Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation

Javier Cámara *, Pedro Correia*, Rogério de Lemos†, David Garlan‡, Pedro Gomes§, Bradley Schmerl‡ and Rafael Ventura*
*University of Coimbra, Portugal. Email: {jcmoreno, pcorreia, ventura}@dei.uc.pt
†University of Kent, UK. Email: r.delemos@kent.ac.uk
‡Carnegie Mellon University, USA. Email: {garlan, schmerl}@cs.cmu.edu
§Critical Software, Portugal. Email: pedro.m.gomes@criticalsoftware.com

***Abstract*—Although architecture-based self-adaptation has been widely used, there is still little understanding about the validity and tradeoffs of incorporating it into real-world software-intensive systems which already feature built-in adaptation mechanisms. In this paper, we report on our experience in integrating Rainbow, a platform for architecture-based self-adaptation, and an industrial middleware employed to monitor and manage highly populated networks of devices. Concretely, we reflect on aspects such as the effort required for framework customization and legacy code refactoring, performance improvement, and the impact of architecture-based self-adaptation on system evolution.**

I. Introduction

Architecture-based self-adaptation [7], [9], [10] is regarded as a promising approach to building flexible and dependable software systems able to autonomously adapt to changes in the conditions prescribed by their environment at run-time. Although there is previous experience in applying architecture-based self-adaptation in practice [2], [5], [6], [8], the common denominator for the existing case studies is that they deal with target systems in which self-adaptive capabilities are designed and incorporated from scratch. However, in practice, many legacy systems have some adaptation mechanisms already built-in (and often tightly coupled with the rest of the system).

Currently, there is little understanding about the feasibility and tradeoffs of implementing architecture-based self-adaptation in such systems. This paper tackles this issue by addressing two fundamental questions: (i) Can architecture-based self-adaptation be applied to legacy systems *that have existing self-adaptation encoded in them*?, and (ii) What is the effort associated with improving adaptation behavior in such systems using architecture-based self-adaptation?

To answer these questions, we report on our experience in applying architecture-based self-adaptation to an industrial middleware system developed at Critical Software called Data Acquisition and Control Service (DCAS), which is used to monitor and manage highly populated networks of devices in renewable energy production plants.

For the implementation of our prototype we used Rainbow [7], a framework that provides a reusable infrastructure for the engineering of self-adaptive capabilities to monitor, decide, and act on situations that require system adaptation.

To achieve our goal, first, we removed built-in adaptation mechanisms in DCAS in order to obtain a version that could be integrated with Rainbow, thus allowing us to replicate on our Rainbow-based prototype the adaptation behavior of the original DCAS. Secondly, since DCAS was slow in recovering its performance in situations in which devices were persistently slow in reporting data, we assessed the difficulty of modifying adaptation behavior using architecture-based self-adaptation when using our Rainbow-based prototype.

The rest of this paper is organized as follows. Section II provides a general description of DCAS. Section III briefly describes the Rainbow approach for architecture-based self-adaptation and summarizes applications to other case studies. In Section IV, we describe the approach followed for the integration of Rainbow and DCAS. Section V provides an evaluation of different aspects regarding the process of integration and results obtained. Section VI concludes the paper and indicates directions for future work.

II. Data Acquisition and Control Service (DCAS)

The Data Acquisition and Control Service (DCAS) is a middleware from Critical Software 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)[1], which is a platform that provides asset management support for power producing companies based on renewable energy sources. The overall csEMS architecture aims at high scalability, flexibility and customization with management capabilities that enable the operation of control centers independently of the underlying application (*e.g.*, wind, solar, etc).

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

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

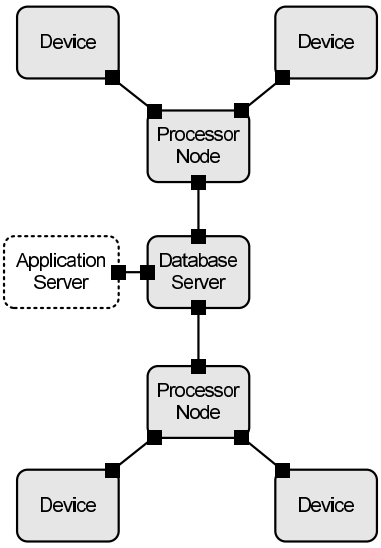

Figure 1. Architecture of a DCAS-based system

- **Devices** are equipped with one or more sensors to obtain data from the application domain (*e.g.*, from wind towers, solar panels, etc.). Each one 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 the rate at which the device should be polled for data, or which are 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 instance of DCAS.
- **Application server** is connected to the database server to obtain data, which can be presented to the operators of the system or processed automatically by application software. However, the DCAS service is application-agnostic, so the application server will not be discussed in the remainder of this document.

