Incorporating Architecture-Based Self-Adaptation into an Adaptive Industrial Software System

Javier Cámara[1], Pedro Correia[2], Rogério de Lemos[3], David Garlan[1], Pedro Gomes[4], Bradley Schmerl[1], Rafael Ventura[2]

Abstract

Complex software-intensive systems are increasingly relied upon for all kinds of activities in society, leading to the requirement that these systems should be resilient to changes that may occur to the system, its environment, or its goals. Traditionally, resilience has been achieved either through: (i) low-level mechanisms embedded in the implementation (e.g., exception handling, timeouts, redundancies), which are unable to detect subtle but important anomalies (e.g., progressive performance degradation); or (ii) human oversight, which is costly and unreliable. Architecture-based self-adaptation (ABSA) is regarded as a promising approach to improve the resilience and reduce the development/operation costs of such systems. Although researchers have illustrated the benefits of ABSA through a number of small-scale case studies, it remains to be seen whether ABSA is truly effective in handling changes at run-time in industrial-scale systems. In this paper, we report on our experience applying an ABSA framework (Rainbow) to a large-scale commercial software system, called Data Acquisition and Control Service (DCAS), which is used to monitor and manage highly populated networks of devices in renewable energy production plants. In

Email addresses: jcmoreno@cs.cmu.edu (Javier Cámara), pcorreia@dei.uc.pt (Pedro Correia), r.delemos@kent.ac.uk (Rogério de Lemos), garlan@cs.cmu.edu (David Garlan), pgomes@criticalsoftware.com (Pedro Gomes), schmerl@cs.cmu.edu (Bradley Schmerl), ventura@dei.uc.pt (Rafael Ventura)

[1]Institute for Software Research, Carnegie Mellon University, USA.

[2]Department of Informatics Engineering, University of Coimbra, Portugal.

[3]School of Computing, University of Kent, UK and CISUC, University of Coimbra, Portugal.

[4]Critical Software, Portugal.

the approach followed, we have replaced some of the existing adaptive mechanisms embedded in DCAS by those advocated by ABSA proponents. This has allowed us to assess the development costs associated with the reengineering of adaptive mechanisms when using an ABSA solution, and to make effective comparisons, in terms of operational performance, between a baseline industrial system and one that uses ABSA. Our results show that using the ABSA concepts as embodied in Rainbow enabled an independent team of developers to: (i) effectively implement the adaptation behavior required from such industrial systems; and (ii) obtain important benefits in terms of maintainability and extensibility of adaptation mechanisms.

Keywords: Architecture-based self-adaptation, Evolution, Middleware, Rainbow

1. Introduction

The increasing reliance on software systems to carry out virtually all activities in society has led to the requirement that these systems be resilient in face of internal faults, changing resources, varying loads, and even changing usage requirements. Traditionally such system resilience has been achieved in two ways: (i) through low-level mechanisms embedded in the system implementation, such as, exception handling, timeouts and redundancies; or (ii) through the use of human oversight. Neither of these is adequate for certain systems. Embedded mechanisms lack flexibility and are typically unable to deal with subtle but important kinds of anomaly (e.g., progressive performance degradation), whereas human oversight is both costly and unreliable.

To address these deficiencies over the past decade there has been considerable attention given to new paradigms for improving system resilience through the use of autonomic, or self-adaptive, techniques [1]. One of the more promising approaches within the field of autonomic computing is architecture-based self-adaptation (ABSA) [2, 3]. ABSA adopts a feedback control systems point of view, centralizing problem detection and automated repair in a supervisory

control layer that monitors and adapts a system at run-time (See figure 6). Decision making (both for problem detection and for repair) by the control layer is aided by the use of architectural models of the system and its environment.

Proponents of ABSA have argued that this scheme has a number of inherent advantages. First, it is automatic and improves reliability while reducing the costs with respect to human oversight. Second, it decouples the control and adaptation logic from the system implementation. This allows one to more easily modify and reuse resilience-improving mechanisms across systems since such mechanisms are localized and explicit. Third, the use of architectural descriptions provides access to systemic information that can be used to perform sophisticated analysis and detection of subtle anomalies, such as, transient server failures or progressive performance degradation. Moreover, developers can add self-adaptation to legacy systems for which the code may not be available.

Researchers have illustrated these benefits through a number of case studies [4, 5, 6] that typically employ small-scale examples and prototypes that are intended to be representative of larger industrial systems. However, it remains to be seen whether ABSA can be truly effective in a large-scale industrial software system that has significant constraints on resources, timing, and commitments to legacy implementations. Is ABSA able to effectively implement the adaptation behavior required by such large-scale systems? Does ABSA provide clear benefits in terms of maintainability and extensibility of adaptation mechanisms?

In this paper, we shed light on these questions using a significant case study by incorporating an ABSA solution on a large-scale commercial software system. The system, called Data Acquisition and Control Service (DCAS), is used to monitor and manage highly populated networks of devices in renewable energy production plants, and has been in use commercially for over 5 years. DCAS has a number of properties that makes it an ideal example for investigating the benefits and drawbacks of ABSA. First, the system has strong requirements for robustness and optimal performance, making it a good candidate for autonomic adaptation. Second, to meet its requirements, DCAS currently deploys both forms of traditional resilience mechanisms – embedded fault recovery mech-

anisms implemented at the object class level for certain kinds of automated resource allocation, and human oversight for handling other kinds of resource management. Third, it is representative of a large class of distributed systems involving control of physical devices where high availability and efficient resource management are crucial to commercial success.

Our goal, by using this case study, is to investigate how well ABSA works by replacing some of the existing adaptive mechanisms in DCAS with those advocated by ABSA proponents.This approach had two important benefits. First, it allowed us to make effective comparisons between a baseline industrial system, typical of other systems that use traditional adaptation mechanisms, and one that uses an ABSA solution. Second, it allowed us to assess the costs associated with reengineering adaptive mechanisms to use the new paradigm. This latter point is important since most systems that are candidates for ABSA are likely to have legacy commitments to resilience-enhancing approaches. If the costs of switching over from those mechanisms to ABSA are too high, then we would be unlikely to see its widespread adoption, whatever its benefits might be.

For implementing DCAS following the ABSA principles, we used the Rainbow framework [8], a reusable infrastructure for the engineering of self-adaptive capabilities to monitor, decide, and act on situations that require system adaptation. To investigate the questions outlined above: (i) we removed built-in adaptation mechanisms in DCAS to obtain a version that could be integrated with Rainbow, thus allowing us, first, to replicate in a Rainbow-based version of the DCAS original adaptation behavior (embodied in a prototype – Rainbow-DCAS), and second, to assess the effort in doing this replacement; (ii) we then investigated ways in which Rainbow could be used to improve a perceived problem with the existing DCAS repairs – specifically, performance problems for situations in which devices are persistently slow in reporting data – and assessed the difficulty of modifying adaptation behavior using ABSA in Rainbow-DCAS; (iii) finally, we investigated the use of ABSA to automate some adaptations that in the original DCAS were handled by human oversight – specifically, a scale-out adaptation mechanism that allows the system to extended with new

processors.

In [7] we reported on an initial exploration of these topics. In this paper, we revise and extend our initial findings by reporting on how ABSA was used to automate adaptations originally handled by humans. Moreover, we include new sections on: (i) lessons learned (Section 6), which includes a comprehensive summary discussing the most important findings during our experience, (ii) related work (Section 8), which contrasts our experience with other similar experiences in applying self-adaptive frameworks (e.g., requirements-driven) to legacy systems, and (iii) threats to validity (Section 7).

The rest of this paper is organized as follows. Section 2 provides a general description of DCAS. Section 3 briefly describes the Rainbow platform for architecture-based self-adaptation. In Section 4, we describe the approach followed for the integration of Rainbow and DCAS, including re-implementation of existing adaptation mechanisms using ABSA, as well as, the extension of the Rainbow-based version of DCAS with an automatic scale-out mechanism. Section 5 provides an evaluation of different aspects regarding the process of integration and extension, describing the results obtained. Section 6 describes some lessons learnt. Section 8 overviews related work, whereas Section 7 discusses threats to validity. Section 9 concludes the paper and indicates directions for future work.

2. Data Acquisition and Control Service (DCAS)

The Data Acquisition and Control Service (DCAS) is a middleware from Critical Software [5] that provides a reusable infrastructure to manage monitoring and (non-automatic) control of highly populated networks of devices. In particular, the middleware is designed to be seamlessly integrated with Critical's Energy Management System (csEMS)[6], which is a platform that provides asset management support for power producing companies based on renewable

[5] http://www.criticalsoftware.com

[6] http://solutions.criticalsoftware.com/products_services/csems/

energy sources. The overall csEMS architecture aims at high scalability (with deployments that monitor networks of up to several thousand devices), flexibility and customization with management capabilities that enable the operation of control centers independently of the underlying application (e.g., wind, solar, etc). csEMS has been deployed across more than 15 different countries on 4 continents.

The basic building blocks in a DCAS-based system (Figure 1) are the following:

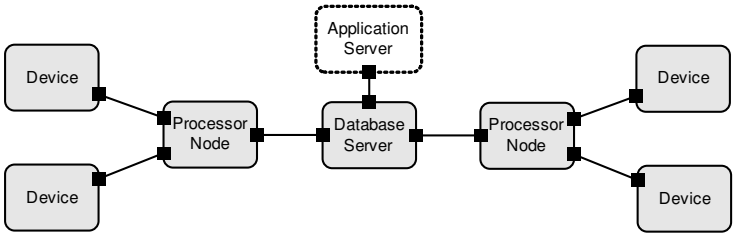

Figure 1: Architecture of a DCAS-based system

- **Devices** are equipped with one or more sensors to obtain data from the application domain (e.g., from wind towers, solar panels, etc.). Each of these sensors has an associated *data stream* from which data can be read. There may be different types of devices connected to the network, each type with its particular characteristics (e.g., protocols, type of data collected, etc.). Each type of device has an associated *device profile* that specifies e.g., the rate at which the device should be polled for data, and the expected value ranges for the data being collected.

- **Database server** stores all the information collected from devices, as well as, configuration data for the system (e.g., device profiles, etc.).

- **Processor nodes** pull data from the devices at a given rate (configured in the device profile), and dispatch this data to the database server. Each processor node executes an independent instance of the DCAS middleware,

which is implemented as a Windows service [7].

- **Application server** is connected to the database server to obtain data, which can be presented to the operators of the system or processed by application software. However, the DCAS middleware is application-agnostic, so the application server will not be discussed in the remainder of this document.

The typical DCAS-based system presents a blackboard architecture in which the database server acts as a centralized data manager into which processor nodes running the DCAS service write information collected from devices connected to the network.

