A TECHNOLOGY PROPOSAL TO REALISE ENTERPRISE APPLICATION INTEGRATION

***Rafael Z. Frantz*[1], *Rafael Corchuelo*[2], *Fabricia Roos-Frantz*[1] and *Sandro Sawicki*[1]**
[1]Unijuí University, Department of Exact Sciences and Engineering, Ijuí, Brazil
[2]University of Seville, Department of Computer Language and Systems, Seville, Spain

Abstract

All over the years, enterprises have been accumulating a variety of applications in their software ecosystem to support their business processes. As a result, a software ecosystem is an heterogeneous set of IT assets (data and functionality) of the enterprise. Enterprise Application Integration (EAI) discipline aims to provide language and tools to support the development of integration solutions. Enterprises are always looking for how to optimise the use of resources, which includes their IT assets. Thus, building EAI solutions that allow to reuse actual data and functionality is mandatory to optimise the current or to provide support for the new business processes that emerge in an enterprise. The need for building EAI solutions has been pushing the development of languages and software tools, which can be used to model, implement, and run integration solutions. In the EAI community, the support for integration patterns is a trend, both in languages and software tools. In this chapter we introduce a software tool to support the realisation of Enterprise Application Integration. This tool is a Java-based software development kit that we provide to implement and run EAI solutions based on integration patterns. We have conducted a series of experiments to evaluate our proposal against real-world integration problems and the results indicate that it is viable and can be used to solve real-world integration problems.

PACS: 05.45-a, 52.35.Mw, 96.50.Fm

Keywords: Enterprise Application Integration

1. Introdution

All over the years, enterprises have been accumulating a variety of applications in their software ecosystem [1] to support their business processes. As a result, a software ecosystem is an heterogeneous set of IT assets (data and functionality) of the enterprise. It is common to find technology gaps within software ecosystems, chiefly because they run different operating systems, they have different data models to represent the same business entity, they contain applications that were developed using different programming languages, and the applications usually are not ready for integration. Enterprise Application Integration (EAI) discipline aims to provide language and tools to support the development of integration

solutions. Enterprises are always looking for how to optimise the use of resources, which includes their IT assets. Thus, building EAI solutions that allow to reuse actual data and functionality is mandatory to optimise the current or to provide support for the new business processes that emerge in an enterprise.

The need for building EAI solutions has been pushing the development of languages and software tools, which can be used to model, implement, and run integration solutions. Figure 1 introduces the layers involved in an EAI solution. The highest layer is the most abstract and deals with the business process model. At this level, the most popular languages are the Business Process Model and Notation (BPMN) [2] and the Business Process Execution Language (BPEL) [3]. Whereas the former language provides a standard graphical notation to model business processes at a computation independent level and targets the activities and the flow of the business process, the later language focuses on building executable orchestration models and can be seem as complementary to BPMN. Application integration models require languages that can provide constructs to capture the most fundamental concepts involved in modelling EAI solutions such as messages, tasks, ports, processes, and adapters. Hohpe and Woolf [4] have documented several conceptual integration patterns that have been adopted by the EAI community as a cookbook when building integration solutions. Neither BPMN or BPEL provide support for these integration patterns. In Frantz et al. [5] we have introduced a Domain-Specific Language that can be used to design application integration models. This language realises the conceptual integration patterns documented by Hohpe and Woolf and provides constructs to the aforementioned fundamental concepts involved in modelling EAI solutions. We use this language to model the application integration solutions introduced in this chapter.

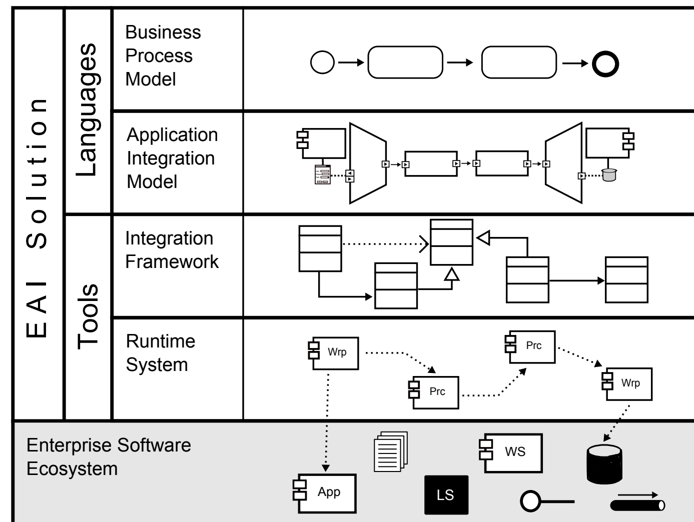

Figure 1. Layers of an Enterprise Application Integration Solution.

An integration framework is a software development kit that allows for the implementation of application integration models into executable code. Roughly speaking, an integration framework is composed of two layers: core and toolkit. The core provides concrete implementation for messages, ports, and processes, and an abstract implementation for

task and adapter. The toolkit extends the core layer to provide a set of concrete tasks and adapters. A task is the implementation of an integration pattern and an adapter is the piece of software that implements the low-level communication protocol necessary to interact with the IT assets being integrated. A runtime system supports the execution of EAI solutions, and usually includes monitoring features so that it is possible for software engineers to gather statistics about the execution of their integration solutions.

In the past few years several integration frameworks have emerged. Camel [6], Spring Integration [7], and Mule [8] are amongst the most successful open source integration frameworks. Camel provides a fluent API [9] that software engineers can use programmatically or by means of a graphical editor. In both cases, the EAI solution is implemented using a Java, Scala, or XML Spring-based configuration files. Spring Integration was built on top of the Spring Framework container, and provides a command-query API [9]. This tool can be used programmatically or by means of a graphical editor. EAI solutions are implemented using either Java code or an XML Spring-based configuration file. The architecture of Mule got inspiration from the concept of enterprise service bus. Software engineers count on a command-query API [9] to use this tool programmatically, or a workbench to design and implement EAI solutions using a graphical editor. EAI solutions are implemented using either Java code or an XML Spring-based configuration file. Camel, Spring Integration, and Mule incorporate a runtime system so that it is possible to execute the code of EAI solutions implemented using these technologies.

In this chapter we introduce a Software Development Kit to implement and run integration solutions. It was designed to be used by means of a command-query API [9] and provides support for the implementation of application integration models designed using our Domain-Specific Language. This Software Development Kit also includes a runtime system with some basic monitoring features. We have conducted a series of experiments to evaluate our proposal against real-world integration problems and the results indicate that it is viable and can be used to solve real-world integration problems.

The reminder of this chapter is organised as follows: Section 2 introduces our Software Development Kit to support the implementation and execution of EAI solutions; Section 3 presents four case studies to which we have applied our technology to demonstrate that it is viable and can be used to solve real-world integration problems; and, Section 4 concludes this chapter.

