Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation

Giuseppe Valetto
Telecom Italia Lab, Columbia University
Via Reiss Romoli 274
10148, Turin, Italy
+39 011 2288788
Giuseppe.Valetto@tilab.com

Gail Kaiser
Columbia University
Department of Computer Science
New York, NY 10027, United States
+1 212 939 7081
Kaiser@cs.columbia.edu

ABSTRACT

We have developed a model and a platform for end-to-end run-time monitoring, behavior and performance analysis, and consequent dynamic adaptation of distributed applications. This paper concentrates on how we coordinate and actuate the potentially multi-part adaptation, operating externally to the target systems, that is, without requiring any a priori built-in adaptation facilities on the part of said target systems. The actual changes are performed on the fly onto the target by communities of mobile software agents, coordinated by a decentralized process engine. These changes can be coarse-grained, such as replacing entire components or rearranging the connections among components, or fine-grained, such as changing the operational parameters, internal state and functioning logic of individual components. We discuss our successful experience using our approach in dynamic adaptation of a large-scale commercial application, which requires both coarse and fine grained modifications.

Categories and Subject Descriptors

D.2.4, D.2.5 [**Software Engineering**]: Software/Program Verification – *reliability, validation;* Testing and Debugging – *diagnostics, error handling and recovery, monitors.*

General Terms

Measurement, Performance, Reliability.

Keywords

Dynamic Adaptation, Dynamic Reconfiguration, Perpetual Testing, Distributed Systems, Software Process Enactment, Workflow, Coordination, Agents, Mobile Code.

1. INTRODUCTION

Distributed computing is becoming a commodity. Users rely upon distributed systems for a number of value-added services that pervade their everyday lives, from work, to commerce, to entertainment, to social interaction. Services such as Web-based collaboration suites, electronic B2B and B2C, on-demand multimedia content provisioning, and ubiquitous personal messaging are built on top of a networking infrastructure as distributed systems, often constructed by composition. The complexity of the behavior and interrelationships of these "systems of systems" becomes increasingly harder to manage, and it is in general impossible to analyze all facets of their functional and non-functional characteristics in advance. That aggravates the critical problems of managing the provisioning of the service and maintaining the intended application-level, "soft" quality of service (QoS). In order to resolve poor performance or failures, often service is interrupted, the underlying application is taken down (at least in part), and the spiral of software lifecycle iterates back to the installation or deployment phase, and sometimes even to earlier development phases.

While such a drastic response may be obligatory at times, it is desirable when possible to resolve problems with lesser impacts and costs – *while the system is running and without bringing it down.* Our research addresses the manageability and soft QoS of distributed applications, by using external automated facilities for their *dynamic adaptation*. By that term we mean any controlled set of actions aimed at modifying, at runtime, the structure, behavior and/or performance of a target software system, typically in response to the occurrence and recognition of some adverse condition(s).

Our approach centers on process-based coordination of mobile agents. An external automated entity dispatches onto the target system one or more mobile software agents to carry out the needed dynamic adaptation. Each agent travels to and then operates within an execution environment that is co-located with a target component or actualized connector. The only preparation required of the target is its instrumentation or wrapping with a set of *host adaptors*: those are employed to mediate agent access to portions of the internal state and behavior of target units, exposed for dynamic adaptation purposes. In certain cases, a single agent can traverse a set of target components and connectors in sequence; in many others a community of agents needs to perform activities simultaneously on multiple target units. In at least the latter case, a coordination mechanism is needed. For those coordination purposes we employ process technology: we design the concerted adaptation as a process among these software agents, and employ a process enactment (or workflow) engine to orchestrate the agents at runtime.

Others have also proposed to exploit results from process or agent research, or their combination, to control the behavior and performance of a running application, either as a promising generic coordination mechanism [4] [6], or attacking specific aspects of dynamic adaptation: for instance, dynamic service composition and management [12], deployment [10], self-modification [13], and "perpetual testing" (see http://www1.ics.uci.edu/~djr/edcs/PerpTest.html).
The major distinctions of our approach are the decoupling of the adaptation facilities from the target system and further the independence from any underlying support framework, which together enable a wide spectrum of adaptation, with varying granularity, from the configuration of the target architecture as a whole, to the pairwise interactions between components, down to the tweaking of the inner state of single components.

In a preliminary paper [3], we introduced our concepts, model and system – called Workflakes - for applying highly automated process- and agent-based dynamic adaptation facilities "from the outside" of a given target system. At that time we had implemented only a proof of concept of Workflakes, using a "mockup" of the University of Massachusetts Juliette decentralized process system [5]. We have more recently completed and evaluated an operational implementation based on the Cougaar open–source decentralized system as the runtime process engine (see http://www.cougaar.org).

Workflakes has been developed as part of the Kinesthetics eXtreme (KX) infrastructure [21] for the Continual Validation of distributed systems [2]. KX defines a "meta-architecture" that superimposes upon generic distributed applications an OODA (Observe - Orient - Decide – Act) monitoring and control loop - also known as monitor/analyze/respond. KX features instrumentation and probing of target components and connectors, application-aware functional and extra-functional gauging derived from the analysis of posets of probe-generated messages, and automated actuation. Workflakes takes within KX the roles of automated decision support and actuation coordination, that is, choosing and performing dynamic adaptation processes, as deemed necessary on the basis of input by the KX gauges that diagnose the state of the target system.