The main objective of DCAS is collecting 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. To achieve this objective, a DCAS-based system shall be able to scale up, making use of the computational resources in the node(s) where it is running, and scale out, supporting the deployment of several instances of the service within the same system to extend the number of connected devices.

A. DCAS Structure and Functionality

A different instance of DCAS 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:

- **Service Engine** is in charge of orchestrating all the flow of data among the different components of the service.
- **Polling Scheduler** triggers the process to perform requests to devices according to their scheduled time of execution.
- **Data Requester** performs requests to devices.
- **Data Persister** stores the information obtained from devices into the database.
- **Alarmer** raises alarms if the data coming from the devices is corrupted (*e.g.*, values out of expected range).
- **Data Stream Manager** manages the information regarding the *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.

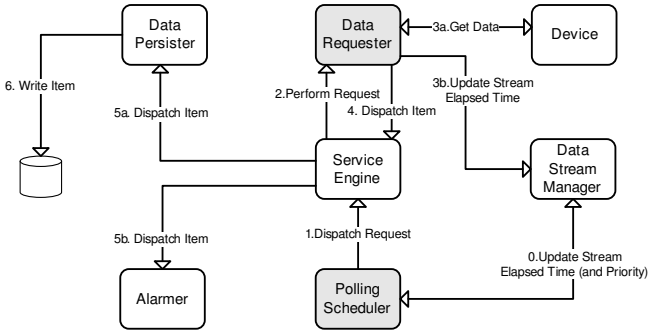

Figure 2. DCAS service operation

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:
   - 3a. Communicates with the device, retrieving the requested data and packing it into an item.
   - 3b. Updates the elapsed time information of the stream from which data has been read in step 3 (it is worth reminding that a device can have one or more data streams assigned from which data is read).
4. The item is dispatched by the Data Requester to the Service Engine.
5. The Service Engine:
   - 5a. Dispatches a copy of the item to the Data Persister.

5b. Dispatches a copy of the item to the Alarmer.

The information of a data stream is updated according to the time elapsed since the request for data is performed by the Data Requester, until a response is received from the device. Moreover, the Polling Scheduler continuously updates the priorities of the scheduled requests according to the information updated in the Data Stream Manager (see step 0 in Figure 2). Further details about this issue can be found in Section II-B1.

Two important components in DCAS for achieving the desired quality goals 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.

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 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, and a collection of processes, called *Data Requester Processor Pollers* (DRPPs), that dequeue requests from the secondary queue and retrieve the data from the appropriate device according to the specific contents of the request.

Concretely, 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 queue.
2. A process called *Data Requester Poller* retrieves a request from the primary 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) 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 cycles through the queue processing requests in the following manner:

1. The PSP dequeues the request in the first position of the queue.
2. The PSP clones the request retrieved from the queue and dispatches the clone to the service engine.
3. The queue retrieves an updated value for the elapsed time of the data stream targeted by the request and computes a priority for it 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.

B. Adaptation Mechanisms

In Section II-A, we have described the structure and functionality of DCAS. In this section, we focus on the existing adaptation mechanisms of DCAS that are aimed at maintaining the performance of DCAS under different loads. These adaptation mechanisms respond to failing devices, increased number of devices, and changing data rates.

1) Rescheduling: The rescheduling mechanism affects the Polling Scheduler, and is aimed at avoiding the degradation of performance of the system caused by devices which fail to respond in a timely manner (or do not respond at all) when polled. In a nutshell, the mechanism consists in decreasing the priority 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 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 is 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.