2. The Software Development Kit

We have worked on a technology to support the realisation of Enterprise Application Integration. We refer to this technology as Guaraná SDK. Guaraná SDK is the Java-based software development kit that we provide to implement and run EAI solutions based on integration patterns. It was designed to be used by means of a command-query API [9]. This tool is composed of two layers, namely: core and toolkit. The former provides a number of classes and interfaces that implement the abstractions of our Domain-Specific Language [5], and the latter extends some abstractions in the core to provide a general-purpose toolkit, ready-to-use implementations of building blocks, such as tasks and adapters. By core we mean a layer that provides abstract and concrete implementations. Our core also includes a runtime system to execute EAI solutions. In the following sections we describe

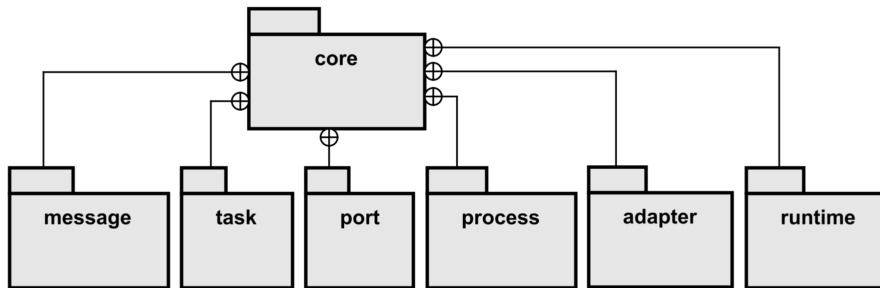

Figure 2. Packages of which our core is composed.

the core and the toolkit layers.

2.1. The Core Layer

The core layer is composed of six packages. Figure 2 provides an overview of this layer and in the following subsections we describe each package.

Messages

Messages are used to wrap the data that is manipulated in an EAI solution. They are composed of a header, a body and one or more attachments, cf. Figure 3. The header holds several meta-data properties, including: message identifier, expiration time, and list of parents. The message identifier is a UUID that uniquely identifies every message; the expiration time allows to set a deadline after which a message is considered outdated for further processing; the list of parents allows to track which messages originate from which ones, i.e., it helps find correlated messages. The body holds the payload data. Attachments allow messages to carry extra pieces of data associated with the payload, e.g., an image or an e-mail message. Messages implement two interfaces so that they can be serialised and compared, respectively. Serialisation is required to deep copy, to persist, and to transfer messages, and comparison enables a workflow to process messages according to their priority.

Tasks

The task package provides the foundations to implement domain-specific tasks in specialised toolkits. Figure 4 shows the task model in our proposal. A task models how a set of inbound messages must be processed to produce a set of outbound messages. Tasks communicate indirectly by means of slots. A slot is an in-memory priority buffer that helps transfer messages asynchronously so that no task has to wait until the next one is ready to start working. Tasks become ready to be executed according to a time criterion or a slot criterion. In the former case, a task becomes ready to be executed periodically, after a user-defined period of time elapses since it became ready for the last time; in the later case, it becomes ready when the appropriate set of messages is available in its input slots. For instance, a merger is a task that reads messages from two or more slots and merges them into one slot; this task can transfer messages as they are available. Contrarily, a context-based

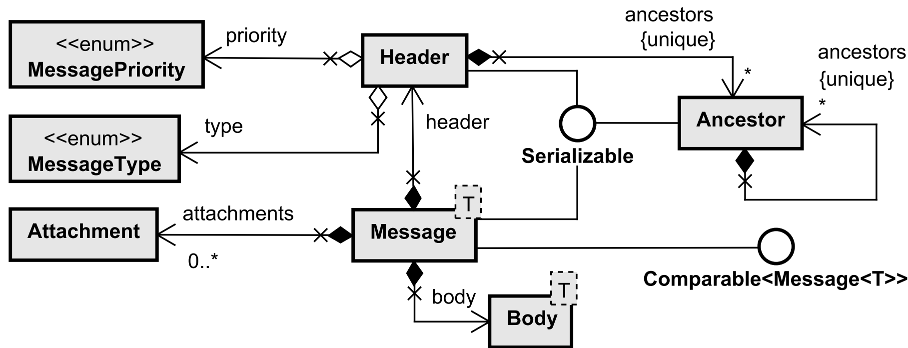

Figure 3. Message model in the core.

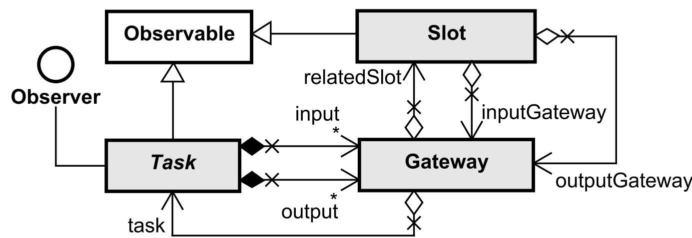

Figure 4. Task model in the core.

content enricher is a task that reads a base message and a context message from two slots and uses the later to enrich the former; this task cannot become ready to be executed until two messages are available in its input slots. Slots and tasks are both observable objects, which means that they can notify other objects of changes to their state; in addition, tasks are observer objects since they monitor slots.

Ports

Ports abstract away from the communication mechanism used to interact with the environment, cf. Figure 5. Note that every port must be associated with a process, and that we distinguish between one way and two way ports. The former are unidirectional ports that allow to read/write messages; the latter are bidirectional ports that allow to either send a message and wait for the answer (responder ports) or to receive a message and produce a response message (solicitor port). Internally, ports are composed of tasks, namely: a communicator, a pipeline, and a mapper. Communicators are the tasks that allow to actually read or write a message, namely: in communicators are used to read a message in raw from; contrarily, out communicators are used to write a message in raw form. By raw form, we mean a stream of bytes that is understood by the corresponding process or asset. The pipeline helps pre- or post-process a message in raw form to decrypt/encrypt, decode/encode, or unzip/zip it.

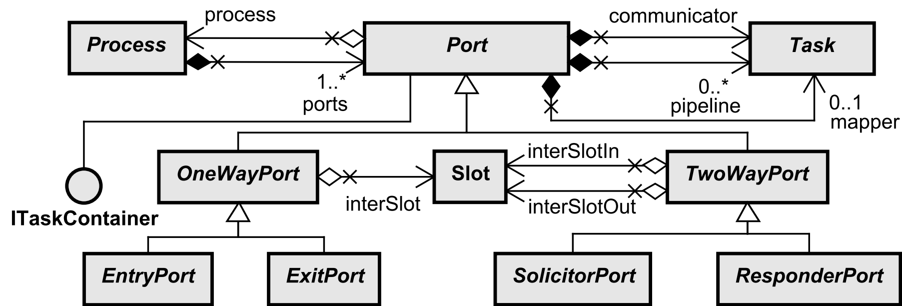

Figure 5. Port model in the core.

