POLITECNICO DI MILANO

Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Multi-level Service Monitoring

Relatore: Prof. Ing. Sam GUINEA

Tesi di Laurea Specialistica di:
Stefano SAINAGHI - Mat. 751496

Anno Accademico 2011/2012

Contents

List of Figures

List of Tables

Abstract

Nowadays, applications are more and more realized as service compositions, that is, assembling small, independent, accessed on-demand, possibly heterogeneous pieces of code in an easy and flexible way. Cloud computing takes the service abstraction one step further and applies it to new kinds of resources, both hardware and software. On the other hand, service-based applications are way more complex to manage than traditional ones. They, in fact, are naturally structured on multiple, interdependent layers. This means we need a new way to control their performance, which would take into account all the functional and non-functional dependencies and would be able to react to the frequent changes of services.

There are many existing runtime service management solutions. However, in the context of cloud computing, they can only be considered partial solutions: some of them intervene at specific levels only, while others do not consider the adaptation process in its totality. This thesis refers to a project in which runtime management is obtained through a multi-level MAPE cycle. The approach consists of four phases that are applied concurrently to multiple layers. The four phases are: monitoring and correlation; analysis of adaptation needs; identification of cross-layer strategies; adaptation enactment.

This thesis focuses on the realization of the first phase in the cycle: monitoring. The work proposes an approach in which we extract independent execution data from each layer, and correlate them to obtain a clear and holistic representation of the application's functional and non-functional behaviour. In addition, the thesis presents two performance visualization tools. The first is real-time, and can show live trends. The second uses historical data to allow the designer to "drill down" to discover the reasons behind past application failures. The entire approach has been tested with various load simulations on a distributed application.

Descrizione della tesi

Oggigiorno, le applicazioni vengono sempre più spesso realizzate tramite composizioni di servizi, ovvero combinando tra loro blocchi di codice solitamente piccoli, tecnologicamente eterogenei, indipendenti e utilizzabili "on demand". Per facilitare ciò, negli ultimi anni sono state realizzate diverse tecnologie, e molte altre ne arriveranno in futuro. Il cloud computing, poi, ha comportato un'ulteriore innovazione, estendendo di fatto il concetto di "affitto" a qualunque tipo di risorsa informatica, dal comune programma software fino all'infrastruttura vera e propria.

Le applicazioni basate sui servizi, però, sono molto più difficili da controllare di quelle tradizionali. La loro struttura naturale, infatti, si articola su più livelli: da un livello "generale", astratto, al culmine, ogni aspetto dell'applicazione (struttura interna dei processi, dipendenze) viene approfondito discendendo la gerarchia, fino ad arrivare all'ultimo strato in cui sono contenuti gli effettivi elementi implementativi. Assumendo una base "cloud", inoltre, anche l'infrastruttura fisica sottostante deve essere presa in considerazione, aggiungendo un ulteriore livello in fondo allo stack. Tutto questo rende decisamente più complicato il mantenimento delle prestazioni, in quanto i diversi livelli applicativi sono legati indissolubilmente tra di loro, e operare modifiche ad uno di essi influenza inevitabilmente anche gli altri. Occorre, perciò, adottare un processo che tenga conto della struttura multi-livello e di tutte le possibili dipendenze, funzionali e non, tra i componenti dell'applicazione. In secondo luogo, i servizi stessi sono soggetti a continue evoluzioni, che non possono certo essere monitorate completamente da un analista "umano": per questo motivo, quindi, un ipotetico middleware di controllo deve anche fare in modo che l'applicazione si adatti "da sé", reagendo automaticamente a tutti i possibili cambiamenti.

Le soluzioni che sono state presentate fino ad ora sono solamente parziali, specialmente se considerate nell'ambito "cloud": alcune, sulla base delle applicazioni tradizionali, si occupano di un singolo livello e trascurano gli altri; altre, invece, si focalizzano su una specifica fase del processo adattativo, curando poco il resto. Il progetto di cui questa tesi è parte definisce un approccio integrato basato su un ciclo real-time di auto-adattamento, suddiviso, sulla base del noto modello MAPE (Monitor, Analyse, Plan and Execute), in quattro macro-fasi: raccolta di informazioni dall'applicazione; determinazione dei requisiti di adattamento; identificazione della strategia multi-livello da adottare; effettuazione della strategia sull'applicazione. Ognuna di tali fasi opera nell'ambito multi-livello, definendo perciò delle strategie che siano attente a tutti gli elementi possibilmente condizionati da ciascuna azione.

Il contributo pratico dato da questa tesi al progetto consiste nella realizzazione del primo dei quattro blocchi, delegato alla fase di monitoring. Quella che è stata implementata è un'architettura flessibile, costituita da piccoli sotto-blocchi funzionali, disaccoppiati tra loro, in grado di estrarre dati da ogni strato dell'applicazione, processarli per ottenerne indici di prestazione più rilevanti e, soprattutto, produrre informazioni correlate, in modo da poter avere una rappresentazione chiara e completa del comportamento dell'applicazione stessa. In aggiunta, esulando dagli scopi del ciclo, sono stati sviluppati due tool di visualizzazione, uno operante in tempo reale e l'altro differito, "storico", che permettono all'analista di, rispettivamente, osservare l'andamento "live" dell'applicazione e/o effettuare un'analisi approfondita, avendo a disposizione tutti i dati correlati, del suo comportamento in un dato periodo nel passato. L'intero sistema è stato poi accuratamente verificato su una reale composizione distribuita, operandovi dei test di carico ed analizzando i dati registrati dal sistema.

Credits

Al professor **Sam Guinea**, per le svariate mattinate perse a spulciare codici indecifrabili e, più in generale, per tutto l'aiuto datomi durante tutto l'arco di questo lavoro nonché per avermi permesso di fare questa bella e interessante esperienza.

Alla mia **famiglia**, per l'ormai consueta fiducia incondizionata.

Ai miei **amici universitari**, per i momenti di divertimento passati insieme in questi sei anni.

Ai miei due **nonni**, per l'onore di essere stato loro nipote.

A **me stesso**, che forse era il caso che sacrificassi un po' meno per questa (in fondo) stupida università.

Chapter 1

Introduction

Services have subverted the traditional dogmas of Information Technology, leading to a more sustainable use of computer resources. To become a standard for everyone, though, they first have to resolve several problems, the most important being performance uncertainty. This thesis aims to provide a valuable solution to it.

The chapter is divided in three sections. Section 1.1 explains what services are in the IT world and which challenges they bring. Section 1.2 gives a brief illustration of the proposed approach. Section 1.3 summarizes the implementation work at the center of the thesis. Section 1.4 sketches instead the arguments treated in the various chapters of the essay.

1.1 Problem

Until (relatively) few years ago, IT was a strictly personal concept. Everyone, from single individuals to huge organizations, used to build his system entirely by himself, following the "in-house" policy. The costs to be sustained to maintain this situation, though, soon became prohibitive, while at the same time the rapid diffusion of the Internet, along with reusability-enabling languages (as Java, C/C++, XML and so on), made cooperation through digital media incredibly easy and cheap. This context encouraged the take-off of on-demand services, in which the user could request and access a (Web) resource at a time specified by himself; since he did not acquire the actual resource, but rather its utilization, he was able to manage his costs much more efficiently, and reduce waste. Due to that, the on-demand business grew larger and larger, and IT officially entered the "service-based era".

An excellent definition of services, with regard to the IT environment, has been given by Prof. Mike Papazoglou and Dr. Dimitrios Georgakopoulos [1], two of the most eminent personalities in this field:

> Services are self-describing, open components that support rapid, low-cost composition of distributed applications. Services are offered by service providers –organizations that procure the service implementations, supply their service descriptions, and provide related technical and business support. Since services may be offered by different enterprises and communicate over the Internet, they provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration. Service descriptions are used to advertise the service capabilities, interface, behavior, and quality. Publication of such information about available services provides the necessary means for discovery, selection, binding, and composition of services.

Let's clarify the most important points in such statement:

- a service is basically a component, that is, a reusable unit of composition with a specific interface contract. In addition, it must include a document, written using an appropriate paradigm, which illustrates its capabilities (self-descriptiveness), and must be accessible through publicly available standards (openness);
- services ensure distributed computing, that is, the splitting of the application in several smaller tasks to be executed independently by dif-

ferent entities, even unaware of each other, communicating over the Internet;

- all the information a service consumer needs to know in order to utilize it are contained in its description, which must be supplied by the provider itself.

Evolution in this area has ultimately brought to cloud computing[1]. This can be seen as a new way to intend a generic computer resource, both software and hardware: no longer a product, which is owned and thus entirely controlled by the customer who bought it, but rather a service (in the broad sense of the term), which is managed by another subject and accessed on demand by the user through the Internet (or "the cloud", from which the name). Three types of resources can be provided: application (software as a service or SaaS), that is, any specific-purpose software program; platform (PaaS), that is, a fully-equipped execution environment in which to run a custom application; and infrastructure (IaaS), that is, hardware devices as virtual machines, storage systems and so forth. The success of this technology is so big that almost all IT giants are now "moving to the cloud", some starting to sell web-based versions of their trademark products (as Microsoft's Office), others irrupting in the market with brand-new, apposite offers (as Google's Chrome OS).

Applications based on services, though, are much more complicated to manage than traditional ones. In fact, they have a multi-level nature, because their structure is realized as a stack, in which each tier (or layer) presents the application from a different perspective. The top layer in such a stack is also the most abstract one, providing a general, high-level schema of the composition; going down the hierarchy, the content of layers becomes more detailed and concrete, inspecting the sub-processes internal to each service; at the very bottom, finally, the actual implementation artefacts are found. There is not a fixed paradigm to implement the hierarchy: as we will see in Chapter 2, application developers can create their own structure, deciding how many tiers it is made of and which information to put at each of them.

Figure 1.1 gives a graphical idea of this fact, by means of a simple application example. In this case, the top layer, called Application schema, contains a representation of the composition; this, as can be seen, is made by three services, A B and C respectively, that are executed in sequence. The bottom layer (Service implementations) includes the actual entities used to

[1]http://en.wikipedia.org/wiki/Cloud_computing

carry out each service. A and B are both integrated projects (C++ and Java ones respectively); service C, instead, is in fact a service-based application itself: therefore, the hierarchy features at least one more level in the middle. The physical devices (three virtual machine in the figure) on which the services are executed must be taken into account as well, and thus constitute the ultimate, deepest tier.

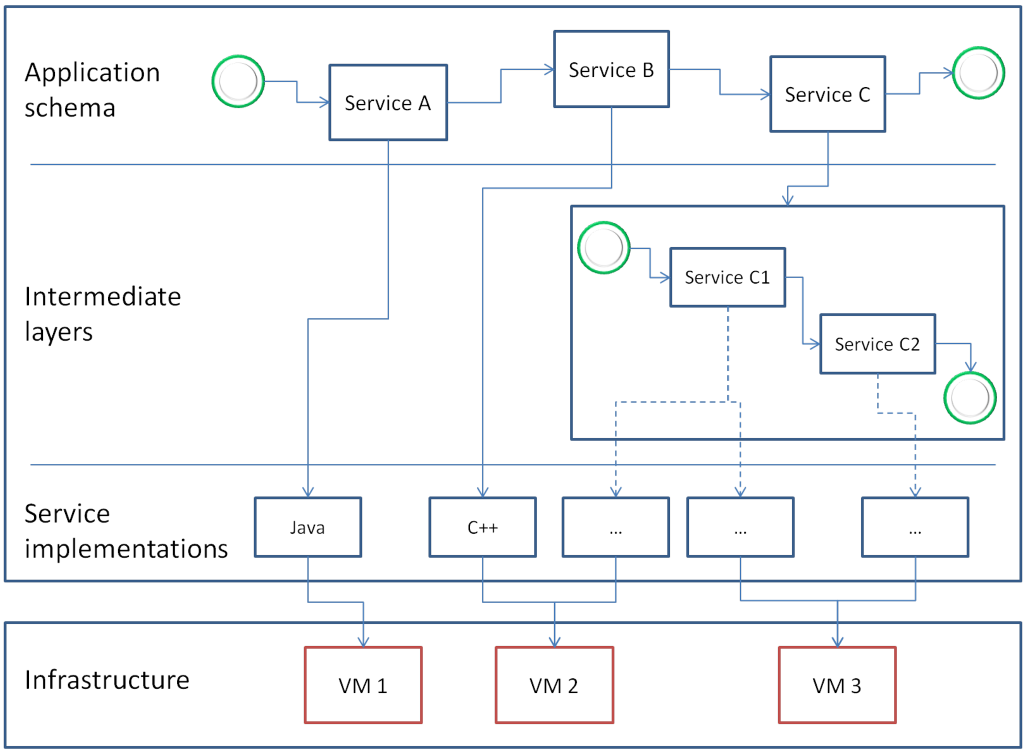


Figure 1.1: Multi-layer nature of a Service-based application

Service compositions generally imply much more complex monitoring and adaptation. First of all, each service in a composition can be implemented with a different technique, and thus requires a different control approach; this problem, however, is in part mitigated by their loose coupling, which allows to treat them independently. Second, most services are provided by external suppliers, and only their interface is available; while being ready to use, thus allowing the developer to avoid implementing the entire application by himself, they are like black, unopenable boxes, on which control can be performed from the outside only. Third, services evolve continually, with a rate so high that human overseeing is not possible; it is the application itself that must keep up with all the changes.

Multi-layering, in addition, makes matters even harder: in case a performance anomaly is detected, the analysis could have to "drill" down various levels to find the actual reason. Moreover, application layers are nearly always dependent on each other, and thus limiting the fixing to the single level not only is not enough to resolve the problem, but could even worsen the situation, causing new problems at other levels. To operate valuable maintenance, a self-executing, integrated approach is thus required, which would take into account all the cross-layer dependencies that can be possibly found to effectuate a correlated analysis and auto-operate complete adaptation strategies.

1.2 Solution

Maintaining adequate performance is an essential prerogative for every service provider. Being Web services a quite new technology, though, expertise on their adaptation is still at an embryonal stage: so far, loads of studies have been conducted about the problem, but none of them has led to a satisfactory solution. In particular, they are all somewhat partial in their content, privileging certain application layers, or adaptation steps, at the expense of the not-less-important others. The solution proposed in this work, instead, devises an architecture providing a fully-comprehensive, end-to-end, multi-layer adaptation process suitable for service compositions.

The studied approach consists of an appropriate variant of the well-known MAPE (acronym of Monitor, Analyse, Plan and Execute) loop [2], which provides a paradigm for applications to auto-adapt themselves. Figure 1.2 shows the high-level diagram of the projected system.

The picture represents a cycle (included in the dashed box) which is attached to the application and operates "beneath" it, transparently to its users. Each iteration of such cycle schedules four sequential, independent macro-phases (each corresponding to a single MAPE stage):

- Monitoring and correlation, where information about performance is retrieved from the application;
- Analysis of adaptation needs, where performance is examined to find out what is not working correctly;
- Identification of multi-layer adaptation strategies, where a sequence of adjustments is planned which would fix the malfunctioning components and all the ones influenced by them;
- Adaptation enactment, where the strategy is finally operated into the application.

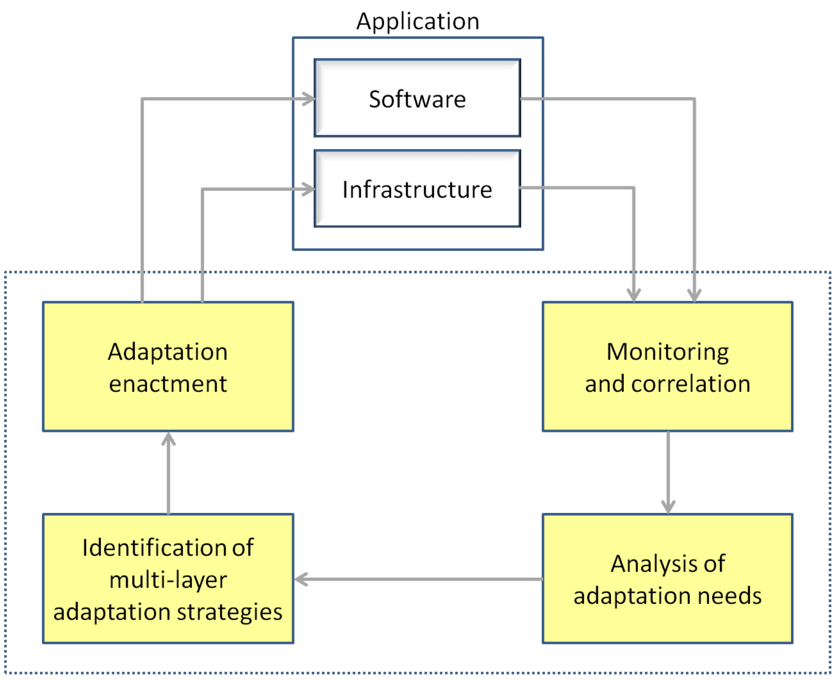

Application
Software
Infrastructure
Adaptation enactment
Monitoring and correlation
Identification of multi-layer adaptation strategies
Analysis of adaptation needs

Figure 1.2: Self-adaptation system overview

Two distinct macro-layers are identified in the controlled application: the software layer, corresponding to the application in its strict meaning (that is, its code); and the infrastructure layer, regarding all the hardware devices involved, be them virtual or not. This is actually a simplification, since an application can be made of any number and type of layers.

Each of the four "blocks" has been here extended, with regard to the "classical" MAPE one, to be able to deal with multi-layer service compositions. The monitoring block encapsulates all the data regarding entire dependency chains in a single information unit. The analysis block uses a top-down procedure to get to the real source of the problem. The identification block (perhaps the most important one) considers the adaptation actions at a dependency-chain scope, looking for all the components possibly affected by an action and integrating the strategy to fix them as well. The enactment block, finally, produces two sets of adaptation actions, one for each application layer.

The innovation brought by this project is that it provides a complete architecture for self-adaptation in the service framework, covering all the steps from detection of failures (violations of SLAs or other application objectives) to enforcement of the actions correcting them. Such an architecture improves both the performance and robustness of service-based applications, in fact amplifying their pros and attenuating the effect of their major cons. On the other hand, having to include a huge amount of reasoning within itself, especially in the identification block, the cycle could be particularly slow in its iterations, possibly conditioning in negative the application as well. It will be fundamental, therefore, to reach a compromise between completeness and complexity.

1.3 Practical contribution

This thesis focus on the first phase of the cycle, that is on "Monitoring and correlation". The purpose of a monitoring system, when attached to a service composition, is to take data from all its layers and turn them into useful information for the next step; this information, as the second part of the label reads, must be correlated, that is, it must join values of cross-layer, interdependent components in the same structure.

This is achieved through an appropriate composition of three different kinds of components. Collectors are software sensors that are attached to the application's various layers. They are used to collect raw execution data from the system. Calculators elaborate these raw data into more significant metrics. Finally, correlators merge metrics coming from different layers to construct a holistic understanding of the system's behaviour. To communicate with one another, instances exchange messages (here called events) by means of a publish-and-subscribe middleware; this way, users can easily create all the processing sequences they need, using a common, loosely coupled mechanism to link adjacent instances.

This thesis also presents two advanced monitoring visualization tools. The first tool provides live tracking of metrics, allowing the user to control the statistics about his application as soon as they are extracted. The second tool is instead a more functional dashboard, in which the past trends of correlated metrics can be retrieved from a database and displayed together. This latter instrument is of particular importance, because it permits business analysts to perform a drill-down process, getting to the actual source of the problem; at the same time, moreover, they can verify whether the metrics they decided to keep tracked represent good indicators for the application and, in the negative, refine the monitoring configuration.

1.4 Thesis content

Chapter 2 summarizes some of the studies conducted on the subject so far, grouping them according to the loop phase they focus on. Reading it, it can be seen how a complete solution, which would take all the key factors into account and would be clearly delineated at the same time, does not exist yet.

Chapter 3 allows the reader to understand “the real deal”, that is, the infrastructure proposed to face the problem. First, it is described how the structure of applications is represented in this work; then, for each cycle block, it is explained the contribute it brings, along with the operations carried out in order to accomplish it.

Chapter 4 regards the practical aspect of the thesis, in which the first block of the cycle (that is, the one in charge of monitoring) has been realized. The implemented system is here illustrated exhaustively, from the applications it can be operated on to the components it is made of and how they are combined together.

Chapter 5 documents the tests executed to demonstrate the functioning of the implemented system. Precisely, the setup of the test-bed application is elicited, the adopted strategy is depicted and the correspondent results are commented. Also, the impact of monitoring on the performance is here discussed.

Chapter 6 closes the thesis with general considerations about the work as well as some indications about future efforts in this field.

Chapter 2

State of the art

Service-based computing is in continuous evolution and expansion, and so are the studies of its possible adaptation techniques. So far, though, most of them concentrate on one phase of the previously depicted cycle, paying too little attention to the other ones. Others, instead, do treat the entire adaptation process, but they do so in a very high-level, abstract way, leaving too many important questions unresolved. Here, the content of some of such studies is presented, tacitly asserting the innovativeness of the work treated in this report.

The chapter is divided in four sections. Sections from 2.1 to 2.3 contain studies regarding the first three phases of the loop respectively (the fourth phase, adaptation enactment, is a "pure-action" stage where the studied strategy is simply applied, and as such it is implicitly included in every study). Section 2.4 describes instead a complete solution based on something else than the four-step cycle.

2.1 Monitoring and correlation

2.1.1 Monere

Monere [4] is an online monitoring system for cross-domain Web service compositions. Developed at University College London, its purpose is to improve the availability of applications by shortening the time to diagnose their possible failures; in particular, it exploits the properties of single components to get as much knowledge as possible about the application structure and performance, and to provide analysts with useful information that will let them locate problems faster.

The architecture of Monere is the same for each domain it runs within, and consists of three key components: a Monere server, a set of Monere agents and the Monere Information Service (MIS). The agents perform the operative task, that is, they are in charge of gathering metric measurements and discovering all the possible dependencies among application elements; the server receives all the information from agents operating within its domain, persists them and shows them through a detailed user interface; the Information Service allows cross-domain interaction by making available information about components in its domain to agents running in remote domains. To carry out its function, an agent is attached a set of plug-ins, each of them representing a particular resource type and including the instructions to discover it and retrieve its metrics; several resources operating at any level have their plug-in representation, making it possible to monitor a huge number of indicators, from latency of Web services down to utilization of OS devices.

What really allows to exploit such a great amount of data to the fullest, though, is dependency discovery. Monere, in fact, looks for all the possible relations between the application components and builds the related dependency tree, which makes metric correlation immediate. There are three stages in the dependency discovery process: first, the agents, thanks to their plug-ins, discover all the components running in their own domain; next, components are analysed to identify the services, executed in the same domain or not, they rely on, and establish the respective dependencies; finally, agents send all the information they obtained to their server, which puts them together to determine the application tree.

A great quality of Monere is that it does not require modification on the monitored application, as it makes use of already included component-specific APIs. In terms of performance, instead, experiments run to confront the two cases of Monere enabled and disabled, respectively, found a relative slowdown of about 8%, which is neither an excessive nor a negligible value. Other reliable comparative tests, though, showed an increase of failure detections as well as an average reduction in diagnose time, both of about 22%.

