2011 Third IEEE International Conference on Coud Computing Technology and Science

An Efficient Orchestration Engine for the Cloud

Rafael Z. Frantz
UNIJUÍ University
Ijuí, RS, Brasil
rzfrantz@unijui.edu.br

Rafael Corchuelo
University of Sevilla
Sevilla, Spain
corchu@us.es

José L. Arjona
Intelligent Integration Factory, Inc.
Huelva, Spain
arjona@i2factory.com

***Abstract*—The Cloud is evolving as a cost-effective solution to run services that support a variety of business processes. It is not surprising then that Orchestration as a Service (OaaS) is gaining importance as a means to integrate the many services a typical company runs or outsources in the Cloud. OaaS requires very efficient orchestration engines: the faster they work, the less customers have to pay and the more customers can be served. In this paper, we report on a new orchestration engine; we have performed a series of stringent experiments that prove that it outperforms a state-of-the-art orchestration engine in widespread use. Our conclusion is that our proposal is an efficient, solid orchestration engine ready for the Cloud.**

I. Introduction

Enterprise Application Integration (EAI) is a field of Software Engineering whose focus is on providing methodologies and tools to integrate the many heterogeneous services of which typical companies' software ecosystems are composed. Mapping business processes onto these services and integrating them appropriately seems to be a cornerstone for successful companies. Furthermore, the Cloud [26] is revolutionising the way companies run their services: the pay-per-use paradigm has proven an effective way to reduce IT costs without sacrificing quality, which is attracting an increasing number of companies.

Enterprise Service Buses (ESBs) lay at the heart of many current EAI solutions. Their main constituents are an array of adapters and an orchestration engine. The adapters abstract software engineers from the burden of wiring services using specific-purpose technologies. The orchestration engine, aka integration engine, provides the runtime support a collection of orchestration processes require. Roughly speaking, an orchestration process retrieves data from some services and routes them to other services; note that, in general, routing involves both transferring and transforming data. Orchestration processes can be described using a variety of languages, including WS-BPEL [4] and new languages that are based on the well-know catalogue of Enterprise Integration Patterns (EIPs) that was compiled by Hohpe and Woolf [16], cf. [8, 10, 11, 19].

The shift towards the Cloud is promoting that orchestration processes are run as services, as well (Orchestration as a Service, OaaS) [17]. OaaS is intimately related to the Platform as a Service service model, in which providers provide their customers with a computing environment to which they can deploy their software packages, including their orchestrations [26]. Wlodarczyk and others [33] mentioned that ESBs that work on the cloud are the key to taking EAI a step forward: Inter-EAI. This motivates the need for more efficient orchestration engines in the Cloud: the more efficient the engine, the less users have to pay to run their orchestration processes; similarly: the more efficient the engine, the more customers a provider can serve using the same resources. This has motivated mainstream players to provide new technologies for OaaS: Microsoft has recently launched the .NET Workflow Service[1] as part of the Azure Services Platform; CSC[2], has launched the Cloud Orchestration Services and Trusted Cloud Services initiative; Cordys has also launched their Enterprise Cloud Orchestration[3] initiative to promote OaaS.

In this paper, we report on a new orchestration engine called Guaraná RT[4]. Section §II, reports on the related work; Section §III, describes our proposal; Its salient features are that it relies on an efficient runtime that uses a configurable thread pool that works at the task granularity level, i.e., instead of allocating threads to process instances or routes inside them, it allocates threads to individual tasks inside processes. According to our survey of the literature, this seems to be a novel technique that has proven to work very well in heavily-loaded scenarios; The experimental results in Section §IV prove that Guaraná RT outperforms a state-of-the-art open-source orchestration engine in widespread use: Apache Camel. Section §V, sketchs our main conclusions.