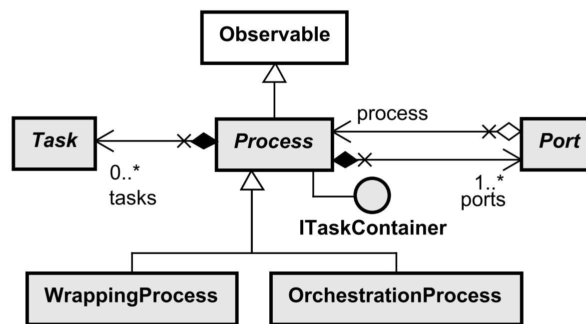

Figure 6. Process model in the core.

The pipeline in an input port ends with a mapper task that transforms the resulting stream of bytes into a text document, for instance; the pipeline in an output port begins with a mapper that transforms the text document into a stream of bytes. Bidirectional ports can actually be seen as a combination of an input and an output port.

Processes

Processes are the central processing units in an integration solution, cf. Figure 6. They are composed of ports and tasks, extend class `Observable`, and implement interface `ITaskContainer`. The reason why processes are observable is that they are just an abstraction that helps organise groups of tasks that co-operate to achieve a goal; from the point of view of our proposal, they are just a container that reports which of their tasks are ready for execution to an external observer. The `ITaskContainer` defines the interface to add, remove, and search for tasks in objects that may contain tasks. A process may have several observers, e.g., to log or to monitor its activities; however, the most important one is a runtime system, which we describe in the following section.

Processes serve two purposes, namely: there are processes that allow to wrap applications and processes that allow to orchestrate the workflow. The former are reusable processes that endow an application with a message-oriented API that simplifies interacting with it. Implementing such a wrapping process may range from using a JDBC driver to

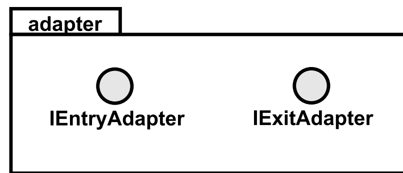

Figure 7. Adapter model.

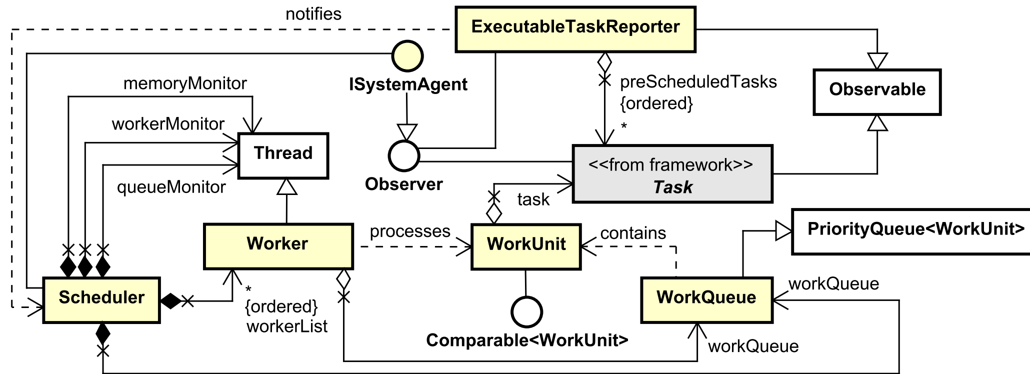

Figure 8. Engine model in the toolkit.

interact with a database to implementing a scrapper that emulates the behaviour of a person who interacts with a user interface. Orchestration processes, on the contrary, are intended to orchestrate the interactions with a number of services, wrapping processes, and other orchestration processes. Independently from their role, processes are composed of ports and tasks.

Adapters

This package aims to provide the foundations to implement adapters in specialised toolkits, cf. Figure 7. Adapters are the piece of software that implements the low-level communication protocol necessary to interact with the processes or assets involved in an integration solution. The core layer provides four interfaces to describe the operations used by ports to read, write, solicit, and respond.

The Execution Engine

The `engine` package provides an implementation to the task-based runtime system on which Guaraná SDK relies, cf. Figure 8. `Scheduler` is the central class in this package. At run-time, a scheduler owns a work queue, a list of workers, and three monitors. The work queue is a priority queue that contains work units to be processed. A work unit consists of a task to be executed and a deadline. In most cases, the deadline is set to the current time, which means that the corresponding task can execute as soon as possible; there are a few

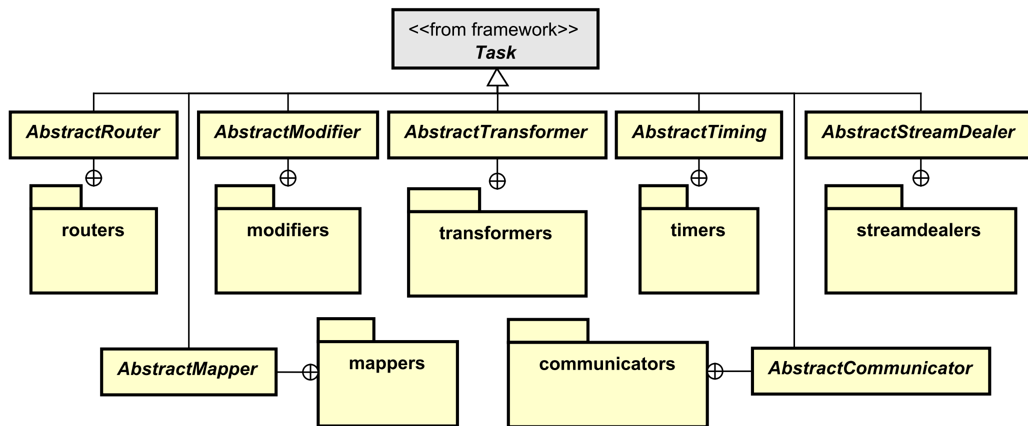

Figure 9. Task model in the toolkit.

time-dependent tasks for which the deadline is set to a time in future, e.g., a timer that ticks every minute or a communicator that polls a service every ten minutes. Workers are an extension to the `Thread` class and they have a reference to the work unit that they have to execute. The monitors are intended to gather statistics about how memory is used, the time tasks take to complete, and the size of the work queue. They have been implemented as independent threads that run at regular intervals, gather the previous information, dump it to a file, and become idle the sooner as possible.

2.2. The Toolkit Layer

The core provides two extension points, namely: `Task` and `Adapter`. We have designed a core toolkit that provides extensions to deal with a variety of tasks that support the majority of integration patterns in the literature [4], and provide active and passive adapters that enable the use of several low-level communication protocols.