2.1.2 SOA4All Analysis Platform

SOA4All[1] is an EU-funded project aimed at building a scalable, distributed infrastructure allowing its users an easier utilization of services through the Web. The key element of this infrastructure is the Distributed Service Bus (DSB), a middleware dispatching messages in order to provide transparent access to all the services registered to the platform as well as communication between the services composing the platform itself. Users interact with SOA4All by means of its Studio component, which includes three sub-platforms, each addressing a specific set of functionalities; among them, the Analysis Platform (AP) [5] supplies functionalities to monitor the service performance and retrieve knowledge from it.

The AP is made by five components:

- Monitoring Mediator (or MM), which is the interface between the AP and the DSB. It takes events from the data collectors within the bus and forwards them for processing and, in the opposite direction, it communicates the management decisions that are taken in the module to the bus;

- Basic Event Processor (BEP), which can be seen as the operative element. First, it computes basic metrics from events as soon as they arrive from the MM; then, it sends both raw and derived data to high-level processing components, and updates the graphical features displayed in the User Interface. Furthermore, it manages the storage of events into the analysis warehouse;

- SENTINEL, providing a deeper analysis of the business process. It collects events from monitoring logs and puts them into a complex event processing (see Section A.3 for more details on this technology) tool, where they are correlated and/or further elaborated to generate new, higher-level-format data in real time. Such data will be then examined with the aid of semantic technologies to extract business-level knowledge;

[1]http://www.soa4all.eu/

- Knowledge Analytics (K-Analytics), which gathers data from several sources, from the BEP to user feedbacks, and then uses ontologies to abstract it and obtain more condensed, more conceptual information, in such a way that users can have a better comprehension of their applications. Differently from that of SENTINEL, thus, its input consists of a much bigger amount of data, its output is targeted at end-users (instead of analysts) and its execution takes place at batch-time rather than "live";
- UI widgets, presenting all computed information to the user in a graphical way.

In order to achieve a good compromise between computation and scalability, the SOA4All infrastructure has been studied to be deployed in a distributed fashion. With regard to monitoring in particular, each user domain runs its own Studio component, in which the Analysis Platform cares only about the services used within the domain itself. Finally, a flexible visualization of monitoring data is provided, that is, each user can specify, according to his role and his needs, the information he wants to monitor as well as the means for its representation.

2.1.3 Instance/Class monitoring

Barbon et al. [12] have devised an architecture to monitor service compositions implemented in BPEL[2] (Business Process Execution Language). This tool has three interesting properties: it is non-intrusive, that is, it does not require modifications to the application code; it supports both single-instance monitoring and class monitoring (aggregating values from all the instances of a class); and it provides a high-level language for its configuration, from which it then generates the appropriate (Java) code.

The architecture is in fact an extension of a BPEL engine (precisely, the Active BPEL one), which is a system able to read and execute processes written in BPEL. In particular, three components have been added:

- a Runtime Monitor (RTM), which manages the monitoring process. It contains a Monitor Inventory, storing all the deployed monitor instances, and two Monitor Handlers to handle the execution of currently running Instance- and Class-Monitor Instances (IMs and CMs respectively), each of which controls the value of a specific application property;

[2]http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

- a Mediator (perhaps the most important element), which links the BPEL engine with the extension system by intercepting all the relevant events and messages and handing them over to be processed by the RTM;
- an Extended Admin Console, enriching the web-based administration interface in order to visualize information about monitor instances.

Instance and Class Monitors descend from the same interface, called *IMonitor*, which includes methods to retrieve general process attributes; however, their purpose is significantly different. An Instance Monitor is attached to a single process instance: it is instantiated when a new instance starts; it updates its internal status as the instance evolves; and, finally, it is stopped after the instance stops. A Class Monitor, instead, aggregates information from all the instances of a specific class; therefore, it is instantiated only once, at the beginning, and never terminates.

When a message is received by the BPEL engine, a copy of it is gotten by the Mediator, which forwards it to the RTM; in here, the Instance Monitor Handler operates first, passing the message to the right IMs to compute their new property values; subsequently, the Class Monitor Handler is invoked to do the same for the involved CMs. To update its status, a CM must collect values from all the respective IMs, via a request/response mechanism, and aggregate them into a single parameter.

The language projected to specify the desired monitoring structure is called RTML (Run-Time Monitor specification Language). The document provides its grammar, which indicates the categories of elements recognized, that is, events, IMs and CMs, along with their formatting rules. An event is the basic unit of processing of the system; it could represent the start/end of a process, a message exchanged between two process activities, or a particular situation within the execution (when a variable assumes a certain value or a process reaches a certain state). An IM can monitor either a boolean property, obtained by combining events logically, or a numeric one, resulting from an algebraic formula. A Class Monitor, finally, is defined through the IM grammar extended with aggregate operations.

2.2 Analysis of adaptation needs

2.2.1 Influential factor analysis

Branimir Wetzstein and Florian Rosenberg of University of Stuttgart, together with other people from Vienna University of Technology [10] and

FBK-Irst (Trento) [9] respectively, have elaborated a system which monitors several application metrics and analyses them in order to find interdependencies, extract adaptation requirements and subsequently determine the adaptation strategy to enact. In fact, since it covers all the steps of the self-adaptation cycle shown in Section 1.2, this work could be classified as an overall solution; yet, it concentrates much more on the analysis one, and thus it fits this section best.

The depicted process consists of four cyclic steps. First, at design time, models of the metrics to monitor and the available adaptation actions must be defined. Relevant KPIs and their target values are defined along with the parameters possibly influencing them, distinguishing between Process Performance Metrics (PPMs), which are collected from runtime process instances, and infrastructure QoS indexes. For each action, instead, it must be specified the technique it uses and its impact on the application metrics, that is, which ones are affected positively and which ones are worsened by its enactment. This stage differs from all the others in the fact it takes place off-line and, due to that, it is not physically wired with the blocks immediately upstream and downstream of it; still, the links, and subsequently the inclusion of the modelling block in the cycle, make sense if considered from a "conceptual" point of view.

The next phase includes both monitoring and analysis. It is divided in two major sub-tasks, each of which is subject of one of the studies mentioned above: in the first one, metrics are collected and dependencies are found; in the second one, dependencies are examined and adaptation requirements are extracted. In the former, the integrated application is structured on three layers: process runtime, monitoring and analysis. The process runtime layer contains the process description and the required services. The monitoring layer contains all the tools necessary to gather the metrics from the application, make them available (via publish-and-subscribe, see Section A.2 for more details on this technology) to be treated, store the results and even display them into a dashboard. Finally, the analysis layer includes the process analyser component, which takes the entries from the database and puts them into machine learning; precisely, a decision tree is generated, in which the influential factors that lead to a success (represented by a green leaf) rather than a failure (red leaf) are highlighted.

Once a dependency tree is delivered, it must be examined. Initially, the tree paths leading to undesired behaviours are identified, and the relevant ones, that is, the ones it is more important to prevent, are chosen among them based on predefined criteria. Next, from each selected path, critical factors are retrieved in its nodes, along with their range of values in its branches, and a logical expression (a conjunction of clauses) is created to describe the violation. In the end, all such expressions are logically negated and jointed together in a single one, which goes into a specific tool that outputs its solutions (there could be more than one).

Differently from the upstream two, the third and fourth steps correspond to the ones in the MAPE loop, explained in Sections 3.2.3 and 3.2.4 respectively: in them, indeed, the possible strategies are identified and ranked to pick the best one, which will be finally applied. For each adaptation option received from the previous block, a set of suitable, non-objective-conflicting actions (that is, no action in the set has a negative impact on any metric to be fixed) is generated; if it is empty, the option is automatically discarded. The so-created strategies are then confronted based on the "amount" of negative impact they bring: the one with less negative effects wins.

Experimentations ran on the dependency analysis task by simulating influential factors verify the theoretical expectations almost completely. The only problem arisen in some tests is the non-consideration of indirect influences, that is, when a KPI violation is due to factor A, which in turn is due to factor B, the influence of B on the violation itself is neglected. To resolve this question, it is possible to "drill down", performing dependency analysis on factor A, or alternatively to remove factor A from the metric set. Furthermore, it has been found that the number of nodes in the tree increases with that of the instances taken into account, and thus pruning techniques, or again metric removal, could be necessary to eliminate likely marginal factors.

2.2.2 MoDe4SLA

MoDe4SLA (Monitoring Dependencies for SLAs) [11] is an approach studying the impact of single services in compositions to ensure an easy analysis of failures at runtime. In particular, the work focuses on violations of SLAs (acronym of Service Level Agreements), which are provider-consumer contracts indicating a certain number of constraints (called Service Level Objectives, SLOs) regarding service performance.

The described approach is applied in two distinct phases of the service composition life cycle. First of all, at design time, all the SLAs contracted by the provider (the ones stipulated with customers as well as those did with suppliers) must be interpreted. An SLA document contains a list of points, the SLOs, which details the terms of the contract with regard to the cost and quality of the exchanged service. The very initial operation, hence, is to link each of the SLOs in the consumer SLA with those, contained in the other documents, affecting it: for every single point, the involved elements are identified, formalized and put into a graph, which must represent the dependency structure as accurately as possible. There is no imposed modelling language to implement such a graph.

The next step in the design phase is impact evaluation: the graph is "parsed" in order to numerically quantify the weight of dependencies included in it. To do that, some kind of handbook must be provided which associates each block in the graph with the correspondent algebraic formulas. Once formulas have been generated, parameter values are put into them to compute the expected impact of each factor. At design time, though, some values are undefined: for example, the number of times an item already obtained will be reused instead of re-invoking the respective service, or the ratio with which a service is preferred to another one in a mutually exclusive choice; in cases like these, estimates calculated using previous process instances are exploited.

The second phase takes place during service execution. Data about running instances is stored in the database, which is made by three tables: Composite Service, Service and Message. Then, all the entries related to the same process instance are put together to compute SLOs and compare them to the agreed thresholds; for each parallel, the respective service in the dependency graph is coloured green, yellow or red according to its outcome. In addition, estimations are evaluated against "real-life" data, and impact factors are taken into account. These three elements allow a simple detection of the services violating their SLA, and thus enhance management of service compositions.

2.3 Identification of multi-layer adaptation strategies

2.3.1 Taxonomy-driven adaptation

Popescu et al. [7] have studied an adaptation approach based on taxonomies. In their work, they identify all the possible constraint violations, called mismatches, that can take place in an application, classifying them into specific hierarchies and resolving them using predefined templates. Applications, and subsequently mismatches and templates, are structured on three layers: the organisational layer, which illustrates the application in terms of its actors and the relations among them; the behavioural layer, which includes the flow of execution; and the service layer, which presents the actual services exploited in the process along with their providers.

This approach is based on three key concepts: event, taxonomy and template. Events are used to signal the occurrence of mismatches and start the adaptation process. Taxonomies group same-layer mismatches in tree hierarchies, each with the most general mismatch at the root and the most specific mismatches at the leaves. Templates, instead, are BPEL processes which are activated when a particular mismatch must be solved. Linking taxonomies, templates and the application logic itself to one another by means of events allows great flexibility: each of the three elements can be developed autonomously and rearrange interactions by simply changing the events it uses. Likewise, the use of hierarchies grants efficiency, as the most appropriate template is always operated, and robustness at the same time, as more general templates are available when no exact ones can be found.

The adaptation procedure is carried out in three steps. Initially, when a mismatch occurs, the respective event is triggered and a message is sent to start recovery. Next, a component called Adaptation Matchmaker looks for an appropriate template in the repository. The research leads to one of the following results, from best to worst:

- exact: the template found fixes that precise mismatch, that is, it is associated with its tree node;
- plug-in: the template deals with a more general mismatch, that is, it is associated with a node above in the tree;
- subsume: the template fixes a more specific mismatch (a node deeper in the tree);
- failed: no compatible templates are found.

If a template is found, its action sequence is finally started in order to enact the adaptation. Such actions can themselves trigger new mismatches, which launch their own adaptation process and in turn can trigger more mismatches, and so on; the more “external” process, thus, waits for all its dependencies to be solved to resume its execution. Since mismatches can arise at any layer, this mechanism automatically ensures cross-layer adaptation. It is fundamental, however, that dependencies among templates do not form loops, because that would cause a deadlock; to avoid that, the whole template-dependency chain is always checked prior to its actual activation.

2.3.2 Cost-based optimization

Leitner et al. [8] analysed the issue of finding the best adaptation strategy in service composition on a mathematical basis. In their work, they rank action sequences according to their cost, a broader indicator including not only the penalties due to possible violations of the SLOs indicated in the agreement between customer and provider, but also the price to pay in order to carry out the strategy. Considering this second factor is very important: it is not uncommon, in fact, that performing adaptation costs more than paying penalties, and to take into account only the options complying with SLA can actually imply a much higher expense for the provider.

Since optimization is a proactive task, that is, it must prevent SLO violations rather than fixing them a posteriori, its main challenge is that not all the information are available: indeed, it is not possible to know in advance whether requirements will be violated or not. To solve this problem, the optimizer gets part of its input from a component named Violation Predictor, which uses machine learning to estimate future SLO values from those of historical application instances. Downstream of the optimization process, instead, adaptation actions can be of three types: data manipulation, in which only some attributes are changed without altering the composition; service rebinding, which can be more or less difficult depending on the nature of the replacing service; and structural adaptation, in which the composition is somehow redesigned.

The strategy research is thus formalized as an optimization problem: given a set of SLOs, each with its penalty and measure functions; a set of possible adaptation actions and related (constant, for simplicity) costs; and a transformation function, describing the effect of an action sequence on a composition instance; the objective is, obviously, to minimize the total cost, obtained by the sum of penalties for violated objectives and costs of applied actions, without breaking any application constraint.

Three typologies of algorithms have been contemplated in the study: branch-and-bound, local search and genetic algorithm. Branch-and-bound is a deterministic algorithm which examines all the possible action combinations following a tree path with pre-pruning of suboptimal solutions, which means it avoids deepening the tree if either the current solution already fixes all the SLOs or a conflict has arisen between two actions. Local search, instead, heuristically takes an initial solution and tries to improve it by looking in its neighbourhood, that is, adding or removing one single action at a time. Finally, genetic algorithm (GA) mimics the processes of evolution in biology (selection, crossover and mutation), iterating them a fixed number of times (or generations) to refine the solution action set.

Experimentations performed on all the algorithms basically confirm the theoretical expectations: branch-and-bound, while always finding the optimal solution, requires a huge amount of instances to be examined, and thus is best suitable (with optimizations) for small problems, in which the number of possible sets is limited; both the other two approaches are much faster, and two optimizations of theirs, GRASP for local search and Memetic Algorithm (MA) for GA respectively, provide solutions of good quality as well. Furthermore, and most importantly, it has been demonstrated the utility of both predicting SLA violations and considering prices of adaptation actions into strategy selection, which lead to better performance as well as a significant cost reduction.

2.3.3 VieDAME

VieDAME (Vienna Dynamic Adaptation and Monitoring Environment) [6] is a system extending BPEL functionalities to provide monitoring and runtime adaptation. In this case, runtime adaptation means that, for each activity of the BPEL process, its service (partner link in the BPEL terminology) is determined dynamically, during the execution of the process itself, based on specific QoS attributes. Including monitoring as well, VieDAME could be also regarded as an overall solution; actually, though, the study details the adaptation part much more. Both the new features are added in a non-intrusive way, without changing neither the BPEL process nor the involved services.

The VieDAME system is split in two parts: the VieDAME Core, which takes care of communication among components, and the VieDAME Engine Adapter, which injects the extensions into the BPEL engine using Aspect-Oriented Programming (AOP), a paradigm allowing to force additional code at specific points into the base system. Also, the BPEL description of partner links is extended, so that to include replaceability (boolean attribute), some QoS parameters and a list of services alternative to it. When VieDAME is enabled, partner service invocations (performed via SOAP[3]) are caught by a new device, called Interception and Adaptation Layer (IAL), which effectively puts the Engine Adapter between the engine and the actual services, giving it the possibility to manipulate the requests.

The first component operating in the VieDAME execution flow is the Monitor, in charge of pulling QoS information out of the process. Initially, this element identifies the current BPEL activity, retrieving its name and invocation context; then, it starts a timer to measure its execution time and hands over the invocation to the next component; finally, after either a result is returned or an exception is raised, it stops the timer and persists the invocation instance along with its duration and outcome.

[3]http://www.w3.org/TR/soap/

If the original partner service needs to be replaced by a better one, and if its description marks it as replaceable, the Monitor forwards the invocation to the Selector. This latter must pick the best possible alternative according to a specific criterion, be it availability, response time, or even a simple round-robin policy. Very often, the interface of the alternative service, while being semantically equivalent, mismatches that of the original one in the number or nature of required parameters: in this case, invocations must be further treated by the Transformer component, which exploits appropriate tools to transform the SOAP request and make it compliant with the new interface.

In summary, the VieDAME system provides monitoring by means of its Monitor component, while the runtime adaptation task is carried out by the Selector and Transformer ones. Several load tests have been run to evaluate the impact of each of the three components on a BPEL-based application, detecting a significant performance degradation only upon the inclusion of Transformer.

2.4 Other approaches

2.4.1 QUA for SOA

QUA is a technology-agnostic adaptation framework, that is, an architecture providing application adaptation in a way it is independent of the implementation. To do that, it first analyses the application requirements at a higher, abstract (namely, technology-agnostic) level, and then performs adaptation by exploiting the appropriate technology-specific mechanisms. In [13], the QUA middleware has been applied to Service-Oriented Architectures (see Section 4.2) to achieve multi-layer self-adaptation for them.

The system is divided in three frameworks: the Service Meta-Object Protocol (MOP), the Planning Framework and the Platform Framework. The Service MOP is the key element, as it is responsible of actually abstracting all the technology-specific data from services to produce a minimal set of standardized information. Those include:

- implementation blueprints, that is, "documents" describing the service architecture (in terms of the artefacts composing it);
- a service platform, that is, the runtime environment needed to interpret the blueprint;
- a list of the service's dependencies.

Furthermore, a quality predictor function is used to evaluate the performance of a blueprint in terms of QoS parameters, and a number of utility functions can be exploited by the user to rate the service.

Both the other frameworks exploit the Service MOP in their execution. The Planning Framework must compare all the possible implementations for the application, putting together each service composition and calculating its quality, so to determine the one to be used. Once this is found, the respective Service Platforms in the namesake framework must translate the blueprints referring to its services and finally execute them.

To comply with SOAs, QUA must be able to work across several layers and, at the same time, not to influence its trademark properties, among others loose coupling and interoperability. For the former requirement, a two-layer model has been considered, made by a service interface layer, regarding the inter-communication between services, and an application layer, dealing with the implementation of each service. QUA satisfies this requirement, as in quality analysis (within the Planning Framework) it takes into account the whole application structure to allow the development of cross-layer adaptation strategies, which can then be enacted by means of technology-specific mechanisms at both levels.

To respect the second requirement, QUA must instead bear four characteristics: it must not depend on technology-specific adaptation actions; it must be able to work with loosely as well as tightly coupled systems; it must allow the inclusion of adaptation-specific interfaces, if those are available; and it must allow the inclusion of autonomous systems. A practical experiment, documented in the study, demonstrates all these properties for the system.

Chapter 3

Solution

This study is part of a broader project, documented in [3], which has actually delineated the multi-layer cycle. Precisely, it indicates the guidelines for its four stages, that is, for each of them, what it must produce and how its process should take place. This chapter, therefore, provides a detailed, phase-by-phase description of the loop, while the next one will talk about the implemented monitoring block, on which the work has focused on.

The chapter is divided in two sections. Section 3.1 briefly explains the general structure of service-based applications which has considered in the project. Section 3.2, instead, describes in depth the internal architecture of each macro-block in the cycle.

3.1 Target applications

Service-based applications (SBAs) are intrinsically organized on several logical layers. The minimum number of layers in an SBA is two: in fact, there always are a top level, containing the description of the composition as a whole, in terms of the flow of activities it must follow (BPEL process) rather than the components in it and the relations among them (SCA composite, see Section 4.2), and a bottom level, containing the actual implementations for each piece of the composition. Very often, though, a single service could be itself a composition, and thus rely itself on other services: in such a case, a further layer can be identified and put in between. Repeating this reasoning recursively, it can be easily deduced that there is no fixed limit to the number of layers in an SBA. In this work, the hierarchy of an application will be always referred to as a single macro-layer named the software layer; nevertheless, its multi-level structure will be considered throughout the adaptation cycle.

In an application, though, bad performance could also be determined by the physical infrastructure on which it runs: for example, a service could be slowed down by the saturation of the memory, or by that of the CPU, or even by the network card having to manage a number of packets bigger than usual. It is thus very important to keep controlled the underlying machines as well; in particular, should they itself be provided as services in a cloud environment (IaaS, see Section 1.1), their monitoring has a double importance, as it also allows to verify the respect of possible SLAs pre-agreed with the IaaS provider. In this work, to include in the analysis all the hardware devices and their intrinsic relations with each service they host, and to possibly adapt their performances in case of failure, a further layer named the infrastructure layer has been "artificially" appended underneath the application stack.

In summary, multi-layer in this context means "software layer on top of infrastructure layer", with the software layer implicitly structured on several interdependent levels, the number of which varies depending on the specific configuration of each SBA.

3.2 Multi-layer self-adaptation cycle

3.2.1 Monitoring and correlation

At the beginning of the control loop, application performance must be tracked. The input of the first block must thus be taken directly from the application, and elaborated to produce and output the needed performance indicators, along with as much information about the context in which they were computed as possible. Such data must be expressed and formatted so to be quick to parse and understand, allowing processing blocks downstream of this to easily find the sources of requirement violations.

First of all, data must be physically extracted from running application instances. Therefore, some sensor-like devices are needed; in particular, since they must deal with software entities, rather than real, "touchable" ones, such devices must be software as well. A software sensor is a standalone program, or a piece of code inserted in the application, which is able to intercept data at specified points in its execution. An ideal software sensor would be just like a standard "physical" one: small (that is, lightweight), non-invasive (at least as less as possible) and easy to use. In the multi-layer application model considered in this work, two types of sensors (one for each layer) are needed. At the software layer, a sensor must "capture" the instants of time at which an instance is created, services are invoked, a critical section terminates and so on. At the infrastructure layer, instead, a sensor should record metrics as CPU usage, memory allocation, network traffic etc. on a periodical basis. For both the categories, different implementations can be required depending on the technologies exploited in the application.

Once "raw" data is available from sensors, it must be put in the appropriate processes and mathematical formulas in order to compute the required quality indexes (technically called Key Performance Indicators, KPIs). Since such operations have to take place at runtime, usually on a huge amount of values, and some of them (as, for instance, aggregate functions) can be quite complex, the devices employed in this task will have to feature a significant computing power, capable of supporting high loads and working at high frequencies.

Finally, calculated KPIs must be actually put together with their context-related factors, and that is the job of a correlator. A correlator simply gets values from multiple sources and produces a single element (a data structure rather than a file) containing all of them. This task is as simple in its execution as it is fundamental in the economy of the entire loop: indeed, it highlights all the dependencies across the application tiers, producing all-inclusive records which allow an earlier discovery of failure sources in the analysis phase as well as a prime identification of cross-layer "paths" in the strategy selection phase, that is, all the components that, being related to each other, could require to be modified in order to fix a specific problem. Elements at different layers that need to be correlated are not discovered autonomously by a correlator; rather, they must be indicated as its inputs by the user at design time.