II. Related Work

There are a number of recent papers in which the authors have paid attention to the problem of devising efficient orchestration engines, namely: Höing and others [17] presented the BIS-Grid project, in which they are working on providing the infrastructure required to develop and deploy EAI solutions using grid-oriented technologies; their orchestration engine is thus coarse-grained since their focus is on complex services that are run on a grid infrastructure

[1] http://www.microsoft.com/azure/workflow.mspx
[2] http://www.csc.com/cloud/
[3] http://www.cordys.com/cordyscmscom/enterprisecloudorchestration.php
[4] http://www.guarana-project.net

978-0-7695-4622-3/11 $26.00 © 2011 IEEE
DOI 10.1109/CloudCom.2011.110

and are orchestrated using WS-BPEL; Jia and others [20] reported on a proposal to improve the scheduling of tasks in the context of EAI solutions that build on clustered JBI services; Grounds and others [12] reported on a technique that allows to minimise the scheduling of tasks to increase productivity in cloud environments, which lays at the heart of their cloud workflow system [25]; Song and others [32] presented additional details on how to select workflow tasks optimally in cloud environments. Yang and others [34] reported on a proposal whose focus is on orchestrating services that are distributed across wide-area networks; Panahi and others [28] reported on LLAMA, whose focus is on monitoring, management, and configuration; finally, Kirschnick and others [22] and Dörnemann and others [7] reported on two proposals to support the automated provision of cloud services.

Our focus is on the design an implementation of a runtime system that can support orchestration engines very efficiently. Our survey of the literature reveals that the key to efficiency in EAI scenarios seems to be how threads are managed and allocated to run instances of orchestration processes. Processes are instantiated whenever the appropriate incoming messages are available, which, in turn, depends on the language used to describe them. (A message is an envelop used to transfer data within an EAI solution.) In WS-BPEL, the description must mention which messages can instantiate a process, i.e., messages are correlated externally; contrarily, in EIP-based languages, every incoming message instantiates a process, i.e., messages are correlated internally.

The simplest approach to run a process instance is to allocate a thread to it, like in ProActive [2]; this engine is based on active objects that have their own thread of control, so that method calls can always be asynchronous; this approach is very straightforward and may work quite well in scenarios in which there are not many process instances, i.e., systems that are not heavily loaded, or scenarios in which requests to external services are not common; otherwise, allocating a thread per process instance degrades performance very quickly as the number of service requests increases (realise that threads must remain blocked until a reply is received) [31]. Chrysanthakopoulos and Singh [6], Schippers and others [30], Andrew and others [3] reported on similar proposals that are also based on a per-process instance allocation policy.

A simple improvement is to use a technique that is usually referred to as dehydration/rehydration [29]; the idea is to detect when a thread has been blocked for too long and then serialise its state and allocate it to another process instance that is ready for execution; when the expected reply is received, the original process instance becomes ready for execution and shall be allocated a thread the sooner as one is available. Although dehydration/rehydration is conceptually simple and may perform well in EAI scenarios [24, 29], implementing it is not trivial at all since serialising a thread's state involves serialising its stack and other contextual information required to resume it.

Further improvements to the way threads are allocated require to know about the internals of orchestration processes. In general, they are composed of tasks that are arranged into routes. Tasks include receiving, sending, and routing a message appropriately through an orchestration process to the appropriate services, to mention a few examples. The usual approach is to allow software engineers configure thread pools that are allocated to specific routes within an orchestration process [10, 19]. This approach may work quite well, but has some drawbacks since orchestration processes are tightly coupled with the way they are deployed; in other words, thread pools must be pre-configured so that they can deal the highest foreseen workload, which does not usually lead to optimal performance during off-peak hours. Previous results in the literature suggest that the over-provisioning of stacks tends to lead to quick exhaustion of the virtual address space [13, 21]; furthermore, it is well-known that locking mechanisms often lack suitable contention managers, which is also very problematic regarding high-performance orchestration engines [9].

Event-driven programming provides a few ideas to manage threads more efficiently using inversion of control. Simply put, instead of calling blocking operations, process instances register their interest to be resumed when an event happens, e.g., receiving a reply from a service; when this occurs, the runtime system invokes a call-back method on the process instance. The problems with inversion of control are that the interactive logic of a program is fragmented across multiple event handlers and that control amongst handlers is expressed implicitly through manipulation of shared state [5, 18, 23]. Haller and Odersky presented a proposal in which there is no inversion of control [13, 14]; instead they made process instances thread-less building on the following idea: a process instance that waits in a receive statement is represented by a closure that captures the rest of the process computation; the closure is "piggy-backed" on the thread of the sender when the expected message arrives. If the receiving closure terminates, control is returned to the sender; if it blocks on a new receive statement, control is returned to the sender by throwing a special exception that unwinds its call stack. Adya and others [1] discussed advantages and disadvantages of multi-threading and event-based programming, considering the distinction between manual and automatic stack management, and the problems involved in handling "passive" messages, i.e., messages that need to be polled from its source.