The main objective of a DCAS-based system is to collect data from the connected devices at a rate as close as possible to the one configured in their device profiles, supporting as many connected devices as possible. Specifically, the primary goal in DCAS is providing service while maintaining acceptable levels of performance, measured in terms of processed data requests per second (rps) inserted in the database, while the secondary goal is optimizing the computational cost of operating the system, measured in number of active processes (called Data Requester Processor Pollers or DRPPs – introduced in Section 2.1.1) in the processor nodes. To achieve these quality goals, a DCAS-based system shall be able to scale-up, making use of the computational resources in the node(s) where the middleware is running, and scale-out, supporting the deployment of several instances of the middleware across different processor nodes within the same system to extend the number of connected devices.

2.1. DCAS Structure and Functionality

A different instance of the DCAS service runs in each of the processor nodes of a DCAS-based system. The main components of the service (shown in Figure 2) are the following:

[7] In the remainder of this paper, we refer to the DCAS middleware or DCAS service simply as *DCAS*, whereas the overall architecture is designated as a *DCAS-based system*.

- **Service Engine** orchestrates the flow of data among the different components of the DCAS service.
- **Polling Scheduler** triggers the process for performing requests to devices according to their scheduled rate of generation.
- **Data Requester** performs requests for data from devices.
- **Data Persister** stores the information obtained from devices into the database.
- **Alarmer** raises alarms if the data coming from the devices is unexpected (e.g., values out of the range defined in the device profile).
- **Data Stream Manager** updates the observed *device response time* (i.e., the elapsed time since a particular device is polled until it responds) associated with the different data streams of the devices. This information is internally used by the polling scheduler to adjust the generation rate of requests for data from devices.

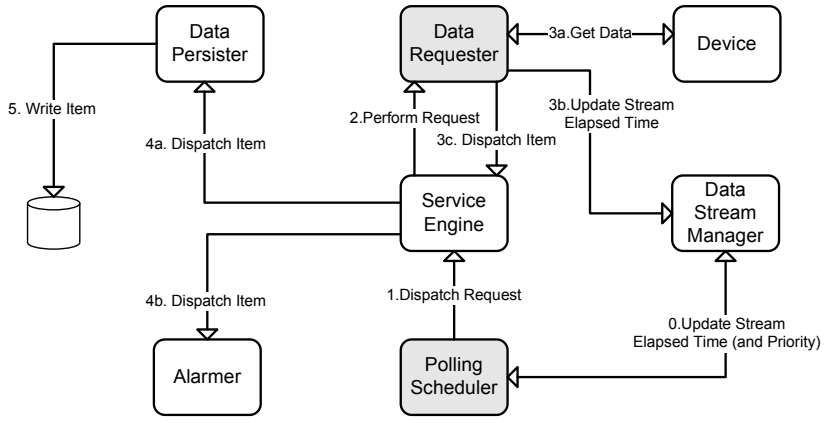

Figure 2: DCAS service operation

Figure 2 illustrates the operation of an instance of DCAS:

1. When the Polling Scheduler determines that the scheduled time for the execution of a request has arrived, the request is dispatched to the Service Engine.
2. The Service Engine forwards the request to the Data Requester.
3. The Data Requester:
   (a) Communicates with the device, retrieving the requested data and packing it into an item.
   (b) Updates the elapsed time information of the stream from which data has been read in step 3a (it is worth reminding that a device can have one or more data streams assigned from which data is read).
   (c) The item is dispatched by the Data Requester to the Service Engine.
4. The Service Engine:
   (a) Dispatches a copy of the item to the Data Persister.
   (b) Dispatches a copy of the item to the Alarmer.
5. The Data Persister writes the item to the database.

The value of the *device response time* of a data stream is continuously updated according to the time elapsed between a request for data made by the Data Requester, and the response received from the device.

Moreover, the Polling Scheduler constantly updates the priorities of the scheduled requests for data from devices according to the information updated in the Data Stream Manager (see step 0 in Figure 2). Changing the priority of devices with low responsiveness helps reducing the frequency with which these devices are polled, avoiding potential degradation of system performance. Further details about this issue can be found in Section 2.2.1.

Two important components in DCAS for achieving the desired quality goals (as expressed in Section 2) are the Data Requester and the Polling Scheduler, which are instrumental in the self-adaption mechanisms of DCAS. The following subsections describe these components in more detail.

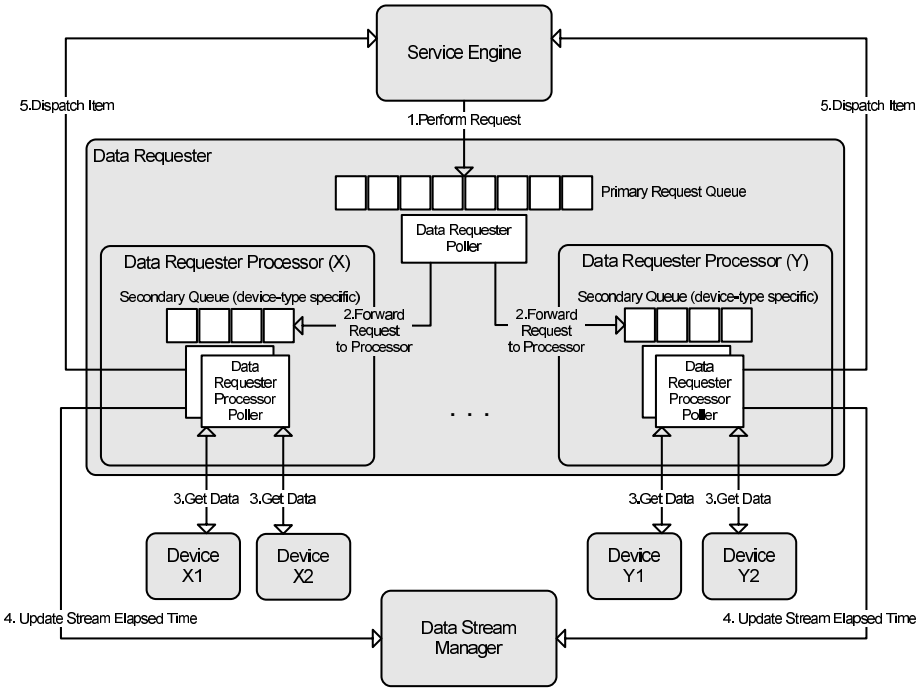

Figure 3: Data requester operation

2.1.1. The Data Requester

The Data Requester is in charge of retrieving data from connected devices. Internally, the Data Requester contains a collection of sub-components, called *Data Requester Processors* (DRPs), which perform requests on devices of a single type (Figure 3), and a *Primary Request Queue* from which requests are distributed to the different DRPs (based on the device type targeted by the request).

Each DRP contains an internal *Secondary Queue* in which device-type specific requests are enqueued. A collection of processes, called *Data Requester Processor Pollers* (DRPPs), dequeue requests from the Secondary Queue, and retrieve the data from the appropriate Device according to the specific contents of the request.

The sequence of events concerning the operation of the Data Requester is as

follows:

1. The Service Engine sends a request to the Data Requester, which is enqueued in the Primary Request Queue.
2. A process called *Data Requester Poller* retrieves a request from the Primary Request Queue, and forwards it to the appropriate DRP. The request is enqueued in the Secondary Queue of the DRP (if the queue is full, the request is discarded).
3. One of the DRPPs in the DRP dequeues the request from the Secondary Queue, and retrieves the data from the device. The communication between the DRPP and the device is synchronous, so the DRPP remains blocked until the device responds or a timeout expires. **This is the main bottleneck regarding performance of DCAS**.
4. When the data is received (or the timeout has expired), the priority associated with data stream from which data was read is updated on the Data Stream Manager.
5. If data has been received, the DRPP packs it into a data item and dispatches it to the Service Engine.

2.1.2. The Polling Scheduler

The Polling Scheduler is in charge of starting the process to request data from devices according to their scheduled time of execution. Internally, the scheduler contains a collection of request queues, each one specific to a particular *polling rate* of devices (or more concretely, data streams – Figure 4). Hence, all the requests to be performed on data streams with the same assigned polling rate are located within the same queue (independently of the type of the device to which they are associated). During the initialization of the service, the information regarding the polling rates of the different data streams is loaded from preconfigured values in the database, and then distributed across the different queues.

Each queue has an associated process called *Polling Scheduler Poller* (PSP), which processes requests in its queue in the following manner:

1. The PSP dequeues the first request of the queue.
2. The PSP clones the request retrieved from the queue and dispatches the clone to the service engine.
3. The PSP retrieves an updated value for the elapsed time of the data stream targeted by the request, and computes the corresponding priority based on the retrieved value.
4. The PSP re-inserts the original request into the queue in a new position that depends on the priority of the data stream. The higher the priority of the data stream, the closer to the first position of the queue the request will be inserted. This guarantees that requests that correspond to data streams with low priority (i.e., those associated with devices that take more time to respond) get processed less often, improving the overall performance of the service.

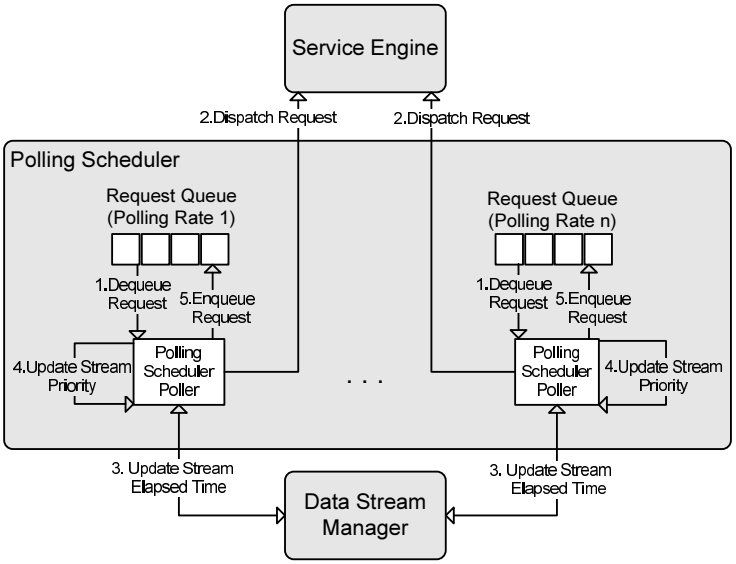

Figure 4: Polling scheduler operation

2.2. Adaptation Mechanisms

Having described the structure and functionality of DCAS, we now focus on the existing adaptation mechanisms of DCAS that are aimed at maintaining system performance under different loads. These adaptation mechanisms respond to failing devices, increased number of devices, and changing data rates.

DCAS implements two adaptation mechanisms to keep an acceptable level of performance while making an efficient use of computational resources: (i) rescheduling aims at avoiding performance degradation caused by devices that fail to respond in a timely manner when polled. It consists in decreasing the polling rate of the data streams associated with the failing devices, so that they are polled less often (thus reducing the average time that DRPPs remain blocked waiting for device data); and (ii) scale-up aims at improving performance by exploiting as much as possible CPU and memory in processor nodes by (de)activating DRPPs as required.

