Designing Multi-Layers Self-Adaptive Complex Applications

Davide Lorenzoli
University of Milano-Bicocca
via Bicocca degli Arcimboldi, 8
20126 - Milano, Italy
lorenzoli@disco.unimib.it

Davide Tosi
University of Milano-Bicocca
via Bicocca degli Arcimboldi, 8
20126 - Milano, Italy
tosi@disco.unimib.it

Salvatore Venticinque
Second University of Naples
via Roma, 29
81031 - Aversa(CE), Italy
salvatore.venticinque@unina2.it

Rosa Anna Micillo
Second University of Naples
via Roma, 29
81031 - Aversa(CE), Italy
rosaanna.micillo@unina2.it

ABSTRACT

The impossibility of statically determining the behavior of complex systems that interact at runtime with heterogeneous devices and remote entities, may lead to unexpected system failures and performance degradation. Recently, self-adaptive applications have been recognized as viable solutions for dealing with systems where size and complexity increase beyond the ability of humans. However, self-adaptive solutions have been always studied in isolation, only involving single layers of the system (e.g. operative system, middleware, firmware, hardware). In this paper we discuss our novel idea of multi-layers deep adaptability of complex systems. We present an integrated approach for designing and coordinating applications with three-layers of self-adaptation. The solution proposed is based on specialized sense-plan-act control loops that interact each others to monitor both specific parts and also the global behavior of the system. These control loops sense unexpected behaviors that can compromise the system, then diagnose the system layer that needs an adaptation action, and finally act the proper adaptation strategy to overcome the revealed problems without user intervention. We validate the approach with a SOA-based application, the Virtual Tour Guide, designed with three-layers self-adaptive abilities in order to overcome functional/non-functional problems that can derive from the integration of heterogeneous and remote third-party services.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*

SOQUA '07, September 3-4, 2007, Dubrovnik, Croatia

Keywords

Self-adaptive systems, distributed systems, service-oriented applications, mobile devices, context awareness

1. INTRODUCTION

Distributed architectures, component based systems and service oriented applications are example of modern software systems. These systems are heterogeneous pieces of code that dynamically cooperate in a decentralized way with loosely coupled control, affecting both complexity and quality requirements of the global system. Users of these systems are themself ubiquitous, heterogeneous and distributed clients that communicate and cooperate through advanced hardware devices and networks, increasing the risk of crashes, failures, or violations of SLAs. For example, how do software systems deal with 1+ billion people using their mobile phones, PDAs, and other wired or wireless embedded devices, communicating with each other, and interacting with other devices such as messaging systems, cars, GPSs, and the Internet?

Traditional design approaches, software engineering techniques, and maintenance cycles are not enough to foresee the dynamic and unpredictable behavior of these systems. What they need is the ability to adapt to new environments, new quality constraints, and new users requirements by updating either the device or the system or both, automatically, at runtime and in a transparent way.

Autonomic computing and self-adaptive applications have been recognized as viable solutions for dealing with systems where size and complexity increase beyond the ability of humans to respond manually, coherently and timely to environmental and system changes. Self-adaptive solutions have been traditionally studied in isolation, only involving single layers and aspects of the system [19]. Highly distributed systems require self-adaptive solutions at different abstraction layers to transparently optimize the communication and the cooperation among different software levels of the system. For example, we can figure-out client-side control loops that interact each other to monitor and adapt the client application; or middleware-side control loops that interact each-other to monitor and adapt the communication

between components; or a combination of them.

Multi-layers self-adaptability is especially demanded for service oriented applications. Service-oriented applications, which integrate third-party services (such as web services, grid services, or remote components), are challenged by the difficulty of keeping consistency between software systems that are maintained by different organizations and may evolve dynamically and independently, because of either changes in the services or dynamic discovery of new services. Service providers may change the implementation independently from clients, e.g., to correct faults or meet new requirements. Clients can locate services dynamically, using service discovery mechanisms that allow clients to discover and connect services based on machine-readable descriptions. This means that client applications may use, in subsequence invocations, different services characterized by different functional and non-functional requirements.

In this paper we present an architecture able to adapt, at different abstraction layers, the interactions among clients and services in order to improve both the availability and the quality of the global application. We propose an integrated solution that works (1) client-side to reveal possible functional mismatches between requested and provided services that can compromise the availability of the global application, and for dynamically adapting the client application accordingly; (2) middleware-side to optimize the binding process at discovery-time and for best matching client requirements and service capabilities; (3) server-side to reveal possible functional/non-functional mismatches that can compromise the quality of the global application, and for dynamically adapting service contents to client-understandable formats. The proposed approach suggests a new way for designing SOA-based applications, requiring a small effort to make the global application self-adaptive. This effort is compensated by a quality improvement of both the application and the service provided.