A typical flow of execution requires all the three subtasks: collection, computation and correlation, strictly in this order. In general, however, the only mandatory operation is collection, which (evidently) has to be performed at the beginning of the process; the other two blocks, instead, are not subject to any kind of restriction: in some cases, it could be necessary, for example, to pass through two back-to-back computation blocks before performing correlation; in others, computation could not be necessary at all; or even, it could take to first operate a correlation block to be able to compute a KPI. The entire architecture, therefore, should be realized in a modular way, so that components implementing each step are independent of each other; in particular, it should be possible to dispose them in whatever order according to the needs, that is, their linkings should be defined in a common, loosely coupled fashion. Moreover, elements should be reusable, in order to be exploited in more than one "chain".

Figure 3.1 represents possible flows of execution in the monitoring step, determined by the paths going from the application to the output KPIs. The blocks internal to this stage (that is, those included in the "Collection" and "Computation and correlation" boxes) have been coloured according to the subtask they relate to.

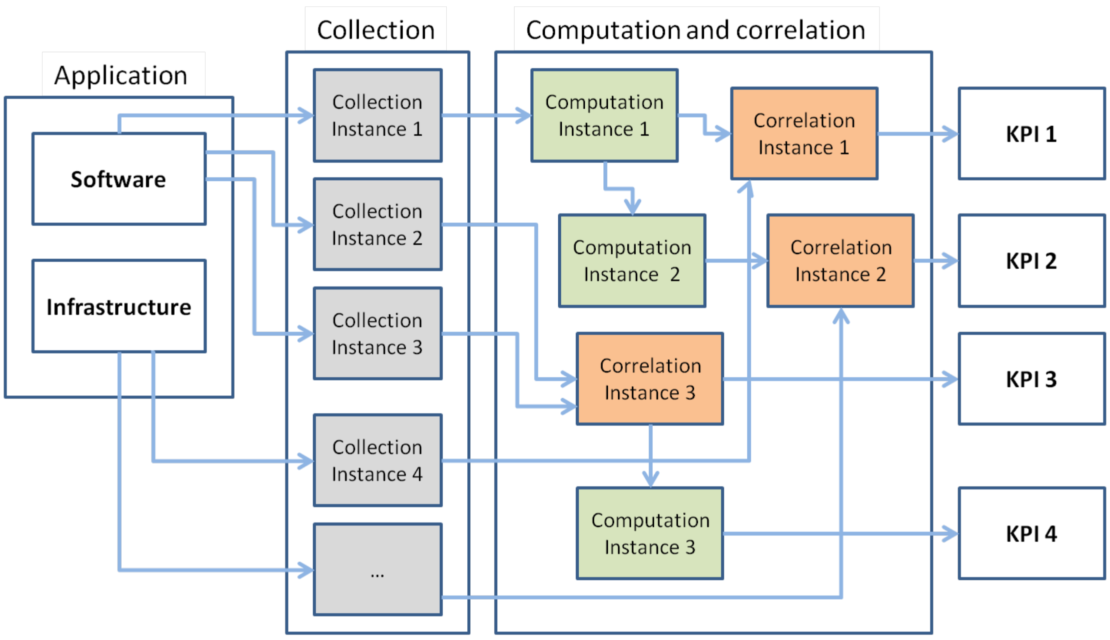


Figure 3.1: Monitoring and correlation

Let's explain the concepts just elicited with an example. Let's say we have to calculate the value of the average response time of the application in the last hour of execution and correlate it with the same metric of each of the n services composing it. To do that, the software-layer sensors must initially get the start and end instants of all the related invocations. Then, $n+1$ computation blocks are necessary, one for the whole application plus one for every single service; each of them must perform two operations: first, respective instants of the invocations occurred in the last one-hour time window must be subtracted (end - start) to obtain their response times; then, the arithmetic mean must be extracted from them. Finally, all the averages enter in a correlation device, where they are put together in a unique KPI object. This way, the services provoking a possible slowdown can be recognized in no time. Another common need could be to associate the response time of a particular service with utilization data coming from the infrastructure it is running on, to discover which parts of the machine are in charge for its bad performance.

3.2.2 Analysis of adaptation needs

Correlated KPIs become the input of the next block, which is required to examine them and check whether the application is doing fine or, otherwise, there have been anomalies and some constraints have been violated. In case failures have occurred, the block must find their causes and, after the analysis, it must define and deliver a set of adaptation actions (even empty, in case no problems are detected), which have to be applied to the system in order to restore its performance.

Incoming metric values are first “studied” and assigned a category, within a subtask called influential factor analysis. This procedure is automated, as it uses a dataset, filled with both values from past process instances and ones opportunely created (by the user), passed through machine learning in order to generate a plausible representation of the application structure. A decision tree is thus generated, in which internal nodes represent attributes, branches coming out of a node represent conditions on that attribute and leaves represent “categories”, each indicating a specific status. However, if the application structure is well known a priori and changes in it do not happen frequently, the whole machine learning step could be avoided, predefining the tree at design time and adapting it only when necessary.

At the end of influential factor analysis, it is clear how things are going and, if they are going bad, the reasons have already been determined. Next, it is necessary to define what has to be done to fix the situation. This stage, called adaptation needs analysis, gets the decision tree (specifically, the path that current data has followed) and an adaptation actions model, containing all the possible modifications that can be enforced into the application structure and the improvement they bring to the performance. Such list is thus scanned, looking for appropriate possible actions according to the tree result; if found, they are output to the next phase. In general, the more flexible the application structure is, the more solution options are available.

This approach exploits the study described in Section 2.2.1. As can be deduced, the flow of actions in the block is strictly fixed, and thus can be performed entirely by a single component. It is important, though, that such component is fully reusable, not depending in any way on the implementation of the application the cycle runs onto. This property, actually, is not that easy to achieve: elements such the historical dataset and the adaptation actions model, as well as incoming data itself, could have several forms according to the type of data they must deal with. For each of these elements, therefore, it is crucial to define a unique way of representation: for the dataset, for example, a fixed, application-independent structure should be imposed or, preferably, a standard, high-level configuration mechanism should be defined for the machine-learning block; the adaptation model, instead, should be provided with a specific language to describe actions in it.

Figure 3.2 shows the flow of execution just explained. Here, the blocks corresponding to the two tasks have been coloured, to be distinguished from those representing data structures.

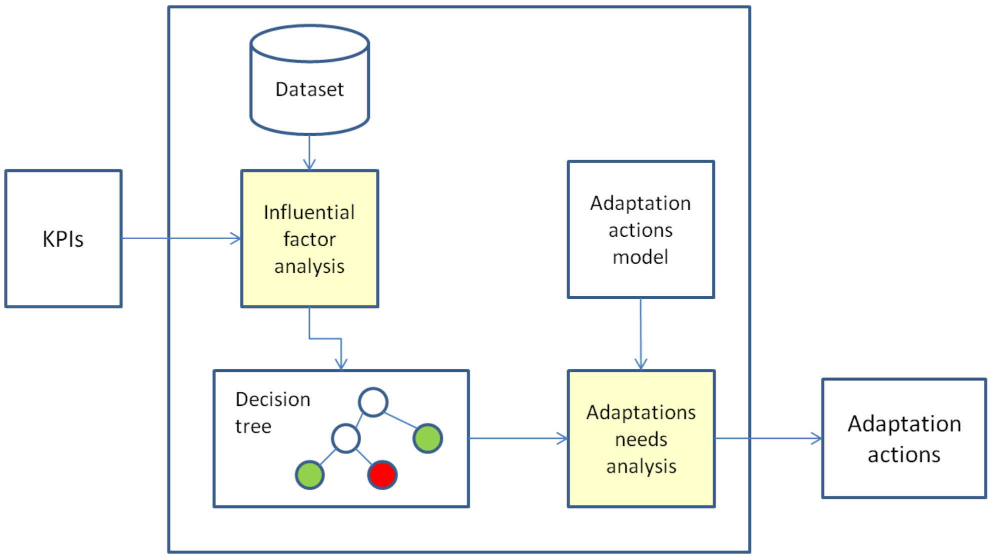


Figure 3.2: Analysis of adaptation needs

3.2.3 Identification of multi-layer adaptation strategies

The third block is responsible of evaluating the practicability of an adaptation action set on the application. First, the analysis of a set produces a list of feasible strategies; each of them is then evaluated, and the one that gets the best score is selected. The solution consists of two distinct action sets, one for each application macro-layer, which are output and will be enforced in the final stage.

Adaptation actions, along with the current application model, are input to a component called the model updater, which simply produces the model-to-be after the enactment of such actions. The new model is taken by the next element, named cross-layer rule engine; this can be seen as the core element for the purpose of the entire study, since it must verify the compatibility of a model with regard to the application constraints imposed by its cross-layer dependencies. To do that, it identifies the components involved in the update and, subsequently, the layers affected along with the respective constraints to be tested. It then calls a series of checkers, one for each constraint, and waits for their response. Meanwhile, it keeps an adaptation tree, where all the developing strategies are tracked; in such tree, each node represents a single action, and thus each path identifies a possible strategy.

A checker is in charge of testing the new model against a specific constraint that must be respected at a specific layer. When invoked, checkers make the opportune verifications, exploiting the appropriate application-dependent adaptation capabilities, and return their verdicts to the rule engine. There are three possible checker responses:

- the model is compatible and no further actions are required;
- more actions are necessary to comply with the constraint;
- the model is not compatible and no actions can be applied to make it so.

When a checker returns a positive answer, the rule engine marks the respective action-node in the tree as a green leaf, thus indicating the path leading to it as a feasible strategy; in the opposite case, the node is marked as a red leaf. In the middle case, instead, the rule engine adds the new actions to the path in the tree and send them back to the updater in order to re-adapt the model, in fact starting the whole loop over again; iterations continue until no new actions are triggered by the checkers, so that all the tree paths are identified as either a valid or an invalid strategy. At the end of this process, all the possible strategies must have been considered.

The so-generated adaptation tree is then passed to the last component in the block, that is, the adaptation strategy selector. Its role is to put the pursuable strategies together on a common scale, by evaluating a number of high-level, predefined metrics on each of them. The strategy which, according to such metrics, turns out as the best one is finally chosen and made available in output.

Figure 3.3 represents the components operating in the task and the interactions among themselves. As it can be seen, in this case communication with entities external to the loop takes place in two situations: when the updater operates on the application model, which clearly must be first defined at design time; and when checkers test their constraints by exploiting pluggable, application-specific capabilities. It is necessary, therefore, to devise a universal language for the application model, and also to project a unique interface for the checker "sockets", to which the appropriate functionalities will be plugged.

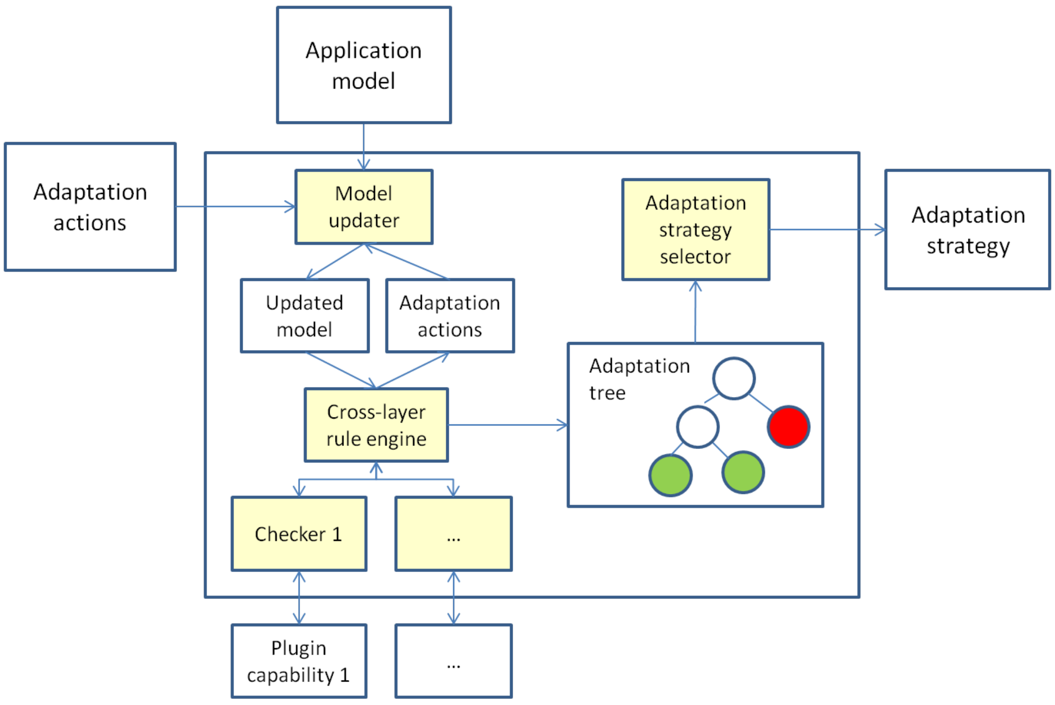


Figure 3.3: Identification and selection of multi-layer strategies

3.2.4 Adaptation enactment

In the fourth and final step, adaptation is physically operated on the application. Specific devices at both software and infrastructure layers are responsible of translating the input strategy into effective commands on the target components. Though this block communicates with the application with reversed roles as compared to the monitoring one (sender the former, receiver the latter), both have the main challenge in common: dealing with multi-language compositions, that is, applications made by services possibly implemented in different ways. Indeed, while the other blocks can work on data that is abstracted from the low-level implementation details, the first and fourth ones must necessarily use technology-specific instructions. One device for each possible implementation paradigm type should thus (ideally) be provided to achieve a more extensive adaptation.

At software layer, enacting a strategy basically means rewriting the process description, that is, editing component definitions, rather than the linkages between them, to alter the flow of execution. This can be done, for example, by switching the order of execution of two components, or by adding/removing one ore more components, or even by replacing the service used by a component with another one which would still allow to respect the constraints. Two modalities of enactment should be available: at runtime (that is, affecting currently running instances), if adaptation has to be forced immediately and/or for a limited period of time (that is, only the process instances operating in such period would be affected; after it, the standard configuration will be applied again); or "structurally", if it is necessary to modify the composition in a permanent way. Obviously, what modality is best depends on the specific case and is not to be decided in this stage, but rather falls within the plans of the needs analyser.

At infrastructure layer, two categories of actions should be possible. The first category regards relocation of services, that is, moving services between machines (virtual or not), even new ones, to balance the load-per-node factor, or simply to transfer part of the execution to a faster processor. The second category operates instead at a lower level, probably the lowest one possible, as it provides instructions to improve the features of machines themselves (more memory, a more powerful CPU and so forth). In this latter case, adaptation techniques are totally different according to the context: in the virtual one, indeed, it is possible to tune up a machine by just, say, resetting a parameter in a configuration file; at the opposite, dealing with physical servers is much more laborious, as everything must be substituted manually.

Naturally, it should be possible to combine actions from both the categories in the same strategy. Most IaaS providers include automatic scaling, load balancing and many other capabilities among their options, allowing developers not to worry about the underlying system; however, since the pricing of cloud-based services usually follows the pay-as-you-go policy (see

Section 1.1), it is always better to keep an eye on the infrastructure as well, in order to reduce resource waste and optimize costs.

Chapter 4

Implementation

The practical contribution given in this work has been to the first phase of the cycle, that is, the “Monitoring and correlation” one, in which data is pulled out from the multi-layer application and processed into usable information. Unlike the next two ones, the monitoring phase does not contain any intelligence; nonetheless, it is equally as important, as it must produce a set of complete, understandable and uniformly structured indicators out of raw, meaningless numbers coming from technologically different sources.

The chapter is divided in three sections. Section 4.1 provides a brief overview, both graphical and textual, of the implemented architecture. Section 4.2 describes instead the type of applications that can be monitored by the realized system. Finally, Section 4.3 provides a detailed illustration of the architecture components: what their meaning is, how they have been implemented and how they operate.

4.1 The Big Picture

This work concentrates on the first phase of the loop, that is, monitoring and correlation. Figure 4.1 shows the architecture of the system projected to exploit such phase.

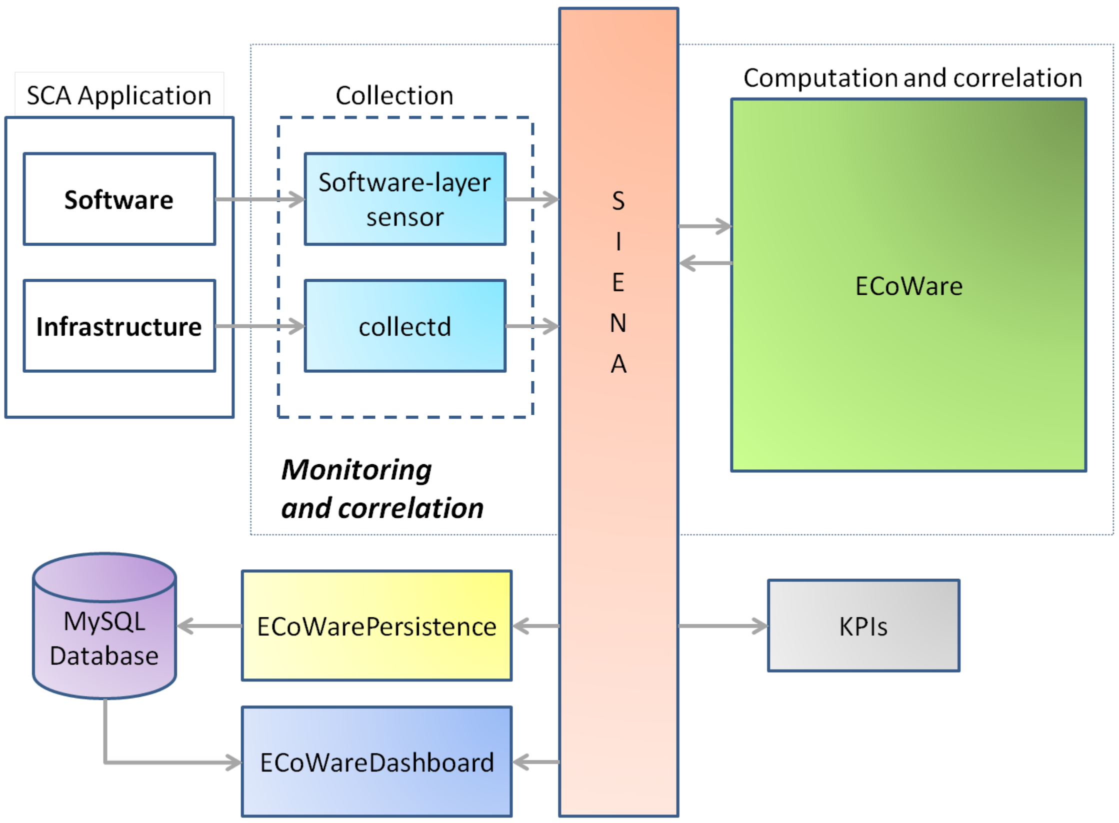


Figure 4.1: Architecture of the implemented features

Besides the (SCA, see next section) application (which provides the input of the phase) and the KPIs (representing its output), seven elements can be distinguished in the picture:

- **Software-layer sensor**: intercepts performance data at the software layer;
- **collectd**: retrieves information from the infrastructure layer;
- **ECoWare**: operates the computing and correlation subtasks;
- **ECoWarePersistence** and the MySQL **Database**: store KPI values;
- **ECoWareDashboard**: displays the calculated metrics on charts;

- **SIENA**: enables communication among all the elements.

The flow of execution in the system coincides with the one depicted in the theory (Section 3.2.1). At first, data is pulled out of the application by the software-layer sensor and collectd; then, data is picked by ECoWare, which transforms it into complete, meaningful information. In addition, here, ECoWarePersistence stores produced KPIs in a database and ECoWareDashboard shows them through appropriate graphs.

Three things can be noticed in the picture. First, the box containing the two sensor blocks (that is, the Collection box) is dashed. That is to say that they can be put together conceptually, with regard to the subtask they operate (and so they correspond to the collection instances in Figure 3.1), but they are distinct and independent in practice: there is not a "macro-component" supervising the subtask and handling their execution. Secondly, the ECoWarePersistence and ECoWareDashboard components are not included in the bigger box, the one named "Monitoring and correlation": in fact, they are not necessary to carry out the task, but rather serve to provide a graphical tracking of the performance, facilitating the analyst on keeping an eye on the application and also allowing us to validate the study. Finally, the SIENA-labelled rectangle crosses the borders of the box. This component, indeed, distributes the messages exchanged between the blocks, and so it does with the newly-produced KPI metrics: therefore, ECoWarePersistence and ECoWareDashboard, but primarily the elements that will implement the next phase, must connect to it in order to access data.

4.2 SCA

Before starting to implement the self-adaptation cycle, it has been necessary to decide what applications to self-adapt. Indeed, while the high-level description of the loop, provided in Section 1.2, is a universally valid model, its implementation depends strongly on that of the controlled application, because the two entities must interact continually with each other. Looking at the diagrams of each cycle phase, it can be seen that there are three specific points in which communication with the application, direct or indirect, must take place: at the very beginning, when performance data must be gathered from the application; (indirectly) in the strategy identification phase, when some checkers may use external, application-dependent capabilities to find further corrective actions; and at the very end, when the adaptation strategy must be actually operated. Components acting at these points, hence, must be necessarily realized according to the technologies and communication protocols used by the application.

The purpose of service-based applications is to put together separate and independent pieces of code, regardless of their language and of the platform they run onto. To make interaction among services possible and simplify the realization of such applications, an architecture, named SOA[1] (Service-Oriented Architecture), has been designed. SOA defines the principles and methodologies to implement a middleware allowing simple integration and orchestration of software of different nature (Java/C++ programs, HTML pages, even legacy systems). Principles consist of the basic properties a service must have to be included, among which loose coupling, abstraction, reusability, statelessness, discoverability and composability. In the SOA philosophy, the user should only provide a high-level description of the composition, indicating its elements and the protocols for their interactions by means of appropriate metadata, and the middleware would take care of the rest, carrying out the actual communication logic.

In this work, analysed applications must be based on the Service Component Architecture[2] (SCA) model. This technology has been created by a major-vendor partnership, including among others IBM, Oracle, Sun Microsystems and Siemens, with the aim to provide a solid, unified platform as a basis for would-be SBA-handling software. For this purpose, therefore, SCA implements the SOA notions, defining a set of specific directives to represent and execute a service composition. In the SCA model, an application consists of elements belonging to three categories:

- **composite**: the project container and the unit of deployment for SCA (that is, the unit needed to make an SCA-based application available over the Internet);
- **component**: an independent block of execution;
- **service**: the basic unit of the process, representing a functionality (usually consisting of one or more APIs) provided by the component it is included into.