Figure 9 presents the extensions to the `Task` class. Below we provide an explanation in which term schema is used to refer to the logical structure of the body of a message. It may range from a DTD or an XML schema to a Java class.

- `Router`: a router is a task that does not change the messages it processes at all, but routes them through a process. This includes filtering out messages that do not satisfy a condition or replicating a message, to mention a few tasks in this category.

- `Modifier`: a modifier is a task that adds data to a message or removes data from it as long as this does not result in a message with a different schema. This includes enriching a message with contextual information or promoting some data to its headers, to mention a few examples in this category.

- `Transformer`: a transformer is a task that translates one or more messages into a new message with a different schema. Examples of these tasks include splitting a message into several ones or aggregating them back.

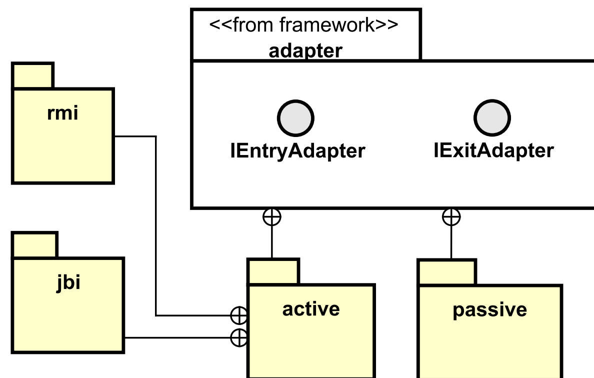

Figure 10. Adapter model in the toolkit.

- `Timer`: a timer is a task that performs a time-related action, e.g., delaying a message or producing a message at regular intervals.
- `StreamDealer`: a stream dealer is a task that deals with a stream of bytes and helps zip/unzip, encrypt/decrypt, or encode/decode it.
- `Mapper`: a mapper is a task that changes the representation of the messages it processes, e.g., from a stream of bytes into an XML document.
- `Communicator`: a communicator is a task that encapsulates an adapter. Communicators serve two purposes: first, they allow adapters to be exported to a registry so that they can be accessed remotely; second, a communicator can be configured to poll periodically a process or application using an adapter.

There is a package associated with every of the previous tasks, which provide a variety of specific-purpose implementations in each integration pattern category.

Figure 10 shows the extension to the adapters. They can be either active or passive. An active adapter allows to poll periodically the process or application with which it interacts; contrarily, a passive adapter is intended to export an interface to a registry, so that other applications or processes can interact with it. Note that entry and exit ports can be implemented using either active or passive adapters.

The `active` package is divided into two packages to provide implementations that are based on JBI and RMI protocols, respectively. Note that supporting JBI adapters allows to plug Guaraná SDK into a variety of ESBs; for instance, our reference implementation is ready to be plugged into Open ESB [10]. This, in turn, allows Guaraná SDK processes to have access to a variety of applications in current software ecosystems, including files, databases, web services, RSS feeds, SMTP messaging systems, JMS queues, DCOM servers, and so on. The `rmi` package provides several implementations that are intended to be used to interact with an RMI-compliant server.

3. Experiments

We have worked on four case studies to which we have applied our proposal in the laboratory to verify its viability. Every case study was designed using our Domain-Specific Language and implemented using our Software Development Kit. The first two case studies are based on real-world integration problems found at Unijuí University (Ijuí, Brazil) and at Huelva's County Council (Huelva, Spain); the next two case studies are based on well-known problems in the literature to illustrate the integration of the applications involved in the scenarios of searching and booking flights and hotels for a travel.

We have conducted a series of experiments to evaluate the previous integration solutions on our Runtime System. We used mock adapters, i.e., adapters implemented in memory that simulate the functionality of a real-world adapter and not on external software. The mock adapters allowed us to save the processing time required by the real-world adapters based on JBI. Furthermore, by using mock adapters the execution of the integration solutions depend only on our Runtime System, and not on other external software. In each experiment we measured the following variables:

Consumption of CPU Time per Thread: This variable measures the average CPU times that the integration solution has consumed to process all of the messages of an experiment. 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. To measure this variable, we run every integration solution with a fixed message production rate, a varying the number of threads (t), and a varying number of messages (m). We introduced a 60-second delay between every two experiments. The message production rate considered was one message every 5 milliseconds, we varied t in the range 1, 2, 4, 6, 8 threads, and m in the range $20,000$, $40,000$, $60,000$, ... , $200,000$ messages. In total, we ran a total of 125 experiments for each integration solution to draw our conclusions on this variable.

Pending Messages: This variable measures the number of messages that had not been processed yet right after the message production finishes. The experiments conducted to measure this variable consisted of running an integration solution with a fixed number of messages per experiment, a varying number of threads (t), and a varying message production rate (r) to simulate heavily-loaded scenarios. We introduced a 60-second delay between every two experiments. The total number of messages in each experiment was $100,000$, we varied t in the range 1, 2, 4, 6, 8 threads, and r in the range 200, 400, 600, ... $3,000$ messages per second. In total, we ran 375 experiments for each integration solution to draw our conclusions on this variable.

We ran these experiments on a machine that was equipped with an Intel Core i7 with four physical CPU threads that run at 2.93 GHz, and had 8 GB of RAM, Windows 7 Professional Service Pack 1, and Java Enterprise Edition 1.6 64-bit installed. Each experiment was repeated 5 times and the results were averaged in order to diminish the effects of unpredictable events in the operating system. In every experiment the body of the messages hold an actual document in XML format. Note that the size of a message being processed by an integration solution varies, since it is modified and transformed throughout the workflow.

Thus, we have computed the average size of the messages that belong to a same correlation processed in an integration solution. The result is an average message size of $1,356.14$ bytes for the Unijuí University solution, $1,317.75$ for the Huelva's County Council solution, $1,498.11$ bytes for searching flights and hotels solution, and $1,435.66$ for the booking flights and hotels solution. (Note that this deviates largely from other experiments in the literature in which the authors used unrealistically small message sizes.)

In the following sections, we first describe the integration problem tackled in each case study; we then provide a solution model; next, we show the experimental results that we gathered and draw our conclusions.

3.1. Unijuí University

This case study consists of a non-trivial, real-world integration problem that builds on a project to enhance the functionality of the call centre application at Unijuí University. The goal is to automate the invoicing of personal phone calls that employees make using the University's phones.