In this paper, we elaborate on our revised Workflakes architecture, as well as evaluate the model's merits and limitations based on experience gained by putting it to test on a real-world, mass-market Internet service. We also discuss a series of conceptual, architectural, technological and applicability aspects that we have discovered to be key to the effective exploitation of dynamic adaptation capabilities.

2. THE WORKFLAKES APPROACH

2.1 Background

Dynamic adaptation as a discipline responds to the observation that a running system is likely to be subject to condition variations that can be external (i.e., in its execution environment) or internal (e.g., degradations, faults, other forms of malfunctioning, or even just consumption of resources over time, etc.). Such variations can influence the provided service in a number of ways, which can only be partially predicted and adjusted for from within the application code itself, e.g., by striving to make the code fault tolerant *a priori*.

This leads to a vision of monitoring, analysis, and response facilities that are somehow intertwined or connected with the target system, to promptly detect changes in its internal and external conditions, and then decide upon and exert appropriate adaptations upon it. Such a vision is often presented as a feedback (reactive) and/or feed forward (proactive) control loop. However, one major crucial issue is how to achieve the decision and action blocks of the control loop. "Standard" APIs have been proposed for probing [25] and gauging [26], but not yet for response – which is less well understood.

Our process-based coordination and agent-based actuation, follows three basic principles: it enables to keep disjoint the adaptation target from the adaptation facilities, which operate "from the outside" with respect to the target; it maintains a clear-cut separation between coordination and computation concerns (as advocated in [1]), to be handled, respectively, by the process engine and the agent infrastructure; and it provides a foundation for reasoning upon any explicit knowledge about the target system's architecture, e.g., as codified in an Architecture Description Language (ADL), to tailor the dynamic adaptation process

with relevant resources, facts and policies [19] [30]. We believe that those principles provide a nice degree of generality and flexibility to our approach: on the one hand they do not impose too many prerequisites on the nature, architecture or implementation of the target system, and on the other hand they are not excessively constraining with respect to the reach and capabilities we can achieve.

In the remainder of this section we discuss in depth the dynamic adaptation approach of Workflakes within the larger context of our KX architecture, and provide some details about the most relevant aspects of the current Workflakes implementation.

2.2 The KX Architecture and Workflakes

KX defines an architecture that is meant to impose an end-to-end OODA loop upon generic distributed target applications. Its goal is Continual Validation, that is, ensuring that given critical functional and extra-functional parameters of the running target are preserved throughout its operation. Continual Validation is not limited to passive monitoring of such parameters, but can be intrusive and therefore adapt the target system either in a reactive (that is, for repair, aiming at restoring the nominal parameters) or a proactive fashion (e.g., for plan-ahead reconfiguration, such as scaling).

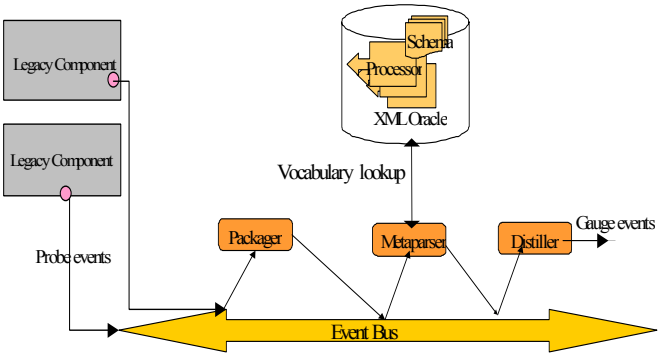

Figure 1: Monitoring and analysis in KX.

The KX *SmartEvents* model [21] fulfills the Observe and Orient OODA steps with a semantics-aware pipeline architecture (see Figure 1), built on top of a publish/subscribe event bus [23]. Streams of XML-formatted events are generated by probes placed within the target code (e.g., via *active interfaces* [22]). Probe events (**Observe**) are propagated to subscribing *Event Packagers*, which variously manipulate the contained information to extract bits of semantic content, with the support of a flexible XML parsing engine (the *Metaparser*) and of a repository of known XML schemas and dynamically applicable transformations (the *XML Oracle*); Event Packagers can also convert probe events in other formats used within the common framework. The contents of multiple streams of events get finally correlated by rule-based *Event Distiller*s to derive application-dependent high-level facts. Distillers function as *gauges* (**Orient**), i.e., they measure, express and report knowledge about the state of the target system: its health, compliance with the parameters subject to Continual Validation, and any other conditions formulated for input to dynamic adaptation decisions.

In the KX context, Workflakes is complementary (and orthogonal) to the SmartEvents model, and is concerned with the Decide and Act part of the OODA loop. KX gauges provide Workflakes with continually updated knowledge about various aspects of the target system, some of which can trigger a dynamic adaptation process (**Decide**). Workflakes' enactment typically results in the end in the orchestrated dispatching of mobile agents that travel to the target components and carry out appropriate local computations (**Act**). As agents terminate their duties, they report back their outcomes to the process engine, possibly triggering other process fragments.