Figure 4.2 provides a diagram of an SCA project. As can be seen, a composite contains one or more components (two in this case, named *AccountServiceComponent* and *AccountDataServiceComponent* respectively), each of them exposing one or more services as their entry points. In addition to its own services, a component must indicate the ones it exploits

[1] http://en.wikipedia.org/wiki/Service-oriented_architecture
[2] http://oasis-opencsa.org/sca

from other components, and that can be done by declaring one or more references; wires linking references with entry points, therefore, symbolize inter-component communication. In this case, each component exposes one service, and *AccountServiceComponent* also includes a reference to the service of *AccountDataServiceComponent*. Both services and references can be "elevated" to the composite scope, in order to, respectively, make them available to the outside (considering them the application's "real deal", that is, its actual functionalities) and access a service of an external application; this operation is called "promotion". Finally, components can require some global properties to be set prior to their execution, and those can be promoted as well. In the example, *AccountServiceComponent* has its service promoted, and the name of the promoted service is *AccountService*; it also has a property, named currency.

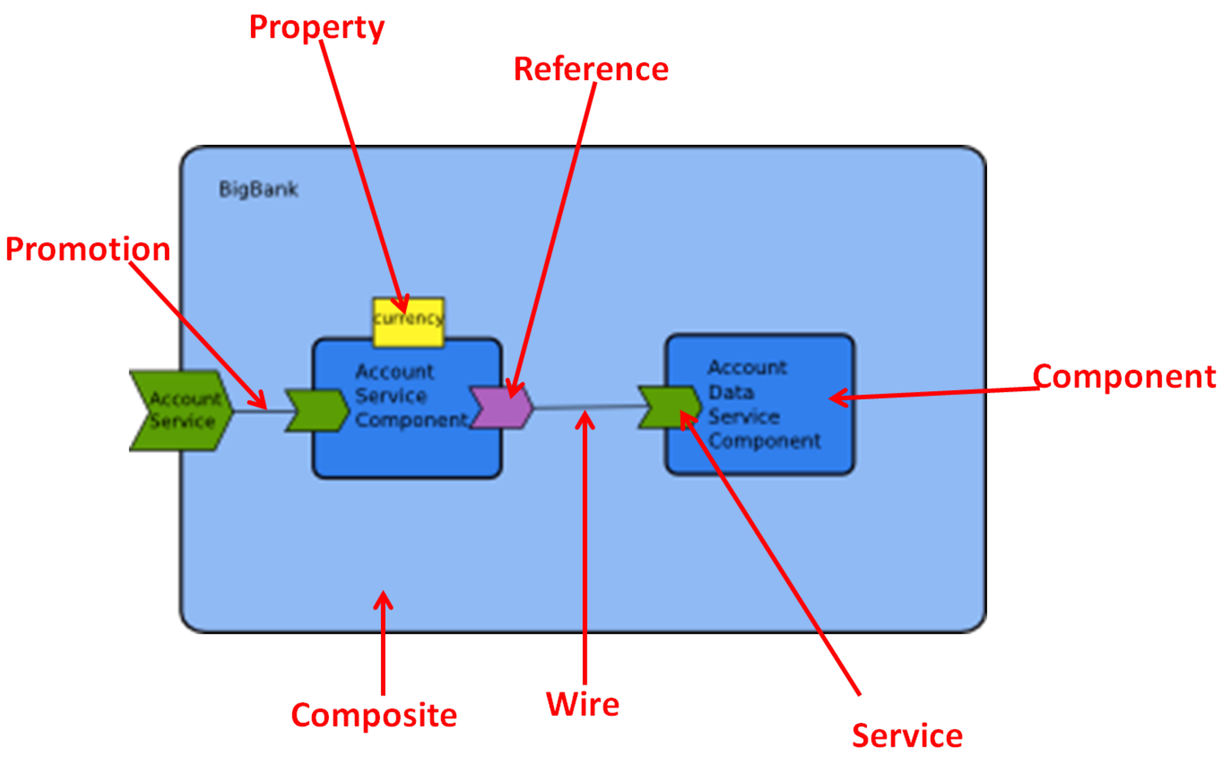


Figure 4.2: Sample diagram of an SCA application

The entire application structure must be described, using appropriate XML features, and saved in a .composite file. Such a file must be structured in the same way as the application diagram would, exploiting the tags related to each artefact. Precisely, in the *composite* section must be listed the descriptions of all included components as well as all the promoted services/references/properties; in the *component* section, in turn, it must be indicated the implementation technology (Java, C++, but also BPEL or SCA composite itself, thus allowing multi-layering) and the location of the actual resource; the *service* section must contain the information about the service interface, and in both the *service* and *reference* sections it must be specified the underlying communication protocol.

Listing 4.1 shows the configuration file for the previous example. XML tags corresponding to the various elements have been here highlighted. It can be noted that in this occasion, in both the component sections, services have not been declared explicitly; that is not necessary, since the particular implementations of the components, which are realized by means of a Java class, allows to automatically consider all the (public) methods included in it as service APIs. As can also be seen, the wire linking a reference with the related service is defined through its *target* attribute, and a promotion is identified by the *promote* one.

```xml
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://account"
    name="Account">
    <service name="AccountService"
        promote="AccountServiceComponent"/>
    <component name="AccountServiceComponent">
        <implementation.java
            class="bigbank.account.AccountServiceImpl"/>
        <reference name="accountDataService"
            target="AccountDataServiceComponent"/>
        <property name="currency">USD</property>
    </component>
    <component name="AccountDataServiceComponent">
        <implementation.java
            class="bigbank.accountdata.AccountDataServiceImpl"/>
    </component>
</composite>
```

Listing 4.1: Sample .composite file of an SCA application

So far, four major independent SCA-based systems have been realized: Apache's Tuscany[3], OW2's FraSCAti[4], Fabric3[5] and ServiceConduit[6]. They all implement the basic SCA concepts, differing in the amplitude of the included functionalities. Being SCA a relatively recent technology, each of

[3]http://tuscany.apache.org/

[4]http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti

[5]http://www.fabric3.org/

[6]http://www.service-conduit.org/

them is in continuous development, every updated version including new component implementation possibilities, binding protocols and management features. Further tools will sure be conceived in the future. In this work, it has been analysed compositions put together and executed on top of either FraSCAti (version 1.4) or Tuscany (version 1.6.2), both written in Java.

4.3 Components

4.3.1 Data collection

As can be seen in figure 4.1, two distinct elements (one for each level) have been used in this implementation to gather data from the application. In particular, the software-layer sensor has been realized itself in two versions, to comply with each of the SCA middlewares employed. At the infrastructure layer, instead, only one tool has been exploited.

Neither FraSCAti nor Tuscany provide specific monitoring APIs among their functionalities. With FraSCAti, though, sensors can be easily forced by means of so-called intents. An intent is an independent SCA composite, which is associated to one or more entities (components, services or references) of the application and activated every time each of such entities is about to be invoked; at that moment, precisely, the code of the intent is injected amid that of the application, so that to be executed just before the invocation is sent and just after its result is returned. To include an intent in the application, the .composite file must be modified, adding the following attribute among those of the interested elements:

requires="intent-composite-name"

where *intent-composite-name* is the name given to the SCA composite representing the intent.

To realize an intent, FraSCAti offers the *IntentHandler* interface; this, in particular, contains only one method, named *invoke()*, which is exactly the one called by the middleware when an invocation is triggered. Having the intent to be an SCA composite, the usual, less intrusive and "cleanest" way of packaging it would be a separate Java project; however, it is also possible to merely include the respective classes, along with the .composite file, together with those of the actual application. Practically speaking, there is no difference between the two approaches; the first one, though, is much more preferable, since it is more adherent to the software-sensor guidelines expressed in 3.2.1.

Listing 4.2 shows the skeleton implementation of an intent class. The point at which the invocation is done is the instruction *ijp.proceed()*, and thus, as indicated, the monitoring code must be written immediately above and immediately below such instruction. For the purposes of this work, it has been necessary to create only one intent, which, both before and after the invocation, first identifies the invoked operation in terms of composite, component, service and method by using appropriate FraSCAti APIs, and then forwards the information to SIENA along with the current timestamp; the only difference between the two moments is that information is labelled as “StartTime” in the first case, “EndTime” in the second (the meaning of all these operations will be clearer in Section 4.3.2).

```
public class FooIntentHandler
  implements IntentHandler {

  // ----------------------------------------------------------------
  // Implementation of the IntentHandler interface
  // ----------------------------------------------------------------

  public Object invoke(IntentJoinPoint ijp) throws Throwable {
    Object ret;
    //
    // PUT HERE CODE TO RUN BEFORE THE JOINPOINT PROCESSING
    //
    ret = ijp.proceed();
    //
    // PUT HERE CODE TO RUN AFTER THE JOINPOINT PROCESSING
    //
    return ret;
  }
}
```

Listing 4.2: Sample intent class

Tuscany, on the other hand, does not provide functionalities to integrate external sensors (or, at least, it does not indicate them clearly in its documentation). The only way, therefore, would be to physically insert opportune lines either into the application code or into that of the middleware itself. In order not to modify the application, the latter option has been chosen and the source of Tuscany has been inspected, to comprehend how it handles invocations and subsequently find the right locations in which monitoring instructions should have been placed. In so doing, it has been found that Tuscany includes a separate Java project for each component implementation and service binding at runtime, and all such projects contain a class implementing the *Invoker* interface and overriding its *invoke()* method. Invocations take place in such method: hence, once the respective instruction is localized, the monitoring procedure can be written “around” it as previously seen with FraSCAti. To retrieve the information identifying an invocation, it has been necessary to look for opportune APIs in this case as well.

From the analysis just delineated, it can be easily deduced that, when it comes to monitoring, FraSCAti is undoubtedly preferable over Tuscany: to include sensors in the latter, indeed, it has been necessary to alter its own implementation. In addition, since Tuscany (and FraSCAti as well) is realized as an SCA-based application itself, adding those instructions directly in its source code results in intercepting all the invocations passing from there, even those referring to internal Tuscany components: this fact, besides capturing a lot of uninteresting service calls, can potentially lead to a significant slowdown in the application. Finally, it can also be noted that, despite offering a much better approach, FraSCAti requires to modify the application, precisely its .composite file, while Tuscany does not; this aspect, though, is surely negligible, because such a modification is operated at the highest level possible, and thus it does not go to alter neither the flow of execution nor the implementation of components.

At infrastructure layer, the focus is not on the application, but rather on the machines it runs onto: a (possibly) lightweight program is therefore needed on each of them to periodically acquire information about memory consumption, network traffic and/or whatever parameters it is important to monitor according to the specific situation. Last, and perhaps most importantly for the purpose of this work, such a program must also make available the gathered data for post-processing. A bunch of free tools, usually daemons (that is, small programs running in background) can be found that are able to collect data and (almost all of them) display performance graphically; furthermore, major IaaS providers come with their own monitoring capabilities. Only few options, however, allow the data they collect to be accessed in a simple and immediate way. collectd is one of them, and that is why it has been chosen to operate in this context.

Figure 4.3 shows a detail about the infrastructure-layer data collection flow, taken from the general schema of Figure 4.1. A new component, called InfrastructureSensor, is here highlighted; this is a Java project, developed outside collectd and then inserted into it in order to gather the metrics intercepted by the tool, translate them into Java data structures and forward them to SIENA. A close description of collectd, explaining its internal flow and how it has been possible to include InfrastructureSensor in it, is provided in Section A.1.

Data collected from the application is then forwarded to be processed

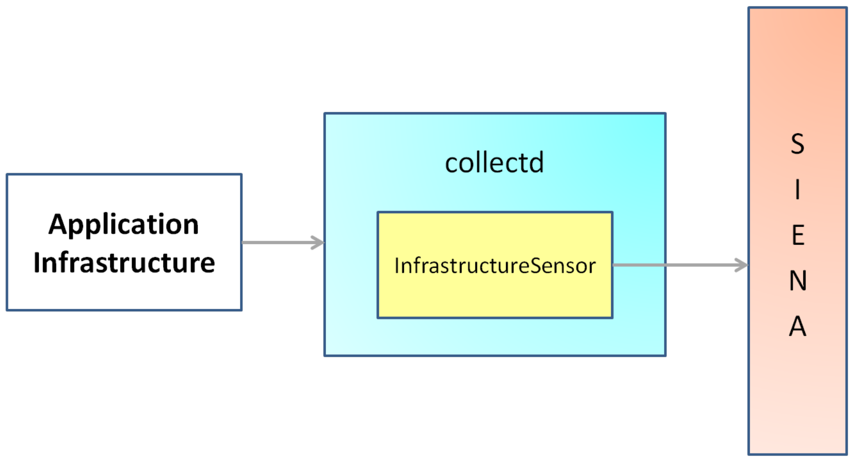


Figure 4.3: Data collection at infrastructure layer

and produce KPIs. As mentioned above, communication among operating elements is carried out by SIENA, which is a Java tool implementing the publish-and-subscribe technology. Publish-and-subscribe can be represented by a channel, or bus, and by several entities connected to it. Such entities, as the protocol name itself suggests, are divided in two groups: publishers, which generate items and place them on the bus; and subscribers, which instead wait for the specific items they are interested in and get a copy of them as soon as these are available on the bus. The main advantage of publish-and-subscribe is that entities act in complete independence (and unawareness) of each other: in fact, the only thing they need to know is the structure of the data they will exchange (that is, the number and type of attributes in it). That is just how relations among components in the monitoring task should be, and that is also why it has been decided to exploit this paradigm to make them interact.

Figure 4.4 details how SIENA enables communication among any two generic system elements. The component acting as the publisher, at the end of its processing, passes the data to a block named *SienaOutputAdapter* (consisting of a Java class), which translates the data in the SIENA format and then performs the actual publication on the bus. At the subscriber end, the *SienaInputAdapter* block operates in the opposite direction: after retrieving the item from the bus, it de-parses it and finally hands it over to its related processing component. Note that the two communication actors are completely decoupled from each other: the publisher does not care about who will receive its data, and so does the subscriber about senders of the items it gets. Section A.2 provides a close description of SIENA.

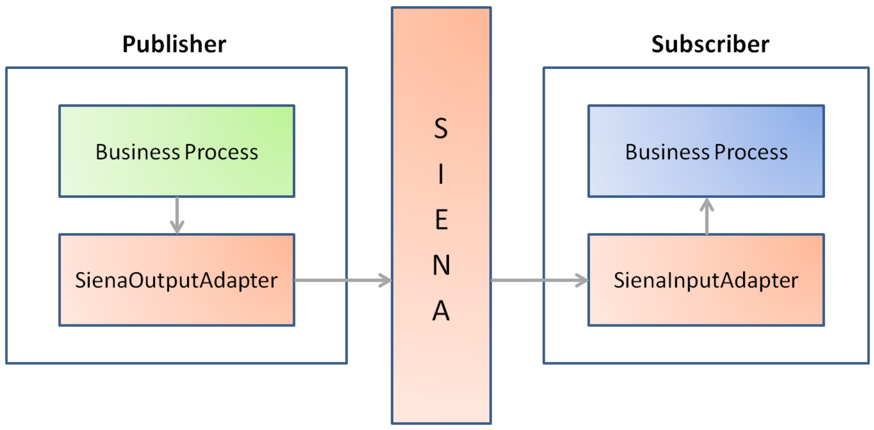


Figure 4.4: SIENA integration in the system

4.3.2 KPI computing and correlation

The computing and correlation subtasks are both performed by ECoWare (the word is a merger of Event, Correlation and middleWare). The decision to include both the operations in a unique (Java) component comes from the fact that, despite being substantially different with regard to their functionalities, they can be carried out through the same process. The bottom line consists of considering all the data flowing into the system as events, each of them uniquely identified by: its type; an alphanumeric code indicating the entity that has produced that event; and a progressive number to distinguish events of the same type generated by the same entity. From this perspective, a calculator is an entity which takes an event, applies some formulas to it and outputs the result in a new event; a correlator, instead, is an entity which gathers two or more different events, encloses them together in a single structure and finally outputs them in a new event instance. Therefore, both can be seen as devices which wait for events, treat them and release new events, the only difference being the specific treatment they operate.

Figure 4.5 illustrates the internal composition, as well as the flow of execution, of a generic instance running in ECoWare. First, events are received from SIENA and translated by the *SienaInputAdapter* block; parsed events are next sent to ESPER, a very powerful event processor, which performs the requested operations on them and outputs the so-generated new events; such output is delivered to the *UpdateListener*, which acts as the ESPER equivalent of a SIENA subscriber and does nothing other than passing all the received data to its embedded *SienaOutputAdapter* block; the latter encapsulates the data into fresh SIENA events which finally publishes. It can be immediately seen that instances internal to ECoWare are loosely coupled to each other, as each single one of them operates in complete autonomy and communicates with outside by means of SIENA. Thanks to that, the two should-have characteristics depicted in the high-level description for such block (Section 3.2.1) have been realized: instances in ECoWare, indeed, can be combined in any (compatible) order and, at the same time, they can be part of multiple processing chains. ESPER is here represented as a black box: Section A.3 describes how it works.

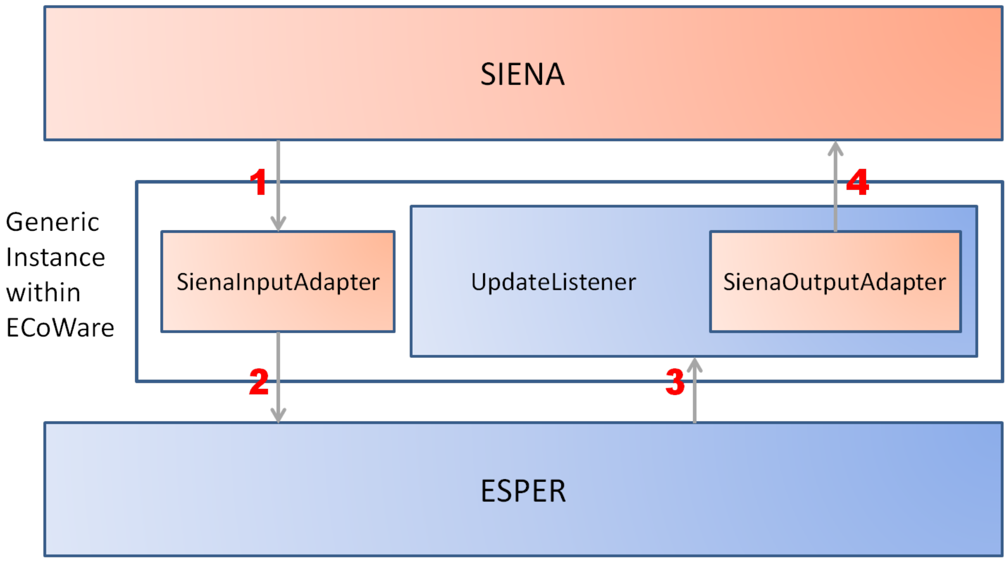


Figure 4.5: Processing flow in an ECoWare component

Three categories are possible for instances in ECoWare: there are calculators and correlators, which role have been already described, and there are also filters. A filter is an entity taking an event and letting it "pass" (which, in this case, means outputting a new event identical to the one received in input) only if its attributes are into a specific range; with regard to this work, this can be useful to detect constraint violations: all the events indicating a good behaviour do not require to be analysed an thus will not pass through the filter, while the other, bad ones will be forwarded. Filtering is in fact one of the operations that can be included among computation features and, therefore, filter blocks can be seen as complementary to calculator ones.

Figure 4.5 shows that all the ECoWare blocks execute a common set of operations, although they differ in the events they process and in how they process them. This led us to define them through a hierarchy. This promotes modularity and allows designers to easily add, update, or remove blocks without a significant impact on the EcoWare infrastructure. The corresponding UML class diagram is illustrated in Figure 4.6. All instances extend from a top *KPIManager* class, which is used to define their common characteristics. The *KPIManager* provides: one or more identifiers for subscribing to event sources; one publication identifier for outputting events; and one abstract *launch()* method that needs to be extended by each subclass. The second level in the hierarchy presents three ECoWare instance categories: computation (*StandardKPICalculator* and *CustomKPICalculator*), filtering (*StandardKPIFilter* and *CustomKPIFilter*) and correlation (*Aggregator*). *StandardKPICalculator* is an abstract class that adds a time interval for incoming events that will be considered in the formula, and a rate at which the computation will be performed. The time interval is defined in terms of a time unit and an amount, indicated by *intervalUnit* and *intervalValue* respectively. The output rate is similarly defined in terms of an *outputUnit* and an *outputValue*. The *CustomKPICalculator* defines a completely open canvas for designers that want to create new KPI calculation blocks. *StandardKPIFilter* is an abstract class that has been extended by *HPFilter*, a high-pass filtering block, and *LPFilter*, a low-pass filtering block. Finally, correlation components are all similar in the sense that they all perform the same operation, and only differ in the events they receive as input; this is why we implemented a single class, called *Aggregator*.

Table 4.1 provides instead an explanation of the calculator, filter and correlator implementations that can be seen in the figure, in terms of events processed and operation performed. The letters X Y and Z in the event-related columns refer to generic events. What the table shows is, first of all, that calculator instances process and produce specific-type events only, while filters and aggregators, on the contrary, can treat every kind of event; secondly, that a filter (as expected) does not modify the event content, but rather compares it with a predefined pattern and then decides whether to discard it or send it through; finally, that a single aggregator can operate on two events, the first being identified as primary and the other as secondary, and outputs new data only when a new primary event instance arrives.

With the exception of correlators, which can be attached to (at most) two event sources, each instance running in ECoWare processes events coming from a single source; therefore, if it is necessary to, say, compute the average response time of two application components, the same number of *RTCalculator* objects must be instanced, that is, one for each component. Figure 4.7 shows an example of a complete KPI production chain. In this case, the software-layer sensor captures the start and end instants of the invocations for service A, generates the respective *StartTime* and *EndTime* events and publishes them with ID “A”. Such events are picked by an *RTCalculator* in-

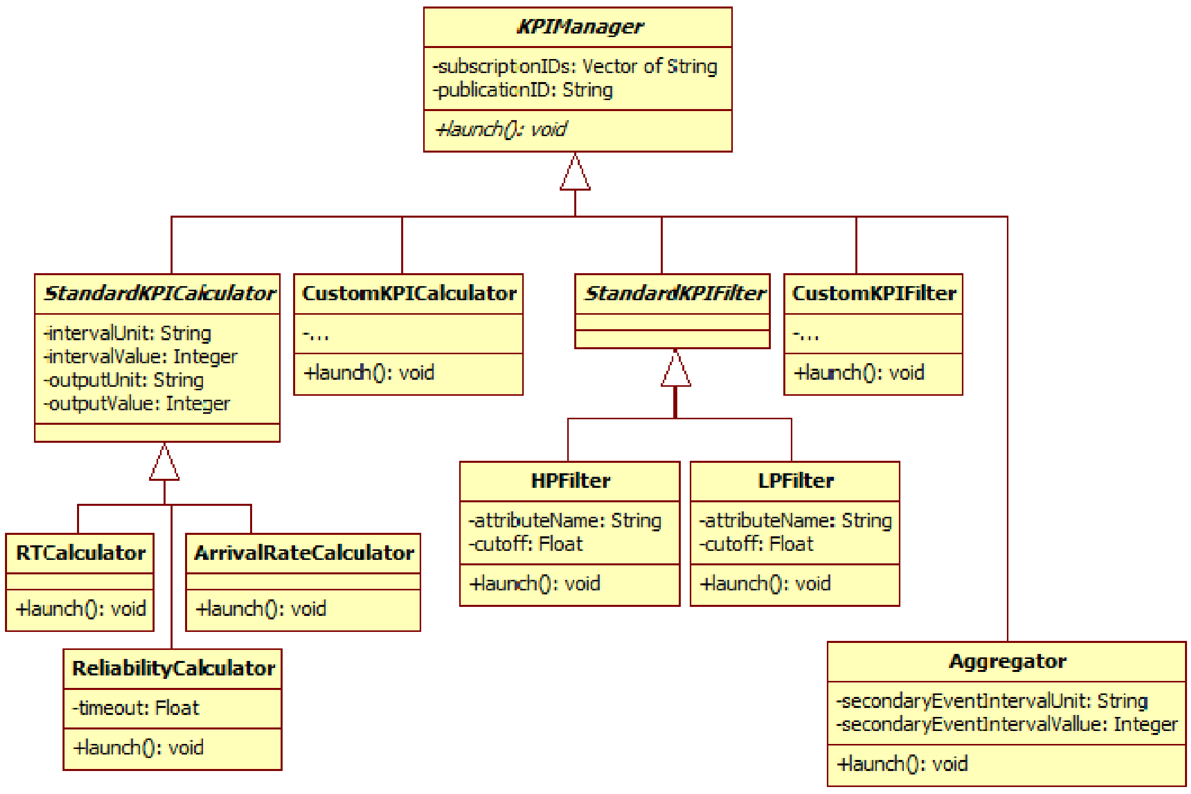


Figure 4.6: ECoWare components hierarchy

Name	**Input events**	**Output event**	**Description**
RTCalculator	StartTime EndTime	RT	$AVG(EndTime - StartTime)$
ReliabilityCalculator	StartTime EndTime	Reliability	$\frac{COUNT(EndTime - StartTime < timeout)}{COUNT(StartTime)}$
ArrivalRateCalculator	StartTime	ArrivalRate	$COUNT(StartTime)$
HPFilter	X	X	Retransmits only the events in which the value of the target attribute is greater than the cutoff.
LPFilter	X	X	Retransmits only the events in which the value of the target attribute is less than the cutoff.
Aggregator	X, Y	Aggregator	Every time a new event X (primary) arrives, outputs it along with each of all the Y events occurred in the specified period.