The Software Ecosystem

The integration solution involves five applications, namely: Call Centre, Human Resources System, Payroll System, Mail Server, and SMS Notifier. Each application runs on a different platform; the Human Resources System and the Payroll System are legacy systems developed in house, and the rest are off-the-shelf software packages purchased by the University. In addition, Mail Server provides a POP3 and a SMTP interface; the other applications were designed without integration concerns in mind, which requires to interact with them by means of their data layer. The Call Centre records every call every employee makes from a University's phone; it can identify who the employee is because they have a personal access code that they have to enter before dialling the number they wish to call. This code is used to correlate phone calls with the information in the Human Resources System and the Payroll System. The Human Resources System provides personal information about the employees. Every month, the Payroll System computes the salary of every employee, including wages, bonuses and deductions. The Mail Server and the SMS Notifier run the University e-mail and short message system services, respectively, and are used for notification purposes.

Every phone call registered by the Call Centre, except for toll-free calls, must be transformed into debits in the Payroll System. Employees can be notified by e-mail and/or short text message about their calls, so that they are aware of the deduction that shall appear in their pay slip. The only assumption we make is that the information registered in the Call Centre, apart from the data of the phone calls and the personal access codes, does not include any other information about the employees. It is the Human Resources System that provides this information.

Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration

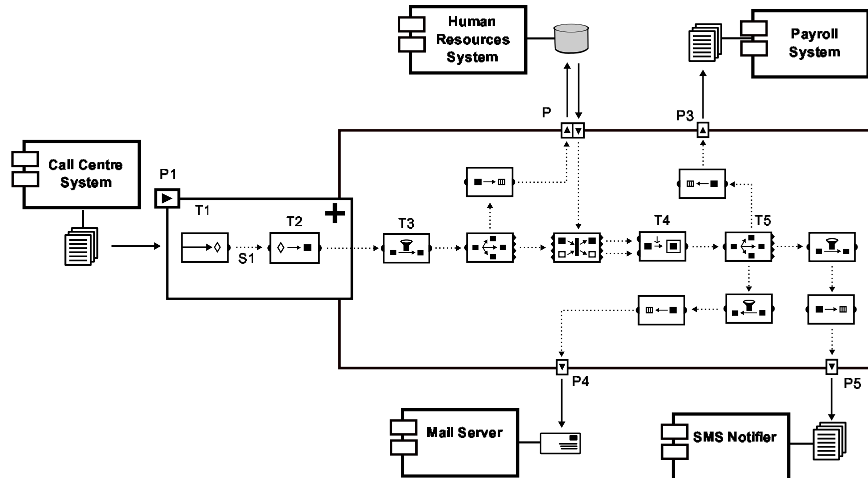

Figure 11. The Unijuí University integration solution.

solution, cf. Figure 11. Some ports use text files to communicate indirectly with the `Call Centre`, the `Payroll System`, and the `SMS Notifier`; we have direct access to the `Human Resources System` database by means of a pair of entry and exit ports. Translator tasks were used to translate messages from canonical schemas into schemas with which the integrated applications work.

The workflow begins at entry port `P1`, which periodically polls the `Call Centre` log to find new phone calls. Every phone call results in a message that is added by communicator task `T1` to slot `S1`. The body of the message holds the polled data as stream. Every port is provided with only a single communicator task, except for entry port `P1` which has a mapper task. This mapper `T2` maps inbound messages onto outbound messages that conform to a canonical XML schema that represents phone call records. Inside the process, task `T3` filters out messages that have a toll-free call. Then, messages containing calls with a cost are replicated to the `Human Resources System`, so that one copy can be used to query this application, by means of port `P2`. Next, task `T4` enriches the other correlated copy with the information returned by the `Human Resources System` and then task `T5` replicates this enriched message to the `Payroll System`, the `Mail Server`, and the `SMS Notifier`. Exit port `P3` writes to the `Payroll System` debit orders that conform to the data model in this application. The copies sent to the `Mail Server` and the `SMS Notifier` go first thought filters to prevent exit ports `P4` and `P5` from receiving messages without enough information (e.g., destination e-mail address and destination phone number, respectively).

Experimental Results

Figure 12 presents the results of our experiments. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since the values we got for R^2 were 0.994, 0.996, 0.998, 0.997, and 0.998 for 1, 2, 4, 6, and

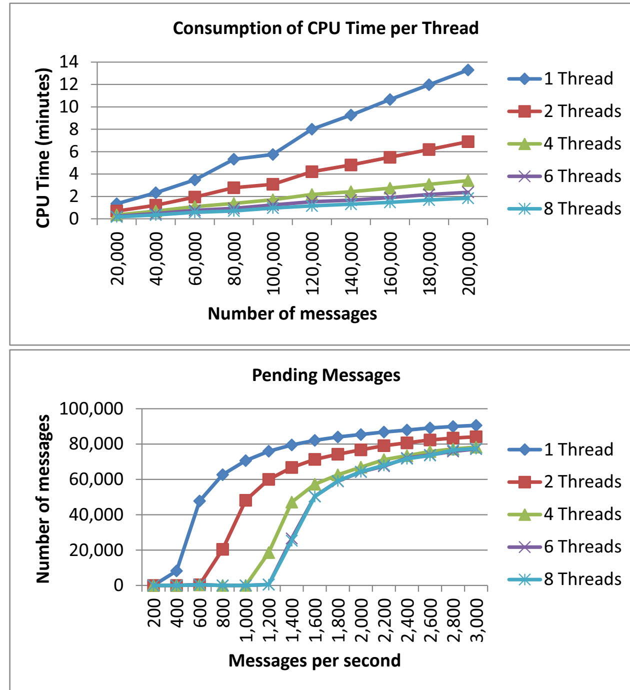

Figure 12. Experimental results for the Unijuí University solution.

8 threads, respectively. The graph depicted for this variable shows that, as a consequence of only having four physical CPU threads available in the processor, running the integration solution with 6 and 8 threads does not help reduce the consumption of CPU time per thread very significantly, as was the case for 1, 2, and 4 threads.

The graph depicted to show the number of pending messages indicates that the integration solution supports a message production rate r until $1,200$ messages per second when using 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. Furthermore, these experiments indicate that for an $r = 3,000$, there is no much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. In this case study, the integration solution only starts to accumulate messages for 1 thread with an $r = 400$. 2 and 4 threads allow to run the integration solution without accumulating messages with an $r = 600$ and $r = 1,000$, respectively.

3.2. Huelva's County Council

This case study consists of a real-world integration problem that builds on a project to automate the registration of new users into a unique repository of the Huelva's County Council. This repository contains information about users that comes from both a local application and a web portal. It is expected that every new user is notified and provided with his/her digital certificate by secure e-mail.