The paper is structured as follows. Section 2 surveys quality issues in complex software systems. Section 3 introduces functional, non-functional problems that may derive from the integration of services, referring to a working example that will be used throughout the paper. Section 4 and Section 5 describe our multi-layer self-adaptive framework to support the design and the execution of complex distributed systems. In Section 6 we summarize the results of our experiments wrt the working example. We discuss related work in Section 7, and finally we conclude in Section 8.

2. QUALITY IN COMPLEX SYSTEMS

Complex software systems are characterized by a large number of heterogeneous and remote parts that interact in different ways and with complex relations among the many required properties. This lead to unpredictable and often unmanageable behaviors that are not easily inferable from formal or informal specifications of the system components. As a consequence, system failures and performance degradation can suddenly arise, directly impacting the global quality of the system.

In particular, the increased complexity will come with many new aspects that may greatly affect system quality and health, and will not be addressed by traditional software engineering best-practices:

Transient faults: in complex systems, new faults (we refer to these faults as transient faults) can stem from unexpected interactions between independently developed devices and applications, unpredictable stress and interferences of the environment, or hardware aging. Transient faults are difficult to reproduce in testing and debugging environments, thus traditional verification and validation techniques, such as end-of-production testing, will not be enough to guarantee the delivering of the required levels of quality, availability, reliability, and security in the applications. In addition, solving a transient fault (revealed during operation) with hand-on intervention would be impracticable in complex systems due to the enormous costs generally associated with post-release bug fixing, the unacceptable scarce flexibility of the stop-repair-redeploy-restart approaches, and the impossibility for traditional debugging techniques to scale to the enormous complexity of the target systems;

Unpredictable working contexts: component-based and service-oriented technologies further contribute to increase the dimensions of the problem, pointing new and difficult questions for software testing and analysis. Traditional software testing and analysis techniques usually assume that testers have nearly complete knowledge of the software under test, as well as of its requirements and execution environment. Unfortunately, this will not be the case in new complex systems. Most of the times, components and services are designed and developed independently by third-party companies, and then acquired, deployed and used in sets of applications with different needs, goals and stakeholders. Producers are not able to identify requirements and profiles of all possible usage contexts, and thus testing of components and services in isolation is not enough. New faults are very likely to emerge every time a component is used in an unforeseen context. Integration testing in the deployment environments can reveal part of the emerging faults, but the effectiveness of traditional techniques is limited by the unavailability of source code, uncomplete functional and non-functional specifications, and limited knowledge about the internal design choices. Finally, even if a component fault is successfully diagnosed, the fixing process often requires collaborating with the component producer, which is difficult when not even impossible to achieve;

Emerging security flaws and threats: not trustable services and components, i.e., designed without the necessary care for security (but still able to interact with the environment), pose a threat to the correct behavior and functionality of the whole system. However, given the presence of heterogeneous (often mobile) devices, whose hardware and software may evolve independently of one another, security requirements are difficult to define a-priori for all devices and applications. Unforeseen security flaws and threats are likely to emerge whenever new interactions are established during operation, leading to the necessity for security to be dynamically checked and enforced by the system itself;

Interoperability issues: in complex systems, heterogeneous devices interoperate each others using remote entities (such as services and components) dynamically discovered. Often, there is no control on the actual implementation of these entities that are being used. For example, service providers may decide to dynamically deploy new versions of their services independently from the clients. In other cases, available services may have different implementations and providers behind an identical interface. This is especially true in Service Oriented Architectures (SOA) where clients

discover and connect services based on machine-readable descriptions [1]. This means that clients can match and connect syntactically equivalent specifications of services available in the network, that semantically differ in small but relevant details that can compromise the stability of client applications. For example, we used the ImageConverter web service [2] that converts images in different encodings. This service takes as input a `string` representing the local path of an image and returns the source image converted in a different format. During our tests, we have experimented several service failures due to wrong interpretations of the local path of the image (caused by a mismatch between the OS of the client and the service).

Being able to diagnose and heal transient faults, automatically adapt to unpredictable working environments, prevent security problems, and dynamically check, assess and guarantee interoperability with remote entities that can have "unstable" and "outside the control" implementations, becomes a prominent issue in complex systems. In this paper we primary focus on quality problems that can derive from interoperability and integration issues.

3. AN EXAMPLE

In this section, we discuss the problems that can derive from dynamically evolving services through a SOA-based Virtual Tour Guide (VTG), a case study that we use throughout the paper. VTG is a lightweight application that integrates several remote services to provide audio and video media-contents about a user-requested cultural tour. When a user needs information of a cultural site, he/she inserts the address of a target location. VTG asks the broker for providers that can satisfy the user request and then creates a complete virtual tour (with maps, multimedia contents, and additional information about hotels, restaurants, transports near to the target cultural site, weather forecast).

