A Uniform Programming Abstraction for Effecting Autonomic Adaptations onto Software Systems

Giuseppe Valetto
Telecom Italia Lab
Torino, Italy
Giuseppe.Valetto@tilab.com

Gail Kaiser
Columbia University
New York, NY, United States
kaiser@cs.columbia.edu

Abstract

Most general-purpose work towards autonomic or self-managing systems has emphasized the front end of the feedback control loop, with some also concerned with controlling the back end enactment of runtime adaptations – but usually employing an effector technology peculiar to one type of target system. While completely generic "one size fits all" effector technologies seem implausible, we propose a general-purpose programming model and interaction layer that abstracts away from the peculiarities of target-specific effectors, enabling a uniform approach to controlling and coordinating the low-level execution of reconfigurations, repairs, micro-reboots, etc.

1. Introduction

Current trends in software development increasingly favor the construction of large-scale, distributed software ensembles that provide new services via the loosely-coupled integration of a mix of pre-existing and new sub-systems.[1] Each sub-system may have been built separately by some third party, and may be complex on its own: for example, it may itself be distributed; it may rely on its own stack of middleware layers; etc.

These large-scale *systems-of-systems* present a technological heterogeneity that poses a significant problem when it comes to the development and the execution of provisions for their *runtime adaptation*. By runtime adaptation, we mean any automated set of actions aimed at modifying the structure, behavior and/or performance of a target software system while it continues operating. Runtime adaptation can be used to address self-management concerns, for instance to (re-)configure, recover from faults, tune extra-functional parameters, and so on. In systems-of-systems, these changes may impact multiple components or modules in multiple different sub-systems, which may have diverse technological underpinnings.

One way to cope with this heterogeneity is to develop the code that is used to effect a desired adaptation step (sometimes called an *effector*) piecemeal and *ad hoc* for each distinct adaptation that can be applied to each target component. A better approach, when feasible, is to tailor (to each relevant adaptation step) generic effectors supplied by some technology that already interacts nicely with one or more of the sub-systems or sub-system components (or with middleware underlying those sub-systems). Either model, however, tends to negatively impact the level of generality of the autonomic techniques and solutions that apply runtime adaptations on systems-of-systems by invoking these effectors. The best solution would be to abstract away from the peculiarities of individual effectors or effector technologies, with a uniform veneer that can be leveraged for simple and consistent interaction with all effectors.

This paper introduces an abstract programming model, represented by a limited set of primitives that can be used to describe and direct the various phases of the work of an effector. This set of primitives can be reified as a generic effector API, to effectively hide diversity in effectors. Each effector can still be developed reflecting the technology and other specifics of its intended target component, which ensures efficiency in the adaptation implementation; however, all effectors can be managed uniformly through the

[1] The Gartner Group calls these systems "composite applications": "*Composite applications … enable the development of new application systems by combining brand-new logic and transactions exposed by pre-existing, legacy applications*" [1]. "*Monolithic, isolated application stovepipes are being left behind. New systems are partitioned, distributed, integrated ….*" [2].

effector API according to the underlying programming model. This enables generality in the autonomic control facilities that interact with their adaptation targets via the effectors. While conceived for complex systems-of-systems, of course the same programming model can be conveniently adopted for more homogeneous systems.

In the next section, we present a general model of self-management infrastructures, intended to be applicable to most autonomic computing approaches, within which our programming model "fits" – this material can be viewed as our "problem statement". We then motivate the proposed programming abstractions using examples extracted from an industrial case study [12] [14]. This case study has provided the authors with a set of requirements, on the basis of which the proposed abstraction layer and its primitives have emerged. We then discuss the adoption of our programming model and the implementation of the corresponding generic effector API within the Kinesthetics eXtreme (KX) platform for the runtime adaptation of complex systems-of-systems. Previous papers on KX [7] [8] [9] [10] [11] emphasized the monitoring, analysis, planning and execution coordination aspects of its autonomic infrastructure; here we present our effector programming model and API for the first time.

2. Autonomic Management Model

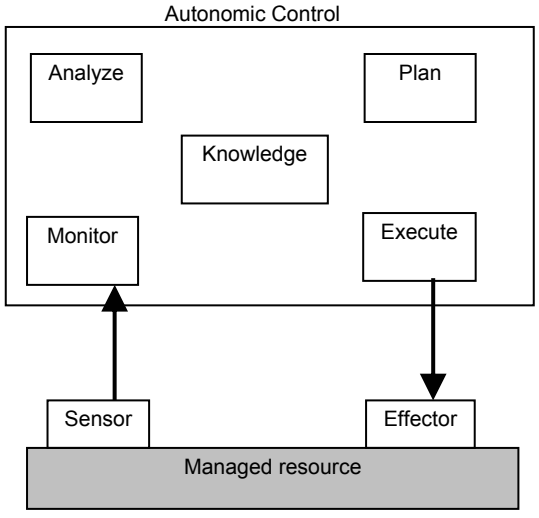

Figure 1: IBM MAPE-K reference model.

As a general rule, autonomic computing techniques assume the presence of provisions that enable to dynamically adapt some aspects and elements of the system while it is in operation. The necessity of those provisions as first-class entities in an autonomic system is highlighted, e.g., within the MAPE-K (Monitor, Analyze, Plan, Execute -Knowledge) reference model for autonomic control loops, proposed by IBM [3], displays a high-level view of a MAPE-K framework, including its interactions with some system component to be adapted (i.e., a *managed resource*). In particular, the *Execute* element of a MAPE-K loop clearly relies on some means that can be used to carry out the planned changes onto the managed resource: in software systems terms, that equates to some pieces of code (some effectors) that can be invoked or otherwise activated to effect those changes. Most other models and architectures for general-purpose autonomic systems also highlight effectors as first-class entities: for example, the Rainbow platform for architecture-based adaptation developed at CMU [20] (depicted in Figure 2), and the Accord component framework [24].