III. Our Orchestration Engine

In this section, we describe our orchestration engine from both a structural and a dynamic perspective. In the paragraphs below, we make it clear that the emphasis of our design is on allocating the available threads to tasks, instead

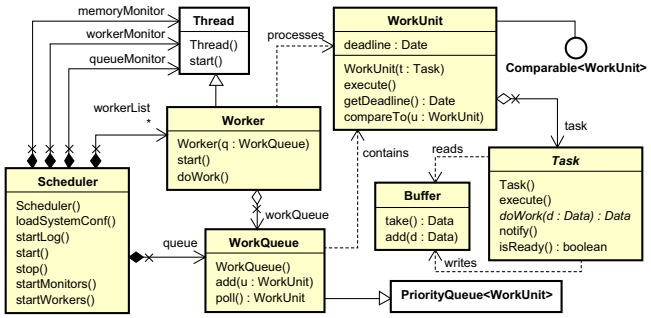

Figure 1. Static model for the task-based execution engine.

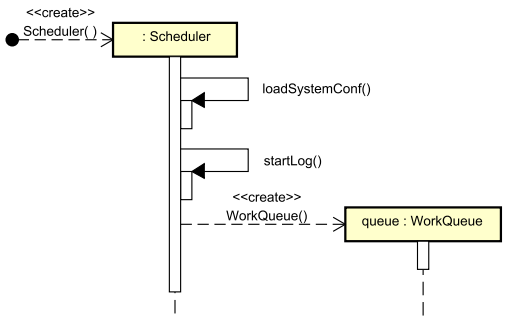

Figure 2. Initialising the engine.

of processes or routes within them. Our results prove that this is very effective.

A. Structure

Figure §1 presents a class diagram that models the main structural components of our proposal. Scheduler is the central class since its objects are responsible for co-ordinating all of the activities in an instance of our orchestration engine. Note that this class is not a singleton since we do not preclude the possibility of running several instances concurrently. At run-time, a scheduler owns a work queue, a list of workers, and three monitors.

The work queue is a priority queue that stores work units to be processed. A work unit has a reference to a task and a deadline. Note that class Task is abstract, which means that our engine is not bound with a particular set of tasks; this allows to create specific-purpose task toolkits that can be plugged into the engine. Guaraná DSL is a complementary proposal that provides a number of task toolkits to design EAI solutions in different contexts [11]. Usually, the deadline of a work unit is set to the current time, which means that the corresponding task can execute as soon as possible. If the deadline is set to a time in future, then the corresponding task is delayed until that time has elapsed. This is very useful to implement tasks that need to execute periodically, e.g., a communicator that polls a service from time to time.

Class Worker extends the standard Thread class, i.e., objects of this class run autonomously. Each worker is given a reference to the work queue, from which they concurrently poll work units to process.

The monitors gather statistics about how memory, CPU cores, and the work queue are used. The memory monitor registers information about both heap and non-heap memory; the worker monitor registers the user- and the system-time worker objects have consumed; and, the queue monitor registers the size of the queue and the total number of work units that have been processed. Monitors were implemented as independent threads that run at regular intervals, gather the previous information, store it in a file, and become idle the sooner as possible. Class Buffer represents in-memory

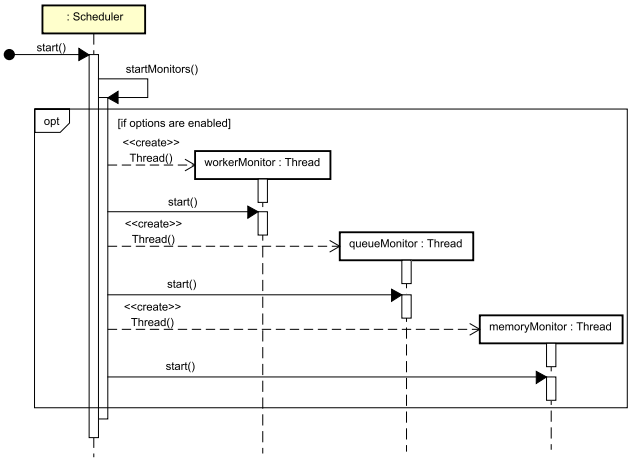

Figure 3. Creating and starting monitors.

buffers from/to which tasks read/write messages. This allows them to work in total asynchrony from each other.