The heterogeneity of the user's devices that can use the VTG application (such as PDAs, notebooks, smart phones, mobile phones, 4th Generation phones) and the dynamic integration of different evolvable services may lead to several problems. Devices are characterized by different hardware and software capabilities (such as battery life, screen size and resolution, memory capability, operative system, running services, processes and applications, network connection, etc). Moreover, depending on the dynamically bound remote services, returned multimedia contents may take different encodings, formats, and file size. Finally, service performances and service qualities are extremely variable because they strongly depend on the hardware and software profile of each service provider.

Consider for example the request for a tour in Los Angeles. The simplest request takes as input a `string` that indicates the cultural site or location of interest. First of all, the application contacts the broker, asking for a valid implementation of a MapProvider service. Figure 1 shows an excerpt of the operation that characterizes the specification of the request. The service takes as input the address of the location (of complex type `Address`) and a token to access the service (of type `string`). The complex type Address is the sequence of address information such as city name, country name, etc...

[1]WSDL (Web Service Description Language) or GWSDL (Grid WSDL) are examples of service description languages

[2]available at `www.xmethods.com`

```
1  <wsdl:portType name="MapProvider">
2    <wsdl:operation name="getMap">
3      <wsdl:input message="impl:getMapReq" name="getMapReq"/>
4      <wsdl:output message="impl:getMapResp" name="getMapResp"/>
5    </wsdl:operation>
6  </wsdl:portType>
7
8  <wsdl:message name="getMapReq">
9    <complexType>
10     <sequence>
11       <element name="address" type="impl:Address"/>
12       <element name="token" type="xsd:string"/>
13     </sequence>
14   </complexType>
15 </wsdl:message>
16
17 <wsdl:message name="getMapResp">
18   <complexType>
19     <sequence>
20       <element name="getMapReturn" type="xsd:String"/>
21     </sequence>
22   </complexType>
23 </wsdl:message>
24
25 <complexType name="Address">
26   <sequence>
27      <element name="city" nillable="true" type="xsd:string"/>
28      <element name="country" nillable="true" type="xsd:string"/>
29      <element name="houseNumber" nillable="true" type="xsd:string"/>
30      <element name="intersection" nillable="true" type="xsd:string"/>
31      <element name="postalCode" nillable="true" type="xsd:string"/>
32      <element name="stateProvince" nillable="true" type="xsd:string"/>
33      <element name="street" nillable="true" type="xsd:string"/>
34   </sequence>
35 </complexType>
```

Figure 1: Excerpt of the WSDL description for a MapProvider Web Service

(of type `string`). The invocation of the getMap operation produces as result a getMapReturn element (of type `String`) that represents the map associated with the target location.

This WSDL description does not provide information about non-functional aspects, and can syntactically match many WSDL specifications of MapProvider services available from common repositories, that semantically differ in important details. For instance, some providers can require input addresses expressed in an extensive form (e.g. Los Angeles), while others can support only the code of the location (e.g. L.A.). Some providers can return maps in 2D or 3D rendering, with different encodings (e.g., JPG, PNG, EPS, proprietary formats, etc.), image sizes (e.g., 3264x2448, 1024x768, 320x240, etc.) and image weights (from few KBs to several MBs). Since heterogeneous devices can be connected to different map providers at different times, we can experience both relevant failures client-side (due to inhomogeneous semantical implementations of the map provider service) and also a perceivable degradation of the system performances. What happens if the client device is a PDA with an LCD screen of 320x240 pixels and it is connected to a map provider that only returns image with a size of 1280x1024 pixels? What happens if the client device is a smartphone able to display only JPG encodings and it is connected to a map provider that does not manage JPG images? Why wasting bandwidth in delivering into the network images of several MBs when the client application is a 3rd Generation phone with a limited system memory? Why stressing an overloaded map provider if an unloaded provider can offers a service that equally satisfies the user profile and the QoS client requirements? What happens if, during the discovery phase, no one available map services completely match the client requirements?

4. THE APPROACH

Single-layer approaches and classical stop-update-redeploy-

restart methodologies are not feasible to designing software architectures of complex distributed and service-oriented applications. Self-adaptive solutions, which have been recognized as a viable way for overtaking these problems [15], are being experimented in several application domains, but never with the focus on multi-layers adaptability. Engineering a classical self-adaptive system requires building a sense-plan-act control loop into the system to monitor its dynamics, diagnose unexpected behaviors, and automatically adapt the global system to avoid system failures [18]. While all the existing approaches in literature rely on a single sense-plan-act control loop instantiated for each system, we believe that a multi-layer self-adaptive approach is needed in highly distributed and complex applications. "Monolithic" self-adaptive solutions are unfeasible in distributed and service-oriented systems where several architectural levels interact by common interfaces developed and maintained by independent organizations. These systems draw upon the idea of "software above multiple devices" i.e. the software composing the global system runs above hundred of machines, thus making unmanageable a solution based on a centralized control loop.