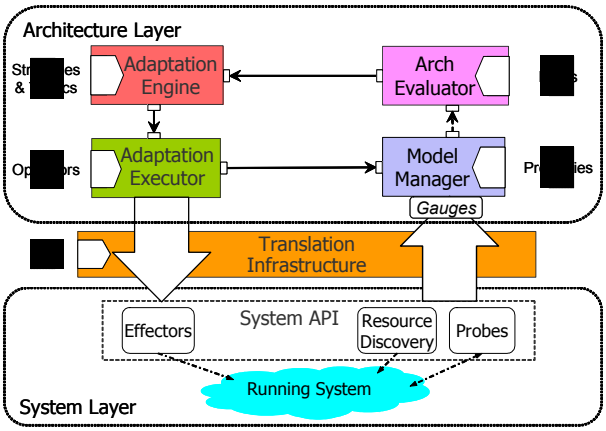

Figure 2: CMU Rainbow architecture.

An effector can take many forms. Examples include: a resident algorithm or module running inside a software component; a set of proprietary operations that can be invoked from outside the component by some supervising entity or peer component(s); functionality exposed externally in compliance with some standardized programmatic interface (such the JMX management framework[2] or the Common Information Model (CIM) family of standards[3]); mobile code that is dispatched to be executed onto the component [13]; or entirely external utilities in the environment that can manipulate arbitrary components or processes in some manner, e.g., process migration across hosts [21].

The kind and reach of the adaptation(s) made possible by an effector – or a related set of effectors - may vary considerably, ranging from the tuning of internal parameters within a single module, to the re-

[2] See http://java.sun.com/products/JavaManagement.
[3] See http://www.dmtf.org/standards/standard_cim.php.

arrangement of the architectural configuration and the distribution layout of a complex distributed system, to the controlled shut down and/or restart of the managed resources, as advocated in Recovery-Oriented Computing (ROC) [4]. Similarly, the nature and characteristics of effectors can be very different, and depend on multiple factors, such as the choices taken when designing the target system and any middleware technology upon which that system could rely - choices made in many cases without automated adaptation in mind - as well as the scope and goals of the autonomic solution intended to be applied to the system. By influencing the availability (or lack) of certain features - such as introspection, extensibility, openness, exposition of suitable programmatic interfaces, etc. – those factors determine how internal effectors are activated and external effectors imposed.

When the target of runtime adaptation is monolithic, or mostly homogeneous - for instance, a system developed by a single stakeholder and/or with a single pre-integrated suite of software technologies - it may be possible to select and adopt a single technique for all effectors and all runtime adaptation needs. In the context of systems-of-systems, however, many of the options above can co-exist; moreover, the technologies adopted for the adaptation provisions might not interoperate well, particularly when combined in unanticipated and "creative" ways designed to cope with the self-management requirements imposed *a posteriori* on the constructed ensemble. Since systems-of-systems are by definition large-scale, widely distributed, and possibly administered under the ownership of multiple stakeholders, an autonomic controller intended to exert end-to-end concerted runtime adaptation on the whole system may face challenging heterogeneity problems.

3. Motivating Example

3.1. Runtime adaptation with KX

To facilitate the definition and development of complex, end-to-end adaptations that require multiple coordinated steps and impact multiple sub-systems, our KX autonomic infrastructure relies on an engine, named Workflakes [11] [14], for the enactment of runtime adaptation logic in the form of a process, or workflow. The workflow plan is structured as a task decomposition hierarchy. Leaf (atomic) tasks produce the intended side-effects of the workflow, that is, to effect changes onto the target system via the selection and execution of a specific effector. Workflakes hence considers effectors as first-class resources, and its design includes an effector catalog, that is, a repository that associates each leaf task with the possible effectors it might instantiate.

In the absence of our effector programming model, how the Workflakes engine must interface to and activate an effector to enact a leaf task depends entirely on the nature and technology of the corresponding effector. The same would hold for any other autonomic controller without an effector abstraction layer.

The first release of Workflakes, used in the case study described in Section 0, was limited to orchestrating effectors that could be exposed through a single technology. Mobile (Java) code was our choice. For one thing, that paradigm was particularly apt for an exo-structure like KX, which aimed from the start to "autonomizing legacy systems" from the outside [9], as opposed to infrastructure-level autonomic mechanisms, such as adaptive middleware (like IQ-Services [25] or ACT [26]), or kernel-level provisions (like Q-fabric [29], for resource management). Furthermore, we could take advantage of an in-house mobile code technology, named Worklets, which our lab had previously developed for unrelated purposes [22], and which we could tailor to runtime adaptation requirements.

Worklets - like other mobile code systems - require at the receiving end of the code transfer the presence of a "landing dock", which can receive and execute the incoming code. In Worklets, that dock is called a Worklet Virtual Machine (WVM) and is embedded in a Java Virtual Machine (JVM). WVMs allow incoming Worklets to be activated and also to interface to any adaptation code already residing within target system components. However, this approach had several drawbacks: the reach of Workflakes-orchestrated runtime adaptation was limited to targets into which WVMs could be embedded (via manual or automated instrumentation); we were forced to wrap with WVMs all kinds of effectors, even when other methods to expose those effectors existed natively; and, of course, the target system had to be implemented in Java.

3.2. Case study description

The case study regards a multi-channel instant messaging (IM) service for personal communication. The runtime environment (see Figure 3) consists of a typical three-tiered server farm, incorporating a mix of commercial software elements, such as a load balancing package at the front end; J2EE enterprise application servers as the backbone of the middle tier, and an Oracle relational database at the back end; proprietary applications, in particular the IM server hosted in the middle tier - which may or may not be

wrapped within a J2EE Web application - and the distributed shared state repository that allows multiple replicas of the IM server to operate in an undifferentiated way as a collective service; and black box or legacy subsystems, typically providing some specific functionality, for example access to the service through certain channels, such as SMS or WAP, and gateways to the mobile communication network.