Scale-up and rescheduling run in two separate control loops embedded in different sub-components of the processor node (data requester and polling scheduler, respectively). Moreover, the C# adaptation logic that corresponds to these control loops is scattered across different parts of the code, and based on low-level information that indirectly indicates which aspect of the system needs to be improved. For instance, if the size of a data request queue associated with a particular data requester remains close to zero consistently, the scale-up adaptation mechanism considers this as an indicator of good performance, implying that there are active DRPPs which probably are not necessary and have to be deactivated. On the contrary, if the queue size increases consistently, scale-up tries to increase performance by activating new DRPPs.

A third adaptation mechanism for scaling-out is only available as a manual operation carried out by a human operator in DCAS, and deals with incorporating additional processor nodes running the DCAS service when the system cannot maintain an acceptable performance level when using all available resources in the set of active processor nodes.

2.2.1. Rescheduling

The rescheduling mechanism affects the Polling Scheduler, and is aimed at avoiding performance degradation of the system caused by devices that fail to respond in a timely manner (or do not respond at all) when polled. In a nutshell, the mechanism consists in decreasing the polling rate of the data streams associated with the failing devices, so that they are polled less often (thus reducing the amount of time that Data Requester Processor Pollers - or DRPPs - remain blocked waiting for device data).

To illustrate the rescheduling process, we introduce the following concepts:

- **Device Response Time (DRT)** is the time that it takes for a device to respond when polled by a DRPP.
- **Sample Rate (SR)** is the preconfigured value for the rate at which a device should be polled, and is fixed throughout the execution of DCAS.
- **Sample Rate Delay (SRD)** is an increment that can be added to the sample rate to poll devices less frequently. When the execution of the DCAS service starts, the SRD for all devices is equal to zero. Moreover, throughout the execution of DCAS, all devices responding in a timely manner should have an SRD equal to zero.
- **Effective Sample Rate (ESR)** or **polling rate** is the rate at which devices are effectively polled (ESR=SR+SRD).

Figure 5 illustrates the adaptation process followed for rescheduling. The process starts by checking if the device response time is above its effective sample rate:

- If the device response time is indeed above effective sample rate, the algorithm checks if the number of consecutive checks in which device response time for the device has been above effective sample rate (represented by counter CI) exceeds a threshold F (preconfigured value). If the threshold F has not been crossed, then counter CI is incremented. Otherwise,

counter CI is reset to zero and the sample rate delay for the device is incremented [8] (thus resulting also in the increment of the effective sample rate).

- If the device response time is below the effective sample rate, the algorithm checks if the number of consecutive checks in which device response time has been below sample rate (represented by counter CD) exceeds threshold F. If threshold F has not been crossed, counter CD is incremented. Otherwise, counter CD is reset to zero, and only if the sample rate delay is greater than zero, the sample rate delay is decremented.

2.2.2. Scale-up

The scale-up mechanism affects the behavior of the Data Requester, and is aimed at improving the performance of the system by exploiting as much as possible the resources (CPU and memory) of the processor node in which a DCAS service instance is running. This is achieved by adding or removing Data Requester Processor Pollers (DRPPs) in the secondary queues of Data Requester Processors (DRPs) as required. In concrete terms:

- If the size of the queue of the DRP remains close to zero, the system is running as expected, so nothing needs to be done. Indeed, if the queue size is consistently zero after a fixed number of consecutive checks, the scale-up mechanism considers that there are active DRPPs which probably are not necessary and starts removing them (one at a time).

- If the queue size of the DRP increases consistently during a fixed number of consecutive checks, scale-up tries to increase performance by adding new DRPPs.

It is worth observing that the addition of new DRPPs does not always result in a proportional increment in the number of requests processed per time unit

[8] The concrete details regarding the calculation to increment and decrement the sample rate delay are not discussed in this document.

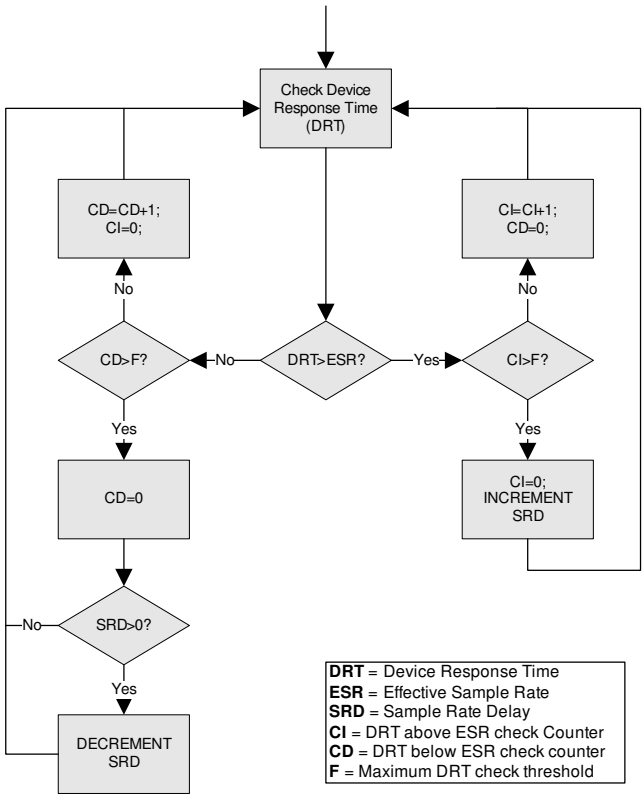

Figure 5: Flowchart of the rescheduling adaptation process

since the system is limited by the throughput of the devices being polled.

2.2.3. Scale-out

When devices are connected to the network at run-time, they can be dynamically incorporated to processor nodes in the DCAS-based system, which are activated progressively according to the demand determined by the system's workload and operating conditions.

Scale-out is supported in the original version of DCAS only as a manual process carried out by a human operator. This is a slow and demanding process in terms of effort required to carry out all the necessary operations to incorporate the new devices. When the DCAS-based system is unable to cope with the given configured data rates while using the maximum of available computational

resources within the current set of active processor nodes, it writes an entry to the log in the database to notify this event to a human operator. Then, a new instance of the DCAS service must be manually deployed, and devices re-attached across the different service instances (i.e., processor nodes), according to the particular situation. Each service instance is unaware of the existence of others, but there is a basic mechanism implemented so that each instance gets only the data streams it should process. Specifically, data stream entries in the database include a DCAS instance identifier that indicates which service instance should process its requests. Then, each DCAS instance reads data stream entries in the database upon initialization, ignoring the data streams in which the DCAS instance identifier does not match its own.

The process followed by a human operator to perform scale-out in a DCAS-based system consists of the following steps:

1. determine which (possibly new) devices need to be attached to a processor node,
2. decide which of those devices can be attached to a currently active processor node, and which must be attached to a new one,
3. insert the appropriate DCAS instance identifier in the data stream entries in the database,
4. restart active processor nodes that have been assigned with new devices, and
5. deploy and activate new processor node(s).

Once the scale-out process has been completed, newly deployed processor nodes are able to carry out scale-up and rescheduling as described in this section, according to changing conditions of the devices that they are attached to.

Note that the original developers of DCAS considered implementing an automated version of scale-out by embedding additional adaptation logic in the system, but discarded it because it was found to be rather challenging. In particular, the DCAS middleware is implemented as a Windows service originally designed to run in a single processor node. Hence, its code is oblivi-

ous to other instances of DCAS running alongside it in other processor nodes. As a result, the automation of a scale-out process in the original version of DCAS using traditional means would have implied major code refactoring, along with the inclusion of algorithms for distributed decision-making. In contrast, using ABSA as an alternative solution enables implementing scale-out on an external architecture-based adaptation layer able to observe and control the (de)activation of the different instances of the DCAS service running on different processor nodes (as shown in Section4). This allows centralized decision-making based on a global view of the state of the system and its environment provided by architecture models updated at run time.

3. The Rainbow Approach

Rainbow is a platform supporting architecture-based self-adaptation of software systems. It has the following distinct features: an explicit architecture model of the target system which is updated at run-time, a collection of adaptation strategies, and utility preferences to guide adaptation. Rainbow is aimed at reducing engineering effort by incorporating an explicit representation of adaptation knowledge [8]. Rainbow is comprised by a customizable framework that can be applied to a wide range of systems, and a language to represent human adaptation knowledge called *Stitch*.

The Rainbow framework (Figure 6) includes mechanisms for: monitoring a target system and its environment (using the observations for updating the architectural model of the target system), detecting opportunities for improving the target system's quality of services (QoS), and deciding the best course of adaptation based on the state of the target system [9]. The main components of

[9] Rainbow embodies what is known as the MAPE-K control loop proposed in IBM's Autonomic Computing initiative [9] to Monitor relevant variables in the system, Analyze whether adaptation is required, Plan the best course of action, and Execute adaptation, with the addition of a shared Knowledge base (founded on architecture models in the case of Rainbow), acting as a cornerstone of the process.

the framework are:

- **Architecture Evaluator** evaluates the model to ensure that the target system is operating within an acceptable range, as determined by a set of architectural constraints. If the evaluator determines that the system is not operating within the accepted range, it triggers adaptation.
- **Adaptation Manager** chooses a suitable adaptation strategy based on the current state of the target system (reflected in the architectural model).
- **Strategy Executor** executes the adaptation strategy chosen by the adaptation manager on the running target system via effectors.
- **Model Manager** updates the architecture model using the information observed in the running target system by the monitoring mechanisms in the translation infrastructure (probes and gauges).

Rainbow leverages the notion of *architectural style* [10] to exploit commonalities between systems, providing reusable infrastructures with explicit customization points that can be applied to a wide range of systems: (i) the architecture model of the target system customizes the model manager; (ii) architectural constraints related to adaptation goals customize the architecture evaluator; (iii) style operators and their mappings to target system effectors customize the strategy executor; and (iv) utility preferences and a collection of adaptation strategies with their associated cost-benefit impacts customize the adaptation manager.

Providing this substantial base of reusable infrastructure with explicit customization aims at reducing the cost of developing self-adaptation mechanisms.

Building upon the elements of the architectural style, Rainbow provides the *Stitch* [11] language to represent human adaptation knowledge using three high-level concepts:

- **Operator** is the most primitive unit of execution and represents a basic configuration command provided by the target system (corresponding to

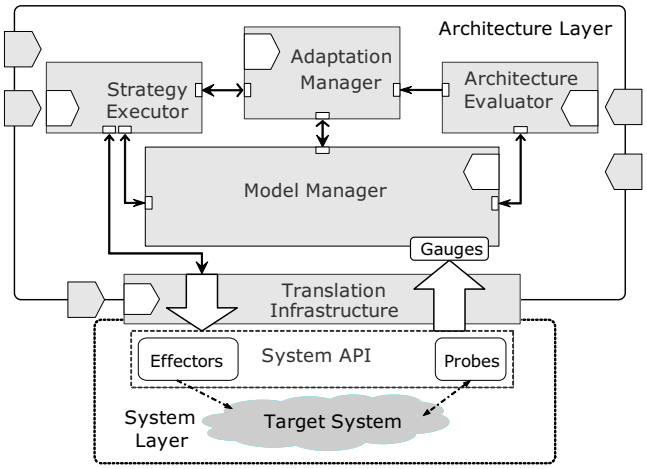

Figure 6: The Rainbow framework [8]

a system-level effector). They are defined in the architectural style of the system.