In this paper we figure out the possibility of distributing in multiple abstraction layers ad-hoc control loops that are orchestrated by a supervisor, in order to avoid side effects when a piece of the application is adapted. A layered self-adaptive solution can improve the modularity and scalability of the system. Moreover, it can better handle, in a transparent way, both the heterogeneity of clients' and servers' resources and also different techniques and technological choices that characterize each system actor. We propose a three-layers self-adaptive architecture in which each single part of the system is monitored and autonomously adapted by a single self-adaptive layer. We enrich each critical part of the system with a specialized control loop that interacts with the other layers in order to either meet subsystem requirements or to globally optimize the system behavior in order to constantly guarantee both the availability and the quality of the global system.

The first adaptation layer works client-side to mediate interactions between the client application and the other entities (such as components, services, remote middlewares, and brokers) to reveal possible interoperability and integration problems that can happen due to functional mismatches between the involved entities, and to dynamically adapt the client application accordingly. The second adaptation layer works middleware-side to manage the discovery process and then optimize the match among the request of the client and the actual abilities of the service. To this end, we introduce the concept of user profile (i.e., as device capabilities and user preferences) to enrich the discovery process. The third adaptation layer works server-side to reveal possible functional/non-functional incompatibilities that can compromise the quality of the global system, and then upstream adapt the interactions among clients and services to common and well agreed formats.

4.1 Client-side Adaptation Layer

The client-side adaptation layer consists of a set of control loops instantiated and customized for each set of remote services that are dynamically integrated into the application and that comply with a specific description. The customization of each control loop consists of three main phases:

Set-up: in this phase, software integrators analyze the service description to identify ambiguities that may result in implementation dependent mismatches. The analysis is driven by specific fault taxonomies (see [24] for more details on the taxonomy);

Diagnosis: in this phase, software integrators design test cases to reveal the occurrence of the possible mismatches identified in the setup phase. Test cases are defined according to guidelines that describe possible test strategies for different faults categories [24];

Adaptation: in this phase, software integrators design ad-hoc adaptation strategies that are automatically executed when a service mismatch is revealed at run-time. Adaptation strategies are defined according to domain specific adaptation patterns for recurrent types of mismatches [24].

Customized control loops are then deployed and instantiated at run-time into the client application. Each instantiated control loop enhances the client/service communication by monitoring the service behavior. When a service misbehavior is detected (e.g., due to server-side implementation updates or dynamically discovered services that change the semantic of the service) diagnosis mechanisms are triggered and the test suite is automatically executed on the target service to reveal possible mismatches. If the diagnosis mechanisms reveal any mismatch, associated adaptation strategies update the structure and the behavior of the client application to solve the identified functional problems and optimize the interaction with the target service (e.g. by converting input/output parameters or values into formats that are understandable both by the client and also by the remote service, or adding local support for service functionalities that are temporarily missing or misunderstood).

For instance, in the VTG presented in Section 3 we can figure out a scenario in which a client application invokes the MapProvider service to retrieve a map. The client application provides its profile in the form:

```
< ScreenSize = 267x232 pixels >
< Video = supported >
< VideoCodec = mpeg4 >
< Audio = supported >
< AudioCodec = Mp3 >
```

and then asks the dynamically bound MapProvider for a map of Los Angeles `getMap(Los Angeles)`. The service returns to the client the JPG image of size 267x232 pixels. Unfortunately, the client application is a java midlet that supports only the PNG encoding. To make the client application self-adaptive, we customize a control loop for the MapProvider service category. We design a simple test case that automatically (1) invokes the linked MapProvider for a sample location; (2) verifies the retrieved file extension; (3) if the file encoding differs from PNG fail the test, otherwise signal test success. Finally, we design an adaptation strategy, triggered when the test case fails, that automatically invokes a third-party service that provides the support for converting images in different encodings.

4.2 Middleware-side Adaptation Layer

Self-adaptation at middleware layer can provide added value functionalities such as profiling, content personalization, context awareness, advanced service brokering mechanisms, and virtual home environment.