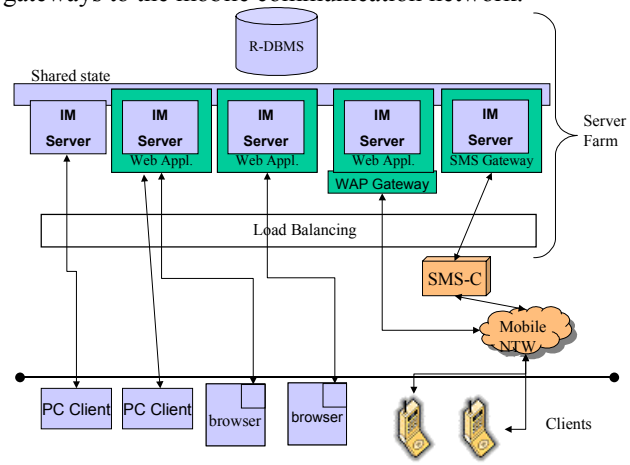

Figure 3: The IM service architecture.

The case study aimed mainly at *facilitating service management* by the system administrators in charge of such a complex distributed application, and *enhancing the QoS* perceived by end-users. Runtime adaptation focused mostly on the middle (application) tier. Even in that restricted context, heterogeneity of target components and effector technologies represented a source of complications. The major adaptation targets were the IM servers, the corresponding Web-based components (proprietary), and the BEA Weblogic J2EE application servers hosting those Web applications (third-party commercial products). The IM software did not natively expose specific provisions for automated management or self-tuning; on the other hand, since it was under one co-author's control, it was amenable to instrumentation with mechanisms, such as the WVMs, for the purposes of the case study. In contrast, the commercial BEA software natively exposed full-fledged and extensible programmatic facilities for the management of the application server and some general features of the hosted Web applications, by means of a JMX-based framework, with its set of MBeans.

For some of the goals of this case study, mobile code turned out to be very convenient: appropriately coded Worklets were shipped to the various service elements, in accord with adaptation logic described by workflow plans running in the Workflakes engine. For example, Worklets, which carried classes and configuration code for the IM servers to the various hosts in the server farm, were appropriate to carry out deployment, initialize the service from scratch, execute re-configurations on the fly, apply patches and roll-out new releases of an active service. All these tasks were beneficial in terms of service management automation.

Another kind of adaptation addressed responsiveness issues in the Web-based component of the IM service. KX monitored thresholds on the size of the queue of pending HTTP requests to the IM Web applications, and Workflakes responded by tuning the number of threads assigned by the application server to the Web application: this adjusted the degree of parallelism in processing client requests, kept the queue size low, and hence ensured satisfactory system responsiveness to Web users accessing the IM service.

The effectors used to manipulate the threading model of the IM Web application were implemented as JMX MBeans. That approach could have permitted effector activation simply by remote messaging; however, since at that time Workflakes had adopted Worklets technology as the sole supported activation mechanism, we were forced to introduce a landing dock for Worklet effectors, embedded in the application server. This dock included a WVM and presented to incoming Worklets a local JMX-based interface to the same MBeans that could have been accessed remotely from Workflakes itself. Besides representing a substantial complication, this implementation was made possible only by the extensibility features of the BEA Weblogic management subsystem, and would have been difficult, or even infeasible, in less open contexts. Moreover, at all times when thread tuning was required for the runtime adaptation of the IM service, Workflakes had to ship a new instance of the same mobile code snippet to the application server, an unnecessary overhead.

Figure 4 illustrates the complications that we had to introduce in order to force upon that situation the specific model mandated by Worklets mobile agent effectors. Case a) in the figure represents a hypothetical situation that would have occurred if the adaptation engine had been able to exploit the native JMX activation mechanisms of the provided MBeans; Case b), instead, shows the more convoluted interaction model required by Worklets.

At the same time we recognized that drawback, we also realized that moving away completely from Worklet effectors to embrace a technique that could be a better fit in that specific case would have not been a solution, since we would have likely lost the afore mentioned benefits of using Worklets for other purposes we had observed in the same case study.

In retrospect, the general lesson that could be drawn from the case study was clear: it is implausible, in particular in the context of systems-of-systems, to be able to select a single "one size fits all" effector technology that can be re-used efficiently across diverse technological settings and for all application requirements. Therefore, we began to consider how we could instead accommodate and hide diversity with a model that could approach the interaction with effectors in an abstract, uniform way.

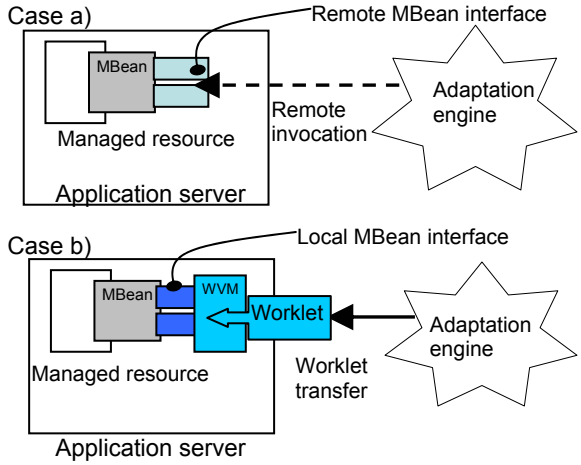

Figure 4: Interfacing JMX adaptation provisions to Worklets.

4. Proposed Model

Three main principles contribute to our effector programming model. They are:

1. The distinction between the *activation mechanisms* and the *adaptation provisions* of effectors.
2. The strict separation in runtime adaptation between the computations effecting changes and the adaptation logic according to which those computations are invoked.
3. The generalization of the work carried out by an effector according to a limited number of phases.

The first principle allows to distinguish within effectors two parts: the computation (i.e., the piece of code) that carries out in practice the desired changes, and which represents the actual *adaptation provision* available on the target component; and the *activation mechanism* that can be used to expose that computation to a variety of autonomic applications. That distinction can be expressed in terms of interface vs. implementation and information hiding: it helps isolate the heterogeneity problem within the technological underpinnings of activation mechanisms, and provides a focus for resolving that heterogeneity through abstraction.