B. Dynamics

Schedulers are configured using a simple XML file with information about the number of workers, the files to which the monitors dump statistics, the frequency at which they must run, and the logging system used to report warnings and errors. Figure §2 shows the sequence of operations involved in the initialisation of a scheduler. The first operation loads the configuration file and analyses it; then, the logging system is started, and a work queue is created.

Note that engines are not started when they are created. It is the user who must decide when to start them using the start() operation. Roughly speaking, this operation causes the invocation of two other operations, namely: startMonitors() and startWorkers(). The former starts the monitors that have been activated in the configuration file, cf. Figure §3, and the later creates and starts the workers.

Figure §4 shows the sequence of operations required to create and start the workers. Note that they are started asynchronously by invoking operation start(). The business

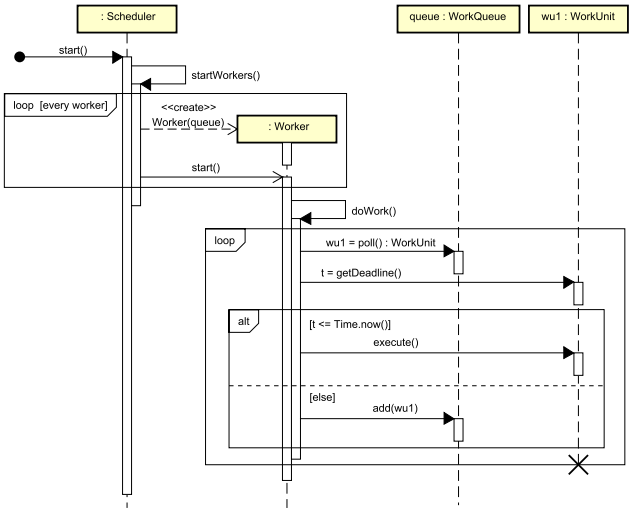

Figure 4. Creating and starting workers.

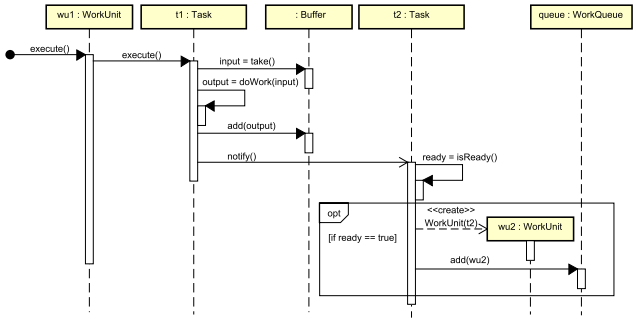

Figure 5. Executing a WorkUnit.

logic of a worker is defined inside its doWork() operation. This operation implements a loop that enables the workers to poll the work queue as long as the scheduler is not stopped. When a work unit is polled, the worker first checks its deadline; if it has expired, then the task can be executed immediately; otherwise, the work unit is delayed until the deadline expires.

Processing a work unit consists of invoking operation execute() on the associated task. This operation first packages the input messages, which are read from the appropriate buffers, and then invokes operation doWork(), which depends completely on the task toolkit being used. Then, the task writes its output messages to the appropriate buffers, which in turn notify the tasks that read from them. These tasks determine then if they become ready for execution or not; in the former case, the tasks create new work units and append them to the work queue, cf. Figure §5.

IV. Evaluation

In order to evaluate our proposal in practice, we selected a well-known benchmark in the literature of EAI: the Café

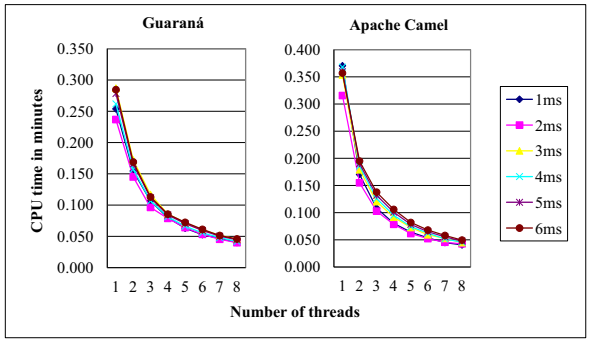

Figure 6. CPU execution times.