Table 4.1: Functionalities of sample ECoWare instances

stance, which in turn produces and forwards an RT event marked with "Y", its own publication ID. The RT event is then passed through a high-pass filter, which checks it against its cutoff (100 ms in this case); if the value is bigger, the event is republished with the filter ID ("Z"). At last, an aggregator correlates the filter-made event with those, labelled by "X", coming from collectd and the application infrastructure layer, and outputs several *Aggregator* tuples with ID "A1". The figure also highlights how easy it is, in this context, to put blocks in cascade with one another: in fact, the only thing to do is make publication and subscription IDs of every two adjacent blocks (the one upstream and the one downstream respectively) match.

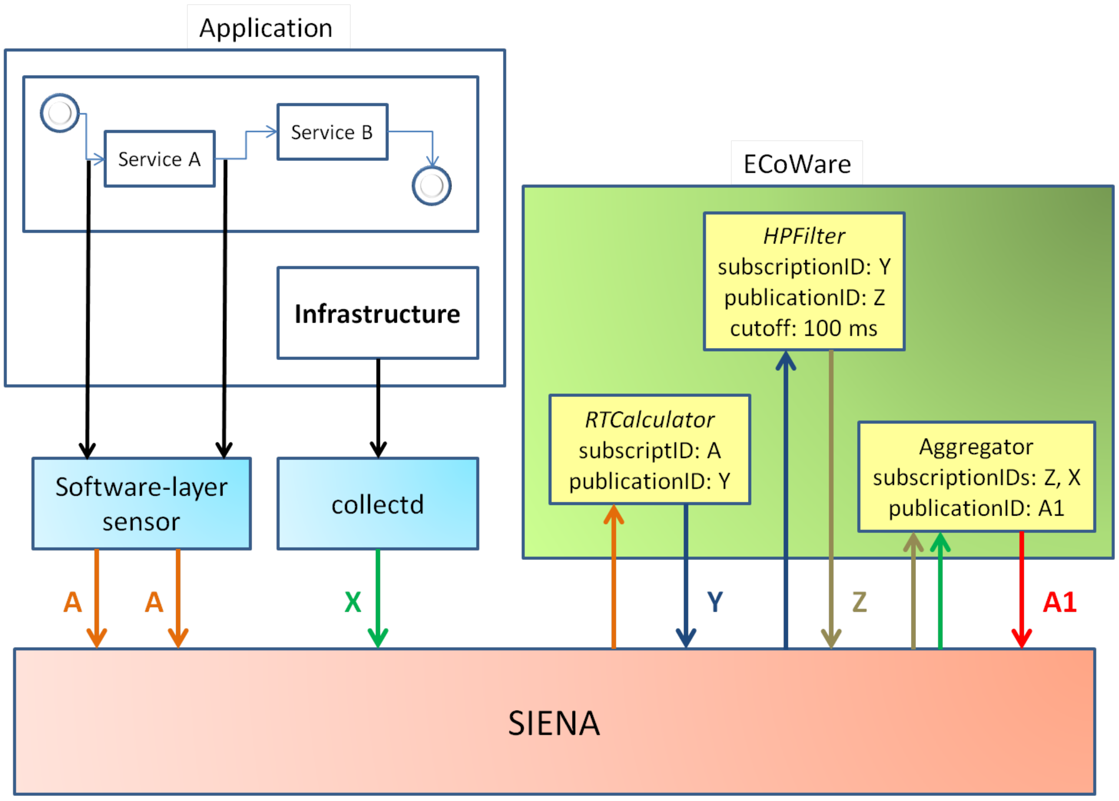


Figure 4.7: Sample KPI production chain

4.3.3 Output visualization

As said at the beginning of the chapter, visualizing the application performance into graphs is not among the purposes of the cycle. In this particular context, though, this extra-feature can be useful, as it permits to validate the entire monitoring structure, allowing to easily verify experimental results against theoretical expectations. In addition, charts are still a helpful instrument for eventual business analysts, letting them understand the situation quickly and perhaps fix small problems in less time than the self-adaptation cycle itself.

The component in which KPI trend visualization has been implemented is called ECoWareDashboard. Two displaying modalities are provided: live, to observe the evolution of metrics in real time; and off-line, to bring all the correlated data together in a single view and thus obtain a more complete comprehension. To realize all the graphical tools, a Java library named JFreeChart[7] has been exploited. This is a very good plug-in, including a great amount of APIs to draw and manipulate several types of charts; among those, the one fitting the purpose of this work best has been the time series chart, allowing to display sets of (time, value) pairs on a Cartesian plane.

Live graphs are implemented by the *KPIVisualizer* class. Instances of such class operate in an extremely simple and straightforward way: first, they subscribe to SIENA for the event of interest; as new data arrives, they extract the timestamp and the value of the target attribute, create the respective (time, value) pair and add it to the chart. Since the only aim of a live chart is to show data in quasi-real time, that is, as soon as it is computed, there is no need for storage or any other additional features. Figure 4.8 illustrates the layout of a live chart. Some parameters can be configured in it, as the bounds for its Y-axis (if not specified, the chart will automatically resize according to the dataset) and, more importantly, the dimension of its dataset, that is, how many values to keep displayed. If the latter is fixed, the dataset is treated as a finite, first-in-first-out (FIFO) queue: when a new event arrives and the dataset is already full, the temporally oldest value is removed; this way, the chart becomes a sliding time-window, representing the concept of "live data" even better.

An off-line (also, historical) graph, instead, is much more functional, as it allows to visualize the past trend of a particular metric in conjunction with those of all the factors implied in its calculation; this kind of dashboard provides an all-inclusive KPI displaying system, allowing an immediate recognition of the culprits in case of failures. To realize such a tool, however, extra-computation must be included, in order to persist the data of interest and made them retrievable for future examination.

As can be guessed from Figure 4.1, the persistence task is performed in the ECoWarePersistence block. Its flow of execution consists merely of connecting to SIENA to get the opportune events and, subsequently, storing

[7]http://www.jfree.org/jfreechart/

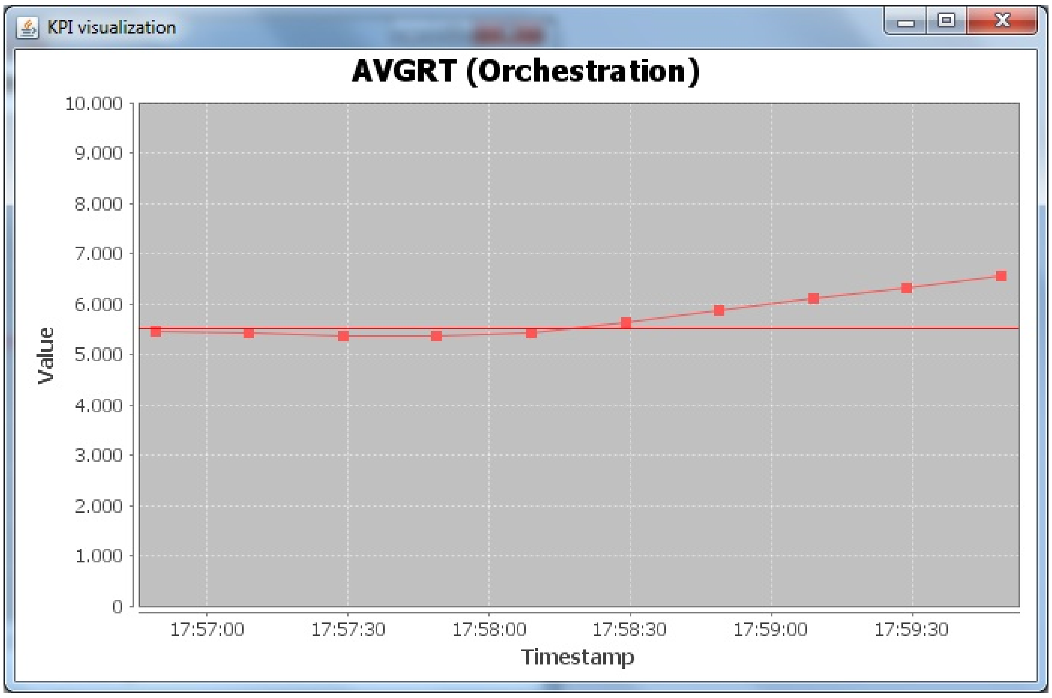


Figure 4.8: Live chart

them into a MySQL database. However, there is a tricky question. The dashboard must show complete graphs for the primary KPI and all its correlated metrics; often, though, ECoWare correlators take their primary input from filters, which provide only a part of all the computed values (that is, the ones matching their patterns). To resolve this point, two separate classes has been defined: *KPIPersistenceManager*, which takes and stores all the original, unfiltered values of the primary event; and *AggregatorPersistenceManager*, which extracts secondary metrics from correlated events and links them to the respective primary item. Likewise, the database, which E-R diagram is depicted in figure 4.9, includes two tables: *KPIItem*, storing primary events, and *SecondaryEventItem*, which describes KPI-correlated metrics (indicating the name of the related attribute to distinguish it among all the ones possibly contained in the same *Aggregator* event). The technology exploited for operating the persistence is JPA[8] (Java Persistence API), provided by the EclipseLink[9] plug-in.

Persistent data is retrieved and displayed on a complete panel by ECoWareDashboard's *AggregatorVisualizer*. Figure 4.10 illustrates the features of

[8]http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html
[9]http://www.eclipse.org/eclipselink/

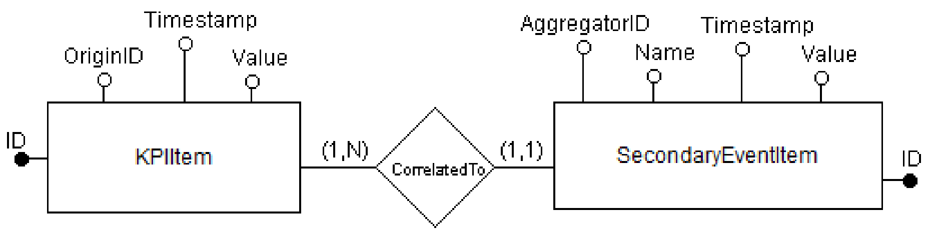


Figure 4.9: Entity-Relationship diagram of the database

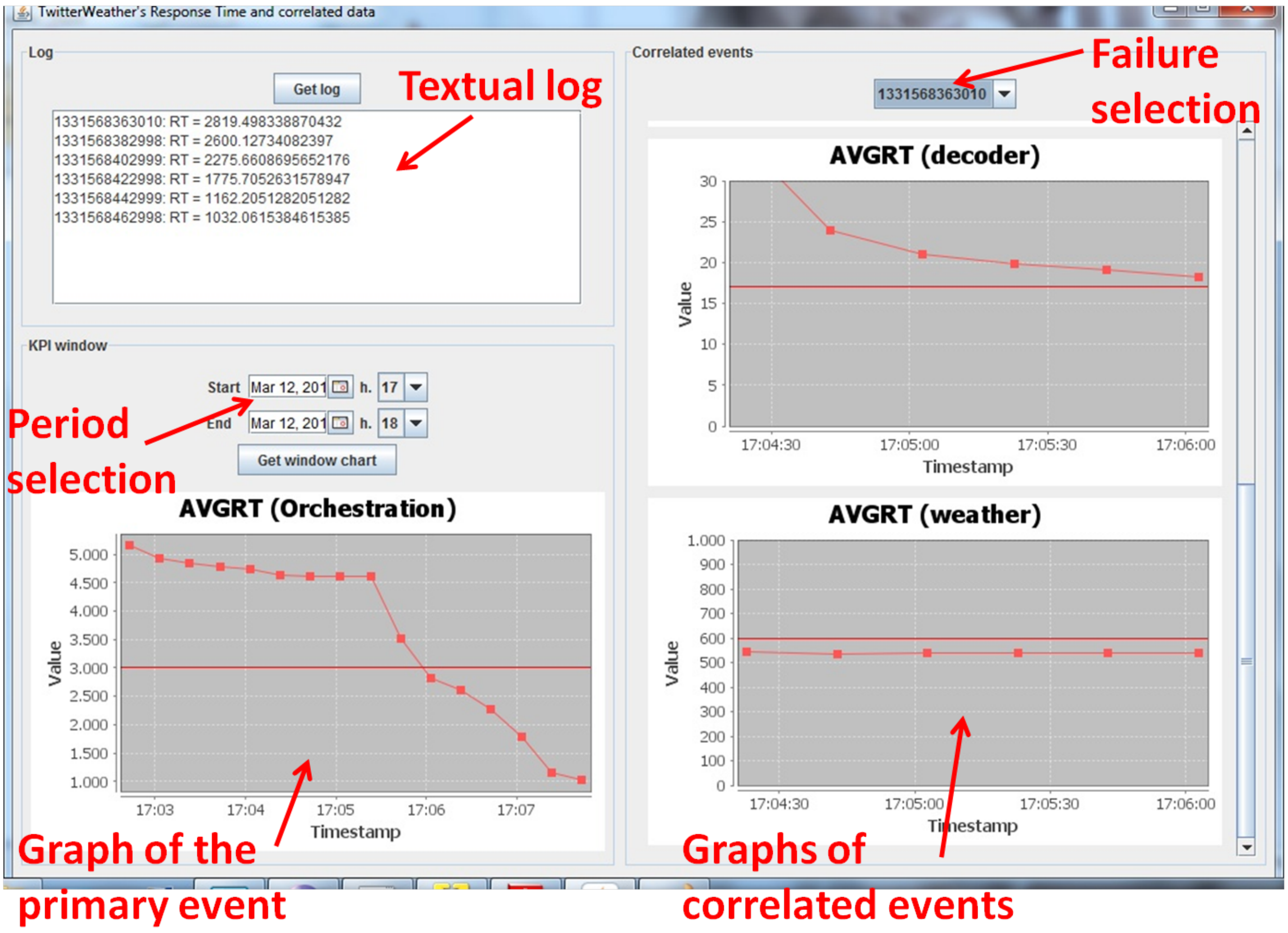
such a panel and how the user can interact with it. Usually, the user first gets a textual list of all the occurred failures; then, he selects a period in which failures took place, specifying its start and end hours (feature provided by the JCalendar[10] plug-in) and gets the respective primary-KPI trend on the bottom-left chart; finally, he can choose a particular failure value, through the top-right drop-down menu, and examine the trend of the factors leading to it.

4.3.4 Block configuration

Each distinct macro-block operating in the system must be preconfigured according to the needs of the overlying application. collectd and its embedded InfrastructureSensor must be told what devices of the infrastructure they have to "suck" information from, plus the rate at which they must do that; in ECoWare, all the required calculator, filter and aggregator instances must be declared along with the parameters "shaping" them; ECoWarePersistence must be given details about the primary and secondary events to be persisted; and ECoWareDashboard, similarly, must know the number and type of the charts to launch, as well as the metrics they will contain. Only exception with regard to this aspect are the sensors at software layer, which consists of few, application-agnostic code lines and need to know just where to send the data they acquire (that is, the location of the SIENA server, see Section A.2).

Apart from collectd, which provides its own configuration file to be set (see Section A.1), all the other elements must be input a file containing the configuration parameters, so that, as soon as the element is started, such file is parsed and the appropriate instances are created and launched. However, since all the elements have been implemented in an ECoWare-like fashion, that is, they are all made by independent objects, configuration is delegated

[10]http://www.toedter.com/en/jcalendar/



```
1331568363010: RT = 2819.498338870432
1331568382998: RT = 2600.12734082397
1331568402999: RT = 2275.6608695652176
1331568422998: RT = 1775.7052631578947
1331568442999: RT = 1162.2051282051282
1331568462998: RT = 1032.0615384615385
```

Figure 4.10: Historical dashboard

to such objects themselves. Doing so, modularity is further enhanced: in each component, the "scaffolding" is developed once and remains nearly unchanged, while specific internal entities are managed in autonomy.

The format chosen for the configuration files is XML, for two reasons: first of all, it allows to define data structures, even complex ones, in a simple and understandable way; second, it can be easily handled in the Java environment via the Document Object Model[11] (better known as DOM) APIs, already included in the Java standard library. In this work, the XML configuration file of a generic block must be organized in sections (or nodes), each of

[11]http://www.w3.org/DOM/

which profiling one of the instances required in the block; when a section is encountered during the parsing of the document, the block first recognizes the class therein treated (usually by looking at its name), and thus invokes the respective constructor, passing the section itself (that is, an XML *Node*) among the parameters so that it can auto-configure itself.

Chapter 5

Case study

The behaviour of the implemented system must be verified within a plausible environment, in which the main characteristics and challenges of a real service composition are reproduced. For this reason, a test application has been taken and separated in four pieces, each run at a distinct location in the Internet. A simulation has been executed and then repeated with increased application load. Finally, all the information extrapolated from the runs have been interpreted.

The chapter is divided in three sections. Section 5.1 gives a complete description of the test application. Section 5.2 provides the details about the experiments, first showing the testing tool exploited and explaining how the entire system (application plus monitoring) has been configured, and then reporting the two runs in all their aspects. At last, Section 5.3 analyses the statistics collected by the testing tool, to determine the overhead caused by monitoring and compare it to other factors.

5.1 Application overview

Figure 5.1 illustrates the SCA diagram of the application on which the monitoring infrastructure has been tested. It is one of the possible variants of Tuscany's tutorial, *store*, a simple program simulating an online food shop.

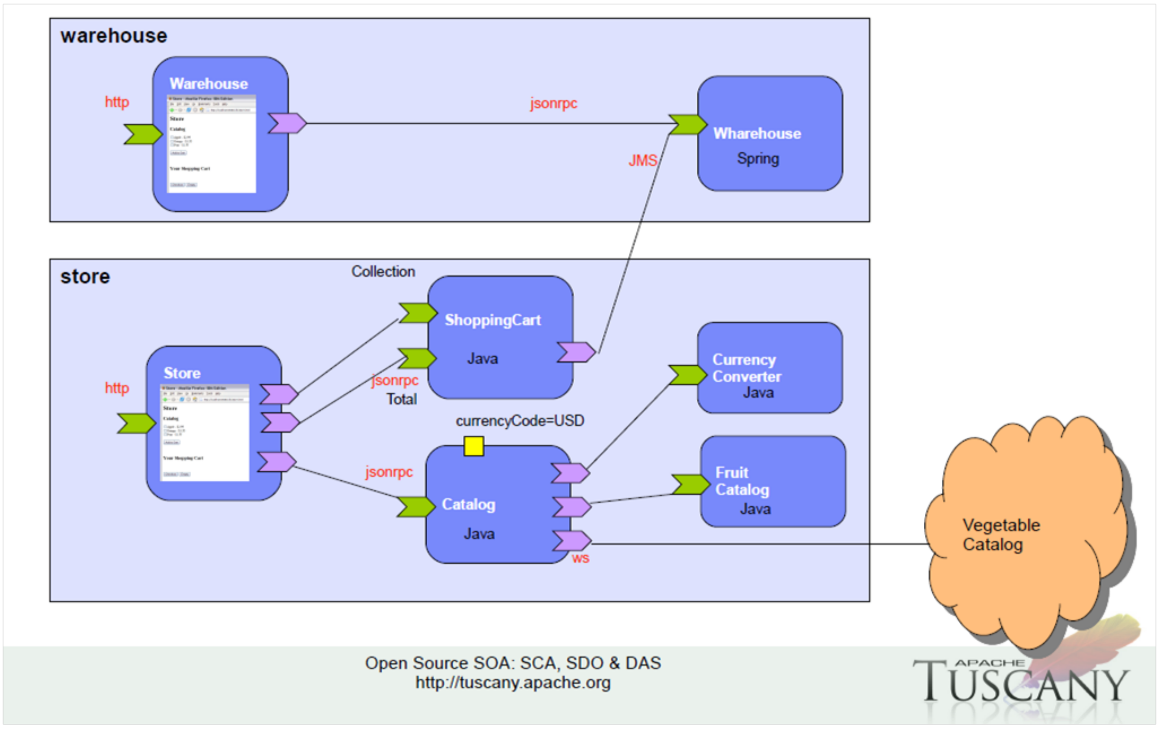


Figure 5.1: SCA diagram of the test application

As can be seen, the application is made by two composites: *store*, the front-end one, allowing users to select and order products, and *warehouse*, the back-end one, memorizing the committed orders. The orchestration component is *Store*, a mono-page Web site; for its purposes, *Store* makes use of two components: *Catalog*, which lists all the store items, putting together the ones contained in *FruitCatalog* with those in *VegetableCatalog* (here represented as a Web service external to the composite; actually, it is included in it), and in turn exploits the *CurrencyConverter* component to determine the price for items according to the chosen currency (which has to be specified in the *Catalog* property); and *ShoppingCart*, which represents the cart where the user puts the items that he wants to purchase. The *warehouse* composite contains itself an HTML interface, allowing only to visualize the pending orders in the depot (realized by means of Java, not Spring[1] as written in the

[1]http://www.springsource.org/

figure); both the components are named *Warehouse* in the figure.