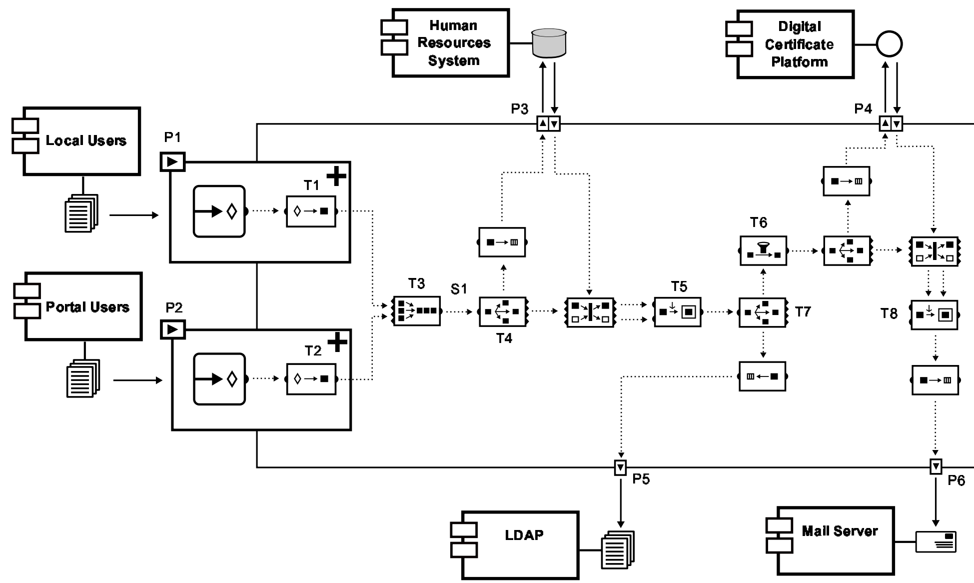

Figure 13. The Huelva's County Council integration solution.

The Software Ecosystem

The integration solution involves six applications, namely: Local Users, Portal Users, LDAP, Human Resources System, Digital Certificate Platform, and Mail Server. Each application runs on a different platform, and, except for the LDAP, the Digital Certificate Platform, and the Mail Server, they were not designed with integration concerns in mind.

The Local Users is the first application developed in house; it aims to manage the county council information systems' users. Note that, this is a standalone application and does not provide an authentication service. The Portal Users is an off-the-shelf application that the web portal uses to manage its users. In addition, a unique repository for users has been set up using an LDAP-based application, so that it can provide authentication access control for several other applications inside the software ecosystem. The Human Resources System is a legacy system developed in house to provide personal information about the employees. It is a part of the integration solution since we require information like name and e-mail to compose notification e-mails. Another application developed in house is the Digital Certificate Platform, which aims to manage digital certificates; it was designed with integration concerns in mind. Amongst other services, this application can be queried to get a URL that temporarily points to a digital certificate that users can download after authenticating. Finally, the Mail Server runs the Council's e-mail service, which is used exclusively for notification purposes.

Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, cf. Figure 13. Some ports use text files to communicate with `Local`

Users, Portal Users, and LDAP; the Human Resources System is queried by means of its database management system; and, the communication with the Digital Certificate Platform and the Mail Server is performed by means of APIs. Translator tasks were used to translate messages from canonical schemas into the schemas with which the integrated applications work.

The workflow begins at entry ports P1 and P2, which periodically poll the Local Users and Portal Users logs to find new users. Every port is provided with only a communicator task, except for ports P1 and P2 that also have a mapper task. In both ports, every user record results in a message that is added by the communicators to their corresponding slots. The body of the message holds the data that has been polled as a stream. Thus, mappers T1 and T2 map the inbound messages onto outbound messages that conform to a canonical XML schema that represents user records. Inside the process, task T3 gets messages coming from both ports and adds them to slot S1. Replicator task T4 creates two copies of every message it gets from this slot, so that one copy can be used to query application Human Resources System by means of port P3, for information about the employee who owns a user record. Next, task T5 enriches the other correlated copy with the information returned by the Human Resources System and then task T7 replicates this enriched message with copies to the LDAP and the Digital Certificate Platform. The new user record is written to the LDAP by means of exit port P5. Before querying the Digital Certificate Platform, task T6 filters out messages that do not include an e-mail address. Messages that go through task T8, which enriches them with the corresponding certificate. Finally, exit port P6 communicates with the Mail Server application to send the certificate and notify the employee about his/her inclusion in the LDAP.

Experimental Results

Figure 14 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since the values we got for the R^2 coefficient were 0.994, 0.994, 0.996, 0.996, and 0.997 for 1, 2, 4, 6, and 8 threads, respectively. The graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. This behaviour is attributed to the limit of four physical CPU threads in the processor. This explains why adding more threads to the integration solution, does not result in a significant reduction of the total CPU time per thread.

The graph depicted to show the number of pending messages, indicates that the integration solution supports a message production rate r until 800 messages per second when using 4, 6, or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. If the message production rate r ranges from $1,600$ – $3,000$, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. If running the integration solution with 2 threads, no messages are accumulated until $r = 600$. A similar behaviour when using 4–8

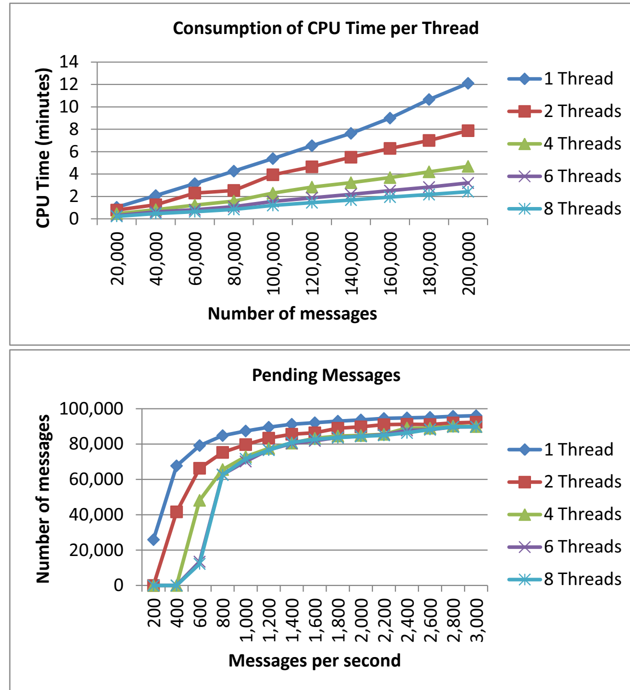

Figure 14. Experimental results for the Huelva's County Council solution.

threads can be observed in this experiment, which is attributed to the limit of four physical CPU threads in the processor.

Note that this case study and the next one are the ones that require more CPU to complete. Note that even in these cases, our proposal is able to handle a workload as high as 400 messages per second without getting collapsed.

3.2.1. Travel Search

Searching for flights and hotels is a well-known integration problem in the context of travel agency business systems. The goal is to devise an integration solution that makes it easier searching for the most inexpensive combination of flights and hotels for a desired travel with a limited budget, so that the customer can decide which one he/she shall book.

