The Adapta Framework for Building Self-Adaptive Distributed Applications

Marcio Augusto Sekeff Sallem
Universidade Federal do Maranhão
Programa de Pos-Graduaacão em Engenharia de Eletricidade
Av Portugueses, S/N, Bacanga, São Luis, MA, Brasil
Email: marciosallem@lsd.ufma.br

Francisco José da Silva e Silva
Universidade Federal do Maranhão
Departamento de Informática
Av Portugueses, S/N, Bacanga, São Luis, MA, Brasil
Email: fssilva@deinf.ufma.br

***Abstract*— Modern computing environments are characterized by a high degree of dynamism that, along with the heterogeneity of computational devices and communication infrastructure, demand the development of a new range of applications that must be able to self-adapt dynamically and transparently according to changes in its execution environment. On a computational Grid, for instance, it is common to notice a high variation on resource availability, node instability, variations on load distribution, and heterogeneity of computational devices and network technology.**

The Adapta framework is a reflective middleware that provides the means to develop self-adaptive component-based distributed applications, separating the business code from the one responsible for adaptation. Adapta also provides a runtime execution environment that monitors computational resources and notifies application components about the occurrence of important events that should trigger reconfiguration actions. Adapta provides a XML based reconfiguration language that defines how the application must adapt in response to environmental changes. Statements of the reconfiguration language can also be applied at runtime, which allows to dynamically change the reconfiguration mechanism itself. This paper describes Adapta architecture, implementation, and evaluation through a concrete case study, where a Grid infrastructure is augmented for incorporating autonomic mechanisms towards a self-healing and self-optimization infrastructure.

I. INTRODUCTION

Modern distributed applications are getting harder to develop, maintain, and configure as environmental dynamism and computational devices heterogeneity increase. For example, consider a pervasive computing environment. The dynamism is exhibited on variations of resources and services availability, intermittent connectivity, huge device heterogeneity and communication technologies, and issues concerning user mobility. Another example is an opportunistic Grid that uses the idle time of distributed resources to perform complex computations. In this environment, it is common to notice a high variation on resource availability, node instability, variations on load distribution, and heterogeneity of computational devices and network technology. The dynamic nature of the Grid infrastructure, its high scalability, and great heterogeneity has turn impracticable its configuration, maintenance, and recovery in case of failures solely by human beings.

This new era of computation demands that application developers address not only the business logic of the system, but also how to make it flexible in order to run on most platforms and adaptable to changes on its executing environment. To cope with those requirements, developers must advance from the static computational paradigm towards a dynamic one. Applications must be able to apply modifications on its own structure and functionality automatically, without code recompiling and human intervention. Configuration and management must be performed on the fly, transparently to end users and not disrupting the service being offered.

Middleware architectures can provide a comprehensive solution guideline and tools to the problem of building effective self-adaptive systems, easing the application developer work. They may introduce runtime environments for context awareness and automatic application reconfiguration. Context awareness, which comprises resource availability and contextual knowledge, plays an important role to dynamic reconfiguration, triggering new adaptations according to the state gathered from the underlying executing environment. Middlewares must primarily address three key questions concerning the design of self-adaptive systems:

- *When to adapt?* How can the system detect that it's time to adapt so that its performance will improve and changes in the environment will not harm system correct functioning?
- *What to adapt?* Which parts, elements, components of the system are subject to being adapted or replaced?
- *How to adapt?* Which adaptive mechanism would be more beneficial to be applied given a certain system and environmental state?

The Adapta framework is a refactoring of the work presented on [2], introducing a well structured reconfiguration language for each aspect of the reconfiguration process (monitoring, environment change detection, and application reconfiguration) and additional support for application reconfiguration, such as: altering application parameters and switching from different application algorithms with a well defined transfer state mechanism. Support for reconfiguring entire application components is also being provided. Adapta is currently being used to introduce adaptive features into a Grid infrastructure, primarily addressing self-healing and self-optimization mechanisms.