- **Tactic** is an abstraction that groups operators to form a single step of adaptation. Tactics are used as primitive actions, and have an associated cost/benefit impact on the different quality dimensions.

- **Strategy** encapsulates an adaptation process, where each step is the conditional execution of a tactic. Strategies are characterized in Stitch as a tree of condition-action-delay decision nodes, where delays correspond to a time-window for observing tactic effects. System feedback (through the dynamically-updated architectural model of the system) is used to determine the next action (*i.e.*, tactic) at every step during strategy execution.

In previous work, Rainbow has been applied to a wide range of systems, however the most widely reported has been the ZNN exemplar [6]. ZNN is an example web server that uses open source, off-the-shelf web servers, load balancers, and databases to implement a simple news site. We have applied adaptation in this context for quality attributes such as performance, cost, and information quality. In addition to this, Rainbow has been applied to manage

and repair the archiving pipeline of a web-based voice talk show and discussion group provider called TalkShoe. In this case, Rainbow would report problems with the production of the MP3 file recordings of the episode and report to a human operator [12]. In both of these cases, self-repair was added to these systems through Rainbow; there was no existing control loop that managed the kinds of adaptations that were implemented in Rainbow. The effort required for doing this for ZNN was 92 man-hours, and for TalkShoe, 34 man-hours. We will discuss these numbers in more detail in Section 9.

4. Implementing ABSA Mechanisms in DCAS

In this section, we describe the process followed for incorporating ABSA into DCAS, which consists in evolving DCAS to enable its integration with Rainbow, re-implementing DCAS scale-up and rescheduling mechanisms using Rainbow, and extending DCAS with an automatic scale-out mechanism.

4.1. Evolution of DCAS

Evolving DCAS to integrate it with Rainbow involves: (i) removing the logic that corresponds to the two control loops in which the scale-up and rescheduling mechanisms reside; and (ii) implementing the translation infrastructure between DCAS and Rainbow to enable their communication.

Previous case studies in which Rainbow has been applied [11, 6] describe systems that typically feature components that already include public interfaces to access their functionality (e.g., starting/stopping a web server, etc.). In contrast, implementing the translation infrastructure between DCAS and Rainbow required exposing part of DCAS internal functionality through a public interface, enabling communication with Rainbow for extracting system information through probes and effecting changes through system-level effectors. To achieve this, we implemented a lightweight server component embedded in DCAS that enables the exchange of information between a running instance of the DCAS service and Rainbow using TCP sockets. Figure 7 illustrates the translation

infrastructure used between Rainbow and DCAS. According to the diagram, probes and effectors in Rainbow act as clients of the TCP Server, which in turn acts as a mediator between the actual probes and effectors embedded in DCAS and the probe and effector clients in Rainbow. Below, we exemplify the flow of data related to probes and effectors, as depicted in Figure 7:

- Probes embedded in DCAS update the values of probed variables in the Local Data Store of the TCP Server, pushing updates whenever variables change (P1a and P2a). Then, when a Probe Client in Rainbow requests the value of a particular variable (P1b), it is directly served from the Local Data Store to the probe client (P2b). This approach was chosen due to the difficulty of invoking the necessary operations to retrieve data in DCAS from the TCP Server. Specifically, information such as queue sizes or number of active pollers in the Data Requester, as well as, information relative to device data streams could not be obtained from the TCP Server, so different parts of DCAS code were instrumented to extract this information and update it in the Local Data Store of the TCP Server.

- Effector clients in Rainbow send requests for command execution to the TCP Server (E1), which forwards them to the Effector embedded in DCAS

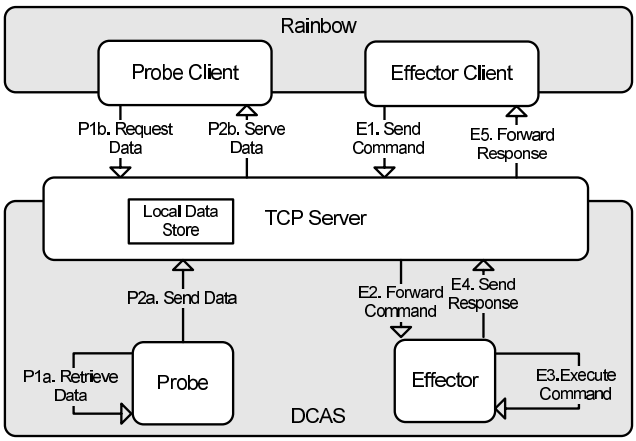

Figure 7: DCAS-Rainbow translation infrastructure

(E2). Next, the Effector executes the command (E3) and returns a response to the TCP Server, which states whether execution was successful (E4). Finally, the TCP Server forwards the response to the Effector Client in Rainbow (E5).

4.2. Re-implementation of Scale-up and Rescheduling

Implementing the ABSA version of the scale-up and rescheduling adaptation mechanisms is carried out by customizing the Rainbow framework by: (i) modeling the architecture of a DCAS-based system; (ii) scripting the adaptation logic (Stitch adaptation tactics and strategies); and (iii) implementing the client side of probes, gauges and effectors. Note that the implementation of the client side of probes, gauges, and effectors is trivial and therefore not discussed in the remainder of this paper.

Architecture Modeling. We can identify two quality objectives for the self-adaptation of a DCAS-based system: (A) performance, and (B) cost. Performance analysis is captured by the number of requests per second (rps) stored in the database server. Cost analysis identifies the number of active pollers in data requesters as the primary contributor to cost.

Table 1 displays the major elements of the blackboard architectural style for DCAS, including architectural types, properties, and operators. Properties sampleRateDelay, effectiveSampleRate, and deviceResponseTime in DeviceT can be mapped into the concepts of rescheduling adaptation, discussed in Section 2.2.1. Property numPollers in ProcessorNodeT corresponds to the number of active pollers (DRPPs) in the Data Requester of a processor node, whereas property queueSize corresponds to the size of its Primary Requester Queue, and queueStatus to the growth rate of the queue (negative values indicate that the number of elements in the queue is shrinking). Finally, property rps in DBServerT indicates the number of requests per second stored. The ProcessorNodeT.increasePollers() operator increases the capability of a processor node by activating a new Data Requester Processor Poller in its Data Requester, while decreasePollers() deactivates it. The

DeviceT.changeSampleRateDelay(sampleRateDelay : int) operator sets the effective sample rate of the data streams in a device by setting the value of its sample rate delay.

Type	**Property**	**Operator**
DeviceT	sampleRateDelay effectiveSampleRate deviceResponseTime **location**	changeSampleRateDelay(sampleRateDelay :int) **assignDeviceToPN(location :int)**
ProcessorNodeT	numPollers queueSize queueStatus	increasePollers() decreasePollers() **enablePN()** **disablePN()** **restartPN()**
DBServerT	rps **numDevices** **numUnassignedDevices** **numUnprocessedDevices**	

Table 1: DCAS architectural style elements. Elements in bold face correspond to the extended version of the DCAS architectural style for scale-out

Scripting Adaptation. Using the architectural operators defined in DCAS architectural style, we specified two pairs of tactics with opposing effects. One pair adds (i) or removes (ii) pollers, whereas the other pair increases (iii) or decreases (iv) the sample rate delay of the streams associated with a device. When performance is low, objective A - related to performance, suggests that the system should activate additional pollers (using tactic (i) above), if the processor node has not exhausted the resources assigned to DCAS (memory and CPU), or otherwise increase the sample rate delay of devices with higher response time using tactic (iii). When rps remains close to the top of its expected range, objective B - related to cost, suggests that the system should reduce cost

by deactivating pollers (using tactic (ii)) which may not be required to maintain an acceptable level of performance in the system.

Based on the tactics described above, we designed a baseline set of strategies for system adaptation to balance the different quality objectives in the system. This set of adaptation strategies is able to reproduce the original adaptation behavior of DCAS (as described in Section 5.2.1). Regarding scale-up adaptation, we define a pair of strategies to increment and decrement the number of DRPPs in processor nodes as needed (IncreasePerformance and ReduceCost, respectively. For rescheduling, we define another pair of strategies to increase, and reduce, the sample rate delay for devices which fail to respond in a timely manner (strategies increaseDelay and DecreaseDelay, respectively).

IncreasePerformance. Strategy IncreasePerformance first specifies its applicability condition, which is used by Rainbow's Adaptation Manager during strategy selection to determine whether the strategy should be considered for adaptation. In this particular case, the condition is defined using the conjunction of the predicates: (i) styleApplies (line 1), which checks whether the model defines the architectural types used in the strategy; (ii) rpsViolation (line 2), which determines if the system is experiencing low performance (rps below threshold MIN_RPS); and (iii) !maxLazyStreams, which holds only if the number of active pollers in the system is not greater than the number of data streams with low responsiveness. Moreover, the predicate qShrinking defines whether the growth rate of the queue in the processor node is negative.

In the body of the strategy, node t0 (line 6) executes tactic addPoller if the corresponding guard is satisfied (in this case qShrinking evaluating to false, i.e., the queue in the processor node is growing). To account for the delay in observing the outcome of the tactic's execution upon the system, t0 specifies a delay window of 5000 milliseconds [10] (end of line 6). After the end of t0's time window, the guard for nodes t0a (line 7) and t0b (line 8) are evaluated. If the guard

[10] Observation delays are determined experimentally in accordance with the adaptation logic implementing the original adaptation mechanisms in DCAS.

in t0a is satisfied, the queue is considered to be still growing, hence the tactic to add another poller is executed again. Otherwise, the guard in t0b is going to be satisfied (guards in t0a and t0b are mutually exclusive), and the strategy is going to end its execution returning a success status (indicated by keyword done, line 8) since the size of the queue in the processor node is already decreasing.

```
1  define boolean styleApplies = Model.hasType(M, "ProcessorNodeT");
2  define boolean rpsViolation = exists s : T.DBServerT in M.components |
       s.rps < M.MIN_RPS;
3  ...
4  strategy IncreasePerformance
5  [ styleApplies && rpsViolation && !maxLazyStreams ]{
6    t0:(!qShrinking)->addPoller()@[5000 /*ms*/]{
7      t0a:(!qShrinking)->addPoller()@[5000 /*ms*/]{
8      t0b: (qShrinking)->done;
9      }
10   }
11   t1: (qShrinking) -> done;
12 }
```

Listing 1: Stitch strategy to increase performance via activation of DRPPs.