The interactions between KX gauges, the Workflakes process engine, and the actuator agents are entirely automated, that is, no human intervention is needed to close the loop; however, either gauges or Workflakes (or both) may be configured to report their status externally, and human-oriented steps may of course be inserted in the process, for instance to allow for auditing or confirmation before initiating a particularly critical adaptation.

2.3 Workflakes in Detail

The current implementation of the Workflakes system, embodying the dynamic adaptation principles and features discussed above, relies on the integration of the worklets mobile agent platform [24] [3] with a Cougaar-based process engine specialized for agent coordination.

Worklets are multi-transport mobile agents written in Java. Each worklet acts as a carrier of one or more computational snippets, named *worklet junctions*. Agent transport facilities are provided by Worklet Virtual Machines (WVMs) residing at the origin and destination. WVMs also define an execution environment for worklets, exposing via a *host adaptor* an interface to match the computational capabilities of incoming worklet junctions, and to enable access to portions of the state and behavior of the target component. A host adaptor provides a degree of abstraction and information hiding to junctions that implement a given kind of functionality, thus mediating any idiosyncratic properties of different native implementations of target elements, and presenting them consistently for adaptation purposes. A junction is deposited at each WVM corresponding to a step of the worklet's programmable trajectory, and is executed there. The execution of a deposited junction can be micro-controlled via a *worklet jacket.* Jackets are interpreted by the WVM and can regard a variety of optional aspects, such as junction priorities, pre- and exit conditions, subsumption of a previously (or possibly subsequently) arriving worklet,

repetition, suspension or resumption at time intervals or upon other conditions, and so on. Partial or final results of the junction execution can in some cases influence the remaining trajectory of the corresponding worklet. Further details are outside the scope of this paper.

The Workflakes process enactment engine is a modification of Cougaar, specifically constructed to dispatch, coordinate and receive worklets, and exploits this capability in two fundamental ways. First, worklets originate from WVMs that are incorporated within the decentralized task processors (*clusters*) of Cougaar, and travel to the target components to be adapted. One of the major responsibilities of the process is therefore to decide what are the most appropriate *actuations*, i.e., (sets of) worklet junctions to be dispatched, in order to fulfill a given dynamic adaptation task. For that reason, the process engine integrates a repository of junction descriptions and a junction factory, so that *actuation junctions* can be treated as first-class process resources that need to be assigned and instantiated by the process logic according to the task at hand. Furthermore, Workflakes uses worklets also to dynamically load process definitions onto task processors, either with a pull or a push modality. Specific *process definition junction* types have been implemented to that end, which enable the dynamic deployment via worklets of portions of the process to the most convenient task processor for execution.

Such process delivery may for example be used in the pull modality to incrementally retrieve process fragments when requested to handle certain gauge events, or in the push modality for on-the-fly process evolution across a distributed Workflakes installation.

The Workflakes implementation relies on the Cougaar concept of *plugins*. Plugins allow to customize the functionality of Cougaar clusters by inserting components that implement a particular logic or a specific capability,

and get access to the process state and other data via the Cougaar distributed blackboard. As shown in Figure 2, we use several Logic Data Model (LDM) plugins to import and convert KX gauge events in terms of process facts, to maintain the target system architectural knowledge, and represent the junction repository; an Expander plugin to load process definitions and spell them out as concatenations of tasks; an Allocator plugin to map tasks to actuation junctions as needed; an Executor plugin that employs the resident WVM to handle worklet instantiation and shipment.

In Workflakes, plugins are initially idle and devoid of any hardcoded logic related to any particular process; for that reason, we call them *shell plugins.* The set of shell plugins launched within a cluster at start time is therefore merely indicative of the kinds of service and functionality that the cluster is meant to offer within the overall Workflakes engine. Shell plugins can get activated at any time via the injection of the abovementioned process definition junctions; from that moment on, they acquire a definite behavior that is coded in those junctions, and start taking part in the enactment of the process in a modular fashion.

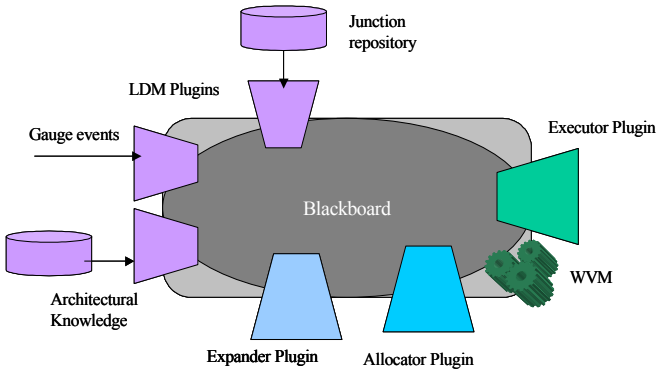

Figure 2: Cougaar Representation of a Workflakes Task Processor.

3. DYNAMICALLY ADAPTING AN INTERNET MASS-MARKET SERVICE

3.1 Application Context and Results

We have been experimenting with a multi-channel instant messaging (IM) service for personal communication, which operates on a variety of channels, such as the Web, PC-based Internet chat, Short Message Service (SMS), WAP, etc. The service is currently offered on a 24/7/365 basis as a value-added service to thousands of users. It is implemented using J2EE and relational databases.

