to challenging issues that stem, in part, from the conflicting demands of application developers seeking simplicity and flexibility, administrators focused on reducing costs, and users demanding performance. An event-based runtime architecture, coupled with formal SLA models, can address some of these challenges and afford many benefits to the end-to-end development of business processes including efficient resource utilization, dynamic process deployment, automated monitoring, and intelligent resource discovery.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: Business Process Management (2003)
2. Amazon Web Services, `http://aws.amazon.com`
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A Berkeley view of cloud computing. Tech. rep., Univ. of California, Berkeley (2009)
4. Channabasavaiah, K., Holley, K., Edward M. Tuggle, J.: Migrating to a service-oriented architecture. IBM developerWorks Technical Library (2003)
5. Chau, T., Muthusamy, V., Jacobsen, H.A., Litani, E., Chan, A., Coulthard, P.: Automating SLA modeling. In: CASCON. Toronto, Canada (2008)
6. Hu, S., Muthusamy, V., Li, G., Jacobsen, H.A.: Transactional mobility in distributed content-based publish/subscribe systems. In: ICDCS (2009)
7. IBM: Web service level agreements (WSLA) project, `http://www.research.ibm.com/wsla/`
8. Li, G., Muthusamy, V., Jacobsen, H.A.: A distributed service-oriented architecture for business process execution. ACM Trans. Web 4(1) (2010)
9. Muthusamy, V., Jacobsen, H.A., Chau, T., Chan, A., Coulthard, P.: SLA-driven business process management in SOA. In: CASCON. Toronto, Canada (2009)
10. Natis, Y.V.: Service-oriented architecture scenario. `http://www.gartner.com/DisplayDocument?doc\_cd=114358` (Apr 2003)
11. Paschke, A., Schnappinger-Gerull, E.: A categorization scheme for SLA metrics. In: Service Oriented Electronic Commerce (2006)
12. Wang, H.J., Fan, X., Howell, J., Jackson, C.: Protection and communication abstractions for Web browsers in MashupOS. In: SOSP (2007)
13. Yahoo Pipes, `http://pipes.yahoo.com`
14. Yan, W., Hu, S., Muthusamy, V., Jacobsen, H.A., Zha, L.: Efficient event-based resource discovery. In: DEBS (2009)