The second principle is about separation of concerns: coupled with the first, it allows to regard the end-to-end runtime adaptation of large-scale distributed systems as the interplay between a repertoire of known and available effector computations that had been variously coded in or around the different subsystems, and adaptation logic that is capable and in charge of coordinating those computations as needed, by acting on their activation mechanisms. In terms of the MAPE-K reference model, the adaptation logic corresponds to the *Plan* element, as opposed to the *Execute* element, which activates the effectors. In order to make explicit and strengthen that separation, our model advocates the presence within an autonomic control system of an adaptation engine, whose role is to enact the logic that governs each runtime adaptation, and that remains disjoint from effector invocation mechanisms.

Those first two principles inspired our adaptation engine from the start, and can be already recognized in the description of the first release of Workflakes from Section 3.1. WVMs represent the activation mechanism for Worklet-based effectors, since they allow to run incoming code that interfaces to any adaptation provisions residing on target system components. Furthermore, the presence of Workflakes within KX satisfies the second principle, since it takes the role of the engine that enacts the adaptation logic, and directs – but remains separate from – effector computations.

Finally, the third principle – and the main contribution of our model – derives from the observation that the work of an effector onto its target can be often segmented in a handful of operations, which – at a high level of abstraction – do not depend on implementation choices. The interplay between the adaptation engine and some effector, requires, at a minimum, the following: the engine must instantiate the effector, or otherwise bind to a pre-existing instance; optionally, it can pass parameters to it; it then invokes its activation mechanism; on the other side, the effector needs to report back to the engine any results that become available as the byproduct of its execution. Those operations provide an abstract view of major phases or stages that can be recognized in the cycle of activity of an effector, and have hence inspired the set of primitives employed by our programming model.

4.1. Operational semantics

We have turned the conceptual phases of an effector activity outlined above into a small set of primitives that constitute a common interface for all activation mechanisms, fully decoupling the adaptation logic and the effectors. Each activation mechanism used for some given effector technology must be standardized to expose these same primitives, and implement them in the most convenient way, given the characteristics and constraints of that technology.

To make a practically useful interface that effectively enables transparent interaction with effectors, the primitives must present clear and consistent operational semantics. We have defined the following primitives:

- *Lookup*: this primitive represents a preliminary step for the adaptation engine, which is used to identify the type of effector that is necessary at each specific stage during some runtime adaptation and – if applicable – even the particular instance of that type that must be activated.
- *Recruit*: this primitive allows the adaptation engine to get a handle on the effectors that have been identified through the Lookup primitive. Depending on the situation, as well as on the kind of effector involved, it may be sub-categorized in one of two ways:
    - as *Instantiate*: implements the Recruit semantics in cases where a new instance is needed by the adaptation engine;
    - as *Bind*: implements the Recruit semantics in cases in which a suitable instance already exists and is available to the adaptation engine.
- *Configure*: this primitive carries out any initialization and customization work that may be necessary on the recruited effector; for example, it may pass to it the parameters suitable for the adaptation task at hand.
- *Activate*: this primitive is in charge of actually launching the execution of the effector computation; this may include the deployment of the effector onto the target component that needs to be adapted.
- *Relay*: this primitive provides a way for the effector, once it is activated and its work is under way, to report back to the engine any relevant data that it generates or observes. Since the work of an effector can have a relatively long duration and can occur asynchronously with respect to its activation, it is not usually convenient to model the passing of results in a request/response fashion. It seems more appropriate and general to equip the effectors in use with a data conduit (established during the configuration phase), which the effectors can employ whenever they need to relay data back to the task processor.

With the exception of Lookup, it is evident how all the primitives tend to be strongly dependent in their implementation on the technologies employed to develop the effectors, with their idiosyncratic properties. However, they collectively represent a uniform and abstract mode of interaction with all kinds of effectors.

Lookup is the only primitive that does not require direct coupling between the adaptation engine and the effectors. Their relationship is in that case mediated, since Lookup assumes the availability of an *effector catalog* (see Figure 5), which becomes therefore an integral part of our model That catalog is a repository of knowledge about effectors (types as well as instances) and responds to queries issued by the engine, whenever the engine needs some effector to proceed with the runtime adaptation. Each query must result in enough information to enable the Recruit stage that follows to either Instantiate the effector (e.g., a class name or identifier, or an address from which the corresponding executable code can be downloaded), or Bind to it (e.g., a reference to an active effector instance, or an address of a registry where the active instance is indexed).

In practice, the effector catalog must include some mechanism (such as associated meta-data) for the purpose of describing, discriminating among, and selecting suitable effectors for each task of a runtime adaptation process, and for the computing environment to be effected. No specific assumptions are, however, imposed at this level on the nature and format of the effector catalog; also the queries by the engine may in principle derive from a variety of application-dependent situations and can be expressed in a multitude of ways. For those reasons, Lookup remains, like the other primitives, an abstract operation in our model, and its implementation is fully dependent on the implementation chosen for the catalog itself.

4.2. Programming model for an effector API

As a counterpart to the conceptual interface represented by the primitives described in Section 4.1, and in order to facilitate its application, we have devised a design, according to which the activation mechanisms of effectors can be organized into a generic effector API, and a programming model that goes with it.

The main idea is the organization of the primitives comprised in the interface into three subsets (or *slots*), which become available at different times during the

execution cycle of an effector and that are implemented separately.