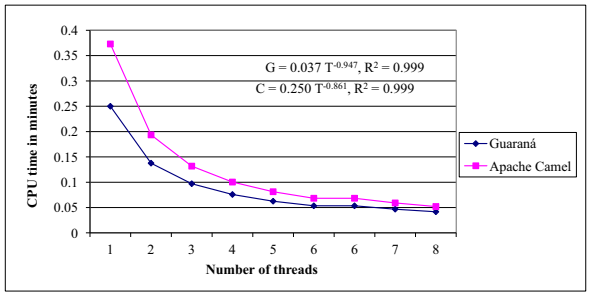

Figure 7. CPU execution times: Guaraná vs. Apache Camel

integration solution [15]. This benchmark is conceptually simple, but illustrates well what typical orchestration processes do: they read messages from a service, split them into pieces, route the pieces to different services, wait for their replies, assemble some result messages, and route them to the appropriate services.

In our experiments, we compared Guaraná RT 1.3.0 and Apache Camel 2.7.2. We used the Café implementation that is provided by the Apache Camel team and ported it to run on Guaraná RT for comparison purposes. The experiments consisted of processing a total of 250 000 input messages that were fed into the Café integration solution in 25 bursts of 10 000 messages each. The bursts were executed with message production rates of 1–6 milliseconds to simulate a heavily-loaded scenario, and we introduced a 10-second delay between every two bursts. Each experiment was repeated 25 times, and the results were averaged after discarding outliers using the well-known Chevischev's technique[5]; only 0.02% of the results were considered outliers, which makes it clear that the experiments were quite stable. The experiments were repeated using a thread pool with 1–8 threads; note that Apache Camel requires the thread pool to be configured inside the orchestration processes, which required to stop the engine, make the appropriate changes to recompile, and to re-start it. We ran our experiments

[5]Nisbet and others [27] provided a good summary of the statistical techniques we have used; the tests were performed using the SPSS statistical toolkit.

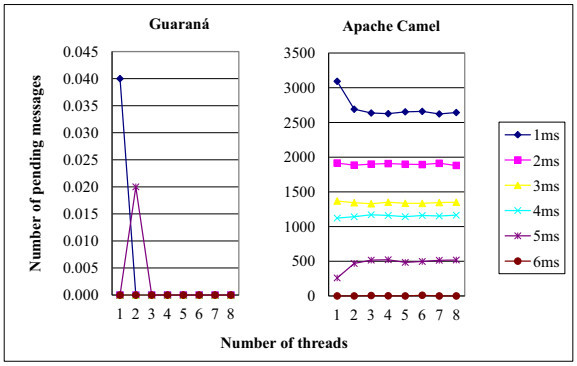

Figure 8. Pending messages after a burst completes.

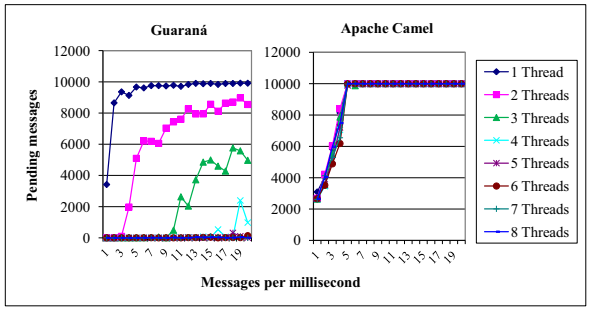

Figure 9. Pending messages after a stringent burst completes.

on a university cloud in which we had access to a virtual computer that was equipped with an Intel Core i7 that run at 2.93 GHz, 3 GB of RAM, Windows 7 Pro SP1, and Java Enterprise Edition 1.6. This makes actual times of little interest, since the same infrastructure was used concurrently by other users. This is the reason why we focused on measuring CPU times.