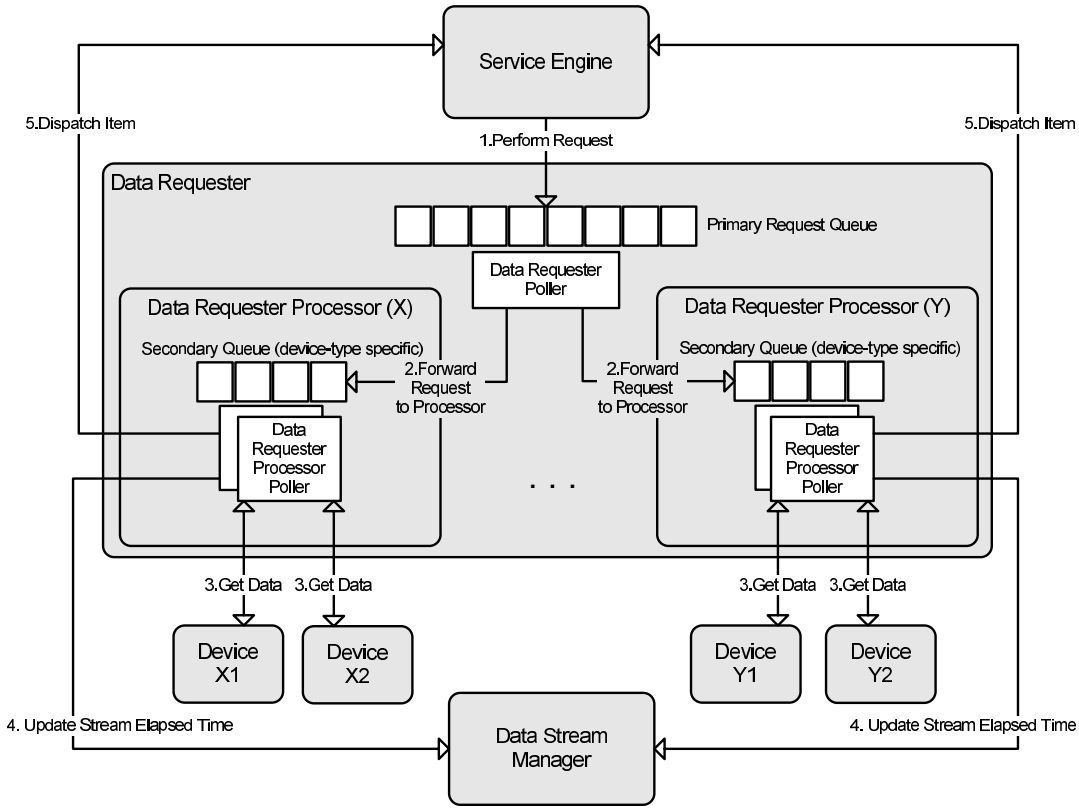

Figure 3. Data requester operation

- **Effective Sample Rate (ESR)** 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 [2] (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.