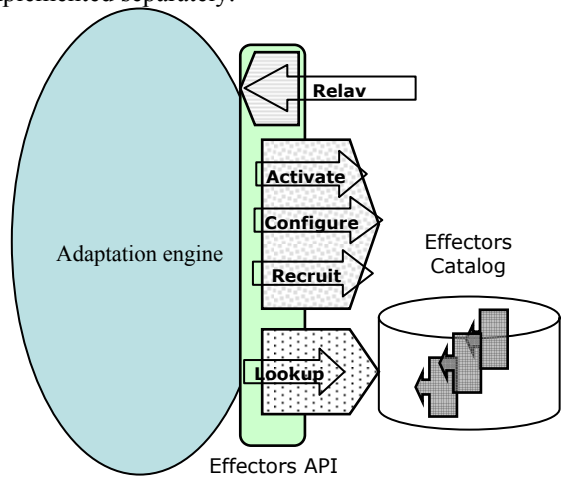

Figure 5: Design of the effector API.

The slots – as shown in Figure 5 – are:

- The Catalog slot, which comprises only the Lookup primitive. Since the implementation of this slot depends only on the nature of the effector catalog, and not on any specificity of the effectors listed in that catalog, that slot is always available, as it provides a conduit to issue queries to the catalog. A discussion on implementation options for the catalog follows in Section 5.
- The Activation slot, so called because it encompasses the actual activation mechanism of an effector, by grouping together the Recruit, Configure and Activate primitives. Their implementation is technology-dependent: to accommodate multiple implementations, this slot can be filled by adopting a plugin mechanism. Multiple plugins that expose those primitives can be developed according to the various available effector technologies, and can be loaded into the slot dynamically. A plugin is selected and used every time a certain effector is looked up from the catalog, to allow the adaptation engine to interact with the effector respecting the semantics of the primitives in the slot, and at the same time in compliance with the technology of that effector.
- The Relay slot, which comprises the Relay primitive, is also implemented by means of technology-dependent plugins. The plugin for this slot is selected, and passed to the instantiated effector as part of its Configure stage. It creates a communication channel from the effector back to the adaptation engine. To support the semantics of the Relay primitive and at the same time comply with the technological underpinnings of each activated effector, that communication channel must be implemented differently for each plugin, taking into consideration, whether the effector operates synchronously or asynchronously with respect to its invocation, or even whether the effector is able to pass back any data at all. For asynchronous cases, like in pub/sub messaging or mobile code dispatching, a callback on the adaptation engine that is activated when the Relay primitive communicates result data back can be appropriate. For synchronous cases, like for example remote method invocation, the invocation of Relay can be implemented as a necessary post-condition of Activate operation.

With the introduction of those slots and the related plugin-based programming model, we keep the interaction with effectors not only simple, but also independent from implementation concerns from the point of view of the adaptation engine.

To describe in detail how the programming model works, we rely on the sequence diagram in Figure 6. That diagram shows in detail how each of the stages of the execution of an effector proceeds. Specifically, it depicts a use case in which the Recruit primitive is mapped onto Instantiate, i.e., it requires the creation of a new instance of some type of effector, following the query to the effector catalog performed in the Lookup stage. The diagram also shows that the Relay stage in this case leverages a callback mechanism to transmit back the results of the effector execution. Notice that the diagram distinguishes the invocation of the primitives (displayed in bold italic font) by the adaptation engine through the API and onto the various plugin-based slots, from the implementation-dependent actions are carried out as a result of those invocations.

The diagram also outlines the flow of information that is exchanged during the various stages (in parentheses on the arrows: *q* stands for query, *q_res* for query result; *handle* indicates the handle to the recruited effector; *c_info* stands for the information needed for the effector configuration; *res* for the results of the effector computation that are transmitted back to the engine). Finally, the diagram includes the interactions that affect the three slots described above (in the diagram, they are *L-Slot* for the Catalog slot, *A-Slot* for the Activate slot, and *R-Slot* for the Relay slot): it shows when the Activate slot and the Relay slot are filled with the appropriate plugins, respectively on the basis of the effector returned as a result of the Lookup stage, and as part of the Configure stage.