Our goals are twofold: We want to achieve *service optimization*, with respect to the overall QoS perceived by the end users, which can be achieved by adapting the functional and/or extra-functional characteristics of the various service components as well as their interactions. The requirements include on-the-fly architectural modification for scalability, in response to the detection of host- and component-specific load thresholds; and on-the-fly re-configuration of the server farm hosting the service. We also aim to support *dynamic monitoring and control of the running service*, that is, simplify and resolve a number of concerns, related to the continuous management of such a complex distributed application. These requirements involve automated deployment of the service code; automated bootstrapping and configuration of the service; monitoring of database connectivity from within the service; monitoring of crashes and shutdowns of IM servers; monitoring of client load over time; support for "hot" service staging via automated rollout of new versions and patches.

Our study is organized as a series of iterations, which aim at incrementally fulfilling requirements originating from needs

discovered in the field by the service provisioning organization, and elicited from the application development and maintenance team. For each iteration, results are first evaluated in the lab; then new requirements are accepted for the next iteration, while the results produced are delivered and put to test on the field. We have now reached the end of the first iteration.

The service runtime environment consists of a typical three-tiered server farm[1]: a load balancer provides the front end of the service to all end users and redirects all client traffic to several replicas of the IM components, which are installed and operate on a set of middle tier hosts. The various replicas of the IM server all share a relational database and a common runtime state repository, which make up the back end tier, and allow replicas to operate in an undifferentiated way as a collective service.

As shown in Figure 3, some of the IM servers may provide additional facilities, which handle access to the service through specific channels, such as SMS or WAP, and interoperate with third-party components and resources that remain outside of the scope of the service, e.g., the gateways to the cell phone communication network. Those extra facilities wrap the core IM functionality in various ways. Given this kind of modularity, it is possible to achieve the continual validation of all of the service components in a server farm in a rather consistent way, by applying probing, gauging and actuation in the first place on the core IM server components (as we did for the first interation), and extending them as needed to validate any critical features of the additional wrapping components.

The current implementation successfully addresses all of the requirements using a specific set of probes, gauges and actuators on top of the common facilities provided by the KX platform. Workflakes addresses the manageability requirements by taking responsibility to correctly initiate the service software via a set of worklets and a completely automatic process, which replaces the original manual procedures and scripts for the installation, deployment and bootstrapping of service components. This process is enabled by the explicit codification – among the logic and data loaded at startup onto the Workflakes bare engine – of knowledge about the service architecture and the runtime environment of the server farm that hosts the service. Furthermore, Workflakes addresses QoS requirements, responding to scalability needs with a reactive process that orchestrates new deployments of IM servers, and opportune reconfiguration of the load balancer.

After startup, Workflakes selects one of the hosts from its internal representation of the runtime environment of the server farm and sends out a worklet to it. This worklet executes bootstrapping code for the IM server and configures it with all the necessary parameters (such as the JDBC connection handle to the DBMS, the port numbers for connections by clients and other IM servers, etc.). Notice that not only the configuration information, but also the executable code of the IM server is deployed and loaded on demand from a code repository made available to the incoming worklet, taking advantage of a code-pulling feature of the Worklets agent platform; that allows to do away with any preliminary installation of the application code on all machines taking part in the server farm, a fact that greatly simplifies on its own the bootstrapping, staging and evolution of the service. (This approach is also followed in Software Dock. [10])

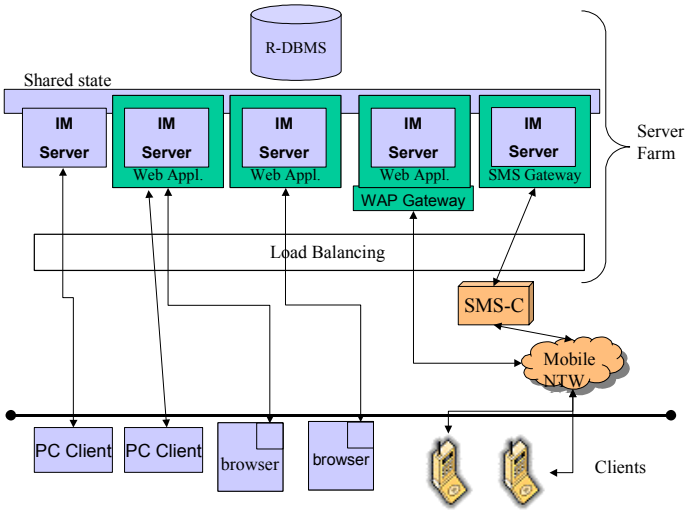

Figure 3: The IM service architecture.

When the worklet instantiates an IM server, certain probes are activated to track its initialization. In the event of an unsuccessful initialization, the likely cause is inferred by KX on the basis of the probes' output and reported to a dashboard GUI for the human management of the service, as well as to the process, which may react by deciding to try to bootstrap an IM server on the same machine again, or on another one. When the initialization is successful, instead, the process dispatches another worklet onto the load balancer, to instruct it to accept traffic for the IM service and pass it to the initialized server at the right host address and port.