Figure §6 presents the average CPU times Guaraná RT and Apache Camel took to complete the experiments. Note that we measured CPU time per thread, i.e., the actual time the available threads took to process the workload, including user and operating system time. Note that the CPU times are almost insensitive to the message rate in each burst, which makes sense because they consisted of exactly the same number of messages; the differences are solely due to the random effects that are inherent to experimenting with a live system. We carried out a statistical study using the well-known non-parametric, two-way Kruskal-Wallis test at the standard confidence level ($\alpha = 0.05$) and we got p-value 0.681, which means that there is no evidence at all that the differences in the CPU times are statistically significant. We then performed a standard power regression analysis on our results to characterise them analytically; the analysis concluded that $G = 0.037T^{-0.947}$, with $R^2 = 0.999$, and $C = 0.250T^{-0.861}$, with $R^2 = 0.999$, respectively, where G denotes the approximation of Guaraná RT's CPU execution time, C the approximation of Apache Camel's CPU execution time, and T denotes the number of threads used. Note that the R^2 coefficient is nearly 1.0 in both cases, which indicates that these approximations are statistically accurate at the standard confidence level, cf. Figure §7. We then performed a standard non-parametric, two-way Mann-Witney test to check if both approximations can be considered statistically equal, and got p-value 0.002, which indicates that there is strong statistical evidence that Guaraná RT outperforms Apache Camel.

Note that the difference between both proposals is smaller as the number of threads increases. This behaviour is not surprising since, other things equal, the total number of messages each thread has to process decreases as the number of threads increases. Note, however, that Guaraná RT takes roughly 2.04 seconds of CPU less in average. At a first glance, this difference does not seem important enough, but it matters when a customer has to pay for that extra CPU consumption or when a provider cannot use those extra seconds to serve more customers. To make the difference more evident, we also measured the number of messages that had not been processed when every message burst finished; we refer to these messages as pending messages. The results are presented in Figure §8. It is not difficult to see how Guaraná RT outperforms Apache Camel by orders of magnitude. Note that when the message production rate is one message every 6 milliseconds, Apache Camel can handle the workload gracefully, but increasing the message production rate to 5 milliseconds has a significant negative impact. When the message production rate increased to 1 millisecond, the average number of pending messages raised to 2703.11, and it does not seem that adding more threads can help handle the workload successfully.

To check Guaraná RT's limit to process messages, we conducted a series of new experiments in which the message production rate was increased from 1 message per millisecond to 20 messages per millisecond. The CPU time both engines required to process these bursts of messages was roughly the same, as expected; contrarily, the number of messages pending to be processed when each burst finished increased significantly. Figure §9 shows the results we got. Note that Guaraná can handle a workload of 20 messages per millisecond quite efficiently with as little as 4 threads; contrarily, Apache Camel gets totally saturated at the rate of 5 messages per millisecond, independently of the number of threads available.

V. Conclusions

In this paper, we have explored a core component of every ESB: integration engines. Current trends in the industry seem to suggest that OaaS is gaining importance at an increasing pace, as companies run more and more services on the Cloud. This motivates the need for very efficient orchestration engines that are specifically tailored to work

in this context in which resources are shared amongst many users who pay per use of the platform. We have reported on Guaraná RT, which is a new orchestration engine that beats a state-of-the-art engine in the market by orders of magnitude. Our conclusion is that Guaraná RT is quite an efficient orchestration engine that is ready to used in real-world integration scenarios in the Cloud. In future, we plan on researching how Guaraná RT can work in elastic environments in which, e.g., more cores can be available at run time.

ACKNOWLEDGEMENTS

Our work was supported by the European Commission (FEDER funds), the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, and TIN2010-09988-E). Rafael Z. Frantz was also supported by the Evangelischer Entwicklungsdienst e.V. (EED).

REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002

[2] B. Amedro, F. Baude, D. Caromel, C. Delbé, I. Filali, F. Huet, E. Mathias, and O. Smirnov. An efficient framework for running applications on clusters, grids and clouds. In *Cloud Computing: Principles, Systems and Applications*, pages 163–178, 2010

[3] B. Andrew, C. Magnus, J. Mark, K. Richard, and N. Johan. Timber: A programming language for real-time embedded systems. Technical report, Oregon Graduate Institute School of Science and Engineering, 2002

[4] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web Services Business Process Execution language version 2.0 specification. Technical report, Organization for the Advancement of Structured Information Standards, 2007

[5] B. Chin and T. D. Millstein. Responders: Language support for interactive applications. In *European Conference on Object-Oriented Programming*, pages 255–278, 2006

[6] G. Chrysanthakopoulos and S. Singh. An asynchronous messaging library for C#. In *SCOOL at OOPSLA*, 2005

[7] T. Dörnemann, E. Juhnke, and B. Freisleben. On-demand resource provisioning for BPEL workflows using Amazon's elastic compute cloud. In *CCGRID*, pages 140–147, 2009