ReduceCost. When DCAS detects small queue sizes (qViolation2) and the minimum level of pollers has not been reached (!minPollers), remove one poller. If queue sizes remain below the threshold after 3 seconds, remove another poller.

```
1  strategy ReduceCost
2  [ styleApplies && qViolation2 && !minPollers ]{
3    t0:(qViolation2)->removePoller()@[3000 /*ms*/]{
4      t0a:(qViolation2)->removePoller()@[3000 /*ms*/]{
5      t0b:(!qShrinking)->done;
6      }
7    t0c:(!qShrinking)->done;
8    }
9  }
```

Listing 2: Stitch strategy to scale-down by deactivating DRPPs.

IncreaseDelay/DecreaseDelay. Increase/decrease sample rate delay of all devices which exhibit response time above/below (tViolation/tViolation2) one step.

```
1  strategy IncreaseDelay [ styleApplies && tViolation]{
2    t0: (tViolation)->increaseSampleRateDelay(M.SRD_INCREMENT)@[5000 /*ms
         */]{
3      t1:(!tViolation)->done;
4    }
5  }
6  strategy DecreaseDelay [ styleApplies && tViolation2]{
7    t0: (tViolation2)->decreaseSampleRateDelay(M.SRD_INCREMENT)@[5000 /*
         ms*/]{
8      t1:(!tViolation2)->done;
9    }
10 }
```

Listing 3: Stitch strategies for rescheduling.

Although this baseline set of adaptation strategies was able to successfully replicate the adaptation behavior of DCAS, we discovered that we could do better since for some cases both in original DCAS and in Rainbow-DCAS using the baseline set of strategies, the adaptation behavior was not enough to recover system performance in a timely manner (please refer to Section 5.2.1 for details). Specifically, we modified IncreasePerformance to add pollers more aggressively by shortening the observation delay between checks in queue sizes, as well as, increasing the number of pollers that can be activated to a maximum that duplicates the number of unresponsive data streams. The improvement obtained by applying these modifications are described in Section 5.2.1.

4.3. Extension of Rainbow-DCAS with Automatic Scale-out.

Extending Rainbow-DCAS with an automatic scale-out mechanism involves: (i) extending the existing translation infrastructure; and (ii) extending the existing customized elements of the Rainbow framework introduced in Section 4.2.

4.3.1. Extending the translation infrastructure.

Extending the existing translation infrastructure is a trivial operation in which the repertoire of messages supported by the TCP server, described in Section 4.1, is extended with requests for data from the new probes and effectors required by the new scale-out mechanism. These are in turn determined by

the extensions to the architectural style and adaptation logic described in the remainder of this section.

4.3.2. Extending the customization of Rainbow.

In order to enable dynamic management of new processor nodes in Rainbow-DCAS, we extended the existing architectural style, as well as, the adaptation logic in Rainbow-DCAS with the corresponding tactics and adaptation strategies.

Architecture Modeling. Table 1 shows the major elements of the DCAS architectural style with additional properties and operators required for scale-out adaptation. Specifically, property location in DeviceT corresponds to the identifier of the processor node to which the device is assigned. Properties numDevices, numUnassignedDevices, and numUnprocessedDevices in DBServerT correspond, respectively, to the overall number of devices in the network, the number of devices which are not currently assigned to a processor node, and the number of devices assigned to a processor node, but whose requests are not currently being processed.

The DeviceT.assignDeviceToPN(location:int) operator assigns a device to a given processor node, whereas operators TProcessorNode.enablePN()/disablePN()/restartPN() activate, deactivate, or restart a processor node, respectively.

Scripting Adaptation. Using the architecture operators described above, we specified tactics that enable Rainbow to assign devices in the network to a processor node, as well as to (de)activate or restart processor nodes as required, according to changes in the number of devices present in the network.

Basing upon these tactics, we implemented a set of strategies that automate scale-out in a Rainbow-DCAS. This set of adaptation strategies reproduces the process followed by a human operator when new devices are incorporated to the network:

AssignDevices. When new devices (i.e., those that are not assigned to a processor node) are detected in the network (unassignedDevices), the strategy assigns the

devices to an already active processor node, if the set of active processor nodes has not reached its maximum capacity (!maxAssignedDevices) by using the tactic assignDStoPN(). The tactic stops assigning devices to the current processor node when the latter reaches the maximum capacity. If the strategy detects that there are still unassigned devices after the first execution of the tactic (node t0a), the tactic is repeated, assigning the remaining devices to the next processor node. This process is repeated until all devices are assigned to a processor node, or all the processor nodes have the maximum number of devices they support assigned.

```
1
2 strategy AssignDevices
3 [ styleApplies && unassignedDevices ]{
4   t0:(unassignedDevices && !maxAssignedDevices)->assignDStoPN()@[3000
        /*ms*/]{
5     t0a:(unassignedDevices)->do[TOTAL_PN]t0;
6     t0b:(!unassignedDevices)->done;
7   }
8 }
```

Listing 4: Stitch strategy to assign devices to processor nodes during scale-out.

ScaleOut. Once all unassigned devices detected in the network have been assigned to their respective processor nodes by the strategy AssignDevices, the ScaleOut strategy kicks in to activate the additional processor nodes required to process the requests coming from the new devices. In particular, the strategy is executed when:

- the maximum capacity of the set of currently active processor nodes has reached its limit (maxAssignedDevices);
- there are devices already assigned to a processor node whose requests are not being processed because the corresponding processor node is not active yet (unprocessedDevices); and
- the number of active processor nodes has not reached the number of available processor nodes in the system.

```
1
2 strategy ScaleOut
3 [styleApplies && maxAssignedDevices && unprocessedDevices &&
      numActivePN<numAllocatedPN]{
4   t0:(maxAssignedDevices && unprocessedDevices && numActivePN<
        numAvailablePN)->activatePN()@[20000/*ms*/]{
5     t0a:(!maxAssignedDevices)->done;
6   }
7 }
```

Listing 5: Stitch strategy to activate processor nodes during scale-out.

So far, we have described the process followed to integrate DCAS with Rainbow, re-implement existing adaptation mechanisms, and extend the adaptation capabilities of the system using Rainbow. This process produced Rainbow-DCAS, a prototype of DCAS which embodies the principles of ABSA. In the next section, we describe how we have used Rainbow-DCAS and our experience during its development as a vehicle to evaluate the effort required to implement an ABSA solution, and contrast its performance with that of the original system.

5. Evaluation

In this section, we evaluate our modifications to DCAS in two dimensions. Firstly, we report on the implementation effort involved when integrating Rainbow and DCAS, including the re-implementation and extension of adaptation mechanisms by using ABSA. Secondly, we evaluate the performance of DCAS when incorporating an ABSA-based solution.

5.1. Implementation Effort

We report here on the implementation effort involved in: (i) evolving DCAS by removing its existing, hardcoded self-adaptation mechanisms, and implementing the translation infrastructure to communicate with Rainbow; (ii) customizing Rainbow to re-implement scale-up and rescheduling adaptation mechanisms using ABSA; (iii) improving the scale-up and rescheduling adaptation

strategies to make the adaptations more responsive to problems; and (iv) implementing automatic scale-out.

The aforementioned tasks were carried out by a team not previously acquainted with DCAS or Rainbow [11].

5.1.1. Evolution of DCAS.

The overall time spent in tailoring DCAS for Rainbow was 145 hours (approximately 3 2/3 work weeks). As can be observed in Table 2, although the implementation of the bulk of the translation infrastructure (TCP Server) did not require much effort, about 55% of the overall time was spent in developing probes and effectors. This stems from the fact that most of the time needed for developing probes and effectors was devoted to code refactoring and instrumentation. This were required to enabling access to the classes and methods needed to obtain probe information, and effect changes in the system (please refer to Section 4.1).

Task	**Time (in hours)**	**%**
Implementing TCP server	15	10.3
Identifying and removing built-in adaptation	40	27.5
Implementing probes	45	31
Implementing effectors	35	24.1
Miscellaneous configurations	10	6.8
Total	**145**	**100**

Table 2: DCAS evolution effort

[11] Although some of the authors of Rainbow and DCAS participated in the study described in this paper, all tasks regarding the implementation of Rainbow-DCAS were carried out by an independent team of three developers at the University of Coimbra without any prior experience with Rainbow or DCAS.

5.1.2. Re-implementing Scale-up and Rescheduling in Rainbow.

We tracked the activities carried out during the customization of Rainbow. The overall effort invested in customization including the modeling of the system's architecture, scripting of the adaptation (developing tactics and strategies in Stitch), and development and testing of the translation infrastructure, including probes, gauges, and effectors amounted to 91 hours (approximately 2 1/3 work weeks).

Task	**Time (in hours)**	**%**
Architecture modeling	20	21.9
Implementing client probes and gauges	22	24.1
Implementing client effectors	12	13.1
Scripting adaptation (tactics and strategies)	35	38.4
Miscelaneous configurations	2	2.1
Total	**91**	**100**

Table 3: Rainbow customization effort for DCAS

Table 3 details the effort devoted to customization. It is worth observing that more than half of the effort (59.1 %) was devoted to the development of the translation infrastructure (probes, gauges, effectors) and the architecture model, whereas the time devoted to scripting adaptation was 38.4%.

5.1.3. Evolution of Scale-up and Rescheduling in Rainbow-DCAS.

Once we had a first version of Rainbow-DCAS, which included a baseline set of adaptation strategies that replicated DCAS adaptation behavior, we evolved the set of adaptation strategies to improve the performance of Rainbow-DCAS. Specifically, in the original DCAS adaptations the system was slow to recover if devices were persistently slow in reporting data.

Table 4 shows the size of the alternative adaptation mechanisms implemented in Rainbow-DCAS and DCAS, as well as, the number of classes involved in each of the adaptation mechanisms in the latter. The data shows that

Item	# SLOC	# Classes
Rainbow-DCAS tactics	88	-
Rainbow-DCAS strategies	57	-
DCAS scale-up	93	2
DCAS rescheduling	115	6

Table 4: Size/Scattering of DCAS adaptation mechanisms

although there is not a substantial difference between the number of lines of source code in Rainbow-DCAS and DCAS (145 lines of Stitch vs. 208 lines of C#), the implementation of adaptation mechanisms in DCAS is scattered across two different sets of classes, hampering the evolution of adaptation mechanisms. However, in Rainbow-DCAS the specification of adaptation is centralized, easing the modification of adaptation behavior. Indeed, we found that **the evolution of the baseline set of adaptation strategies demanded time of an order of magnitude of just minutes, not hours. This contrasts with the effort required to evolve the original adaptation mechanisms in DCAS, which typically demands about 2 man-days to tune** when the middleware is deployed in a new location. Moreover, modifying adaptation mechanisms in Rainbow-DCAS requires just restarting the system after modifying scripted strategies in Stitch, whereas in DCAS the system has to be recompiled and redeployed (two processes that demand additional infrastructure and time).