Following the initial bootstrapping phase, Workflakes takes a reactive role, while the KX platform starts monitoring the dynamics of service usage. Certain probes and gauges are activated to track user activity, such as logging in and out of the initialized server. IM servers have an associated load threshold, which in the case of this particular service is most simply expressed in terms of the number of concurrently active clients in relationship with the memory resources of their host. When that threshold is passed,

[1] A very similar example is employed in [19] as a generic, typical architectural style for Internet service provisioning.

Workflakes is notified and reacts trying to scale the service up. It selects some unused machine that is still available in the server farm, and repeats the bootstrapping process fragment on that machine, including the updating of the load balancer configuration. Of course, this scaling up policy can be repeated as many times as the number of machines in the server farm, Notice that the worklet bootstrapping a new IM server must carry an extra piece of configuration, that is, the indication of some other alive IM server, in order to enable the new instance to synch up with the IM server pool and its shared state, and function as an undifferentiated replica. After a successful initialization of an IM server, client requests begin to arrive at that server via the re-configured load balancer, achieving scalability and thus enhancing reliability and performance overall. Other conditions that can prompt new deployments and bootstrapping of IM servers include failures of some existing server replicas, which are inferred from specific sequences of probe events.

The logic described above is enacted by Workflakes and reified by actuator worklets carrying code for the IM server bootstrap and the load balancer reconfiguration, and feeds off KX gauging events for the reactive parts. It effectively fulfills our deployment, bootstrapping and scalability requirements, supporting at once both the service monitoring and control goal and the service optimization goal in a flexible and dynamic way. Minor changes to the bootstrapping process sketched above enable any service evolution campaign to be expressed as a process with tasks that withdraw from the load balancer old server instances (thus disallowing new traffic to be assigned to them), shut them down when traffic is absent or minimal, and conversely start up, register on the load balancer, and thus make available to users other server instances with the new code release instead.

3.2 Evaluation and Lessons Learned

To date, we have been confronted primarily with what are nowadays typical *application management* [7] concerns for a complex, real-world, mass Internet service, and only secondarily with soft QoS assurance requirements such as scalability. (Notice that this reflects only the set of requirements we have received for the first iteration, rather than the reach of our dynamic adaptation approach.) The granularity of the adaptation policies coordinated by Workflakes has consequently remained relatively coarse, covering aspects such as deployment, instantiation and initialization of service components, and only in some cases finer-grained aspects - notably the reconfiguration of the load balancer. We compare here these more coarse adaptation features with what is available from commercial application management software packages, which typically exploit generic management facilities exposed by the host, such as starting and shutting down an operating system process, inspecting its state and resource usage, and so on.

Target awareness: our probes, gauges and actuators are derived from generic code instrumentation templates that are then instantiated with a limited amount of situational logic (around 15 Java code lines for probes, around 10 for adaptors, less than 100 for worklets in this case), rather than general-purpose (i.e., derived from OS-level monitoring facilities). Thus they provide considerable expressiveness and flexibility in predicating on fine-grained internal characteristics of the application, such as expressing load thresholds in terms of the number of concurrent users, and weighing those thresholds in terms of host capabilities.

Enhanced active management: commercial application management is still primarily concerned with collecting and reporting data. Active management [7] [8] is less mature, still limited to basic actuations, such as scripted instantiations and shutdowns of components, and not particularly customizable or fine-grained. Workflakes process- and agent-based approach emphasizes active management, enabling effective closing of the control loop upon the managed system, either reactively (feedback) or proactively (feed forward).

Highly automated management: here is where the benefit of a full-fledged process engine becomes most evident. Most application management tools report warnings, alarms and other information derived from the monitoring facilities to a dashboard console, where some knowledgeable human operator can recognize situations as they occur, and take actions if needed, with a very limited amount of support, guidance and automation on part of the management platform. Our approach allows to capture the knowledge inherent in the process of managing applications, formalize it as an enactable coordination "program", and automatically execute it. Our platform – in turn - provides the technical means to fulfill that promise. Therefore we offer a high level of guidance, coordination and automation to enforce what is a complex but many times largely repeatable and codifiable process.

Employing Workflakes for application management in the lab shows higher levels of automation, flexibility and repeatability to the deployment of the target service, all benefits which can now be extended to the field. Previous manual procedures had been complex, error-prone and effort-consuming, requiring even several days of work by dispatched application specialists. Moreover, whenever a new installation or release had to be undertaken, the amount of labor did not decrease much, independently from the increased experience of the deployment team. With Workflakes, a completely new installation comports only changing those few elements in the process knowledge base that capture the runtime environment.

Another set of observations from this first iteration is related to the impacts on development. Our team embraced the working hypothesis that no code change in the target system itself was to be made, or requested to the service development team. In other words, we positioned ourselves past the end of the development phase of the project life cycle and just prior to the deployment phase. We treated the target service as a complete legacy, although a legacy for which all the specifications, software artifacts and accumulated project knowledge did happen to be available to us. We were able to superimpose all the needed dynamic adaptation features without interfering with the development process. Notice that another kind of legacy was also addressed: the load balancer we actuated is also commercial software, written in Java by IBM as part of their Web Sphere Edge Server suite. It allows for limited programmatic extensions, and – of course – only the extension API was available to us. We were able to work within these limitations to devise process tasks and actuators that successfully instructed it to accept traffic for certain hosts and ports on the fly.