This paper describes the current work on the Adapta

framework, focusing on its concepts and architecture. It also presents Adapta implementation highlights and how it is being used to add autonomic features to a Grid middleware. The paper is organized as follows: Section II describes the Adapta concepts and architecture, showing its main components and their interactions. Section III shows implementation highlights of each component. Section IV describes the use of Adapta framework towards a self-reconfigurable Grid middleware and its evaluation. Section V compares our approach with related work, while Section VI presents our conclusions and future work.

II. Architecture Overview

Adapta is a framework for developing self-adaptive distributed applications separating the business concerns from adaptation concerns. The application adaptable elements and reconfiguration actions are described using a XML-based language whose code is interpreted and executed during application startup. That code can be modified and re-interpreted on runtime, which allows introduction and removal of new reconfiguration actions on the fly without stopping the running application.

Reconfiguration actions are based on updating application parameters and replacing algorithms with a well defined state transfer mechanism. However, the set of actions is not restricted solely to those two. Experienced developers are welcome to extend Adapta reconfiguration mechanisms to introduce other actions, such as component migration, or replication. Adapta also provides a synchronization mechanism among distributed components that can be used during the reconfiguration process. Reconfiguration actions are combined into strategies that are executed whenever relevant changes on the underlying executing environment are detected.

Adapta provides the application adaptation engine and also a runtime system that regularly monitors resource availability and notifies subscribed components whenever significant environmental changes are detected. Figure 1 illustrates Adapta main components and their interactions. Each Adapta component is itself reconfigurable, since the object model can be modified on runtime applying changes to its functional behavior.

The Monitoring Service (MS) periodically collects information from hardware and software resources. Monitoring is based on Resources and Properties. Resources represent actual hardware and software resources, such as: CPU, memory, network interface, and applications; and Properties are monitorable attributes of resources, such as: CPU load usage, amount of main memory available, network bandwidth and latency, amount of application threads. Properties are associated with a set of *operation ranges*, which are defined by the framework user. For example, one could use the following operation ranges for monitoring the CPU load usage: [0%, 40%), [40%, 75%), and [75%, 100%]. The MS notifies the Local Event Service (LES) located at the same host whenever there is a change on the operation range of a Property.

The Local Event Service (LES) notifies events to subscribed components whenever a determined resource availability condition occurs. Event evaluation is based on a boolean expression provided by the user as part of its definition. To trigger an event notification, the corresponding boolean expression must stay *true* during an amount of time specified by the user, known as the *duration time*. It avoids generating notifications when temporary situations occur, such as a resource usage peak (e.g. a CPU use peak, triggered by starting a heavy program). The MS and the LES should run on every node that can execute a self-adaptive application component.

The Event Processing System (EPS) is a distributed event service that detects composite events from different event sources (distributed nodes). EPS is required for distributed applications where the decision to reconfigure the application should consider the combination of events detected on distinct components. For example, consider an application where component migration takes into account the CPU usage of every node across the network.

Finally, the Dynamic Reconfiguration Service (DyReS) is the adaptation engine that applies reconfiguration actions to the application in response to environmental changes. The framework architecture is based on computational reflection, where DyReS comprises the application meta-level while the base level consists of the application business objects.

III. Implementation Highlights

Adapta offers a high degree of flexibility to the application developer and administrator. Each framework component can be altered at runtime using *AdaptaML*, a XML-based reconfiguration language that describes the component object model. AdaptaML encompass each aspect of the reconfiguration process: monitoring, local and distributed event detection, and dynamic reconfiguration, including the set of adaptable elements and strategies that can be applied to the adaptive application. Framework users can modify and load the *AdaptaML* code of every component on runtime by invoking the `loadObjectModel()` method, assembled at the component interface. The object model is then interpreted and built on runtime, without service disruption and code recompiling, using the Adaptive-Object Model (AOM) pattern [12]. Thus, Adapta is itself reconfigurable. Adapta is built above CORBA middleware and uses Event Channels for event delivery (refer to [10]). We next describe implementation highlights of each Adapta component.