[8] D. Dossot and J. D'Emic. *Mule in Action*. Manning, 2009

[9] A. Dunkels, B. Grönvall, and T. Voigt. Contiki: A lightweight and flexible operating system. In *IEEE Conference on Local Computer Networks*, pages 455–462, 2004

[10] M. Fisher, J. Partner, M. Bogoevici, and I. Fuld. *Spring Integration in Action*. Manning, 2010

[11] R. Z. Frantz, A. M. Reina-Quintero, and R. Corchuelo. A domain-specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, 20(2):143–176, 2011

[12] N. G. Grounds, J. K. Antonio, and J. T. Muehring. Cost-minimizing scheduling of workflows on a cloud of memory managed multicore machines. In *CloudCom*, pages 435–450, 2009

[13] P. Haller and M. Odersky. Event-based programming without inversion of control. In *JMLC*, pages 4–22, 2006

[14] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009

[15] G. Hohpe. Your coffee shop doesn't use Two-Phase commit. *IEEE Softw.*, 22(2):64–66, 2005

[16] G. Hohpe and B. Woolf. *Enterprise Integration Patterns - Designing, Building, and DeployingMessaging Solutions*. Addison-Wesley, 2003

[17] A. Höing, G. Scherp, S. Gudenkauf, D. Meister, and A. Brinkmann. An orchestration as a service infrastructure using grid technologies and WS-BPEL. In *ICSOC*, pages 301–315, 2009

[18] K. Hong, J. Park, T. Kim, S. Kim, H. Kim, B. Scholz, B. Burgstaller, Y. Ko, and J. Park. TinyVM, an efficient virtual machine infrastructure for sensor networks. In *Embedded Networked Sensor Systems*, pages 389–400, 2009

[19] C. Ibsen and J. Anstey. *Camel in Action*. Manning, 2010

[20] X. Jia, S. Ying, L. Hu, and C. Chen. Scheduling active services in clustered jbi environment. In *CloudCom*, pages 413–422, 2009

[21] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *PPPJ*, 2009

[22] J. Kirschnick, J. M. A. Calero, L. Wilcock, and N. Edwards. Toward an architecture for the automated provisioning of cloud services. *IEEE Communication*, December:124–131, 2010

[23] P. Levis and D. E. Culler. Maté: a tiny virtual machine for sensor networks. In *Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 2002

[24] B. Loesgen, C. Young, J. Eliasen, S. Colestock, A. Kumar, and J. Flanders. *BizTalk Server 2010 Unleashed*. Sams, 2011

[25] J. Madden, N. G. Grounds, J. Sachs, and J. K. Antonio. The gozer workflow system. In *IPDPS*, pages 1–8, 2010

[26] P. Mell and T. Grance. Draft nist working definition of cloud computing. http://csrc.nist.gov/groups/SNS/cloud-computing, 2011

[27] R. Nisbet, J. E. IV, and G. Miner. *Handbook of Statistical Analysis and Data Mining Applications*. Academic Press, 2009

[28] M. Panahi, K.-J. Lin, Y. Zhang, S.-H. Chang, J. Zhang, L. Varela, and Y. Zhang. The LLAMA middleware support for accountable service-oriented architecture. In *ICSOC*, pages 180–194, 2008

[29] C. Renouf. *Pro IBM WebSphere: Application Server 7 Internals*. Apress Academic, 2009

[30] H. Schippers, T. Van Cutsem, S. Marr, M. Haupt, and R. Hirschfeld. Towards an actor-based concurrent machine model. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 4–9. ACM, 2009

[31] B. Silvestre, S. Rossetto, N. Rodriguez, and J.-P. Briot. Flexibility and coordination in event-based, loosely coupled, distributed systems. *Computer Languages, Systems & Structures*, 2010

[32] B. Song, M. M. Hassan, and E. nam Huh. A novel heuristic-based task selection and allocation framework in dynamic collaborative cloud service platform. In *CloudCom*, pages 360–367, 2010

[33] T. W. Wlodarczyk, C. Rong, and K. A. H. Thorsen. Industrial cloud: Toward inter-enterprise integration. In *CloudCom*, pages 460–471, 2009

[34] H. Yang, M. Kim, K. Karenos, F. Ye, and H. Lei. Message-oriented middleware with QoS awareness. In *ICSOC*, 2009