The middleware-side adaptation layer consists of a single control loop that manages discovery requests coming from

client applications. This control loop enhances traditional discovery processes of classical SOAs [1] by introducing semantical and profiled searches of new services. The basic requirement for our discovery mechanism is the knowledge of user/client profiles and service requirements. The instantiated control loop collects the profile of the client device and collects QoS constraints imposed by the client application. When the client asks for a new service, the control loop senses the network to discover, from a set of equivalent third-party services, the service that best fits the user requirements. This is done by: (1) syntactically and semantically matching the functional requests of the client with the functional capabilities provided by the services; (2) evaluating the client profile and cutting of all the services that do not satisfy the profile; (3) evaluating the current turnaround, throughput and workload of the services and cutting of all the services that do not fit the client QoS requirements.

For instance, referring to the VTG example we can figure out a scenario in which the client starts a discovery request for a map provider service category. Our discovery platform evaluates the client request `find(Map Provider)`, its profile:

```
< ScreenSize = 267x232 pixels >
< CpuSpeed = 170 MHz >
< VideoCode = mpeg4 >
< Audio = not supported >
```

and the current services workload. The platform discovers, in remote repositories, three potentially available MapProvider service implementations. Two of them match the user request and the client profile (the MapProviderA and the MapProviderB). However, the discovery platform becomes aware that the MapProviderA is overloaded thus returns to the client the binding to the MapProviderB.

4.3 Server-side Adaptation Layer

The server-side adaptation layer consists of a control loop specialized for each service that is dynamically bound with the client application. Each instantiated control loop enhances the service/client interaction by monitoring at runtime the network conditions and the SLA (Service Level Agreement) stipulated with the client. When the control loop detects a disagreement between the provided QoS and the SLA, it evaluates whether an adaptation solution can overcome the performance degradation and the consequent violation of the SLA. Adaptation solutions can, for example, restructure the business process of the involved services to minimize the downtime, or compress contents in lightweight formats to reduce the network congestion. During the adaptation selection, the control loop must continuously take in account the client profile and the device capabilities to avoid adaptation solutions that can both compromise the client stability and also produce undesired side-effects.

For example, referring to the VTG example we can figure out a scenario in which the client application is a laptop with the profile:

```
< ScreenSize = 1280x800 >
< MemoryRAM = 1024Mb >
< AudioCodec = AAC, Mp3, FLAC1.2 >
< DirectX = 9.0 >
< OperativeSystem = MacOS >
```

The client starts a discovery process for a multimedia content provider. The broker discovers only one service implementation that satisfies the client profile (MediaContentD). The client invokes the service asking for multimedia contents of the Los Angeles "Getty Museum". The MediaContentD returns a 3D rendered video (171Mb for 14min. of virtual tour into the museum). The platform dynamically evaluates the QoS constraints of the client and resizes the video in order to avoid potentially SLA violations.

5. ARCHITECTURE

Figure 2 describes the overall system architecture. On the left-side the requestor is composed of two components: a self-adaptive application, and an access manager (in Figure SM - Stub Magda) that is used to exploit the facilities provided at middleware-side (in Figure Magda Platform). The client application uses all the services brokered by the platform just updating the client stubs (in Figure S1, S2, S3) with the remote address of the service implementation dynamically bound. The platform exposes and provides the access to its functionalities by means of a web service (WS) interface. The interactions among requestors and the platform take place just before to invoke a service. The platform must be: non intrusive to maximize the portability; developed according to the SOA model to improve scalability and standardization; must consider services and clients as black box entities to minimize interdependencies with remote entities; must provide a transparent access to the services invoked by the client; must hide to the client its self-adaptive abilities.

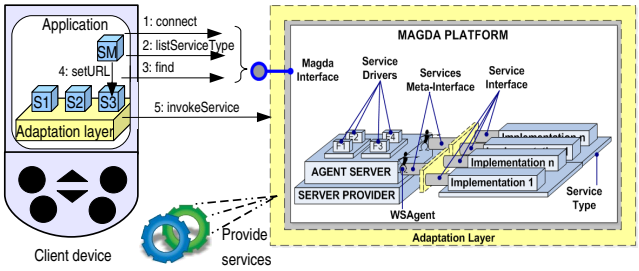

Figure 2: Architectural model

The platform is based on MAGdA (Mobile Agent based Grid Architecture) [21]. MAGdA is composed of a set of agents implemented using the Jade [4] platform. Providers publish services in a dedicated UDDI register. A register authority checks the compliance of the service interface with the supported ones and stores the service record into the register only if the publication requirements have been satisfied. To allow clients to exploit and use the functionalities exposed by the platform, we designed and implemented the platform as a WS characterized by the following interface:

```
public interface MagdaWS extends Remote {
  public String connect(String login, String pwd, String profile);
  public void logout(String token);
  public String[] listServiceType(String token);
  public String find(String serviceType, String token);
  public String[] getTokenList(String token);
  public void closeSession(String token);
}
```