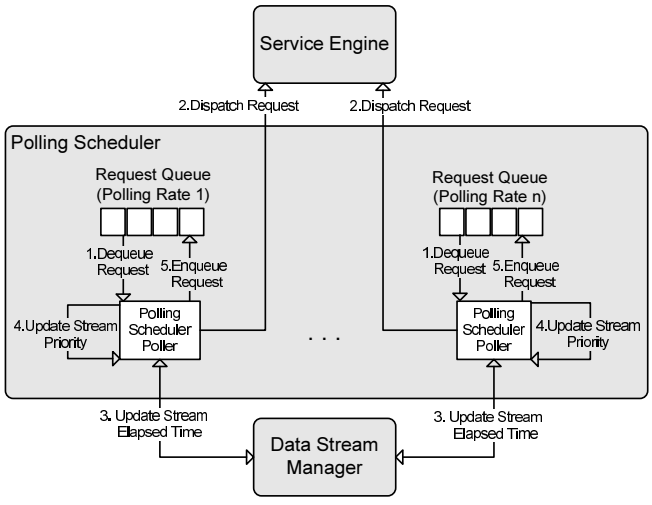

Figure 4. Polling scheduler operation

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

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

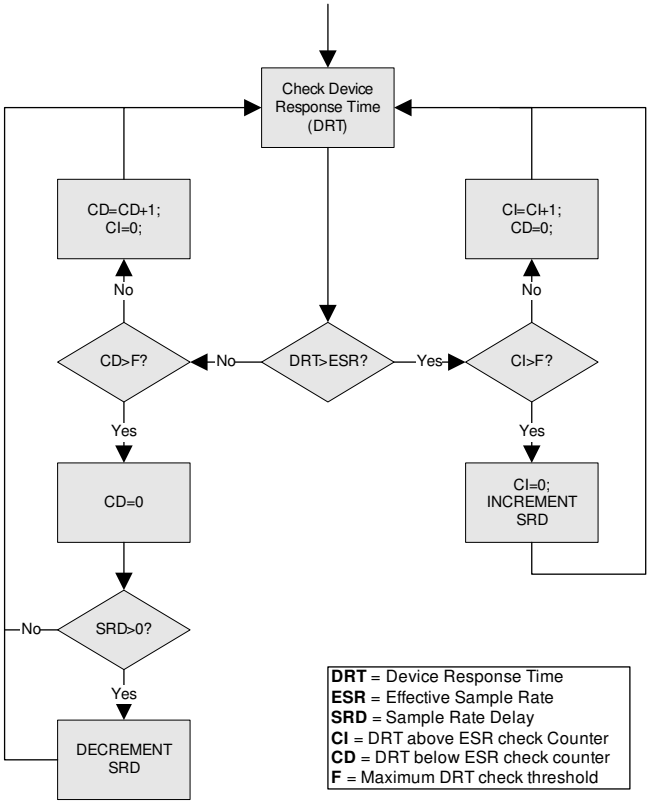

Figure 5. Flowchart of the rescheduling adaptation process

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. Concretely:

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

3) Scale Out: **Scaling out is supported in DCAS only as manual operation**. When the system is unable to cope with the given configured data rates while using maximum computational resources, it writes an entry to the log in the database in order 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 not aware of the existence of others, but there is a basic mechanism implemented so that each instance gets only the data streams it should process. Concretely, data stream entries in the database include a DCAS instance identifier (manually configured by the human operator) indicating which service instance should process its requests.

III. The Rainbow Approach

Rainbow is an architecture-based platform for supporting self-adaptation of software systems, which has the following distinct features: an explicit architecture model of the target system, 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.

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. The main components of the framework are:

- **Architecture Evaluator** evaluates the model to ensure that the target system is operating within an acceptable range, as determined by the 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* [1] 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

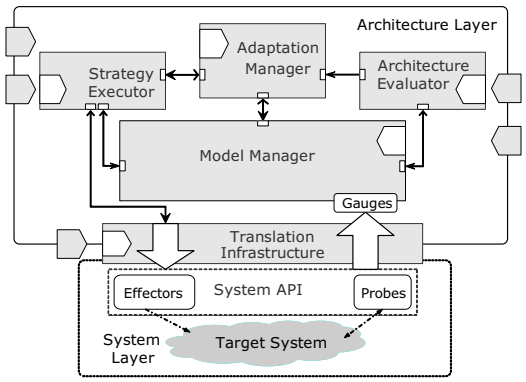

Figure 6. The Rainbow framework

through customization has the advantage of reducing remarkably the cost of development.

Building upon the elements of the architectural style, Rainbow provides the *Stitch* [5] 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 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, we have applied Rainbow to several different kinds of systems, and to adapt to maintain different types of quality attributes. In terms of sytems, 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 [4]. 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 we 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 V.

IV. Integrating Rainbow and DCAS

In this section, we describe the process followed for the integration of DCAS and Rainbow, describing: (i) the evolution of DCAS, carried out to enable its integration with Rainbow; and (ii) the customization of the different elements of the Rainbow framework, including architectural model, operators, tactics, and adaptation strategies.

A. Evolution of DCAS

Previous case studies in which Rainbow has been applied [5], [6] describe systems that typically feature components that 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 the internal functionality in DCAS 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. Probes and effectors in Rainbow act as clients of the TCP server, which acts as a mediator between them and the actual probes and effectors embedded in DCAS:

- Probes embedded in DCAS keep the values of probed variables updated in a data store local to 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. Concretely, 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 TCP server data store.
- Effectors clients in Rainbow send requests for command execution to the TCP Server (E1), which forwards them to the effector embedded in DCAS (E2). Next, the effector executes the command (E3) and returns a response to the TCP server that states whether execution was successful (E4). Finally, the TCP server forwards the response to the effector client in Rainbow (E5).

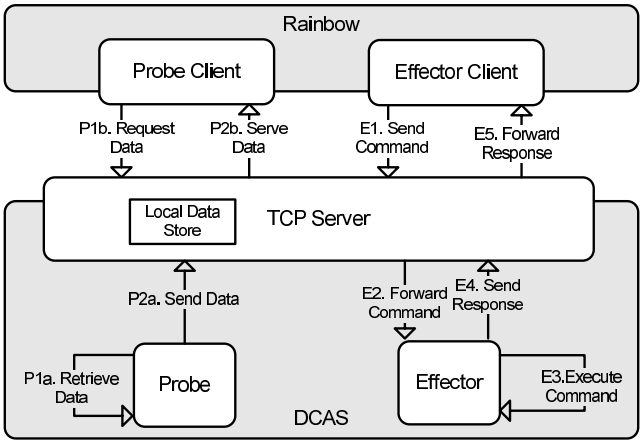

Figure 7. DCAS-Rainbow translation infrastructure

B. Customizing the Rainbow Framework

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 DCAS write information collected from network devices.