A relatively low level of effort was needed to fulfill the goals of our first iteration. The process and actuators for Workflakes consisted of about 1100 lines of Java source code, while KX probes and gauges added another 450 lines (Java and XML). This should be compared with a size of about 36000 lines of Java, JSP and SQL code for the subset of the service components that are currently subjected to probing, gauging and actuation (that excludes the commercial load balancer, for which we cannot produce figures).

We observed, however, that the amount of effort to set up and analyze the target system and its behavior was greater than the effort spent in developing the corresponding solution. In addition, a substantial portion of the code we wrote is intended to capture architectural information, relationships and inferences and represent them to Workflakes and KX. That suggests that being able to capture, describe and expose in an abstract and machine-readable way knowledge about the target architecture may be possibly the single most powerful enabling factor for automated dynamic adaptation of software, which further motivates our plan to exploit formal ADLs in our follow-up research (see Section 5).

4. RELATED WORK

Workflakes advocates the separation between the target system and the facilities used for its dynamic adaptation, at the conceptual, architectural and implementation levels, as a means to simplify the tasks of designing, building, maintaining, evolving and activating/deactivating dynamic adaptation features with minimal disruption to the development or operation of the target. That contrasts with fitting (or retro-fitting, in the legacy case) the target with built-in code, e.g., internal fault tolerance provisions, which for component-based systems typically affect individual components in isolation, rather than working across the whole target system in a coherent, coordinated manner. For example, [18] employs a rule-based inference engine for decision support in application-level QoS assurance. Like in Workflakes, it is a separate coordination entity that enacts scripted management policies by controlling a set of computational actuators. However, the coordinator and actuators must both be embedded with each component subject to adaptation. That requires much heavier instrumentation of the target system, and results in a lack of flexibility in the adaptation actions that can be carried out without re-building the target.

A natural extension of embedded fault tolerance is represented by built-in mechanisms that offer system-encompassing dynamic adaptation capabilities at the price of designing and building the target system as a whole around such facilities. Such an approach often advocates the adoption of middleware with native dynamic adaptation properties. For example, Conic [17], Polylith [15], and 2K / dynamicTao [9] [16] offer a set of reconfiguration primitives as a premium for applications built with and operating on top of those environments.

With respect to our reference problem in Section 3, such an approach would require that all service components, plus all of the server farm's support features, such as clustering, load balancing, state and data persistence and distribution, were assembled from the start with the dynamic adaptation middleware and its requirements in mind. Besides posing a considerable barrier to the dynamic adaptation of legacy software and violating the aforementioned separation between target system and dynamic adaptation facilities, middleware-based dynamic adaptation also introduces a very strong dependency of the former upon the latter, where the spectrum and granularity of possible adaptations is effectively restricted by the set of primitives made available by the middleware. Similar observations apply also to those works that exploit the characteristics of established component frameworks to facilitate certain aspects of dynamic adaptation. For example, BARK [14] deals with dynamic (re-)deployment limited to the Enterprise Java Beans component model.

Several approaches effectively separate coordination from computation, as we do, and some even use process-based capabilities to define, enact and coordinate dynamic adaptation. However, all those we are aware of still employ a middleware-like paradigm and exert the coordination "from the inside", that is, on the target's own computations. For example, [13] introduces the concept of Containment Units, as modular process-based lexical constructs for defining how distributed applications may handle self-

repair and self-reconfiguration. Containment Units define a hierarchy of processes that predicate on constraints and faults, and take action in terms of component substitution and resource reallocation to handle faults within the defined constraints. The enactment of Containment Units is under the responsibility of a process engine that is integral to the system being adapted, and proceeds by directing changes on the target components, which must also be process-aware to some degree.

PIE [11] is another example of a process-based middleware, which supports the assembly and management of a *federation* of components. PIE adds a control layer on top of a range of inter-component communication facilities. The control layer implements process guidance via handlers that react to and manipulate the communications exchanged by the components in a federation. Dynamic adaptation is thus limited to the reconfiguration of the service architectural connectors and is carried out by plugging in appropriate handlers, as directed by the process. In our example, such reconfiguration could be used to adapt the load balancing features and the server-to-server interactions.

TCCS [12] has considerable similarities with Workflakes, since it employs its process engine to direct the work of agents, whose execution in turn carries out the dynamic adaptation tasks. That is consistent with our idea of separating coordination and computation concerns; however, TCCS is the epitome of the middleware approach, since the complex services obtained with it as dynamic compositions cannot exist independently from its process and agent-based framework: TCCS carries out the scheduling and sequencing of all interactions between the service components. Also, the scope of the dynamic adaptation is limited, as in PIE: finer-grained adaptation of the internal computational logic of components remains inaccessible. For example, TCCS, could hardly modify on the fly the routing table of a legacy load balancer product, as we do, to handle the dynamic addition or removal of server replicas.