The `connect` method allows the client to access the platform and to communicate its profile. Advanced functionalities for automatic profiling can be provided for devices which support mobile agents technology. For each successful authentication a personal agent is created in any con-

tainer of the platform which can be distributed geographically. The token returned identifies the personal agent for that session and must be used in all the next requests. The `logout` method closes the session without disposing the personal agent. The `closeSession` method disposes the personal agent. The `listServiceType` method returns the list of service categories which are available in the platform. The `find` method asks the personal agents for a binding to the serviceType provided as an input parameter and returns a new token that identifies the request to be used in order to get the response when it will be available. The `getTokenList` method retrieves the list of tokens for all the pending requests of service discovery.

For each WS, an agent container runs inside the tomcat application server. A specialized agent (we called WSAgent) manages the requests coming from the target WS implementation and also exploits the platform functionalities by collaborating with other agents. Personal agents receive the client profile, compare it with the requirements of dynamically discovered services, and (if necessary) start a recovery action by identifying the set of strategies that can provide a new added value service. The adapted service is realized by composing agents which act like bridges. As shown in Figure 3 agents perform a tunneling of requests coming from clients and return back responses received from providers. Agents-bridge can be "piped" each others. Each agent processes a received SOAP message and forwards it to the next one in the pipe. The client invokes the service using the binding information that links to the head of the pipe; the tail of the pipe receives the request and invokes the service brokered by the personal agent. If the adaptation required by the personal agent for its client is not available a new pipe is dynamically built, and other agent-bridges can be added to handle workload balancing or some other functionality such as security, etc... Each agent can be configured dinamically. It means that when the agent should be created it is possible to define at run-time the modules which will process SOAP requests and responses, where to execute the agent, the previous and the following component of the pipe. A similar idea is used at client-side to provide a mechanism for creating chain of adapters (following the Chain of Responsibility design pattern) thus increasing modularity and distribution of the adaptation logic.

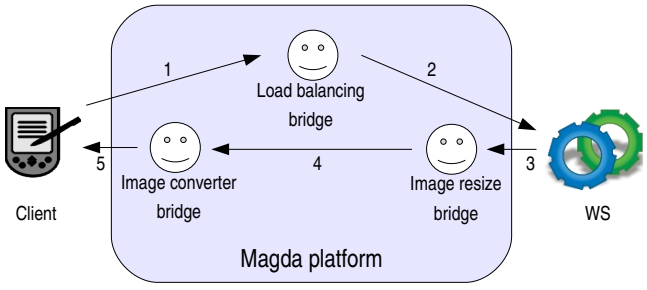

Figure 3: Platform adaptation approach

6. PRELIMINARY EXPERIMENTS

We implemented the VTG case study as a demo server application deployed on different distributed machines. We used Jakarta Tomcat 5.0 as application server, Apache Axis as WS platform, the J2ME Java Wireless Toolkit 2.5 as framework for developing and testing the client application, and the SHIWS plugin (for Eclipse development platform) to make self-adaptive the client layer [10]. SHIWS helps software integrators designing and implementing self-adaptive applications. SHIWS automatically generates the self-adaptive infrastructure, relieving integrators from a potential time-consuming task. A self-adaptive client midlet integrates an access manager that performs the authentication, asks for a service, and invokes it. In the middle, the MAGdA platform has been extended with self-adaptive abilities too. Finally, we set-up the following testbed: three different implementations of the MapProvider service (MapProviderA returns 267x232 PNG images, MapProviderB returns 267x232 GIF images, and MapProviderC returns 800x600 JPG images). We developed the full infrastructure exploiting real web services provided by ArcWeb (`www.arcwebservices.com`) in order to reproduce a real test environment.

In our experiments, accordingly to the midlet client profile and to the load balancing strategy, the platform equivalently selects the MapProviderA (no adaptations are required) and the MapProviderB (client-side adaptations are required to convert the image encoding). When both the providers are overloaded the platform links the MapProviderC (middleware-side adaptations are required to automatically link the MapProviderC, and server-side adaptations are required to resize the returned image). Here, we are able to provide preliminary performance experiments, which have been obtained installing client and server applications on a cluster composed of 7 dual-processors Intel Xeon, 1 GB RAM. Here we use a fake service in order to evaluate the time necessary to satisfy a request, due to transmission time, network latency, marshalling and unmarshalling, waiting time in the server queue. For each request, the client asks a service address and then invokes it. We provide mean response time in the following situations: (1) the platform returns to the client always the same binding address; (2) the platform returns the binding address of a bridge that dispatches the requests to two different providers; (3) the platform starts the bridge when the response time is increasing.