A. Monitoring Service

The Monitoring Service (MS) queries the computational environment using Monitor objects. Each Monitor is responsible for a single Property. Monitors can be dynamically instantiated to introduce new monitoring requirements not known at design-time or replaced on the fly to cope with the diversity of computational platforms.

The Monitor object has two attributes: (a) the frequency, which defines the periodicity in seconds to collect the property value; and (b) the set of property operation ranges which

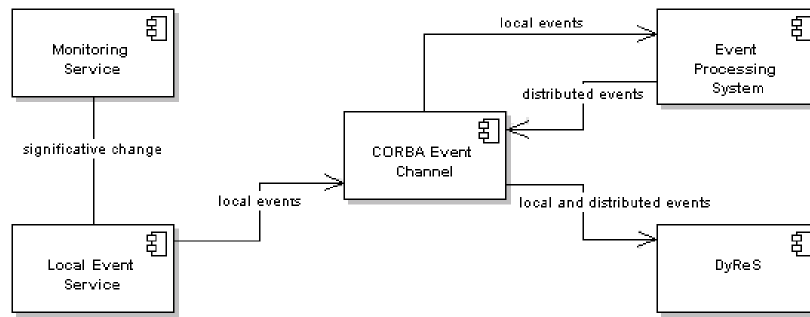

Fig. 1. Adapta Framework

indicates significant changes on property monitoring. The attribute values are also described through *AdaptaML* and can also be altered at runtime.

B. Local Event Service

The Local Event Service (LES) manages *local events* that describe changes on computational resources state gathered from a single host, e.g. CPU, memory, hard disk, and network interface. *Local events* are described using boolean expressions defined by the framework user.

Event detection is taken carefully to minimize the amount of messages sent to nodes. Evaluators check the validity of a *local event* against the current environmental context upon a significant resource change notification. The event boolean expression must stay *true* during the specified duration time in order to trigger an event notification. The duration time feature avoids unnecessary notifications, e.g. a CPU use peak due to the start of a heavy a program.

C. Event Processing System

Adapta distributed event detection component is based on a modified version of the Event Processing System (EPS, [7]). EPS evaluates *distributed events* defined by boolean expressions that comprises two or more *local events*. In this way, EPS consumes *local events* from registered LES, and produces *distributed events* to subscribed components. Typical distributed issues, such as those involving message ordering, loss, or duplication are addressed by EPS using three processing parameters:

- `Detection window`, which indicates the amount of time that a received event is valid, based on its timestamp;
- `Scheduling time`, which indicates the amount of time to wait for processing an event, helping to maintain the correct event ordering;
- `Concurrence time`, which indicates the amount of time that two events should be considered concurrent and treated simultaneously.

D. Dynamic Reconfiguration Service

The Dynamic Reconfiguration Service (DyReS) receives event notifications and starts the application reconfiguration process. DyReS separates the adaptation code from the code that governs the business rules using the reflection architectural pattern [3], which helps to keep application code manageable, facilitating debugging and maintenance.

DyReS consists on a dynamic table containing events and strategies and a reference to its associated Adapta Component Configurator (ACC), responsible for applying the reconfiguration actions. Each strategy encompasses one or more actions that are passed as parameters to the ACC.

The ACC, which extends the Component Configurator architecture [5], introduces three adaptive mechanisms: update of application parameters, replacement of interchangeable algorithms, and synchronization among distributed components. The set of reconfiguration actions is not restricted, and experienced users can extend the ACC to introduce more adaptive mechanisms.

1) Parameter Updating: Application parameters consist of one or more attributes, e.g. video resolution consists of height and width attributes. The parameter updating mechanism uses a callback method approach. The application developer introduces, during design, callback methods to be invoked on every class that has an updatable parameter. The ACC obtains the callback reference and dynamically invokes it using the new values informed by the reconfiguration action.