Apart from the two HTML pages, all the components are implemented in Java; each of them features a single service except *ShoppingCart*, which provides *Cart*, allowing to modify the content of the cart (adding/removal of items etc.) and *Total*, confirming final orders and sending them to the warehouse. The red words in the figure indicate the protocols used to access the referred service: both the Web pages are obviously reached through HTTP requests; *Store* communicates with *Catalog* and the second service of *ShoppingCart*, *Total*, by means of JSON[2]-based RPCs[3] (Remote Procedure Calls), as the two *warehouse* components do with each other; *ShoppingCart*'s *Cart* is exposed in the Atom[4] format; *ShoppingCart* itself calls the warehouse via JMS[5] (Java Messaging Service); the unlabelled wires refer instead to a standard interaction among two Java classes.

Figure 5.2, instead, exploits the application diagram to highlight (even if in horizontal) its hierarchy. In this case, three layers can be determined according to the inter-component relationships, with *Store* at the top, *Catalog* and *ShoppingCart* just underneath it and the rest at the bottom. It can be also noted that the *Warehouse* Web page has been removed from the picture: it does not have a functional role in the store process (indeed, its service is not used by any component) and can be thus neglected.

Finally, Figure 5.3 indicates on the diagram how the application has been deployed in this work, identifying how many nodes have been employed and which capabilities are hosted on each of them. Precisely, the main node contains the *Store* component and constitutes the application access point; the second node includes the *ShoppingCart* component; the third node provides *Catalog* along with all its subordinate entities; the fourth and last node is entirely dedicated to the warehouse.

5.2 Tests

5.2.1 soapUI

A good instrument that can be used to perform exhaustive tests on service-based applications and simulate critical scenarios is soapUI[6], a software able to generate SOAP requests to access Web resources given the location of their

[2]http://www.json.org/
[3]http://json-rpc.org/
[4]http://tools.ietf.org/html/rfc5023
[5]http://java.sun.com/developer/technicalArticles/Ecommerce/jms/
[6]http://www.soapui.org/

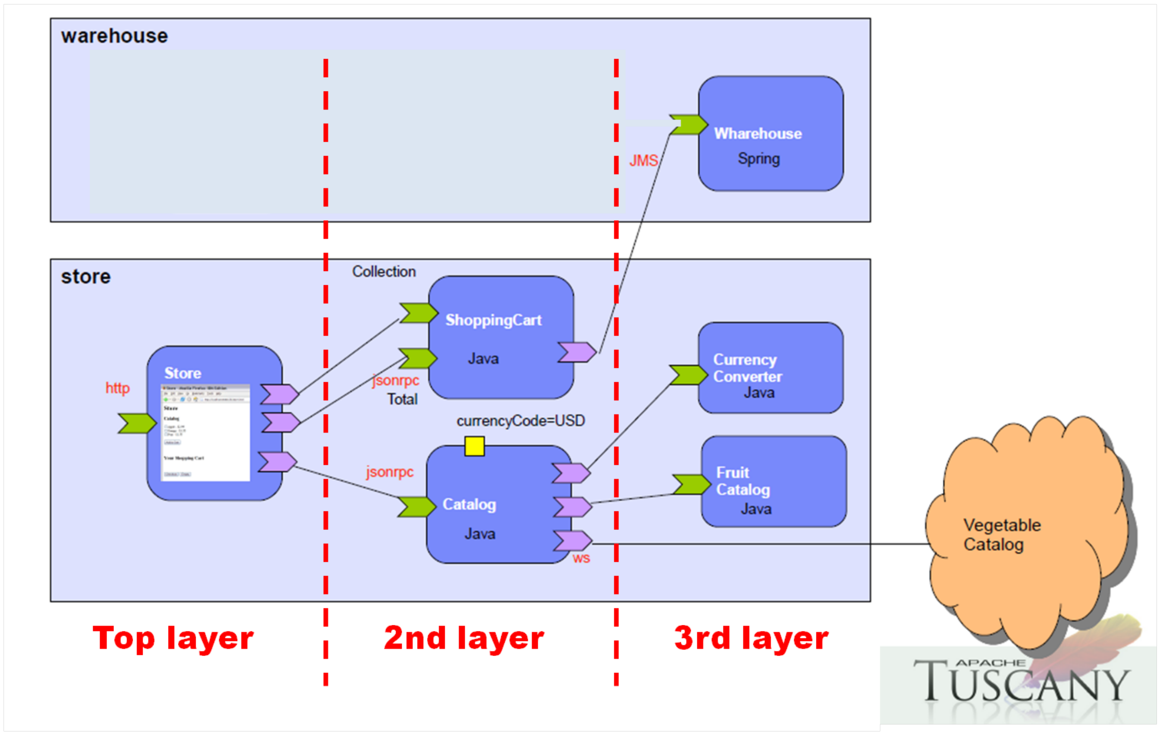


Figure 5.2: Test application - hierarchy

WSDL[7] (Web Services Description Language) document. soapUI makes it possible to operate valuable load experiments, invoking the application for a certain number of times rather than over a specified interval.

Figure 5.4 shows the window for such capability and its possible configuration parameters. Besides the test duration, these include: the number of threads which perform the invocations in parallel; the time a thread must wait before issuing a new invocation after the previous one has terminated; and a random parameter, between 0 and 1, indicating the probability with which such delay is respected. In the central panel, general statistics about the test can be visualized in real time.

To be accessed by soapUI, an application must thus be deployed, along with its WSDL, and exposed as a Web service at a specific address. Doing so is not that easy: a WSDL document, in fact, must include every single detail about the services, and thus writing it is quite laborious, even for small services; moreover, making resources available over the Internet requires to activate a Web server and load the files onto it, which is not difficult but still demands additional effort. Both FraSCAti and Tuscany, however, perform

[7]http://www.w3.org/TR/wsdl20/

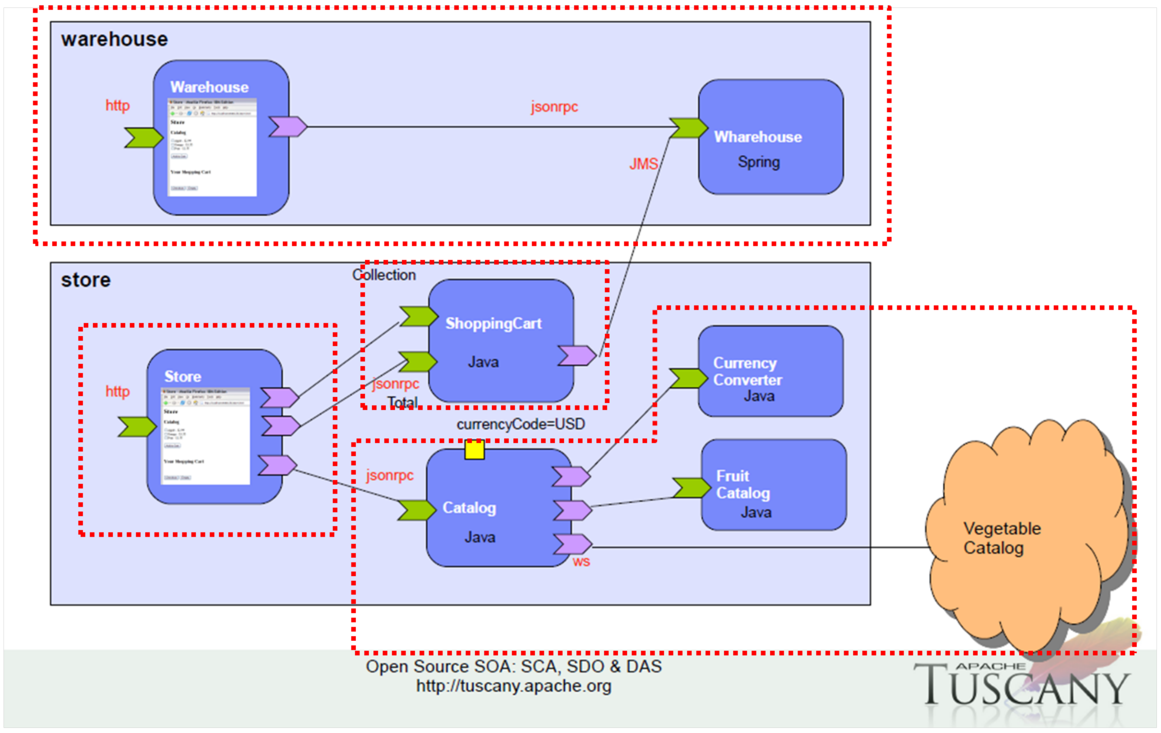


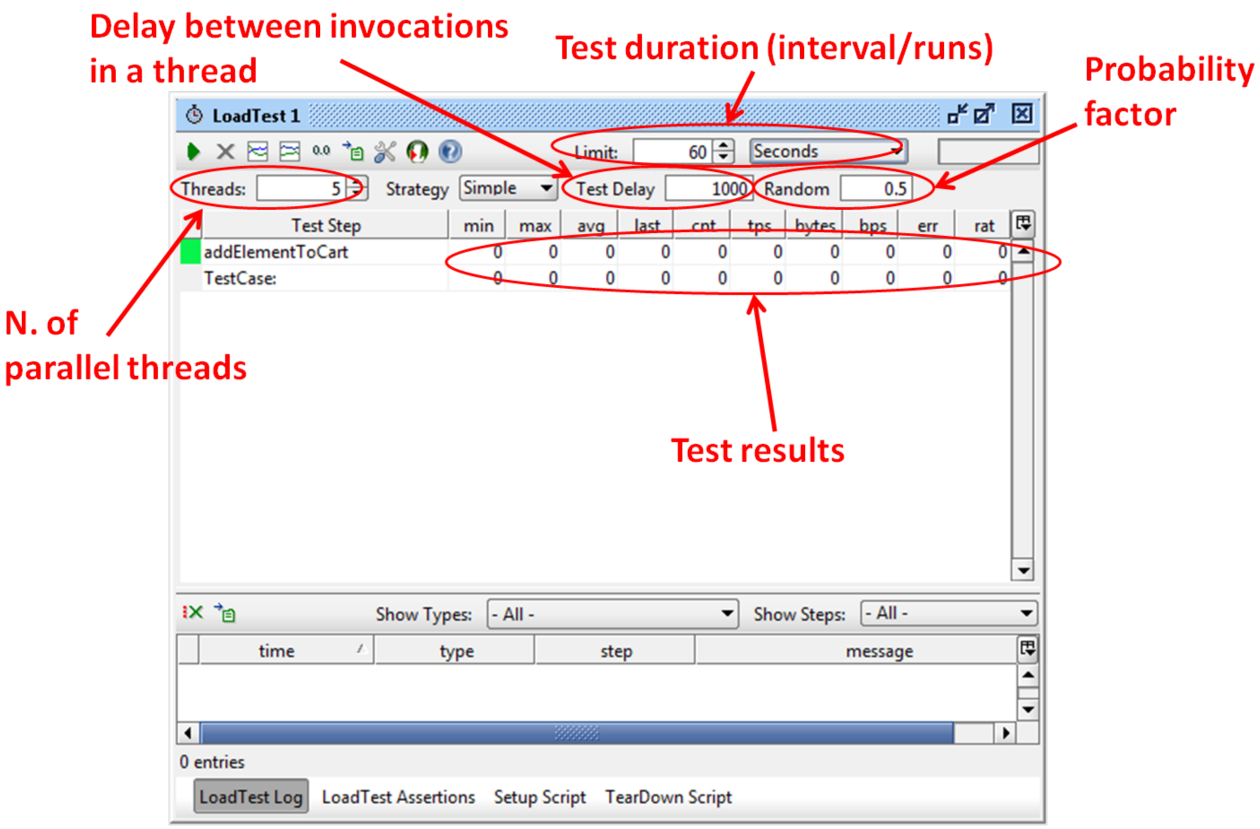
Figure 5.3: Test application - deployment

such tasks by themselves: their execution environments, indeed, include a server, which takes the deployment address of the service and operates the loading automatically, generating its WSDL as well.

5.2.2 System setup

The store application was created by the developers of Tuscany with the only purpose to provide a helpful example, which would easily introduce users to the capabilities of both their product and, more generally, SCA. The program, in fact, can be launched in no time (with two command-line instructions), runs entirely in local and offers a very poor and basic Web interface. Hence, in order to use it as a valid test bed for the monitoring architecture, it has been necessary to intervene and make it “trickier” by modifying some implementation details, still leaving (most of) its business logic unchanged.

First of all, as shown in the previous section, it has been decided to deploy the application on four separate locations. As many composites have been thus created (as the SCA deployment unit is the composite, see Section 4.2), dividing the *store* one in three parts and keeping the *warehouse* one



Limit: 60 Seconds
Threads: 5 Strategy Simple Test Delay 1000 Random 0.5

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
addElementToCart	0	0	0	0	0	0	0	0	0	0
TestCase:	0	0	0	0	0	0	0	0	0	0

Show Types: - All - Show Steps: - All -

time	type	step	message

0 entries

Figure 5.4: soapUI load test window

unaltered. This, actually, is not a difficult task: it is enough, indeed, to copy the .composite file of the application on each node, and comment everything in it except the tags referring to the components to be deployed there. Once distribution has been completed, the services needed to be accessible from outside have been exposed; this is an easy job too, since it just takes to correct, in the .composite file again, the address fields of the related service and reference tags.

Secondly, the original application requires user interaction, not allowing in fact to perform load simulations easily. The HTML-made *Store* component has been therefore replaced with an automatized process encapsulated in a Java class, which has then been exposed as a Web service to be invocable via soapUI. The flow of execution in such class reflects what a standard transaction would be: first, the entire store catalog is retrieved (method *get()* in the *Catalog* service); then, a number (randomly picked between 1 and 10) of casual items is chosen, and each of them is added to the cart, one at a time (*Cart*'s *post()* method); finally, the so-generated order is confirmed and dispatched to the warehouse (using *confirmTotal()* from *Total*, which in turn calls *addOrder()* on *Warehouse*). The introduction of the new component, though, has required to make further adjustments: Tuscany, indeed, does not support the JSON-RPC protocol among two Java artefacts, so the binding of the *Catalog* and *Cart* services have been changed to WSDL; moreover, the Atom format caused *Total* to fail over extensive use, and thus it has been switched to WSDL as well.

The four nodes used for the application deployment consist of the local computer plus three virtual machines provided through Amazon's EC2[8] (acronym of Elastic Compute Cloud) service. The local PC is an Acer Aspire 5738ZG notebook, featuring:

- OS: Windows 7;
- CPU: Intel Pentium T4400 (2.2 GHz, 800 MHz FSB);
- memory: 4 GB;
- disk: 500 GB;
- Internet access: 54-Mbps wireless DSL router.

All the virtual machine are instead EC2 Micro instances, set up as follows:

- OS: 64-bit Linux Ubuntu 10.04;
- CPU: up to 2 EC2 Compute Units, for short bursts (one EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor);
- memory: 613 MB;
- storage: 8 GiB (1 GiB = 2^{30} B) on an independent EBS[9] (Elastic Block Storage) volume;
- I/O performance: low.

From this latter data, it is clear that a Micro machine is not really suitable to host a service; such category, though, is the only one which Amazon provides, if kept under fixed usage thresholds, for free. In this particular case, moreover, services are not that complex, and created no problem for the instances. Each instance must be accessed from the local machine via

[8]http://aws.amazon.com/ec2/
[9]http://aws.amazon.com/ebs/

the SSH (Secure SHell) protocol using putty[10]; to handle it more easily via its GUI, instead, the remote-desktop tool TightVNC[11] has been exploited.

The monitoring system, too, has been organized to work in a distributed context. A collectd instance has been installed on each virtual machine, tracking statistics about the use of CPU, memory and network interface; no infrastructure monitoring has been run on the local node, because all the operation contained in the *Store* component are invocations to the other entities it is linked to. To forward all the gathered information, a SIENA network has been configured: one server has been operated on each peer, with all those running on the virtual machines linked to the "master" server installed in local. This has been decided to reduce the impact of monitoring on the application: doing so, indeed, publication of metric-events is decoupled from its transfer to the processing stage (entirely demanded to SIENA), limiting the application-SIENA communication to a local scope and thus diminishing the overhead due to publish-and-subscribe. Finally, all the ECoWare-related blocks (the processor, the dashboards and the database) have been executed on the local instance.

5.2.3 Ordinary scenario

The first test has been operated in order to measure the performance of the application and its services under "normal" circumstances. The tracked software-layer parameters are:

- the response time of the whole application;
- the arrival rate of the whole application;
- the response time of *Catalog*'s *get()* method;
- the response time of *Cart*'s *post()* method;
- the response time of *Total*'s *confirmTotal()* method;
- the response time of *Warehouse*'s *addOrder()* method.

The appropriate monitoring code has thus been added to Tuscany, and the related ECoWare calculators have been configured. For all the metrics, the values chosen for the interval and output values (see Section 4.3.2 for an explanation of such values) are, respectively, 2 minutes and 20 seconds. From

[10]http://www.putty.org/
[11]http://www.tightvnc.com/

now on, the metrics referring to specific methods will be identified with the name of the service they belong to.

Five macro-correlations have been considered:

- the response time of the whole application with the response times of *Catalog* and *ShoppingCart* as well as with the arrival rate;
- the response time of *Catalog* with the parameters of the underlying infrastructure;
- the response time of *Cart* with the parameters of the underlying infrastructure;
- the response time of *Total* with the response time of *Warehouse* and the parameters of the underlying infrastructure;
- the response time of *Warehouse* with the parameters of the underlying infrastructure.

The same number of off-line charts have been delineated in ECoWareDashboard. Plus, a live graph have been configured for the tracking of the arrival rate in real time.

The parameters used to define the normal circumstances mentioned above, with which soapUI has been configured to run the load test (see Section 5.2.1), are instead:

- test duration: 600 seconds (10 minutes);
- n. of parallel threads: 5;
- delay between invocations: 1000 milliseconds;
- random factor: 0.5.

Figure 5.5 shows the evolution of the arrival rate during the test. Even though the chart bounds are not sized perfectly, it can be seen that its trend is rather regular, with the obvious exception of the initial and final segments.

Figures 5.6 and 5.7 show instead the off-line dashboard representing a snapshot of the first macro-correlation (that is, the one between the response time of the entire application, its arrival rate and the response times of single services), illustrating the first two and last two secondary factors respectively. Likewise, Figure 5.8 shows the trend of *Cart* compared with those of the CPU allocated to the user and the packets received by the network interface. The units of measure are: milliseconds for every time metric, units of scheduling

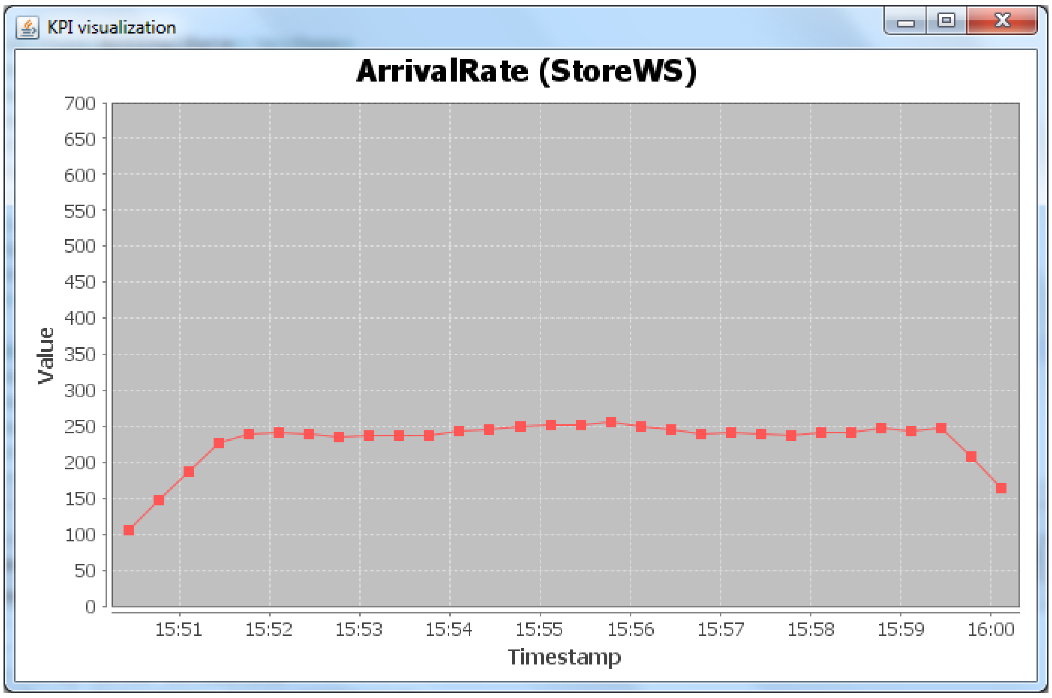


Figure 5.5: Ordinary scenario: arrival rate

for CPU usage, octets per seconds for interface packets. Some of the graphs here may seem very variable, but looking at the Y-axis it can be noted that the range in which points float is actually quite narrow.

Finally, Table 5.1 lists the values which, after evaluating the test results, have been decided as average and peak ones for the monitored metrics. These will be highlighted in the charts of, respectively, secondary and primary events of the appropriate dashboards. All the metrics are expressed in milliseconds except the arrival rate, depicted in number of requests.

Metric	Average	Threshold
Response time (Store)	-	1000
Arrival rate (Store)	250	-
Response time (Catalog)	115	125
Response time (Cart)	114	120
Response time (Total)	120	125
Response time (Warehouse)	8	15

Table 5.1: Ordinary benchmarks for monitored metrics

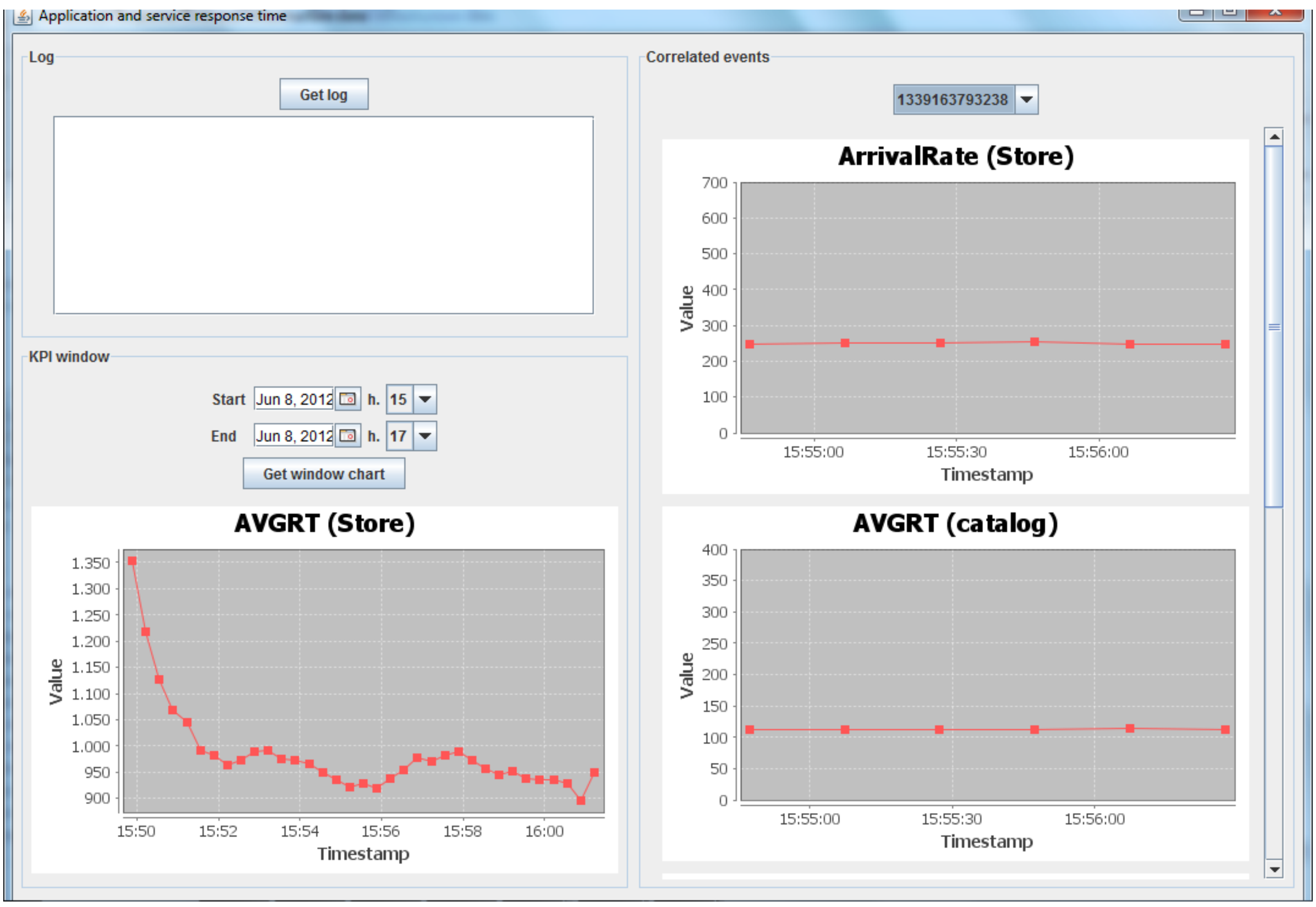


Figure 5.6: Ordinary scenario: AVGRT(Store) vs arrival rate and AVGRT(Catalog)

5.2.4 Altered scenario

Once the parameters defining normal application performance have been established, it is possible to introduce opportune ECoWare filters in the system, to signal when a service exceeds its critical threshold. Therefore, five high-pass filter instances have been configured, each associated to a different response-time metric; no threshold has been defined for the arrival-rate one, thus no filters have been attached to it. Moreover, all the correlators have been modified so to take their primary input no longer from the respective calculator blocks, but rather from the filters themselves, in a way they could operate only in case of failures.