Figure 4 shows the behavior of the system when a single provider is accessed by seven clients, running with eight threads on different nodes. On X axis is shown the number of sent requests for each thread. On Y axis is shown the mean response time (ms). When the experiment is started the number of contemporary messages sent by the clients do not overload the server. When all the servers are waiting for a response the response time reaches a stationary value.

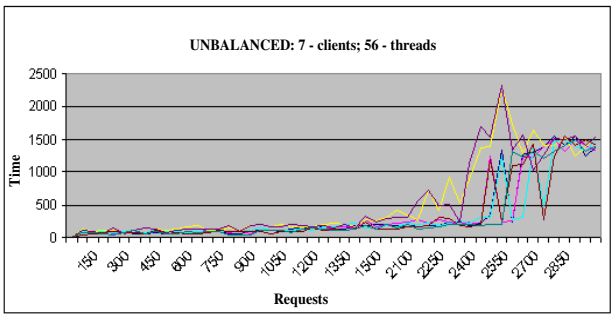

Figure 4: Mean response time without load balancing

In Figure 5 a bridge is used to dispatch the requests to different providers. As it can be seen, the response time decreases. The bridge overhead it is not relevant as it processes just the HTTP message header in order to redirect the request.

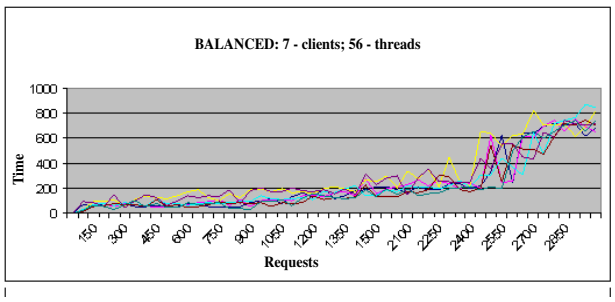

Figure 5: Mean response time with load balancing between two providers

7. RELATED WORK

7.1 Self-* Research Issues

In the last few years, researchers have investigated the notion of self-* systems based on control loops in many application domains with different goals and approaches. These included self-configuring software architectures [7], self-adaptive use and allocation of computing resources [20], self-healing mechanisms [9], and programming models inspired from biology [13]. Control loops and self-* solutions have been also applied to SOA-based applications.

For example, Baresi, Ghezzi and Guinea use assertions (in combination with orchestration languages) to monitor whether or not a service complies to a predefined functional contract: the developers of client applications specify the expected functional contracts with suitable assertions; the assertions are checked at runtime against the responses from services; the identified contract violations trigger the adaptation strategies, such as the selection of alternative services [3]. The approach of Baresi et al. works only at the middleware abstraction layer to increase fault-tolerance, while our approach covers all the three layers of complex systems to improve both the reliability and the QoS of the global application.

Other researches focus on service verification and services composition to identify formal automatized methods for checking and verifying the automatic integration of services [5]. They propose approaches that use automata to describe the interactions among services. However, none of these approaches describes how to identify and solve interface mismatches deriving from implementation changes that are not directly observable in the service specification. Moreover, these approaches only focus on functional problems, without considering non-functional aspects.

Other approaches focus on improving automatic discovery mechanisms by using ontologies and semantic web technologies [22]. These approaches aim to build common reference frameworks to share and reuse data across applications, enterprises, and community boundaries. This would provide powerful instruments to reason about services semantics. Currently, the main difficulty in using these approaches in practical applications is the difficulty of defining generally agreed domain ontologies. Our solution integrates some of these ideas in a more general and powerful framework for the self-adaptability.

7.2 QoS Management Research Issues

QoS management is critical for distributed SOAs because of clients with different QoS requirements and also service providers with different resources, workloads and ever-changing business rules [8]. Concepts and guidelines from ISO/IEC QoS Framework [12] form the basis for the design and implementation of QoS management in SOAs. Recently, new ideas have been proposed to extend the functionality of UDDI registry by introducing additional features such as UX [6] and UDDIe [2]. Here, a QoS model (with QoS certifiers used to verify the claims of web service's QoS) is presented. [16] provides a generalized approach for building a QoS-based registry for SOA. This framework enhances the current UDDI standard to improve the efficiency and the quality of the discovery process, by taking in account user specified QoS parameters. Others propose a new infrastructure for QoS management such as WSLA [14] or SLAng [17] where the authors address the dynamic selection of services via an agent framework coupled with a QoS ontology. [23] proposes to exploit correlations between QoS properties of each Web service to provide good candidates of Web services for the matchmaking process when service brokering is performed. [11] proposes a quality of service assessment model that is able to capture the reason behind the ratings. This allows us to perform quality of service assessment in context, by using only the ratings that have similar expectations. [25] presents an integrated QoS management architecture and solution for the publish/subscribe style of enterprise SOA.