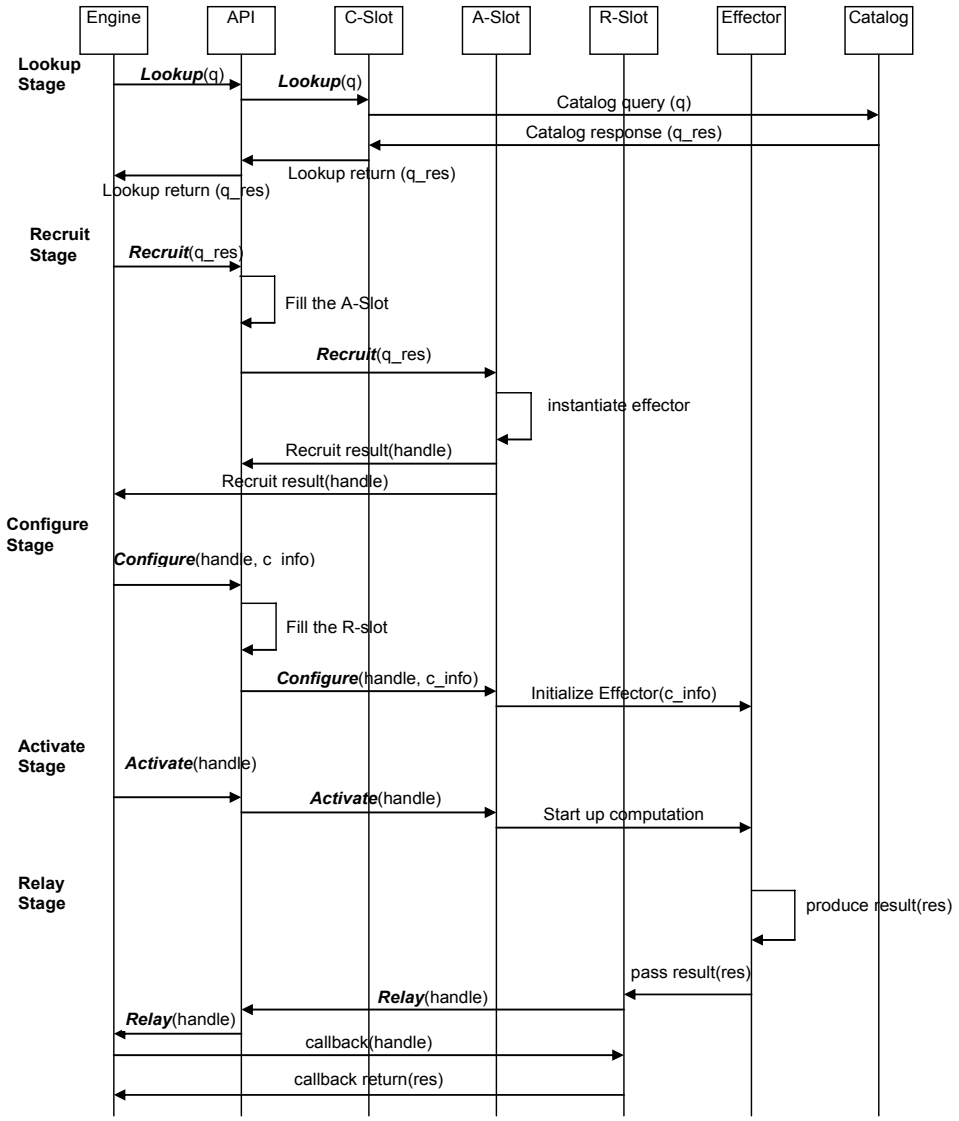

Figure 6: The various stages of effector execution.

5. Implementation

Workflakes has been completely implemented in Java, extending the Cougaar open source platform[4], and customizing it towards the requirements of orchestrating runtime adaptations. Figure 7 shows a view of the main classes in the framework used to implement the programming model described in Section 4.2. All of our experimental work to date has been done with effector technologies that can be programmed in Java; to integrate non-Java effectors, one can rely on the cross-platform interoperability facilities made available by Java, such as the Java Native Interface (JNI)[5].

The `ExecutableTask` interface is used to wrap all kinds of effectors and to expose the primitives of our model. The `GenericEffectorAPI` class implements that interface and is the container for the three slots that provide the interaction channels with each individual effector. It also includes code for the management of those slots: the Figure displays the abstract base classes for the three slots. In order to accommodate some effector technology, it is necessary only to specialize those three classes.

[4] See http://www.cougaar.org/.

[5] http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html.

With respect to the Activation and Relay slots, several specializations have been produced for a variety of interaction models and technologies (not shown in the Figure due to space limitations). We have variously experimented with event-based asynchronous messaging, SOAP-based remote invocation, native Java invocation, and of course our original mobile effectors based on Worklets.

For the Catalog Slot, a specialization is shown, i.e., a catalog that wraps a `Hashtable`. That is a utility, representing the simplest option that enables to directly associate the signature of a leaf task to the corresponding effectors.

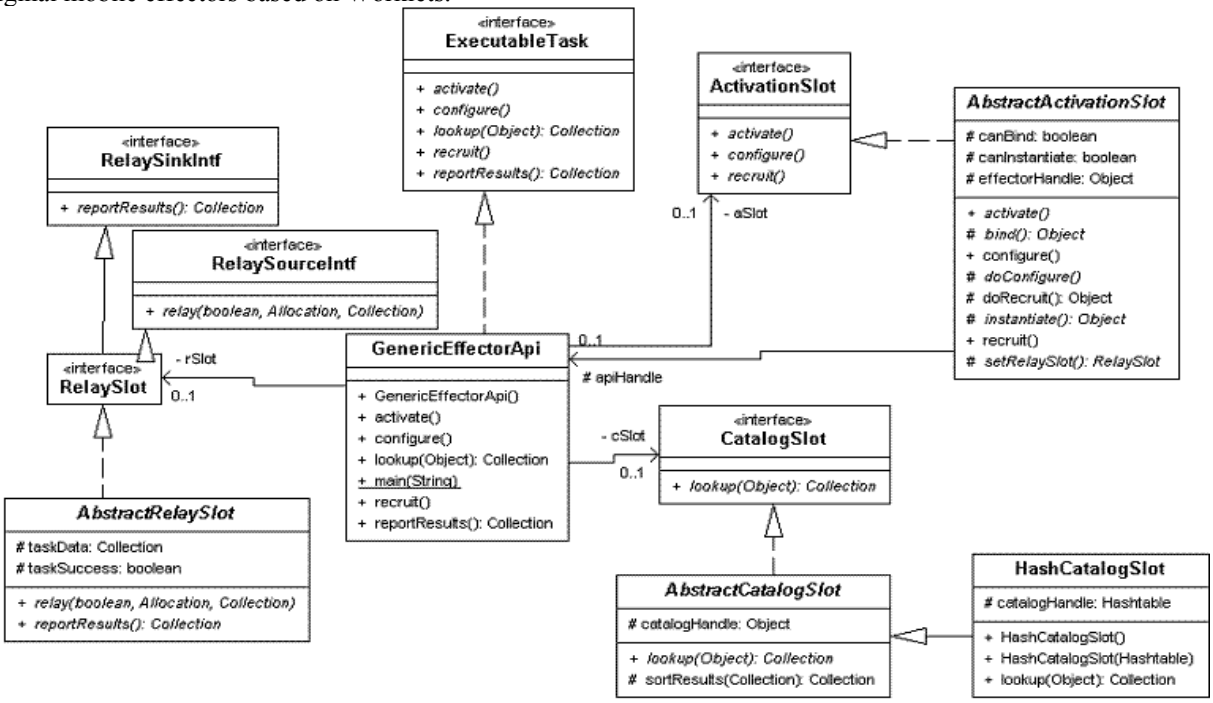

Figure 7: Java framework for the generic effector API.