5.1.4. Implementing Automatic Scale-out.

The time employed in implementing scale-out adaptation in Rainbow-DCAS was approximately 57 hours, out of which the majority (50 hours) were spent in customizing Rainbow, and the rest were used to prepare DCAS for incorporating the new version of scale-out.

Rainbow Customization. Table 5 details the effort required for the different tasks involved in Rainbow customization. While little effort was required to modify the architecture model, the major investments correspond to developing

the probes and gauges needed to monitor the information required by the new properties in the architecture model, as well as, to scripting adaptation tactics and strategies.

Task	**Time (in hours)**	**%**
Architecture modeling	2	4
Implementing probes and gauges	15	30
Implementing client effectors	8	16
Scripting adaptation (tactics and strategies)	15	30
Miscelaneous configurations	10	20
Total	**50**	**100**

Table 5: Rainbow customization effort for DCAS scale-out

Evolution of DCAS. Table 6 shows that little effort was needed to prepare DCAS for incorporating the new version of scale-out since most of the required infrastructure (such as the TCP server, or several effectors that the new adaptation mechanism reuses) was already in place.

Task	**Time (in hours)**	**%**
Modification of TCP server	2	28.5
Implementing effectors	3	42.8
Miscellaneous configurations	2	28.5
Total	**7**	**100**

Table 6: DCAS evolution effort for scale-out

It is worth observing that the overall time required to incorporate scale-out adaptation, which is a far more elaborated mechanism than scale-up and rescheduling (i.e., requiring several intermediate steps and system-wide changes to the architecture), represents only about 20% of the overall effort initially required to incorporate the architecture-based version of scale-up and rescheduling. These numbers indicate that while there is indeed a significant upfront

investment required to incorporate ABSA, this effort pays off not only while evolving existing architecture-based adaptation mechanisms (as shown in Section 5.1), but also when developing new ones, which can easily take advantage of the existing infrastructure.

5.2. Experimental Evaluation

The aim of our experiments is assessing the applicability of architecture-based self-adaptation (ABSA) mechanisms in the context of an application-agnostic middleware, comparing their performance and efficiency with those achieved by DCAS built-in adaptation mechanisms. Specifically, we evaluate the performance of the adaptations to: (i) verify that replicating the adaptations (i.e., scale-up and rescheduling) in DCAS with Rainbow provides similar adaptation performance; (ii) measure the adaptation improvement in Rainbow obtained by evolving scale-up and rescheduling adaptation strategies; and (iii) validate the behavior of automatic scale-out.

5.2.1. Scale-up and Rescheduling

Experimental Setup. For our experimental setup, we deployed both versions of DCAS across three different machines (Figure 8): dcas-db acts as the backend database running on Oracle 10.2.0, dcas-main acts as a processor node, running DCAS, and (dcas-devs) is used to simulate the response of network devices from which DCAS retrieves information (device response simulation is implemented as a simple Web service whose response time can be set in a configuration file). In the case of Rainbow-DCAS (Figure 8, left), Rainbow's master is deployed in a separate machine (dcas-master). All machines run on Windows XP Pro SP3 (DCAS is deployed as a Windows service), and an Intel core i3 processor, with 1GB of RAM.

Our experiments include 100 data streams with a sample rate of 1 second. The duration is 40 minutes (2400s), and the pattern followed is:

1. 0-600s: normal activity to let the system achieve a steady state;

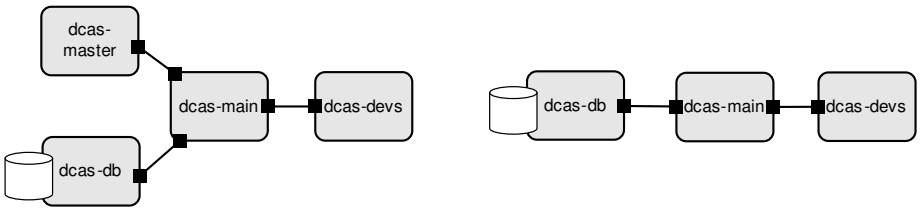

Figure 8: Experimental setup: Rainbow-DCAS (left) and DCAS (right)

2. 600-1200s: disturbance period, during which we induce low responsiveness in data streams (adding a 2-second delay in the response time of 25% of the data streams); and
3. 1200-2400s: system keeps on running with normal activity until the end of the experiment.

To assess the effectiveness and flexibility of the Rainbow approach in the context of DCAS, we carried out two sets of experiments: (i) using a baseline set of adaptation strategies to show that the adaptation behavior of DCAS can be replicated using Rainbow; and (ii) using an evolved set of adaptation strategies to improve adaptation behavior.

Replicating DCAS Adaptation Behavior. Figure 9 depicts the performance (top) and cost (bottom) shown by the different versions of DCAS during the execution of our experiments. Comparing the performance of DCAS with Rainbow-DCAS baseline, we can observe that after the disturbance starts, performance drops in both cases and stays in low levels until the disturbance is removed. Both implementations show a spike in performance when the disturbance is removed, due to the number of accumulated requests in the secondary queues of Data Requester Processors. The removal of the delay in data streams, along with the high number of available active pollers to process the requests in the queues at that point (t=1200s - Figure 9, bottom), causes the sudden increase in performance, which goes back to expected levels almost immediately when queue sizes are reduced back to normal levels. Moreover, the activation of pollers in DCAS presents a slight overshoot compared to the Rainbow-DCAS

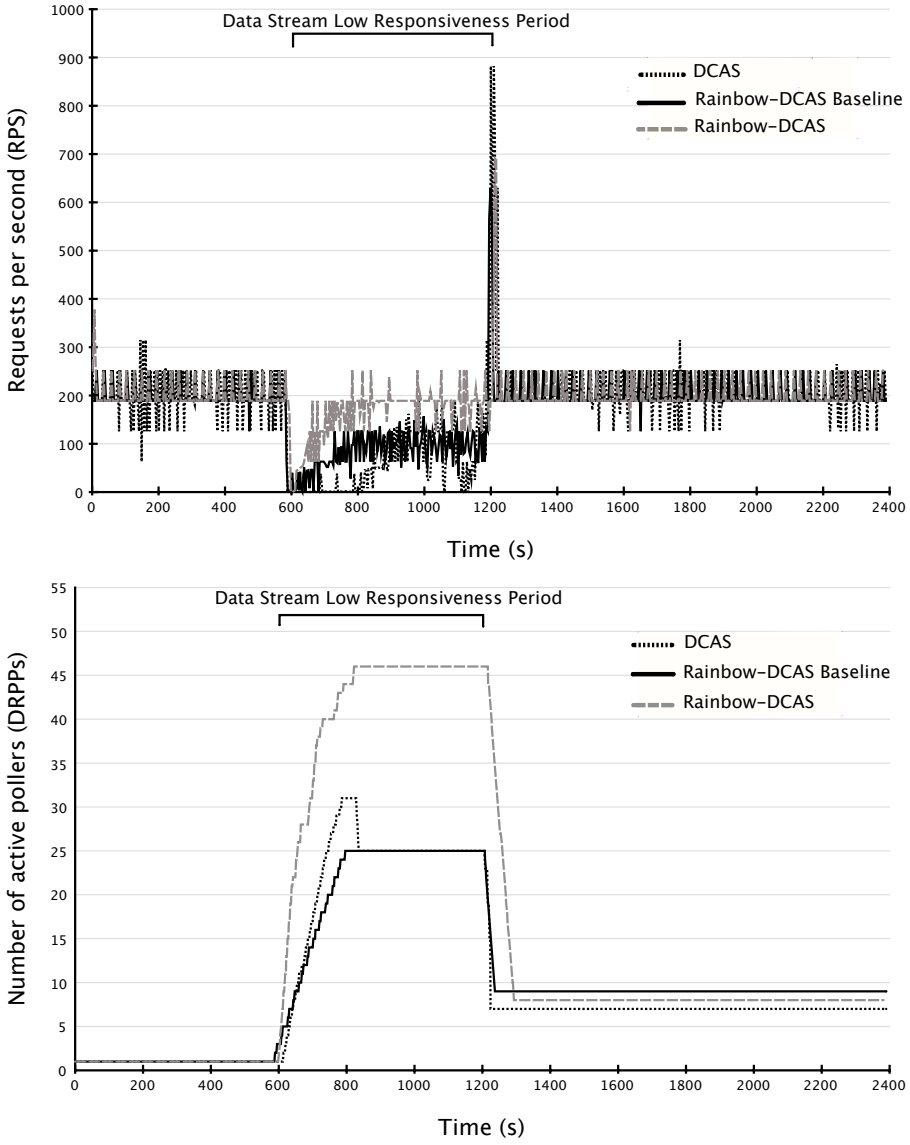

Figure 9: Performance (top) and number of active pollers (bottom)

baseline. This is explained by the longer time periods between the consecutive queue size checks required to activate pollers (as described in Section 2.2.2), compared to the higher frequency of probe updates and shorter adaptation cycle time in Rainbow.

Improving Adaptation Behavior. Once we reproduced the adaptation behavior of DCAS, we evolved the baseline set of adaptation strategies to improve performance during the disturbance period. **Results show that Rainbow-DCAS is able to recover faster than DCAS.** Specifically, when the disturbance period starts, the performance of both DCAS and Rainbow-DCAS degrades initially, going from values in the expected range (200-250 rps) to values in the range 0-50. However, by t=800s, performance in Rainbow-DCAS has been restored to normal levels. In contrast, DCAS does not recover throughout the whole disturbance period, only going back to normal once the disturbance is removed by time t=1200s. Moreover, Rainbow-DCAS is faster in reacting to the disturbance since we modified the adaptation strategies to activate pollers more aggressively when low responsiveness appears in data streams. This comes at the cost of more active pollers, but it is an acceptable solution given that the main priority of the system is performance.

5.2.2. Scale-out

Experimental Setup. To assess the behavior of Rainbow-DCAS during scale-out adaptation, we carried out a set of experiments for which we extended our earlier experimental setup, deploying Rainbow-DCAS across five machines (Figure 10): dcas-db acts as the backend database running on Oracle 10.2.0, dcas-pn0 and dcas-pn1 act as processor nodes running DCAS, dcas-devs is used to simulate the response of network devices from which DCAS retrieves information, and dcas-master hosts Rainbow's master controller. All machines run on Windows XP Pro SP3, and are equipped with an Intel core i3 processor and 1GB of RAM.