We can identify two quality objectives for the self-adaptation of a DCAS-based system: (A) performance, and (B) cost. Performance analysis suggests we monitor the 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.

Type	Property	Operator
DeviceT	sampleRateDelay effectiveSampleRate deviceResponseTime	changeSampleRateDelay (sampleRateDelay :int)
ProcessorNodeT	numPollers queueSize queueStatus	increasePollers() decreasePollers()
DBServerT	rps	

Table I
DCAS ARCHITECTURAL STYLE ELEMENTS

Table I 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 to the concepts discussed in Section II-B1. 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 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.

Using these operators, 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 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 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 V-B1):

IncreasePerformance. When DCAS is experiencing low performance (rps below threshold, rpsViolation), and the number of active pollers is not above the number of data streams with low responsiveness (!maxLazyStreams), activate a new poller if queues are not shrinking, then if queues are still not shrinking after 5 seconds, add another poller.

```
strategy IncreasePerformance
[ styleApplies && rpsViolation && !maxLazyStreams ]{
  t0:(!qShrinking)->addPoller()@[5000 /*ms*/]{
    t0a:(!qShrinking)->addPoller()@[10000 /*ms*/]{
    t0b: (qShrinking)->done;
    }
  }
  t1: (qShrinking) -> done;
}
```

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.

```
strategy ReduceCost
[ styleApplies && qViolation2 && !minPollers ]{
  t0:(qViolation2)->removePoller()@[3000 /*ms*/]{
    t0a:(qViolation2)->removePoller()@[3000 /*ms*/]{
    t0b:(!qShrinking)->done;
    }
  t0c:(!qShrinking)->done;
  }
}
```