Since our model allows to plug in other means to select effectors against tasks, much more sophisticated approaches can be incorporated: for example standard registries like UDDI[8], or ruled-based matching (as in [16]), or even semantic reasoning upon ontologies like those employed in Semantic Web contexts (e.g., using OWL [23]). Some of those techniques may promote a vision in which the execution of effectors on the part of the adaptation engine is not only abstract with respect to the mechanics of the interaction, but also fully *virtualized*. That is, the logic of the runtime adaptation would not need to specify any explicit binding information or be concerned about what effector computations are available: it could simply define – formally, but at a high level - the actions (correction, repairs, optimizations, etc.) that have to be taken at some point, their goals and their constraints. The catalog would then take care to "ground" those actions by choosing the most appropriate effectors, or micro-workflow of several effectors, that can satisfy the requirements of those actions. That kind of virtualization can be seen as a future objective, towards which a uniform abstraction for activating effectors and the separation between adaptation logic and effector computations represent necessary intermediary steps.

[8] http://www.uddi.org

6. Related work

A classic approach to try to impose a degree of uniformity in the interactions between a given system and multiple heterogeneous counterparts is through instrumentation of the latter. Instrumentation can enable the "creation", on board a managed resource, of an activation mechanism that is convenient from the point of view of the autonomic manager. Countless software instrumentation techniques exist, e.g., AIDE [17], ProbeMeister [18], and mediating connectors [19], with some others available commercially. Instrumentation, in general, has however a number of limitations. First, it requires some form of manipulation of the target code (either source or executable), which is not always feasible. Moreover, instrumentation techniques tend often not to port well across different computing platforms. Finally, as highlighted in our experience with the first release of Workflakes (see Section 0), inefficiencies and unnecessary design complications can arise when instrumentation imposes an interaction model that doesn't match or wrap well native adaptation provisions that abide to a different model. We have, however, successfully employed instrumentation for what MAPE-K calls sensors (and Rainbow calls probes), to provide input to the KX monitoring and analysis.

Another approach is to seek uniformity by restricting the adaptations supported to a few simple and "universal" operations: for example, as in some Recovery-Oriented Computing work [4], only shutdown and restart. That is not necessarily as limited as it may sound, since selective and controlled recursive micro-reboots on interdependent sets of elements of different granularity, have shown their effectiveness in particular with respect to self-healing, e.g., to improve the overall availability of complex software ensembles as diverse as operating systems (see [5]), or mission critical distributed systems (see [6]). However, in the case of systems-of-systems, the restart dependencies among components are not always known in advance. Moreover, even the relatively simple operation of rebooting a component is subject to the heterogeneity of platform and software technologies that is characteristic of systems-of-systems. Therefore, the restart operation does not equate to a single effector, but rather to a category of effectors with possibly diverse implementations, and would nevertheless require a programming abstraction like the one we propose, in order to be applied in a uniform way across heterogeneous parts of a system.

The above approaches – which are widely employed and successful to a degree in contexts where heterogeneity is limited - do not particularly promote any form of conceptualization or abstraction. However, for a general-purpose autonomic manager that must handle real-world applications involving systems-of-systems, the benefits of suitable abstractions to interact with managed resources are increasingly recognized.

For example, IBM, in its developerWorks Autonomic Computing Toolkit [15] organizes all interactions with its *touchpoints* onto the managed resource (effectors as well as sensors) around two classes: the first is called the `ManagedResourceTouchpoint`, and resides on the touchpoint; the second is called the `AutonomicManagerTouchpointSupport`, and represents its counterpart within the autonomic manager itself. Those two classes bind to each other via Java RMI, which imposes a client-server synchronous interaction model. The current release of the Toolkit provides only a single standardized operation for the interaction between those two classes: a `sendEvent()` method, available on the autonomic manager, which is suitable especially to implement transmission of monitoring data by sensors, and can possibly be used to provide functionality equivalent to our Relay primitive. Autonomic applications developed with the toolkit are free to specialize the `Touchpoint` and `TouchpointSupport` classes as they see fit, with no specific operational semantics nor generic programming model encouraged or enforced.

IBM also envisions a more comprehensive approach, demonstrated by its Autonomic Management Engine (AME) [16], also part of the Autonomic Computing Toolkit. AME provides a prototype implementation of a full-fledged MAPE-K loop. For the Execute part, AME adopts a plugin-based model to bind to and activate *action launchers*. Each action launcher represents code that is written *ad hoc* to effect changes on managed resources, but all expose to AME the same interface. Each action launcher is also accompanied by an XML descriptor, and a parser class that allows the engine to retrieve and use the specification of the action launcher contained in the descriptor. That specification describes the actions that the action launcher must perform (i.e., the method that it must call), in response to certain events of relevance (called *indications*) that can occur during the earlier phases of the MAPE-K loop. All action launchers implement the `ActionLauncher` Java interface. That interface provides three major operations:

- `setSpecification()` establishes the set of event/action rules for an action launcher;
- `satisfiesSpecification()` indicates whether there exists an action launcher of this type that satisfies a given specification;

- `handleIndication()` passes an event to the action launcher to enable execution of the corresponding action in accord with the loaded specification, and returns the action result.

There are several similarities between our model and AME. First of all, they are both based on the idea of pluggable activation mechanisms for a range of potentially very diverse effectors. Considering the programming abstractions that are supported, the descriptors of action launchers and the related classes and methods can be used for an effector catalog and as a means for Lookup. One difference is that our model is not tied to any given type of catalog or querying, since it has a specific slot for plugging in different catalogs. Our Activate primitive is similar to the `handleIndication()` method. However, that method assumes synchronous execution of the effector computation, since its result is relayed back as the return value of the method. Our model is more flexible since it can also accommodate asynchronous effector execution, by separating the Activate and Relay stages. Our explicit and separate Configure primitive, which is not present in AME, also contributes to flexibility and generality, since it can be used not only to pass parameters every time the effector computation is invoked, but also for any generic configuration needs of the more sophisticated effectors.