Our experiments include two sets of 50 data streams (namely, ds0 and ds1), with a sample rate of 1 second each. The system starts execution with only one set of 50 data streams (ds0), and one active processor node (dcas-pn0) that

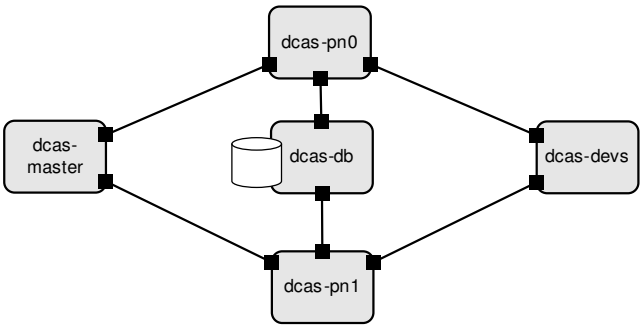

Figure 10: Experimental setup for scale-out experiments in Rainbow-DCAS

processes their data. The duration of each experiment was 40 minutes (2400s), and the pattern followed was:

1. 0-200s: normal activity to let the system achieve a steady state;
2. 200-800s: disturbance period, during which we induce low responsiveness in data streams (adding a 2-second delay in the response time of 50% of the data streams in `ds0`);
3. 800s: activation of the second set of data streams (`ds1`). Total number of data streams is now 100. This triggers scale-out adaptation, which activates `dcas-pn1`, and assigns the processing of data streams in `ds1` to it;
4. 800-1200s: the system runs with both processor nodes active. 50% of data streams in `ds0` are still under induced low responsiveness;
5. 1200-1600s: we induce low responsiveness in 50% of data streams in `ds1`. Both processor nodes are now processing data streams with low responsiveness;
6. 1600-2000s: low responsiveness in `ds0` data streams is eliminated;
7. 2000-2400s: low responsiveness in `ds1` data streams is eliminated. The system keeps on running with normal activity until the end of the experiment.

It is worth observing that during these experiments, scale-out adaptation runs along with both scale-up and rescheduling mechanisms on each of the individual processor nodes.

Extending Adaptation Behavior with Scale-out. Figure 11 shows both the performance (top) and cost (bottom) of Rainbow-DCAS during a scale-out experiment. If we focus on performance, we can observe a drop in rps right after the induction of low responsiveness in ds0 ($t = 200s$). However, it can be observed how performance quickly recovers due to the immediate reaction of scale-up adaptation (noticeable by the sudden increment in active dcas-pn0 pollers in the bottom part of Figure 11). After this, the system keeps on running with the expected performance level for 50 data streams and one active processor node, until the data streams in ds1 are activated ($t = 800s$). At that point, the performance level increases since the scale-out mechanism kicks in, activating processor node dcas-pn1, and resulting in an increased number of processed data requests per second inserted in the database (now coming from both processor nodes).

At this point, we can observe how dcas-pn0 needs to keep 50 active pollers to maintain an acceptable level of performance, where as dcas-pn1 does still operate with the minimum number of active pollers since data streams in ds1 are responding in a timely manner. However, when we induce low responsiveness in ds1 data streams ($t = 1200s$), we can observe how the performance level drops, and the number of active pollers in dcas-pn1 starts to increase.

Finally, the performance level rises again to the expected levels for two processor nodes and 100 data streams. This level of performance is preserved until the end of the execution of the experiment, even when the number of active pollers in both processor nodes has dropped to a minimum level.

6. Lessons Learned

During our experience in developing Rainbow-DCAS, we observed that architecture-based self-adaptation (ABSA) can successfully replicate the adaptation behavior required from an industrial-class software-based system such as DCAS. Moreover, results show a substantial reduction of the effort required to evolve existing architecture-based adaptation mechanisms, and develop new ones once the nec-

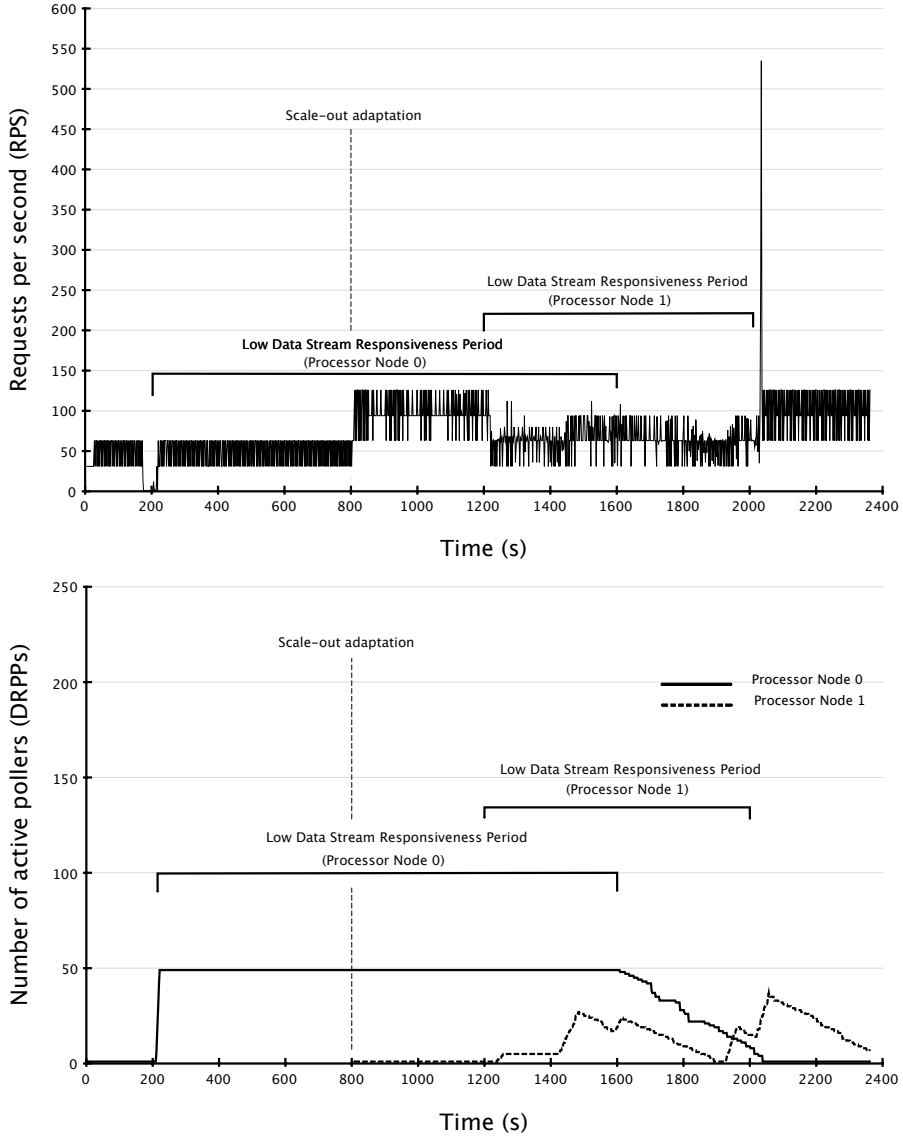

Figure 11: Performance (top) and number of active pollers (bottom)

essary infrastructure is put in place.

To validate our findings, we have compared the customization effort of Rainbow, while integrating it to DCAS, with other two applications of Rainbow. The results of this comparison are shown in Table 7. Results show that **the effort**

Task	**DCAS**	**ZNN**[6]	**TalkShoe**[12]
Architecture modeling	20	13	6
Implementing probes and gauges	22	49	8
Implementing effectors	12	7	5
Scripting adaptation	35	21	8
Miscelaneous configurations	2	2	8
Total	**91**	**92**	**34**

Table 7: Rainbow customization effort

required to implement Rainbow-DCAS is consistent with the numbers reported in previous experiences with Rainbow, with an average time spent in each one of the tasks that ranges between one and two days. However, our DCAS prototype was developed independently, only with scarce consulting provided by the original developers of Rainbow and Critical Software, so development time was partially spent in getting acquainted with Rainbow and DCAS. Hence, we assume that subsequent developments using Rainbow would require less effort.

Beyond the primary observations noted above, we have gained some insight into the development of adaptation mechanisms for this kind of industrial software-intensive system, which is the following.

1. **Development paradigm matters.** In the original DCAS, each adaptation mechanism resides within its own independent control loop in different sub-components of the processor node (i.e., scale-up and rescheduling reside in the data requester and polling scheduler, respectively). This would appear to be a consequence of adhering to a strict object-oriented programming paradigm when developing the embedded code-based adaptation mechanisms in the original DCAS. While enforcing encapsulation and information hiding can lead to good modularization, it also constrains the scope of embedded adaptation mechanisms, restricting their access to information (e.g., for anomaly detection) and actuation to their local scope,

as well as hampering coordinated adaptation across multiple system components. In contrast, the two adaptation mechanisms in Rainbow-DCAS (implemented as the two set of Stitch strategies described in Section 4.2) reside within the same control loop in the external control layer that decides which one should be used, depending on the particular situation, in a coordinated manner.

2. **Explicit information improves adaptation behavior.** As a consequence of the limited scope of embedded adaptation mechanisms, the original DCAS can only use low-level information that indirectly indicates the system's performance. In particular, data request queue growth rates are used to assess whether more pollers should be added or removed from a given processor node, in contrast with using explicit information about the system's performance (i.e., rps). While this solution works well in this particular case, the system is adapting to maintain a growth rate of 0 (i.e., a constant size) in request queues, rather than to meet the actual goals of the system. In general, adapting to control a variable that may not always be correlated with system goals may result in the system failing to adapt in some cases in which adaptation is required (and conversely, to adapt in situations that do not require it). In contrast, Rainbow-DCAS has access to systemic information about whether performance goals are being met. The ability to factor in high level information, like performance, into the decision-making process is possible because of architectural descriptions. These descriptions allow systematic reasoning in terms of the actual goals of the system, rather than ad hoc decisions based on low-level, indirect indicators. Further details about how the use of architecture models have a positive impact on adaptation in DCAS can be found in [13].
3. **Not everything should be managed by ABSA.** Architectural descriptions reify system information and are exploited to reason at a high level about the best way of adapting a system. Hence, engineers developing ABSA mechanisms must make appropriate choices regarding which aspects of system operation could benefit from management by the adapta-

tion layer, and which would add unnecessary complexity and/or overhead, and should therefore be handled directly by low-level mechanisms in the system. For instance, in the particular case of Rainbow-DCAS, details, such as, the specifics of the scheme followed for re-prioritizing devices in rescheduling are abstracted away in the self-adaptive layer and managed directly at the system level.

4. **Sophisticated adaptation demands changes in the monitoring infrastructure.** Although Rainbow has enabled us to implement an automated version of scale-out in Rainbow-DCAS, we have discovered that the fact that the monitoring infrastructure used by Rainbow must be fixed at development-time imposes some restrictions on the solution space. Specifically, the inability to modify the structure of the monitoring and actuation infrastructures at run-time rules out solutions that reassign dynamically part of the devices to different processor nodes for a better distribution of the load. This limitation is currently being addressed in the new version of Rainbow, which will support run-time changes in its monitoring infrastructure.