In contrast to all of the above, Workflakes remains conceptually independent from any support framework, and hence quite general with respect to the reach, granularity and kinds of dynamic adaptation that it can exert. The target of dynamic adaptation is regarded as legacy; the process engine does not do anything directly to that target system, but only deals with dispatching and coordinating the mobile actuator agents. The only requisite with respect to target components and connectors is their wrapping or instrumentation with the host adaptors. Finally, the use of mobile agents guarantees that new forms of adaptation computations can be easily deployed at any time with minimal disruption to service operation.

The ABLE group at CMU also proposes external dynamic adaptation facilities aimed at automated system repair, which are driven by explicating and reasoning about multi-faceted architectural knowledge about the target system [19]. That knowledge is captured in a set of ADL descriptions, which enable to express repair tasks as sets of changes to the architectural model. Those tasks are then delegated to an *Environment Manager* in charge of actuating those changes upon the target system. However, such actuation is currently coded ad hoc for each target application. ABLE and Workflakes can therefore nicely complement each other[2]: Workflakes can take the role of the Environment Manager and at the same time derive its dynamic adaptation processes from an explicit, formal, machine-readable notation.

Also other architecture-driven approaches aim at guiding and controlling the implementation and its configuration to some degree. Some, like ArchJava [27] strive to maintain architectural integrity of an application throughout code development iterations: therefore, they operate in a context disjoint from dynamic adaptation. Others include explicit mechanisms to predicate upon runtime instantiations of the architectural model. For example xADL 2.0 [28] includes provisions not only to bind a model to the artifacts used for its implementation (such as Java classes and artifacts), but also to track down dynamically the runtime instances of such artifacts and link them back to the original elements of the model. A feature of that kind can greatly simplify the interaction between an ADL environment and a dynamic adaptation platform such as Workflakes, because inherently unifies the view of the former over the model and the view of the latter over the "real world" that needs to be adapted.

Willow [20] finally, is a proposed architecture for the survivability of distributed applications, analogous to our KX vision of superimposed OODA capabilities. It is meant to address fault avoidance, fault elimination and fault tolerance. In particular, Willow can implement reactive as well as proactive dynamic adaptation policies, which are driven by codified architectural knowledge, and enacted via a process-based mechanism exerted upon their previously developed Software Dock (re)deployment engine [10]. It appears, however, that the Willow process facility would be restricted to coarse-grained reconfigurations, such as replacing, adding and removing entire components, perhaps even entire nodes, from the target application, while it would presume conventional embedded approaches for local fault tolerance.

[2] In fact, we are working with the CMU authors to integrate their ADL-based approach with Workflakes as an enhancement to our KX platform, not discussed here.

5. CONTRIBUTIONS AND FORECAST

Workflakes not only brings in a considerable level of automation and dynamism to the adaptation of complex applications and value-added services. The principles at its foundation also provide additional benefits. The decoupling of the target system from the dynamic adaptation facilities has proved to be an advantage during the development, test, and maintenance of dynamic adaptation features for a service. The clear conceptual separation between coordination and computation concerns highlights the process element of dynamic adaptation, thus opening the way to auditing, measuring and continuously improving dynamic adaptation processes, and posing a good basis for enabling the reuse of both processes and agents. The independence from any underlying specialized support frameworks or computing platforms results in a wide dynamic adaptation spectrum, in terms of both granularity and applicability.

There are multiple directions for future research. On the applicability side and with the next iterations over the chosen example service, we want to pursue further the issue of application-level QoS assurance, and would like to evaluate the impact of dynamic adaptation processes (and their reuse, assessment and improvement) on service quality levels. We also plan to explore other application domains, with the goal to extend the reach of Workflakes, for example to dynamic and controlled service composition. By that term we intend on-the-fly lookup, recruitment and orchestrated invocation of service components, functional matching and impedance resolution between heterogeneous components with appropriate worklet-mediated connectors, and subsequent control and adaptation throughout the life cycle of the composed service. To that end, we plan to experiment Workflakes in conjunction with the Service & Contract language developed for Cougaar [29]. Finally, we will integrate dynamic adaptation processes with architectural descriptions, to achieve "round-trip", consistent adaptation support, driven from the service model all the way down to the service implementation, and going back up with updates to the model that would reflect the modifications effected onto the running service as a result of actuation.

6. ACKNOWLEDGMENTS

We would like to thank Gaurav Kc for his ongoing development of Worklets (contact gskc@cs.columbia.edu to download the latest version of Worklets), Nathan Combs for help with Cougaar, George Heineman for help with instrumentation issues and techniques for KX and Worklets, David Garlan, Bradley Schmerl, Owen Chang and David Wells for insightful discussions regarding the OODA control loop, Mario Costamagna, Matteo Demichelis, Elio Paschetta and Roberto Squarotti at TILAB for their contribution on KX and Workflakes applicability, and on the IM service, the other members of the Programming Systems Lab for their work on KX. PSL is funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation CCR-9970790 and EIA-0071954, by the Office of Naval Research monitored by Naval Research Laboratory N000140110441, by Microsoft Research, and by a NEC Computers, Inc. equipment grant. The work at TILAB is funded in part by EURESCOM project P-1108 (Olives).

7. REFERENCES

[1] N. Carriero, L. Gelernter, Coordination Languages and their Significance, Communications of the ACM, 35(2):97-107, February 1992.