2) Algorithm Replacement: Families of interchangeable algorithms are a set of objects with the same interface that can be dynamically replaced at runtime. For example, MD5 and SHA-1 comprise a family of hash algorithms. Each family has a proxy that introduces an indirection layer above replaceable objects, keeping dynamic replacement transparent to clients.

The proxy also manages state transfer during the substitution process. The state consists on a set of variables, common to the whole object family that represents the current computation of the active algorithm. Since variables are class fields, the proxy can inspect the object implementing the active algorithm, store its state, and load it inside another object.

Adapta uses a lazy approach for object replacement, instead of an eager one [9]. The lazy approach allows the already running algorithm to complete its execution before being replaced, while the eager one immediately suspends the algorithm execution, performs the replacement, and resumes it from the point where it was suspended. An lazy approach advantage is that it always reaches a valid state, which is not guaranteed by the eager approach, since not every random

execution point (a transient state) is a valid state into the new object.

3) Synchronizing Reconfiguration Actions: In a distributed application, reconfiguration taken on a single component can affect other components or the entire application. For instance, consider a video server that can dynamically replace its encoding algorithm to adjust to current network conditions. The video server must notify its clients to substitute their decoding algorithm in order to maintain consistency during algorithm replacement. Therefore, synchronizing among components is a vital stage.

For synchronization purposes, each ACC manages the component dependencies by maintaining references to other components it depends on (called *hooks*), as well as references to dependent components (called *clients*).

Reconfiguration strategies that require synchronization has to specify whose components they intended to synchronize with and what notification should flow through the dependency chain. A `timeout` attribute is used to avoid that the synchronization process extends itself indefinitely, due to a component crash or network traffic conditions. If the synchronization period exceeds the timeout, the component that sent the notification will perform its own reconfiguration.

IV. Case Study: Towards a Self-Reconfigurable Grid Middleware

InteGrade [4] is a Grid middleware architecture that enables parallel applications to execute in a distributed environment, benefiting from the power of the hardware already available in organizations. InteGrade follows an opportunistic approach, taking advantage of the computing power of idle workstations.

The dynamic nature of the Grid infrastructure, its high scalability, and great heterogeneity has turn impracticable its configuration, maintenance, and recovery in case of failures solely by human beings. We are currently augmenting InteGrade with the Adapta framework, incorporating autonomic mechanisms towards a self-healing and self-optimization infrastructure.

A. Integrade Self-healing Mechanism

InteGrade application fault-tolerance layer encompass four different failure-handling techniques: retrying, replication, checkpointing, and replication with checkpointing. Replication consists on submit the same application with the same set of input parameters a number of times for execution. Each replica represents an active instance of the application, running on a resource different than the other replicas. Thus, as long as not all replicas fail, the application will succeed to execute. When one of the replicas finishes, the Grid middleware must discard (or ignore) the others and return the results to the requesting user. Previously on InteGrade the user manually decided the amount of replicas to be generated as part of the application submission process. We developed an InteGrade version that automatically decides the amount of replicas to be generated for a given application submition based on the currently execution environment MTBF (Mean Time between failures) and the application mean execution time.

We measured the benefits of varying the amount of replicas with a set of simulations. Figures 2 and 3 show the values obtained for the application estimated execution time considering the amount of generated replicas and the execution environment MTBF. The simulations considered an application execution time in the absence of failures of 18 and 36 hours for figures 2 and 3, respectively.