The Software Ecosystem

The integration solution involves two applications, namely: Flights Façade and Hotels Façade. Both applications have APIs that allows for querying several flight and hotel companies, but do not allow to restricts the results by price. Requests to the Flights Façade return information about all of the flights it can find for a specific date, departing, and destination city. Likewise, the Hotels Façade responds with all of the available hotels in a destination city for a specific date. The integration solution must provide an API so that client applications can search for flights and hotels.

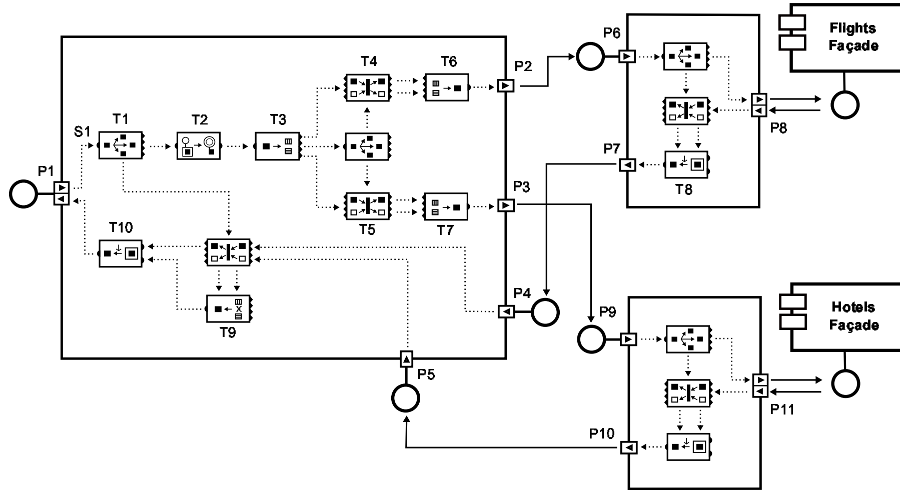

Figure 15. The Travel Search integration solution.

Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process and two wrapping processes, cf. Figure 15. The orchestration process uses a port to publish an API for searching combinations of flights and hotels. The `Flights Façade` and the `Hotels Façade` applications were provided with wrapping processes that allow to query them and remove entries that exceed the maximum affordable price for flight and hotel, respectively. We have decided to implement this functionally outside of the orchestration processes, so that it can be reused in other integration solutions.

The workflow begins at entry port `P1`. This port receives request messages and adds them to slot `S1` inside the orchestration process, from which replicator task `T1` gets them. The replicator creates two copies of every message; the first copy is used to search for flights and hotels, whereas the second is used to remove combinations of flights and hotels that exceed the total budget from the response. Task `T2` promotes the request id from the body of the message to the header of the message, so that it can be used for correlation purposes in tasks `T4` and `T5`. Prior to the correlation, the first copy is chopped by task `T3` into different messages, so that tasks `T6` and `T7` can assemble the outbound messages used to request all possible flights and hotels, respectively. The wrapping process for the `Flights Façade` receives request messages from exit port `P2`, queries the `Flights Façade` application by means of port `P8`, slims the responses in task `T8`, and, then, by means of exit port `P7` writes the responses to the orchestration process. Symmetrically, the wrapping process for the `Hotels Façade` queries its corresponding application with messages received from exit port `P3`. Back to the orchestration process, task `T9` builds every possible combination of flights and hotels returned by the wrapping processes, and, finally, task `T10` slims the response to ensure none of the combinations exceeds the maximum budget.

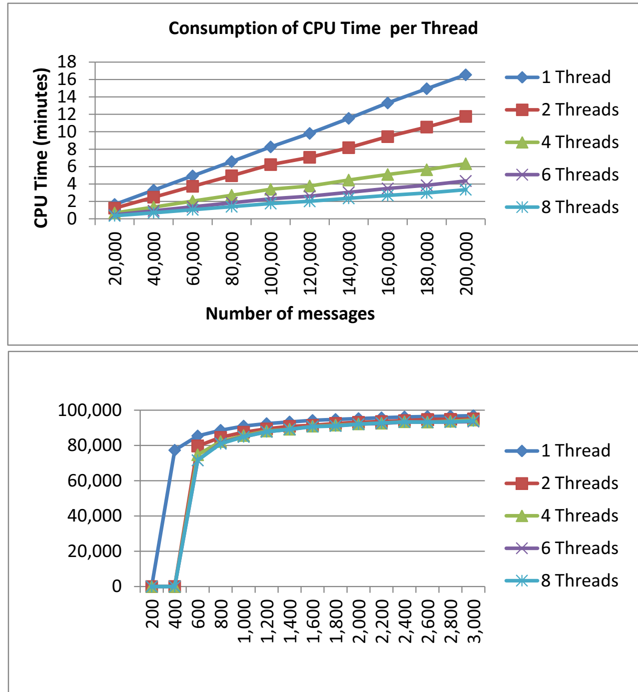

Figure 16. Experimental results for the Travel Search solution.

Experimental Results

Figure 16 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since R^2 is 0.999, 0.998, 0.998, 0.998, and 0.999, and 0.997 for 1, 2, 4, 6, and 8 threads, respectively. Furthermore, the graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. Then, adding more threads apparently does not reduce the consumption of CPU time per thread. This behaviour is attributed to the limit of four physical CPU threads in the processor.

The graph depicted to show the number of pending messages indicate that the integration solution supports a message production rate r until 400 messages when using 2, 4, 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads. If the message production rate r ranges from 800 – $3,000$, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. The results depicted in the graph indicate for all numbers of threads t, the resulting number of pending messages is similar. This behaviour is due to the fact that the Travel Search integration solution uses more tasks that depend on two or more messages, such as the correlator, assembler, and slimmer. This has an impact on the time the integration solution requires to process a request. Furthermore, this integration solution involves three

processes. The greater the number of this type of tasks and the number of processes, the more time is required to process messages.

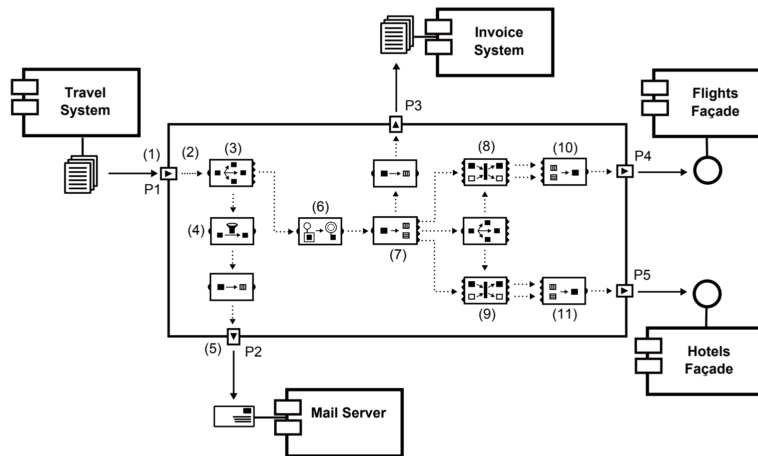

Figure 17. The Travel Booking integration solution.