Subsequently, the load test has been repeated worsening the conditions, that is, increasing the frequency of contemporary requests. The values en-

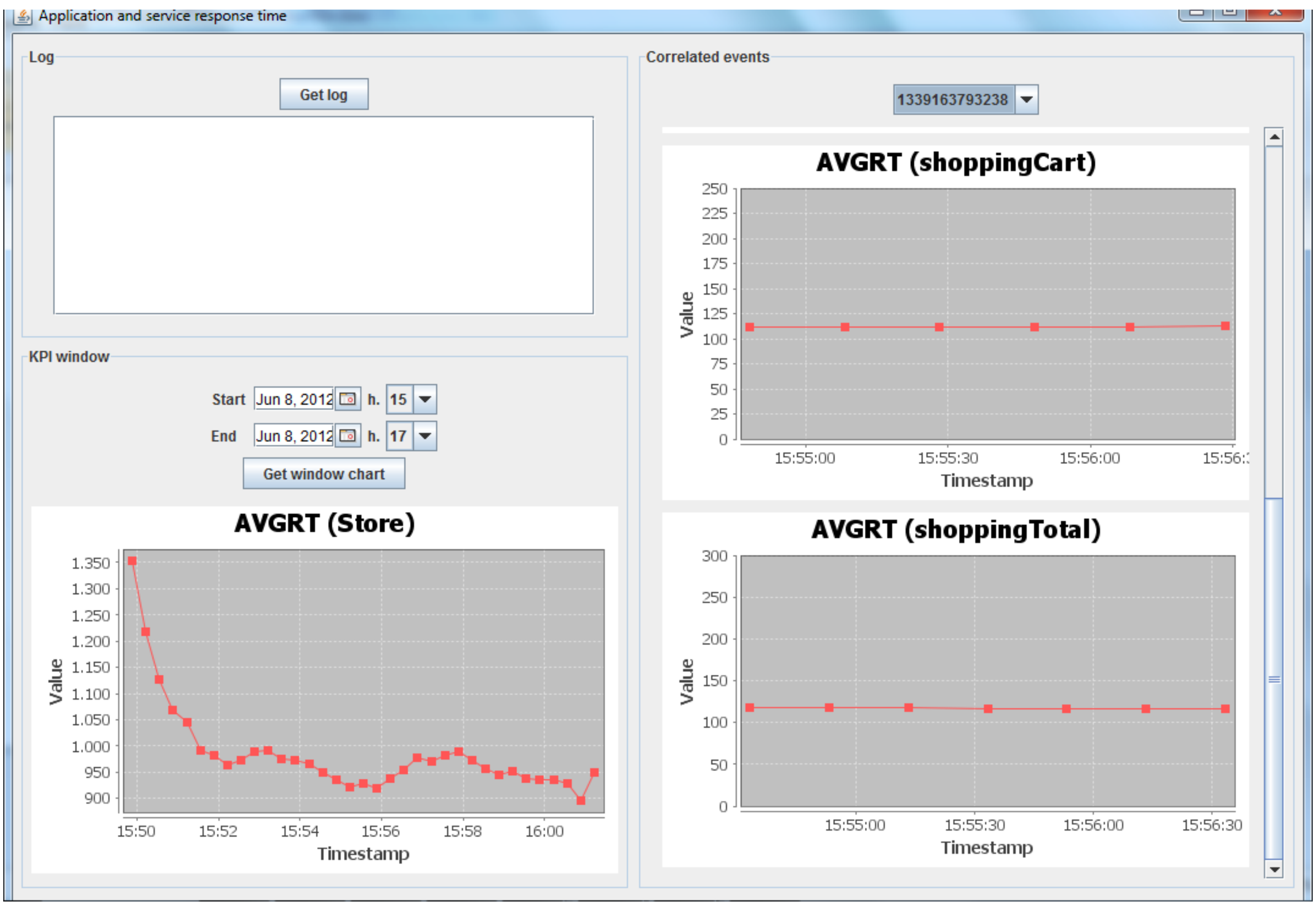


Figure 5.7: Ordinary scenario: AVGRT(Store) vs AVGRT(Cart) and AVGRT(Total)

tered in soapUI are:

- test duration: 600 seconds (unchanged);
- n. of parallel threads: 10;
- delay between invocations: 0 (within a thread, a new invocation is sent as soon as the previous one finishes);
- random factor: 0 (no delay randomization).

Figure 5.9 shows the arrival rate registered in the new scenario. As can be seen, the average here is less than the norm (indicated by a horizontal line on the chart). This could seem surprising, even wrong, given that the

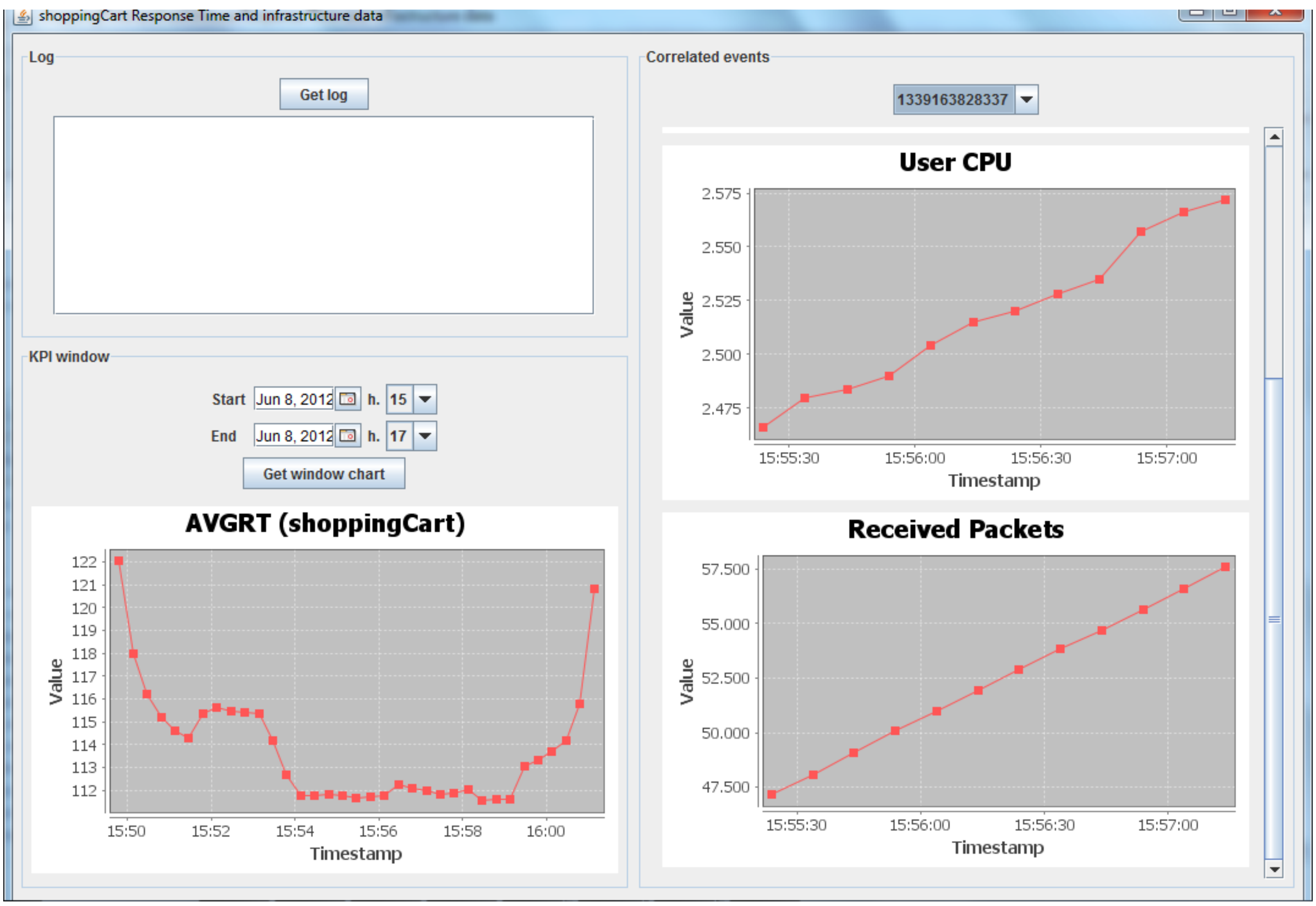


Figure 5.8: Ordinary scenario: AVGRT(Cart) vs user-scheduled CPU and received packets

number of invocations has been augmented. In fact, this is perfectly right: the arrival rate, as shown in Table 4.1, is calculated as the count of all the occurred *StartTime* events generated by the service (*Store* in this case), not the amount of SOAP requests to the application; therefore, since there is more traffic and response times are very likely to be higher, its decrement is more than plausible. The question can arise on the formula itself, which possibly does not reflect the real meaning of "arrival rate"; under these particular circumstances, though, it can be said that the result meets the theoretical expectations.

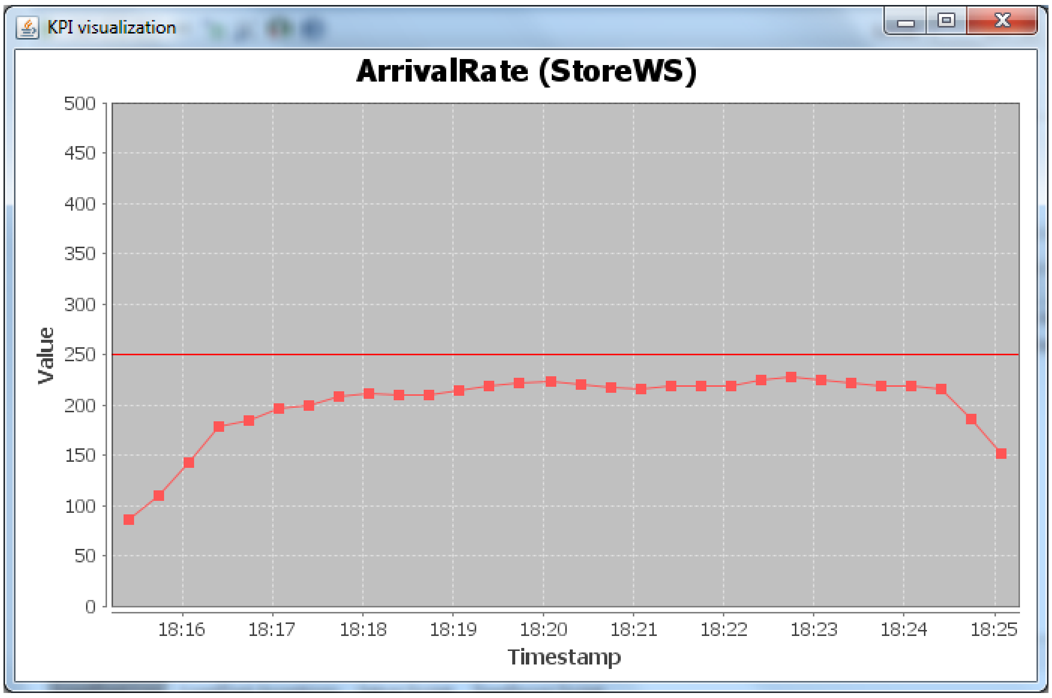
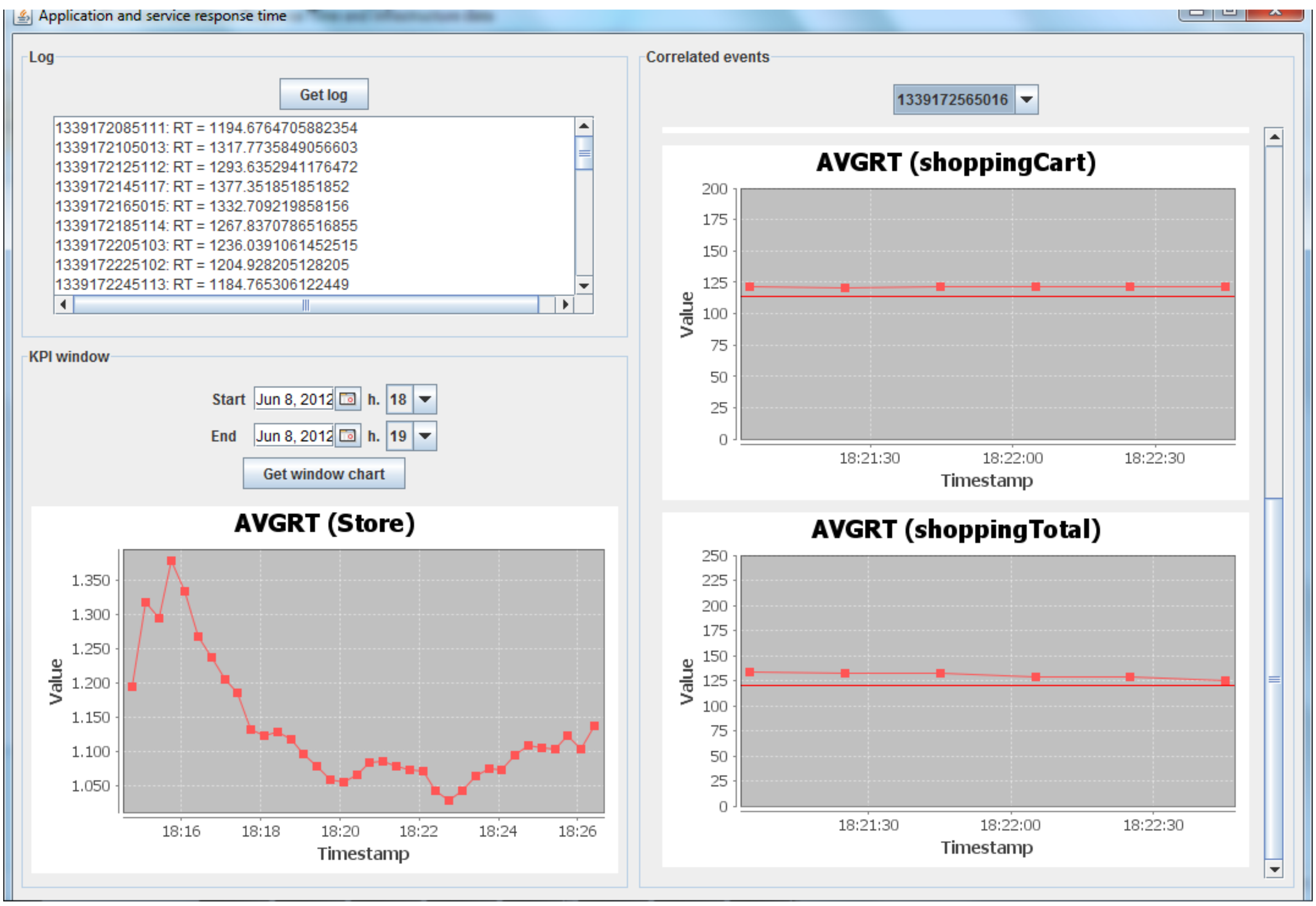


Figure 5.9: Altered scenario: arrival rate

Figure 5.10 reveals how the overall application performance has worsened: all the values in the primary chart overcome the 1-second alert threshold, which, since the chart auto-scales if no bounds are specified, is not even shown. On the right-hand side of the figure, it can also be noted how both the services of *ShoppingCart* have slowed down, with response times well over their normal average. The behaviour of *Catalog*, not visualized here, is instead good, even better than before. These results meet the theory too: indeed, being called on average 5.5 times in a single invocation, *Cart* is the application's most solicited service, and thus it will be the first one dropping; belonging to the same component, hence being executed at the same location, the *Total* one will be negatively affected as well.

Figure 5.11 shows why the *Total* performance, and more generally of the *ShoppingCart* component, is not as good as before. In this case, the response time is rather unstable, with about half of the points over the limit. Focusing on one of such points and analysing the correlated factors, the (predictable) reason is soon found: the number of incoming packets is enormously higher (about three times if compared to Figure 5.8, in which the infrastructure data regards the same component). The improvement of *Warehouse*'s response time (not illustrated in the figure) further demonstrates that the slowdown motivation had to be searched within *ShoppingCart*.



```
1339172085111: RT = 1194.6764705882354
1339172105013: RT = 1317.7735849056603
1339172125112: RT = 1293.6352941176472
1339172145117: RT = 1377.351851851852
1339172165015: RT = 1332.709219858156
1339172185114: RT = 1267.8370786516855
1339172205103: RT = 1236.0391061452515
1339172225102: RT = 1204.928205128205
1339172245113: RT = 1184.765306122449
```

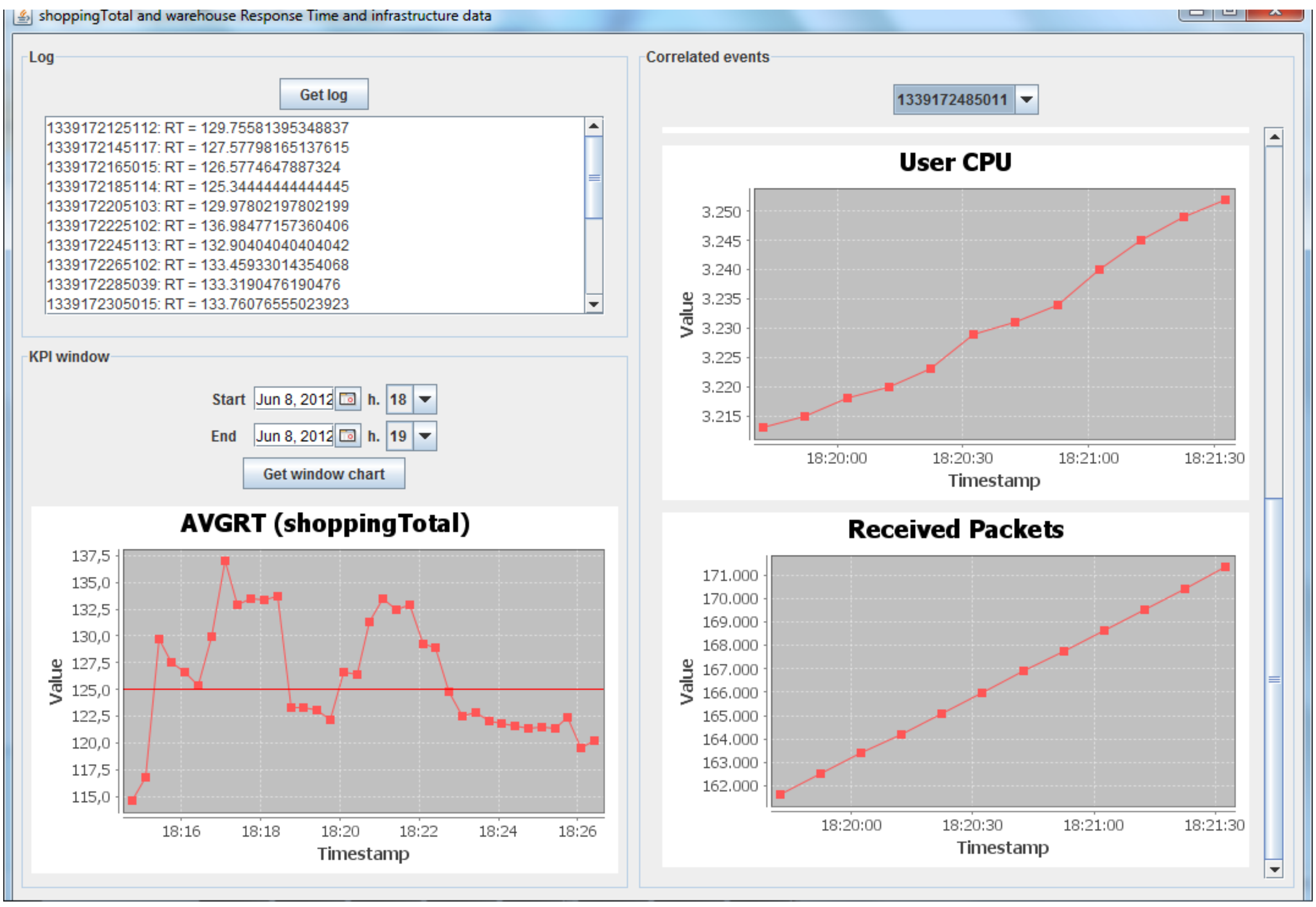
Figure 5.10: Altered scenario: AVGRT(Store) vs AVGRT(Cart) and AVGRT(Total)

5.3 Monitoring impact

Both the experiments just documented have been replicated with monitoring disabled, so to have a rough evaluation of the influence of the monitoring structure on the application performance. Table 5.2 reports the values registered by soapUI in the four tests in terms of response time (minimum, maximum and average) and requests executed, comparing the results of the related cases. Such values are significantly bigger than the ones appraised by ECoWare, as they take into account the "complete" response time, including the time required to issue the SOAP request and receive the response.