7. Conclusions

We propose an abstract programming model for effectors, for use in autonomic computing frameworks that aim to be relatively general-purpose and operate on heterogeneous systems-of-systems, as opposed to implementing self-management, self-healing, etc. capabilities solely for a specific new system or class of systems. This work was motivated by our previous experimentation with such a generic framework, where interfacing to pre-existing adaptation provisions was challenging. Our programming model and sample API implementation distinguish between activation of effectors and their pre-built adaptation provisions, separates the runtime adaptation logic from both activation and adaptation provisions, and exploits the natural progression of the work of effectors into their selection, recruitment, configuration and actual activation, as well as providing for flexible ongoing interactions while the effector performs its work.

We expect to continue refining the effector programming model and API implementations, and plan to apply this approach to a broad range of effector technologies. For instance, one of the authors is working with others on developing a KX-like autonomic infrastructure affording tolerance of intrusions, denial of service, and other security-related attacks [27], as well as on building an eCommerce-oriented testbed intended as a community resource for experimentation with autonomic computing technologies [28].

8. Acknowledgements

Kaiser's Programming Systems Laboratory is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063, and in part by Microsoft Research.

9. References

[1] M. Pezzini, "Composite Applications Head Toward the Mainstream", *Article Top View AV-21-1772*, the Gartner Group, October 16, 2003.

[2] Y.V. Natis, "Predicts 2004: Application Integration and Middleware", *Article Top View AV-21-8190*, the Gartner Group, December 19, 2003.

[3] I.B.M. Corporation, *An architectural blueprint for autonomic computing*, Technical Report, available at: http://www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf, I.B.M. Corporation, April 2003.

[4] G. Candea, and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel", in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[5] M.W. Shapiro, "Self-Healing in Modern Operating Systems", *ACM Queue*, 2(8), November 2004.

[6] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda, "Reducing Recovery Time in a Small Recursively Restartable System", in *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, USA, June 2002.

[7] P.N. Gross, S. Gupta, G.E. Kaiser, G.S. Kc and J.J. Parekh, "An Active Events Model for System Monitoring", in *Proceedings of the Working Conference on Complex and Dynamic Systems Architectures*, December 2001.

[8] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems", in *Proceedings of the 5th Annual International Active Middleware Workshop*, June 2003.

[9] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto, "An Approach to Autonomizing Legacy Systems", in

Proceedings of the Workshop on Self-Healing, Adaptive and Self-MANaged Systems, New York, NY, USA, June 2002.

[10] J. Parekh, G. Kaiser, P. Gross and G. Valetto, "Retrofitting Autonomic Capabilities onto Legacy Systems", *Journal of Cluster Computing*, Kluwer, in press.

[11] G. Valetto, and G. Kaiser, "Using Process Technology to Control and Coordinate Software Adaptation", in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003),* Portland, OR, USA, May 2003.

[12] G. Valetto, and G. Kaiser, "A Case Study in Software Adaptation", *in Proceedings of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, Charleston SC, USA, November 18-19, 2002.

[13] G. Valetto, G. Kaiser, and G.S. Kc, "A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems", in *Proceedings of the 8th European Workshop on Software Process Technology*, LNCS 2077, pp. 102-116, June 2001.

[14] G. Valetto, *Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology*, Ph.D. Thesis, Columbia University, New York, NY, USA, May 2004.

[15] I.B.M. Corporation, *Autonomic Computing Toolkit Developer's Guide 2nd edition*, available at: http://www-106.ibm.com/developerworks/autonomic/books/fpy0mst.htm , I.B.M. Corporation, August 2004

[16] I.B.M. Corporation, *Autonomic Management Engine Developer's Guide v.1.5*, I.B.M. Corporation, 2004.

[17] P.W. Gill, *Probing for a Continual Validation Prototype*, MS Thesis, Worcester Polytechnic Institute, May 2001.

[18] P. Pazandak, and D. Wells, "ProbeMeister: Distributed Runtime Software Instrumentation", in *Proceedings of the 1st International Workshop on Unanticipated Software Evolution*, June 2002.

[19] R.M. Balzer, and N.M Goldman, "Mediating Connectors: A Non-ByPassable Process Wrapping Technology", in *Proceedings of the DARPA Information Survivability Conference & Exposition, Vol. 2*, January 2000.

[20] S. Chen, A. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure", *IEEE Computer,* 37(10), October 2004.

[21] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments", in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, December 2002.

[22] S.E. Dossick, and G.E. Kaiser, "Worklets for Adaptive Workflow", position paper in *CSCW-98 Workshop: Towards Adaptive Workflow Systems*, November 1998.

[23] L. McGuinness, and F. van Harmelen (eds.), *OWL Web Ontology Language Overview*, a W3C Recommendation, February 2004, http://www.w3.org/TR/owl-features/

[24] H. Liu, M. Parashar, and S. Hariri, "A Component-based Progarmming Framework for Autonomic Applications", in *Proceedings of the 1st IEEE International Conference on Autonomic Computing*, New York, NY, USA, May 2004.

[25] G. Eisenhauer, and K. Schwan, "An Object-Based Infrastructure fir Program Monitoring and Steering", in *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, August 1998.

[26] S.M. Sadjadi, and P.K.McKinley, "Trasparent Self-Optimization in Existing CORBA Applications" in *Proceedings of the 1st IEEE International Conference on Autonomic Computing*, New York, NY, USA, May 2004.

[27] A. Keromytis, J. Parekh, P.N. Gross, G. Kaiser, V. Misra, J. Nieh, D. Rubenstein, and S. Stolfo, "A Holistic Approach to Service Survivability", in *Proceedings of the 1st ACM Workshop on Survivable and Self-Regenerative Systems*, October 2003

[28] Y. Diao, J.L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-managing Systems: A Control Theory Foundation", in *Proceedings of the IEEE Workshop on Engineering of Autonomic Systems*, April 2005.

[29] C. Poellabauer, H. Abbasi, and K. Schwan, "Cooperative Run-time Management of Adaptive Applications and Distributed Resources", in *Proceedings of ACM Multimedia*, Juan-Les-Pins, France, October 2002.