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 }
```

Although this baseline set of adaptation strategies was able to successfully replicate the adaptation behavior of DCAS, we evolved them since, in some cases, this behavior is not enough to recover system performance in a timely manner (please refer to Section V-B1 for details). Concretely, 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 results of applying these modifications are described in Section V-B2.

V. Evaluation

In this section, we evaluate our modifications to DCAS in two dimensions. Firstly, we report on the implementation effort involved in (i) customizing Rainbow to apply it to DCAS, (ii) modifying DCAS to remove its existing, hardcoded self-adaptation mechanisms, and (iii) the effort in improving the new adaptation strategies to make the adaptations more responsive to problems. Secondly, we evaluate the performance of the adaptations (i) to verify that replicating the adaptations in DCAS with Rainbow provides similar adaptation performance, and (ii) to measure the adaptation improvement in Rainbow.

A. Implementation Effort

1) Rainbow Customization: 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 (making use of Acme), scripting of the adaptation (developing tactics and strategies in Stitch), and development and testing of the translation infrastructure, including probes, gauges, and effectors amounts to a total of 91 hours (approximately 2 1/3 work weeks).

Task	Time	%
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 II
Rainbow customization effort for DCAS

Table II 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%.

2) Evolution of DCAS: The overall time spent in readying DCAS for Rainbow was 145 hours (approximately 3 2/3 work weeks). As can be observed in Table III, 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 required to enable access to the classes and methods needed to obtain probe information and effect changes in the system (please refer to Section IV-A).

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

Table III
DCAS evolution effort

3) Evolution of 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.

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

Table IV
Size/Scattering of DCAS adaptation mechanisms

Table IV 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, 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 different 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).

B. Experimental Evaluation

The aim of our experiments is assessing the validity of architecture-based self-adaptation mechanisms in the context of an application-agnostic middleware, comparing their performance and efficiency with those achieved by DCAS built-in adaptation mechanisms.

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. 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 memory.

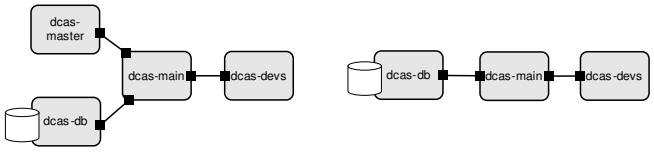

Figure 8. Experimental setup: Rainbow-DCAS (left) and DCAS (right)

Our experiments include 100 data streams with a sample rate of 1 second. The duration is 40 minutes (2400s), and the pattern followed is: (i) 600s of normal activity to let the system achieve a steady state; (ii) 600s of disturbance, 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 (iii) 1200s of normal activity.

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.

1) 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

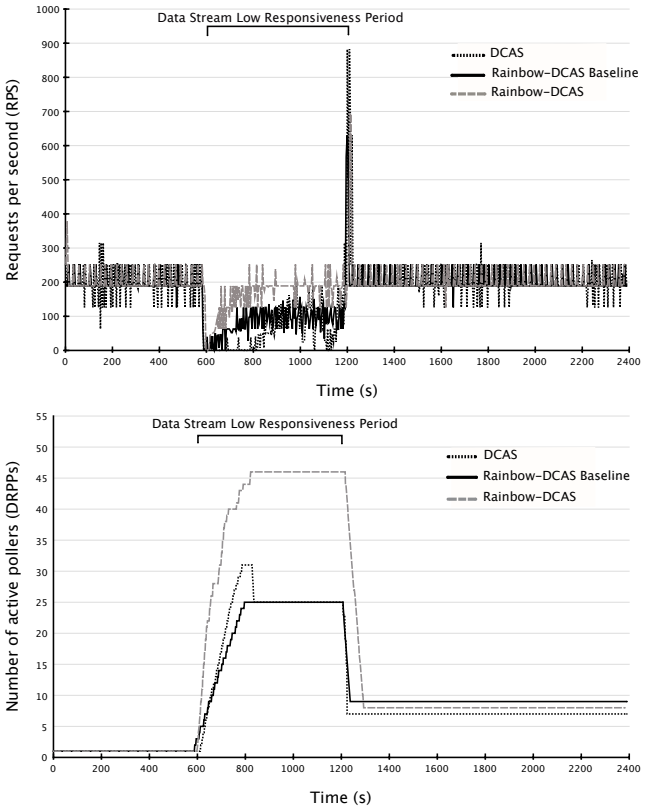

Figure 9. Performance (top) and number of active pollers (bottom)

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 baseline. This is explained by the longer time periods between the consecutive queue size checks required to activate pollers (as described in Section II-B2), compared to the higher frequency of probe updates and shorter adaptation cycle time in Rainbow.

2) 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.** Concretely, 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

Task	DCAS	ZNN	TalkShoe
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 V
RAINBOW CUSTOMIZATION EFFORT

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.

VI. CONCLUSIONS

In this paper, we have assessed the validity of architecture-based self-adaptation in the context of real-world software-intensive systems which already feature self-adaptation mechanisms. To achieve our goal, we independently developed a prototype based on the Rainbow framework for architecture-based self-adaptation of DCAS, an industrial middleware for data acquisition and control in power plants.

Our results show that **architecture-based self-adaptation 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.

Table V compares the customization effort of Rainbow required for the implementation of DCAS, ZNN, and Talk-Shoe. Results show that **the 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.

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, not hours).

According to our observations, we can conclude that, **although incorporating architecture-based self-adaptation in an already adaptive system initially demands an additional effort, 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 [3]).

Future work will deal with the evaluation of architecture-based self-adaptation in other types of legacy software systems to assess the generality of our findings. In the context of DCAS, we will tackle more sophisticated adaptation mechanisms than those currently implemented in DCAS and Rainbow-DCAS, which target scenarios with workloads that feature a fixed number of data streams with varying conditions (*i.e.,* device response times). Scenarios with dynamic workloads that might incorporate new devices at run-time are not currently considered in DCAS, in which scale-out is performed as a manual operation. Concretely, we aim at using architecture-based self-adaptation to overcome the current limitations of DCAS and report on implementing automatic scale-out adaptation in Rainbow-DCAS.

REFERENCES

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

[2] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *SEAMS*, pages 58–67. IEEE, 2009.

[3] F. P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, first edition, 1975.

[4] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, CMU, 2008.

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

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

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

[8] J. C. Georgas and R. N. Taylor. Software engineering for self-adaptive systems. chapter Policy-Based Architectural Adaptation Management: Robotics Domain Case Studies, pages 89–108. Springer-Verlag, Berlin, Heidelberg, 2009.

[9] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In L. C. Briand and A. L. Wolf, editors, *FOSE*, pages 259–268, 2007.

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