Three considerations can be done on such data:

- SOAP has a huge impact on performance. If we compare the average



```
1339172125112: RT = 129.75581395348837
1339172145117: RT = 127.57798165137615
1339172165015: RT = 126.5774647887324
1339172185114: RT = 125.34444444444445
1339172205103: RT = 129.97802197802199
1339172225102: RT = 136.98477157360406
1339172245113: RT = 132.90404040404042
1339172265102: RT = 133.45933014354068
1339172285039: RT = 133.3190476190476
1339172305015: RT = 133.76076555023923
```

Figure 5.11: Altered scenario: AVGRT(Total) vs user-scheduled CPU and received packets

Scenario	Min	Max	Avg	Total requests
Ordinary with monitoring	450	9921	1746.09	1199
Ordinary without monitoring	437	3337	1579.35	1283
Differential	-2,9%	-66,4%	-9,5%	+7%
Altered with monitoring	482	44151	5681.09	1050
Altered without monitoring	465	15126	5097.35	1173
Differential	-3,5%	-65,7%	-10,3%	+11,7%

Table 5.2: soapUI test results

values in the table with those from the *Store* charts, the amount of time due the SOAP protocol is around 50% in the ordinary case and 80% in the altered one, and the latency due to SOAP sees a fivefold increase between the two tests, which is impressive if compared to the 1.27 increase seen by *Store*;

- the minimum response time is almost the same for all the cases, while the maximum response time presents a much higher variability. This latter metric, though, is not fully indicative in itself, as usually soapUI's first invocation is significantly much slower than the rest; however, it is evident that it grows very rapidly with the traffic;
- apparently, the monitoring system has an equal impact in both the cases. While poor with regard to the minimum value, it gradually goes up to an average 10%, a percentage that certainly can not be neglected. However, after running the tests multiple times, we have seen that the response times can sometimes be worse with monitoring disabled than with monitoring enabled.

Given these arguments, it can be finally deduced that, in this case at least, performance is more dependent on the network conditions than it is on the monitoring architecture. In fact, attaching monitoring to the application contributes to degrade such conditions, as a huge number of SIENA-event packets is added to the traffic; this overhead, though, is much less than that caused by SOAP. The third point empowers this theory, clearly showing that, at least in the ordinary scenario, monitoring has a negligible influence compared to the Internet. More generally, however, values rise very quickly as the request rate increases, even alarmingly considering the maximum one.

Chapter 6

Conclusions

This chapter closes the work.

The chapter is divided in two sections. Section 6.1 recaps what has been treated and learned. Section 6.2 talks instead about the work that is yet to be done, with regard to service-based computing in general and this project in particular.

6.1 Final considerations

In this work, it has been approached the problem of performance adaptation in service-based applications, a question more and more important as they are likely to replace traditional ones. In fact, while services are rapidly spreading all over the IT world, valuable monitoring solutions for them are yet to be found; this lack is bad for both the provider, who needs to detect and remove the weaknesses in the application (points of failure, bottlenecks and so on) in order to improve it and get more competitive on the market, and the customer, who needs to keep the QoS well monitored and verify that the SLAs he agreed for are respected. The project at the base of this study presents a self-adaptation process tailored for SBAs, which takes into account their multi-layer nature to devise a complete strategy; here, the very first stage of such process, that is, monitoring, has been realized.

The content of this thesis can be divided in three parts. In the first one, the problem has been introduced by explaining the concept of services in the IT context, identifying the motivations of their rather immediate success as well as the challenges they come with, and giving an initial idea of the overall solution and the monitoring implementation. In particular, it has been argued how a service composition is in its nature structured on more levels (or layers), going from the general application schema at the top down to the physical software and hardware resources at the bottom; it is easy to understand that this complicates adaptation a lot, because it creates several dependency sub-chains which can not be ignored in an adaptation process. Then, the MAPE-like, four-step, self-adaptation cycle has been delineated, with a brief description of the stages it is made of and the interactions it has with the supervised application, and the realized system has been briefly anticipated. Finally, a roundup of some related studies has been presented, implicitly showing that, despite each of them has interesting features that could be exploited, there is no technology capable of solving the issue in a satisfying way at the moment, and thus making evident the innovation that the approach here discussed brings to this field.

The second part has instead provided a detailed illustration of the cycle. Each of the four macro-blocks has indeed been "opened", describing the passages it goes through in order to get its task done and indicating the properties that its optimal implementation should have. Here, it can be seen that every single stage constitutes a fundamental piece for the purpose of the work, and it is also made clear the difference between (self-)controlling a conventional "flat" application and doing the same with a multi-layer SBA: in the first case, the second and third blocks (respectively, the analysis and identification ones) would not even be required, as problems could be detected directly in the monitoring phase (which, besides, would not need data correlation functionalities), and thus immediately resolved by enacting a single set of opportunely predefined actions.

In the third part, the practical (and central) aspects of the work have been described. Initially, the implemented monitoring-and-correlation architecture has been graphically introduced; next, it has been analysed in depth, first indicating what kind of applications it can take care of (that is, SCA-based ones), then entering the architecture itself and explicating it component by component. Finally, the (two) experiments run to demonstrate its behaviour have been documented, presenting all the characteristics of the test program, the conditions in which the tries have been executed as well as their results, and a rough impact evaluation. In summary, it can be concluded that the experiments confirm the main downside of an SBA, that is, unstable and strongly network-dependent performance; the influence of monitoring has instead proved to be rather limited.

6.2 Future developments

The world of services is still in its developmental age: new technologies break into the market without interruption, while existing ones update their versions on a regular basis. The general tendency is to reduce complexity, by devising more agile communication protocols, defining less costly description languages and making services themselves smaller and smaller in order to maximize flexibility. The future is far from clear at the moment; however, one consideration can be done: to assess themselves permanently, services need to find reliable ways to make their performances immune to the inevitable uncertainties characterizing the IP network. Given that, a rather simple prediction sees research in the adaptation field continue to grow exponentially, at least in the same measure as services themselves do.

Much has yet do be done with regard to the here-presented project too, as the remaining part of the loop must be implemented. From the previous reflection, it can be deduced that the blocks most challenging to realize will be those interacting with the application, as they must be able to deal with several different technologies, each coming with its own interface and adaptation APIs; a fundamental factor, therefore, will be the degree of abstraction in such blocks: the more technology-independent they will be, the more reusable and versatile the cycle will. Interesting insights in this direction could be taken from [13] (see Section 2.4.1).

It is equally evident, though, that the current middleware solutions, in the first place, have to grow and improve on this aspect. In the implementation chapter, it has been shown how laborious it is to put monitoring into Tuscany, which in addition does not provide any functionality for run-time rearrangement of compositions; with FraSCAti, the situation is surely better, considering the (few) APIs it comes with to insert intents and dynamically reconfigure composites, but it is still too little. What each SOA-enabling tool should include is complete, simple interfaces allowing to operate performance monitoring as well as, at execution time, all the possible structural changes according to the paradigm it implements.

With regard to the monitoring system in particular, a really useful improvement could instead be to define a unique, high-level configuration language for all its blocks, through which the user can specify all the capabilities he needs in a single document, without having to know how the file of each block should be compiled. This would require to add a new block, taking in input such “global” document and producing the appropriate XML files, that by now have to be created by hand; this way, neither changes should be made to the pre-existing architecture nor application performance would be affected at all, as the new block would have to be run only once, at the launch of the system. About validation, moreover, although the base provided by this work is quite good, it would be opportune to test the system within a more realistic context, that is, on a complex, highly distributed application, in extremely critical traffic conditions and having to run a huge number of ECoWare instances.

Appendix A

Main tools exploited

A.1 collectd

collectd[1] is a lightweight application capable of gathering information about various aspects of the machine on which it is executed. It operates as a daemon (and that is where the final “d” comes from), that is, a program which runs in background instead of under the user control, and it can work on Linux as well as most UNIX-derived systems, among which MAC OS X, but not on Windows ones. In this work, the 5.1 version of collectd has been used.

The architecture of collectd consists of a small core, acting almost exclusively as a communication medium, and a huge number of attachable plug-ins, which carry out the actual functionalities. Some of them are listed in Table A.1. As can be seen, plug-ins belong to, at least, one category: *Read* plug-ins, for instance, are the one retrieving data from specific devices into the system and sending it to the core, while *Write* plug-ins receive collected data from the core and put it into specific-format files; the Network plug-in, instead, belongs to both the categories, as it allows remote instances of the daemon to exchange information.

Name	**Type**	**Description**
CPU plugin	Read	Records the time spent by the CPU in various states.
Memory plugin	Read	Monitors memory utilization.
Interface plugin	Read	Collects information about the traffic of interfaces.
Load plugin	Read	Observes the system load.
CSV plugin	Write	Stores all the collected values in CSV files.
RRDtool plugin	Write	Writes collected values on RRD files.
Network plugin	Read, Write	Enables communication with other collectd instances on different machines.
Java plugin	Binding	Allows Java user-made programs to be run into collectd and communicate with it.

Table A.1: Main collectd plug-ins

The reason why collectd has been preferred to all the other tools, though, is the last plug-in listed in the table, that is, the Java one. This, in fact, embeds a Java Virtual Machine (JVM) into the tool, thus giving the user the possibility to include custom-made programs to achieve specific functionalities. The plug-in also exposes a set of collectd APIs, allowing to communicate with the core in both directions; among them, the *CollectdWriteInterface* interface, which *write()* method is called by collectd every time a new metric has been acquired, has been the perfect instrument to link

[1]http://collectd.org/

the sensor with the rest of the monitoring system. Precisely, such interface has been implemented in the InfrastructureSensor project by the *DataSensor* class, which *write()* method simply takes the notification metrics in the *ValueList* parameter and transforms them into SIENA events which will be finally published.

collectd must be set up by writing an appropriate file, in which one must enable and configure all the plug-ins he wants to be included, as well as global parameters like the frequency of metric collection and the name of the host. In this work, only the basic *Read* plug-ins have been included, as the ones in the table, which do not have any configuration parameters to be specified. To configure the Java plug-in and run a custom program, instead, two operations must be done: first, the program must be exported as a .jar file; then, in the plug-in configuration section, it is necessary to indicate the location of the archive and the names of all the classes in the project which implement a collectd-provided interface (only *DataSensor* in this case). Once collectd is started, each class must be instantiated and registered with the daemon: for this reason, its constructor must be non-parametric, and specify as first instruction the appropriate registration method of the *Collectd* class, according to the implemented interface; in this case, since *DataSensor* implements *CollectdWriteInterface*, the *registerWrite()* method has been used.

A.2 SIENA

SIENA[2] (acronym of Scalable Internet Event Notification Architecture) is a small Java program which implements the publish-and-subscribe technology, thus allowing different, loosely coupled applications to communicate without having to know about one another (that is, without caring about IP addresses). The paradigm constituting the foundation of SIENA (and of publish-and-subscribe in general) is content-based networking, which establishes interaction among peers in a network based on the content of the exchanged messages rather than the physical location of the nodes. In this work, the 1.5.5 version of SIENA has been used.

The architecture of SIENA follows the client/server model, which is very similar to the collectd core/plug-ins one: indeed, it makes use of a standalone, lightweight server, acting as a message dispatcher (that is, what in the figures is represented as the bus) to which publishers and subscribers (clients) connect to, respectively, provide and retrieve messages, called events. The server can be started by simply executing the *StartServer* class, specifying

[2]http://www.inf.usi.ch/carzaniga/siena/

the port for the service in input; clients, instead, must be included and configured in the respective applications, according to the role operated in the system and the events of interest. Servers running on different machines can also be connected together, in order to create a unique, transparently distributed server and provide multiple access points.

A generic SIENA client exploits the capabilities offered by the *ThinClient* class, which must be instantiated passing the address of the server as a textual parameter in the "protocol:address:port" format; the default SIENA protocol, used in this case as well, is TCP. A publisher must invoke its *publish()* method in order to generate an event, passing the event itself as the only parameter. A subscriber, instead, uses the *subscribe()* and *unsubscribe()* methods to, respectively, connect to and disconnect from the server; in the first method (at least), it must provide as first parameter the pattern that the events of interest must match. That can be made by means of either the *Pattern* or the *Filter* class; in this work, the latter has been used. A *Filter* defines a list of attribute constraints, each of which can be added via the *addConstraint()* method, passing the name of the attribute and the target value for it. Before attaching itself to the server, therefore, a subscriber must declare and opportunely configure a *Filter* instance.

Despite making use of the same class, publishers and subscribers are substantially different entities. A publisher, in fact, is activated only "in case of necessity", that is, when a new event is to be delivered; on the other hand, a subscriber is an independent thread, constantly waiting for events and handling them as soon as they arrive. To do that, it must implement the *Notifiable* interface and override its *notify()* method, which is the one called by the server in order to dispatch the events it receives. As already explained in Section 4.3.1 and shown in Figure 4.4, the realized system implements the publisher functionalities through the *SienaOutputAdapter* class and those of the subscriber through the *SienaInputAdapter* class; in particular, the launch of components which need to receive data, ECoWare calculators as an example, merely consists of starting a new *SienaInputAdapter* thread.

The class which encapsulates the messages exchanged within SIENA is called *Notification*. Publishers must translate data into a *Notification* instance before sending it, while subscribers perform in the opposite way to retrieve the original events from it. A SIENA notification is very similar to a Java *Map*, including a set of (attribute name, attribute value) pairs; to add an attribute along with its value to a *Notification*, the *putAttribute()* method must be used; specularly, the *getAttribute()* method is provided to retrieve the value of an attribute given its name; finally, a list of all the attribute names in the notification can be obtained through *attributeNamesIterator()*, returning an *Iterator* instance. Attributes of the Java primitive types can be included in a notification directly, while complex data structures must be first serialized into byte sequences.

A.3 ESPER

ESPER[3] (version 4.4.0 has been used in this work) is a Java program providing Complex Event Processing (CEP), that is, it gets streams of messages, called events, and elaborates them in real time. CEP works in the opposite way as classical Relational Database Management System (RDBMS) processing does: indeed, while in the latter the database is already populated (and changes on it are usually small compared to its dimension) and is accessed through variable queries, in CEP it is the queries itself at the center, since they are fixed (predefined at design time) and operate as functional blocks, receiving events as soon as they are generated and outputting the result of their manipulation as new events. Having to deal with real-time computing, CEP tools must guarantee high throughput and low latency, as well as support a wide range of operations, including complex ones as pattern detecting, filtering, joins, data windows and so forth.

Intra-ESPER communication is carried out, again, following the publish/subscribe paradigm. The architecture of the program is thus identical to the SIENA one: a central server (here called engine) and several clients attached to it. There are two differences, though. The first one is in the nature itself of the applications: ESPER, in fact, provides not only message dispatching, but also, and primarily, event processing, and thus its server includes the logic necessary to elaborate events on top of communication. Secondly, the ESPER engine, differently from the SIENA server, is not a standalone component, but rather must be contained in the application using it, together with all its clients; this also explains why it has been necessary to use SIENA in this work, that is, to allow inter-component communication.

In ESPER, the engine is represented by the *Configuration* class, containing all the execution parameters. An instance of such class must thus be created in the initialization phase of the application, which is also the only point at which it can be profiled directly (that is, by exploiting the *Configuration* APIs). In modular applications like ECoWare, though, each block has its own needs in terms of engine features, and must specify them within its scope in an independent way; for that reason, engine configuration in ECoWare is distributed between single components, which receive

[3]http://esper.codehaus.org/

the Configuration instance allocated at initialization time among the input parameters of their constructors.

The first properties a block must configure are the events needed for its processing. In ESPER, an input event can be represented in several ways, as a *Map* instance, an XML *Node*, or even a specific POJO (Plain Old Java Object), while output events are usually encapsulated into *Map*s only; in this work (and later it will be clear why), it has been decided to use the *Map* representation universally. To include a new event definition into the engine, the *Configuration*'s *addEventType()* method must be called, providing its name and the related attribute map. For each event, it is not necessary to define its structure entirely, but rather it is enough to indicate the attributes that will be explicitly used in the computations; this ensures a greater flexibility in event representation, allowing to change the structure of an event without having to modify the code of the blocks operating on it. In ECoWare, furthermore, calculator and aggregator blocks define also their output events, so that to make eventual downstream filters independent from event types.

As said before, though, after the initialization phase the engine configuration can no longer be accessed directly; still, it is possible to edit it via the *EPServiceProvider* interface, an instance of which can be obtained by calling the *getProvider()* method on the *EPServiceProviderManager* class, passing the configuration itself as a parameter. From a provider, the configuration is reached in two steps: first, the *getEPAdministrator()* method must be called to get an *EPAdministrator* instance; second, a *ConfigurationOperations* object must be retrieved, invoking *getConfiguration(). ConfigurationOperations*, finally, allows to add new event descriptions through its *addEventType()* method, which has the same signature of the "original" one.

Once all the required events have been specified, it is necessary to define the operations to process them, which is done in ESPER by declaring and activating one or more statements. An ESPER statement is none other than a query, stored in the engine, which waits for events and, as soon as a new one arrives, performs its operations and output the result. A statement must be written in EPL (Event Processing Language), a query language extending traditional SQL to deal with real-time streams. The most important features added by EPL are:

- event windows (or views): queries can be set to consider a limited group of events in a stream rather than all of them. For example,

  FROM event-name.win:length(x)

operates only on the last x events of *event-name*,

FROM event-name.win:time(x)

takes only the *event-name* events occurred in the last x seconds,

FROM event-name.win:length_ batch(x)

batches events and releases them when their count is x (can be done with time as well). If nothing is specified after the stream name, the statement operates on new events only; to retain all of them, it is necessary to add the *keepall()* instruction;

- fixed output rate: the OUTPUT clause can be added at the end of the statement to stabilize the output rate, so that the query will output a result at fixed time intervals rather than when specific conditions are met;

  OUTPUT (FIRST|LAST|. . .) EVERY x (seconds|minutes|. . .)

  OUTPUT . . . AT date

  OUTPUT . . . WHEN trigger-expression

  are all examples of such clause;

- row limit: the number of tuples to be output can be limited by the LIMIT clause. For instance,

  LIMIT x

  returns no more than x rows;

- unidirectional joins: when two streams are joined, if nothing is specified after the stream name in the FROM clause, the query operates every time a new event arrives in any stream. Adding the UNIDIRECTIONAL keyword after one of them, as in

  FROM x UNIDIRECTIONAL, y,

  it will produce an output only when the new event comes from x, simply retaining the event if it comes from y; this feature has been used in ECoWare to implement correlators.

Statements can be added to the ESPER runtime by means of, again, the *EPAdministrator* interface: its *createEPL()* method, precisely, includes a statement, passed as a textual parameter, into the engine, which activates it immediately.

Listing A.1 shows the EPL statement exploited in ECoWare to calculate the average response time of several invocations on the same service. In this case, a new (time stamp, value) pair is produced every *outputTime*, in which the first number is obtained via the ESPER's *current_ timestamp()* single-row function, while the second number is the KPI value computed over the invocations occurred in the last *intervalTime*. The WHERE clause specifies the requirements used to join the two events referring to a single invocation: they must come from the same source (here identified as x) and have equal invocation numbers (which of course must be unique within service x). The SNAPSHOT keyword in the OUTPUT clause indicates instead to consider in the computation all the events included in the windows; in this situation, though, that would not have been necessary, since it is done automatically with aggregate functions.

```
SELECT AVG(et.timestamp - st.timestamp) AS value,
    current_timestamp() AS timestamp
FROM StartTime.win:time(intervalTime) AS st,
    EndTime.win:time(intervalTime) AS et
WHERE st.originID = 'x' AND et.originID = 'x' AND
    st.processNumber = et.processNumber
OUTPUT SNAPSHOT EVERY outputTime
```

Listing A.1: Sample ESPER statement

To get the events produced by a statement, a listener must be attached to it. An ESPER listener is an object implementing the *UpdateListener* interface, which is the equivalent of the *Notifiable* interface in SIENA; in this case, though, the method called by the engine to communicate updates is named *update()* (opposed to SIENA's *notify()*) and has two parameters, containing the insert stream (that is, the events just output) and the remove stream (the events which just left the window) respectively. Elements in both arrays implement the *EventBean* interface, from which data can be retrieved in two ways: by de-parsing it into the original structure, through the *getUnderlying()* method, or by getting the value of each single attribute, given its name, through several invocations of *get()*.

In ECoWare, as shown in Figure 4.5, each block includes exactly one *UpdateListener* instance, which is attached to the "final" statement, that is, the one producing the block output events. Every block works only on newly generated events, and thus the second parameter of *update()* is never used. From the figure can be also deduced that the task of *UpdateListener*s in all ECoWare blocks is solely to transfer received events to SIENA: for this reason, a unique listener class, named *EventListener*, has been implemented to comply with all blocks. An instance of such class simply takes each event, retrieves its parameters, puts them into an aptly labelled *Map* and finally passes it to its *SienaOutputAdapter* module. The choice of using *Map*s to represent the events makes is now clarified: with *Map*s, in fact, the processing of *EventListener* can be written in a block-independent way, that is, without the need to know the details of each block; POJOs, instead, would have required to define a different listener for every block, to deal with the specific structure of its output event. Moreover, events created from aggregated functions (that is, the type of operations provided by most ECoWare calculators), of which ESPER does not know the underlying structure, are automatically stored into a *Map*: this further justifies the decision.

Bibliography

[1] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. In *Communications of the ACM - Service-oriented computing, Volume 46, Issue 10*, 2003.

[2] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. In *Computing systems, Volume 15, Issue 1*, 2001.

[3] Sam Guinea, Gabor Kecskemeti, Annapaola Marconi, Branimir Wetzstein. Multi-layered Monitoring and Adaptation. In *Proceedings of the 9th international conference on Service-Oriented Computing*, 2011.

[4] Bruno Wassermann, Wolfgang Emmerich. Monere: Monitoring of Service Compositions for Failure Diagnosis. In *Proceedings of the 9th international conference on Service-Oriented Computing*, 2011.

[5] Adrian Mos, Carlos Pedrinaci, Guillermo Alvaro Rey, Jose Manuel Gomez, Dong Liu, Guillaume Vaudaux-Ruth, Samuel Quaireau. Multi-Level Monitoring and Analysis of Web-Scale Service Based Applications. In *ICSOC/ServiceWave Workshops*, 2009.

[6] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th international conference on World Wide Web*, 2008.

[7] R. Popescu, A. Staikopoulos, P. Liu, A. Brogi, and S. Clarke. Taxonomy-Driven Adaptation of Multi-layer Applications Using Templates. In *Proceedings of the 2010 IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, 2010.

[8] Philipp Leitner, Waldemar Hummer and Schahram Dustdar. Cost-Based Optimization of Service Compositions. In *IEEE Transactions on Services Computing*, 2011.

[9] R. Kazhamiakin, B. Wetzstein, D. Karastoyanova, M. Pistore, and F. Leymann. Adaptation of Service-Based Applications Based on Process Quality Factor Analysis. In *ICSOC/ServiceWave Workshops*, 2010.

[10] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, Frank Leymann. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference*, 2009.

[11] Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M.C.: Monitoring Dependencies for SLAs: The MoDe4SLA Approach. In *Proceedings of the 2008 IEEE International Conference on Services Computing*, 2008.

[12] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Runtime monitoring of instances and classes of web service compositions. In *Proceedings of the 2006 IEEE International Conference on Web Services*, 2006.

[13] E. Gjorven, R. Rouvoy, and F. Eliassen. Cross-layer self-adaptation of service-oriented architectures. In *MW4SOC'08*, 2008.