5. **Explicit adaptation logic reduces evolution effort of adaptation mechanisms.** We found that the evolution of the baseline set of adaptation strategies in Rainbow-DCAS demanded time of an order of magnitude of just minutes, not hours. This contrasts with the effort required to evolve the original adaptation mechanisms in DCAS. Critical Software reported that the effort demanded to tune adaptation behavior when the middleware is deployed in a new location was about 2 man-days. The reason being, in the original version of DCAS, modifying the adaptation logic requires changes in code scattered across different system components. On the other hand, Stitch strategy code is centralized in a single location and some aspects of adaptation behavior (e.g., timing issues) are made explicit in the code, making them easier to interpret and modify.

Although some of the elements described in our experience are specific to

DCAS, there are techniques and artifacts that can be reused when incorporating ABSA to other legacy systems. In particular, the implementation of explicit feedback loops by using a customizable framework for self-adaptation can be employed across different systems, provided that: (i) a suitable description of the system's architecture can be extracted. Specifically, this description should facilitate developers the process of identifying the points in which the system should be monitored and actuated upon via effectors in order to introspect and control the system's state, respectively, and (ii) the source code for the system is available if the legacy system does not provide any interfaces that enable access to points of monitoring and actuation.

Moreover, in the specific case of Rainbow, many of the customization elements such as effectors, probes, gauges, adaptation scripts, and even architecture models can be reused across systems that share the same architectural style with little or no modification.

7. Threats to Validity

Regarding the internal validity of our study, the main threat concerns the measurement of results for implementation effort, which may be distorted by prior experience of members of the development team with either Rainbow or DCAS. However, although some of the authors of Rainbow and DCAS participated in the study described in this paper, all tasks that concern the implementation of Rainbow-DCAS were carried out by an independent team of three developers at the University of Coimbra without any prior experience with Rainbow or DCAS.

With respect to external validity, the main concern is the limited scope of our study, since it is restricted to a particular class of systems. In particular, our results are set in the context of DCAS and Rainbow, and generalization requires experimenting with further types of controllers and systems. However, despite the recent appearance of other frameworks for developing self-adaptive systems such as DYNAMICO [22], Zanshin [20], or StarMX [4], Rainbow is one

of the few frameworks that have been widely available for experimentation in the specific context of ABSA, to the best of our knowledge. Experience reports on engineering self-adaptation with new frameworks, such as the requirements-oriented Zanshin [20], are starting to pave the way for a more comprehensive understanding of new self-adaptation paradigms in practice. Our study is, as far as we know, the first of these experience reports that provides insight into the feasibility of replacing legacy embedded and manual adaption mechanisms by ABSA in industrial-scale software systems.

8. Related Work

Different approaches in the literature for developing adaptive systems range from those that adopt a prominently control-theoretic perspective [14], to others that employ requirements [15, 16, 17], or architecture models [1, 18, 19] to reason about the best way to adapt the target system at run-time.

Some of the proposals include reusable frameworks that facilitate incorporating self-adaptation mechanisms in legacy systems. StarMX [4] is a generic open-source framework that targets primarily systems in the Java domain. Management of non-Java systems can be achieved via Web Services or JNI, making StarMX a potential candidate for implementing self-adaptation in legacy systems. However, the framework has been evaluated to the best of our knowledge only on J2EE applications. Zanshin [16] incorporates an adaptation framework based on a feedback-loop architecture that has been evaluated using different systems. In particular, an experience report [20] describes the process of designing and developing adaptation scenarios on a simplified ATM system. Although the original system does not feature explicit legacy adaptation mechanisms, it is based on feedback loops, allowing adaptation mechanisms to be implemented by integrating the Zanshin framework with the ATM software via aspects, rather than from scratch. Although the report does not provide any quantification in terms of effort devoted to development, it identifies the localization and modification of the original implementation to support monitoring and adaptation

execution as one of the more challenging steps of the process. This is consistent with the results in our study, in which the modification of the original system demanded more effort than the implementation of the new adaptation mechanisms.

Prior experiences in engineering adaptation with Rainbow [6, 12] dealt with simple systems that did not include any legacy adaptation mechanisms. Moreover, although these studies provide detailed measurements of the effort invested in implementing self-adaptation, developers were part of the team that created the ABSA framework.

Various reports in self-adaptive systems [3, 21] point out the need for real data about engineering costs and effectiveness of applying ABSA to real systems. We see this work as a step in this direction.

9. Conclusions

In this paper, we assessed the benefits of architecture-based self-adaptation (ABSA) in the context of large-scale commercial software system, called Data Acquisition and Control Service (DCAS). This system is a middleware that already incorporates self-adaptation mechanisms, and is used to monitor and manage highly populated networks of devices in renewable energy production plants. To perform our evaluations, we independently developed a system that integrates DCAS with Rainbow, which is a framework for supporting architecture-based self-adaptation of software systems.

Our results show that **ABSA can successfully replicate the adaptation behavior required from an industrial-class software-based system**, such as DCAS. Regarding the overall distribution of the effort, approximately 60% was used to evolve DCAS for its integration with Rainbow, whereas the remaining time was spent in customizing Rainbow. Once the baseline set of adaptation strategies used to replicate DCAS adaptation behavior was completed, **incremental changes to evolve and improve Rainbow-based adaptation mechanisms demanded little time** (on the order of minutes, rather

than hours or days). Our experience indicates that **although incorporating ABSA in an already adaptive system initially demands an upfront effort in terms of specification and development, this investment pays off by substantially reducing effort in further system evolution** (in particular considering the fact that, typically, most of the overall effort is devoted to system maintenance [23]). We have also observed that centralized global adaptation improves adaptation by facilitating access to systemic information, and effectively enabling coordinated adaptation. However, we noted that not everything should be managed by ABSA, and that special attention should be paid to deciding which aspects of the system must be managed by low-level local adaptation mechanisms.

In this paper, we have also reported our experience using the Rainbow framework for implementing automatic scale-out adaptation in DCAS. Scale-out adaptation enables the system to deal with dynamic workloads that might incorporate new devices at run-time (something that the original DCAS does not address since scale-out is performed as a manual operation). Even though scale-out adaptation requires system-wide changes and is more elaborated than scale-up and rescheduling, our experience showed that **the effort required to implement automatic scale-out only represents a small fraction of the overall effort initially required to incorporate the architecture-based version of the scale-up and rescheduling**. This evidence indicates that while there is indeed a significant upfront investment required to incorporate ABSA, this effort pays off not only while evolving existing architecture-based adaptation mechanisms, but also when developing new ones that take advantage of the existing infrastructure.

Despite the wide range of proposals available for engineering self-adaptive systems, a recent literature review [21] puts forward the fact that research in this area is primarily evaluated using simple applications, and that collaborations between academic and industrial partners are very rare. Our study is, to the best of our knowledge, the first experience report that provides insight into the feasibility of replacing legacy embedded and manual adaption mechanisms by

ABSA in industrial-scale software systems.

Future work will deal with the evaluation of ABSA, using other types of legacy software systems and adaptation frameworks to assess the generality of our findings. In the context of DCAS, we will tackle more sophisticated versions of scale-out adaptation mechanisms than those currently implemented in the original DCAS and Rainbow-DCAS by incorporating new features under development in Rainbow, such as dynamic probe and effector placement. Finally, we will also investigate general criteria to decide what adaptations should be global and centralized, versus local and distributed.

10. Acknowledgements

Co-financed by the Foundation for Science and Technology via project CMU-PT/ ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN with COMPETE ref.: FCOMP-01-0124-FEDER-012983.

References

[1] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: L. C. Briand, A. L. Wolf (Eds.), FOSE, 2007, pp. 259–268.

[2] B. H. Cheng, R. de Lemos, et al., Software Engineering for Self-Adaptive Systems: a Research Roadmap, in: Software Engineering for Self-Adaptive Systems, Vol. 5525 of LNCS, Springer, 2009, pp. 1–26.

[3] R. de Lemos, et al., Software engineering for self-adaptive systems: A second research roadmap, in: Software Engineering for Self-Adaptive Systems II, Vol. 7475 of Lecture Notes in Computer Science, Springer, 2013, pp. 1–32.

[4] R. Asadollahi, M. Salehie, L. Tahvildari, Starmx: A framework for developing self-managing java-based systems, in: SEAMS, IEEE, 2009, pp. 58–67.

[5] R. Calinescu, L. Grunske, M. Z. Kwiatkowska, R. Mirandola, G. Tamburrelli, Dynamic QoS Management and Optimization in Service-Based Systems, IEEE Trans. Software Eng. 37 (3) (2011) 387–409.

[6] S.-W. Cheng, D. Garlan, B. R. Schmerl, Evaluating the Effectiveness of the Rainbow Self-Adaptive System, in: SEAMS, IEEE, 2009, pp. 132–141.

[7] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. R. Schmerl, R. Ventura, Evolving an adaptive industrial software system to use architecture-based self-adaptation, in: SEAMS, IEEE / ACM, 2013, pp. 13–22.

[8] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, IEEE Computer 37 (10) (2004) 46–54.

[9] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36.

[10] G. Abowd, R. Allen, D. Garlan, Using style to understand descriptions of software architecture, ACM Transactions on Software Engineering and Methodology 4 (1993) 319–364.

[11] S.-W. Cheng, D. Garlan, Stitch: A language for architecture-based self-adaptation, Journal of Systems and Software 85 (12) (2012) 2860–2875.

[12] S.-W. Cheng, Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation, Ph.D. thesis, CMU (2008).

[13] J. Cámara, P. Correia, R. de Lemos, M. Vieira, Empirical resilience evaluation of an architecture-based self-adaptive software system, in: QoSA, ACM, 2014, pp. 63–72.

[14] A. Filieri, C. Ghezzi, A. Leva, M. Maggio, Reliability-driven dynamic binding via feedback control, in: Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012, pp. 43–52.

[15] J. Whittle, P. Sawyer, N. Bencomo, B. Cheng, J.-M. Bruel, Relax: a language to address uncertainty in self-adaptive systems requirement, Requirements Engineering 15 (2) (2010) 177–196.

[16] V. E. Souza, Requirements-based Software System Adaptation, Ph.D. thesis (2012).

[17] L. Baresi, L. Pasquale, P. Spoletini, Fuzzy goals for requirements-driven adaptation, in: Requirements Engineering Conference (RE), 2010 18th IEEE International, 2010, pp. 125–134.

[18] D. Menasce, H. Gomaa, S. Malek, J. Sousa, Sassy: A framework for self-architecting service-oriented systems, Software, IEEE 28 (6) (2011) 78–85.

[19] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (1999) 54–62.

[20] G. Tallabaci, V. Silva Souza, Engineering adaptation with zanshin: An experience report, in: SEAMS, 2013, pp. 93–102.

[21] D. Weyns, T. Ahmad, Claims and evidence for architecture-based self-adaptation: A systematic literature review, in: Software Architecture, Vol. 7957 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 249–265.

[22] G. Tamura, et al., Improving context-awareness in self-adaptation using the dynamico reference model, in: SEAMS, IEEE, 2013, pp. 153–162.

[23] F. P. Brooks, Jr., The Mythical Man Month: Essays on Software Engineering, 1st Edition, Addison-Wesley, 1975.