Our solution implements the QoS management architecture at the middleware layer as a service for Information Brokers. Major contributions include a general integrated QoS management architecture, a flexible and extensible XML-based QoS language, innovative resource models, and a policy-based resource management solution.

8. CONCLUSIONS

Quality assurance in complex and distributed software applications is a non trivial issue. Self-* solutions can increase the quality of these applications by injecting autonomic abilities into the system. In this paper, we have proposed the design, implementation, and an initial experimental evaluation of a three-layers self-adaptive framework that optimizes the interactions among heterogeneous clients and remote entities in distributed architectures.

The initial experimental results we have presented show the applicability and the effectiveness of our approach, in particular, the platform ability of balancing the load of the servers, the efficient redistribution of resources, and the client ability of autonomously preventing integration mismatches.

We are extending the VTG case study with new services and new functionalities, and we are conducting new experiments to investigate in depth the benefits of the approach.

9. REFERENCES

[1] Reference model for service oriented architecture 1.0. Technical report, OASIS, October 2006.

[2] A. S. Ali, O. F. Rana, R. Al-Ali, and D. W. Walker. Uddie: An extended registry for web services. In

Proceedings of the Workshop on Service Oriented Computing: Models, Architectures and Applications at SAINT Conference. IEEE Press, 2003.

[3] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In *Proceedings of the Int. Conf. on Service Oriented Computing (ICSOC)*, 2005.

[4] F. Bellifemine, A. Poggi, and G. Rimassa. Jade: a FIPA2000 compliant agent development environment. In *Agents*, pages 216–217, 2001.

[5] D. Beyer, A. Chakrabarti, and T. Henzinger. Web service interfaces. In *Proceedings of the Int. World Wide Web Conf. (WWW)*. ACM Press, May 2005.

[6] Z. Chen, C. Liang-Tien, B. Silverajan, and L. Bu-Sung. Ux Ű an architecture providing qosaware and federated support for uddi. In *Proceedings of the 2003 International Conference on Web Services*, 2003.

[7] S. Cheng, A. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, pages 276–277, May 2004.

[8] C.Wang, G. Wang, A. Chen, and al. A policy-based approach for qos specification and enforcement in distributed service-oriented architecture. In *Proceedings of the 2005 IEEE International Conference on Services Computing*. IEEE Press, 2005.

[9] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *Proceedings of the Workshop on Self-healing systems*. ACM Press, 2002.

[10] G. Denaro, M. Pezzè, and D. Tosi. SHIWS: a self-healing integrator for web services. In *Proceedings of ICSE - Research Demos*. ACM Press, May 2007.

[11] V. Deora, J. Shao, W. A. Gray, and N. J. Fiddian. Supporting qos based selection in service oriented architecture. In *Proceedings of the International Conference on Next Generation Web Services Practices*. IEEE Press, 2006.

[12] I. O. for Standardization. Iso/iec. international standard 13236 technology - quality of service: Framework, Dec. 1998.

[13] S. George, D. Evans, and L. Davidson. A biologically inspired programming model for self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems, (WOSS)*, pages 102–104, Nov. 2002.

[14] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring of service level agreements for web services. Research report RC22456, IBM, 2002.

[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan 2003.

[16] A. Kumar, A. El-Geniedy, and S. Agarwal. A generalized framework for providing qos based registry in service oriented architecture. In *Proceedings of the 2005 IEEE International Conference on Services Computing*. IEEE Press, 2005.

[17] D. D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE Press, 2003.

[18] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc., 2004.

[19] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems, IEEE [see also IEEE Expert]*, 14(3):54–62, May/Jun 1999.

[20] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 604–613, 2004.

[21] N. M. R. Aversa, B. Di Martino and S. Venticinque. Magda: A mobile agent based grid architecture. *Journal of Grid Computing*, 4(4):395–412, 2006.

[22] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 544–552, New York, NY, USA, 2004. ACM Press.

[23] L. Taher, R. Basha, and H. E. Khatib. Establishing association between qos properties in service oriented architecture. In *Proceedings of the Workshop on Service Oriented Computing: Models, Architectures and Applications at SAINT Conference.* IEEE Press, 2003.

[24] D. Tosi. *Self-Managed Solutions for SOA-based Applications*. PhD thesis, University of Milano Bicocca, 2006.

[25] G. Wang, A.Chen, C. Wang, C. Fung, and S. Uczekaj. Integrated quality of service (qos) management in service-oriented enterprise architectures. In *Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf.* IEEE Press, 2004.