[2] J. Salasin, Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA), DARPA. http://www.darpa.mil/ito/research/dasada/.

[3] G. Valetto, G. Kaiser, G.S. Kc, A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems, in 8th European Workshop on Software Process Technology, June 2001. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-001-01.pdf.

[4] A. Wise, A.G. Cass, B. Staudt Lerner, E.K. McCall, L.J. Osterweil, S.M. Sutton, Jr., Using Little-JIL to Coordinate Agents in Software Engineering, in Automated Software Engineering Conference (ASE 2000), September 2000.

[5] A.G. Cass, B. Staudt Lerner, E.K. McCall, L. J. Osterweil, S.M. Sutton, Jr., A. Wise, Little-JIL/Juliette: A Process Definition Language and Interpreter, in 22nd International Conference on Software Engineering, June 2000.

[6] G. Alonso, Workflow. Assessment and perspective, in International Process Technology Workshop, September 1999. http://www-adele.imag.fr/IPTW/IPTW/Papers/Galonso.prn

[7] R. Stump, W. Bumpus, *Foundations of Application Management*, J. Wiley & Sons, 1998.

[8] N.W. Dutton, C. Wentworth, Ovum Evaluates: Service Management for E-business Applications, Ovum Ltd, February 2001.

[9] F. Kon, B. Gill, R.H. Campbell, M.D. Mickunas, Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents, in IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents, September 2000.

[10] R.S. Hall, D. Heimbigner, A.L. Wolf, A Cooperative Approach to Support Software Deployment Using the Software Dock, in 21st International Conference on Software Engineering, May 1999.

[11] G. Cugola., P.Y. Cunin, S. Dami, J. Estublier, A. Fuggetta, F. Pacull, M. Riviere, H. Verjus, Support for Software Federations: The Pie Platform, in 7th European Workshop on Software Process Technology, February 2000.

[12] S.K Shirvastava, L. Bellissard, D. Feliot, M. Herrmann, N. De Palma, S.M. Wheater, A Workflow and Agent based Platform for Service Provisioning, in 4th IEEE/OMG International Enterprise Distributed Object Computing Conference, September 2000

[13] B.Staudt Lerner, J.M. Cobleigh, L.J. Osterweil, A. Wise, Using Containment Units for Self Adaptation of Software, University of Massachusetts at Amherst, September 2001.

[14] M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, A.L. Wolf, Reconfiguration in the Enterprise JavaBean Component Model, in IFIP/ACM Working Conference on Component Deployment, June 2002. To appear.

[15] C.R. Hofmeister, J.M. Purtilo, Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement, in 13th International Conference on Distributed Computing Systems, May 1993.

[16] F. Kon, R. Campbell, M.D. Mickunas, K. Nahrstedt, F.J. Ballesteros. 2K, A Distributed Operating System for Dynamic Heterogeneous Environments, in 9th IEEE International Symposium on High Performance Distributed Computing, August 2000.

[17] J. Magee, J. Kramer, M. Sloman. Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, 15(6):663--675, June 1989.

[18] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer, Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework, in 3rd IEEE International Workshop on Policies for Distributed Systems and Networks, January 2001.

[19] S.-W. Cheng, D. Garlan, B. Schmerl, J.P. Sousa, B. Spitznagel, P. Steenkiste, Using Architectural Style as a Basis for Self-repair, in Working IEEE/IFIP Conference on Software Architecture 2002, August 2002. To appear.

[20] J.C. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, M. Gertz, The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications, University of Virginia, University of Colorado at Boulder, University of California at Davis, December 2001.

[21] P.N. Gross, S. Gupta, G. E. Kaiser, G.S. Kc, J.J. Parekh, An Active Events Model for Systems Monitoring, in Working Conference on Complex and Dynamic Systems Architecture, December 2001. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-011-01.pdf.

[22] G.T. Heineman, Adaptation and Software Architecture, in Proceedings of the 3rd International Workshop on Software Architecture, November 1998.

[23] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and Evaluation of a Wide-Area Event Notification Service, ACM Transactions on Computer Systems, 19(3):332-383, August 2001.

[24] G. Kaiser, A. Stone, S. Dossick, A Mobile Agent Approach to Lightweight Process Workflow, in International Process Technology Workshop, September 1999. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-021-99.pdf.

[25] B. Balzer, Probe Run-Time Infrastructure, Teknowledge, January 2001. http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeRun-TimeInfrastructureDesign.ppt.

[26] D. Garlan, B. Schmerl, and J. Chang, Using Gauges for Architecture-Based Monitoring and Adaptation, in Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[27] J. Aldrich, C. Chambers, D. Notkin, Connecting Software Architecture to Implementation, in the 24th International Conference on Software Engineering, May 2002. To appear.

[28] E.M. Dashofy, A. van der Hoek, R.N. Taylor, An Infrastructure for the Rapid Development of XML-based Architecture Description Languages, in the 24th International Conference on Software Engineering, May 2002. To appear.

[29] N. Combs, Reliable Recruitment and Assembly of Peer-to-peer Services and Distributed Workflow, in Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[30] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbinger, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, An Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems 14(3):54-62, May/June 1999.