3.3. Travel Booking

Not only requires a travel agency an integration solution that eases the process of searching for flights and hotels, but they also need to automate the booking process. Thus, the goal is to devise an integration solution that takes a travel booking request as input and books the flights and the hotel specified.

The Software Ecosystem

The integration solution involves five applications, namely: Travel System, Invoice System, Mail Server, Flights Façade, and Hotels Façade. The Travel System is an off-the-shelf software system that the travel agency uses to register information about their customers and booking requests. The invoice service runs on the Invoice System, which is a separate software system that allows customers to pay their travels using their credit cards. The Mail Server runs the e-mail service and is used for providing customers with information about their bookings. The Flights Façade and the Hotels Façade represent interfaces that allow booking flights and hotels. They both, in addition to the Mail Server, represent applications that were designed with integration concerns in mind. Contrarily, the Travel System and the Invoice System are software systems that were designed without taking integration into account, thus, the integration solution must interact with them by means of their data layer. The only assumption we make is that every booking registered in the Travel System contains all of the necessary information about the payment, flight and hotel, and a record locator which uniquely identifies the booking.

The integration solution must periodically poll the Travel System for new travel bookings, so that flights and hotel can be booked, the customer can be invoiced and provided with a piece of e-mail with the information about his/her travels.

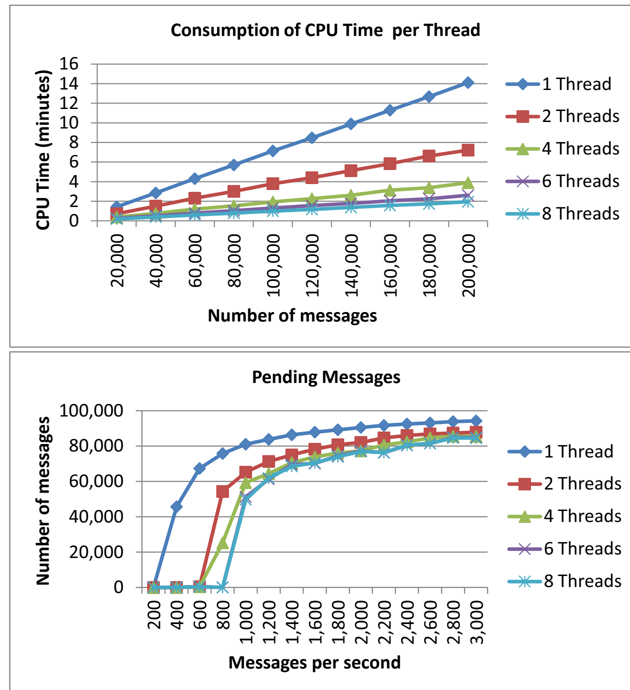

Figure 18. Experimental results for the Travel Booking solution.

Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, cf. Figure 17. Some ports have access to the `Travel System` and the `Invoice System` data layers by means of files. Translator tasks were used in the process to translate messages from canonical schemas into the schemas with which the integrated applications `Invoice System` and `Mail Server` work.

The workflow begins at entry port `P1`, which periodically polls the `Travel System` to find new bookings. Bookings are stored in individual XML files. For every booking, the entry port inputs a message to the process, which is in turn added to slot `S1`. Task `T1` gets messages from this slot and replicates them, so that one copy is used to send the e-mail to the customer and the other is used to prepare the invoice and the booking. The first copy goes through filter task `T2`, which prevents exit port `P2` from receiving messages without a destination e-mail address. Task `T3` promotes the record locator from the body of the message to the header of the message, so that it can be used for correlation purposes in tasks `T5` and `T6`. Prior to the correlation, the second copy is chopped by task `T4` into different messages, so that one outbound message with the payment information goes to the `Invoice System` and tasks `T7` and `T8` can assemble the messages used to book the flights and the hotel, respectively.

Experimental Results

Figure 18 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since R^2 is 1, 0.999, 0.998, 0.999, and 0.998 for 1, 2, 4, 6, and 8 threads, respectively. A great reduction in the consumption of CPU time is achieved when adding more threads until the limit of 4 threads. Then, adding more threads apparently does not reduce the consumption of CPU time per thread, because the CPU has only four physical threads available.

The experimental results that we obtained for the number of pending messages, indicates that this integration solution supports a message production rate r until 800 messages when using 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads. Furthermore, these experiments indicate that for a very high message production rate r of $3,000$ there is no much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. If running the integration solution with 2 and 4 threads, no messages are accumulated until $r = 600$.

4. Conclusion

In this chapter we have introduced a Software Development Kit to implement integration solutions. It is a Java-based command query API [9] and its architecture is organised into two layers, namely: core and toolkit. The former provides a number of classes and interfaces that implement the abstractions of our Domain-Specific Language introduced in Frantz et al. [5], whereas the latter extends some abstractions in the former to provide concrete adapters and tasks that support several integration patterns. The framework layer includes a runtime system, which allows to execute integration solutions. The execution model of our runtime system is totaly asynchronous. Supporting transactions in this kind of execution models requires more research effort. Tasks that comprise the integration workflow of integration solutions, notify they are ready to be executed when they have messages available in all their inputs. For each execution of a task, our runtime system generates a work unit and adds it to a work queue. There can be several threads that listen to this queue and are responsible for their actual execution. We have applied our Software Development Kit to four case studies carried out. In our experiments, we measured two variables: the consumption of CPU time per thread and the number of pending messages. The results indicate that our proposal is viable and can be used to solve real-world integration problems. Every case study was modelled using the Domain-Specific Language introduced in Frantz et al. [5] and implemented using the Software Development Kit introduced in this chapter. Our Software Development Kit was designed to be integrated with Open ESB [10], so that we can reuse its catalogue of adapters.

References

[1] D. Messerschmitt and C. Szyperski, *Software EcoSystemm: Understanding an Indispensable Technology and Industry*. MIT Press, 2003.

[2] "*Business Process Model and Notation Specification Version 2.0*," 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/.

[3] "*Web Services Business Process Execution Language Version 2.0*," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[4] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[5] 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*, vol. 20, no. 2, pp. 143–176, 2011.

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

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

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

[9] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[10] T. Rademakers and J. Dirksen, *Open-Source ESBs in Action*. Manning, 2009.

Reviewed by

Dr. Carlos R. Osuna (crivero@uidaho.edu) - Department of Computer Science, University of Idaho (United States) and

Dr. Carlos Müller Cejás (cmuller@us.es) - Department of Computer Languages and Systems, University of Seville (Spain).