Fig. 2. Mean Execution Time of 18 hours

Fig. 3. Mean Execution Time of 36 hours

By analyzing the simulation results, we can conclude that: a) as the failure rate increases (lower MTBF values), a higher amount of replicas is necessary to keep the application execution time inside an acceptable range (a tolerance value); b) if we fix the MTBF value, it's not profitable, after a certain point, to increase the amount of replicas, since the gain would be minimum while more Grid resources would be used. For example, on figure 2 when the MTBF is 4 or 8, using 2, 4, or 9 replicas would make no significant difference to the application execution time. On figure 3 the same thing can be seen when the MTBF is 8; c) some times great gains can be achieved by slightly variating the number of replicas. For instance, on figure 2 we can see an enormous advantage by using 2 replicas instead of 1 when the MTBF is between 0.5 and 2. Also on figure 3 the same benefit is obtained when the MTBF is between 2 and 4.

The simulation results clearly indicate the benefits of altering the amount of application replicas as the MTBF and application execution time varies. On the other side, the dynamic nature of the Grid execution environment makes it very

difficult (if not impossible) for a Grid user or administrator to manually decide and set the best amount of replicas to be generated for each application submission.

We used the Adapta framework to add to InteGrade the capability of dynamically adjust the number of replicas to be generated. We started by developing a monitor that periodically measures the current Grid MTBF based on a database generated by InteGrade Execution Manager component. This database contains information about each application execution, such as: the global submission and conclusion timestamps and detailed information about the execution of each process that comprises the application execution, including data concerning eventual failures that may have occurred. Figure 4 illustrates AdaptaML monitoring statements. The monitored resource is called *Failure*, which contains a property named *MTBF*. The *MTBF* is regularly collected on intervals of 30 minutes (frequency tag, line 4). The monitor tag, on line 5, points to the class that implements the MTBF monitor, and the path where the binaries are located. Finally, Lines 7-12 shows the MTBF operating ranges.

```
 1: <monitoring>
 2:  <resource name="Failure">
 3:       <property name="MTBF">
 4:            <frequency value="900"/>
 5:            <monitor class="adapta.monitoring.monitors.MTBF" path="/home/adapta/classses/"/>
 6:            <ranges>
 7:                 <range lowerbound="0" upperbound="0.5"/>
 8:                 <range lowerbound="0.5" upperbound="1"/>
 9:                 <range lowerbound="1" upperbound="2"/>
10:                 <range lowerbound="2" upperbound="4"/>
11:                 <range lowerbound="4" upperbound="8"/>
12:                 <range lowerbound="8" upperbound="unlimited"/>
13:            </ranges>
14:       </property>
15:  </resource>
16: </monitoring>
```

Fig. 4. Monitoring language

AdaptaML local event definition is illustrated on figure 5. Whenever there's a change on *MTBF* operating ranges, a *MTBFChanged* event is notified to subscribed components. Its expression is always evaluated to `true` and its duration time is zero, since it's unnecessary to re-evaluate the event. The MTBF computed value is notified along with the event, as an attribute (line 3).

```
1: <local-events>
2:  <event name="MTBFChanged" expression="Failure.MTBF grt 0" duration-time="0">
3:       <attribute name="MTBF"/>
4:  </event>
5: </local-events>
```

Fig. 5. Local events language

InteGrade Global Resource Manager (GRM) is responsible for application scheduling. It was augmented with a reflective layer using DyReS (Section III-D). The GRM reconfiguration statements are presented on figure 6. The updatable parameter is the *MTBF*, which is implemented by the class `autogrid.Grm`. This parameter has one attribute, the class field that stores the *mtbf* value (Lines 2-4). The GRM component is described on Line 6. No hooks or clients are declared because distributed synchronization of reconfiguration actions (Section III-D.3) is not necessary. The reconfiguration strategy is described on Lines 8-10. It consists on an parameter update action, initiated by the *MTBFChanged* event, that invokes the callback `setMTBF` with the MTBF computed value received along with the event. Finally, an algorithm was added to GRM that, given an application submission, computes the number of replicas to be generated based on the current Grid MTBF and the application expected execution time (calculated from previous executions).

```
 1: <reconfiguration>
 2:  <parameter name="MTBF" class="autogrid.Grm">
 3:       <attribute name="mtbf" type="double"/>
 4:  </parameter>
 5:
 6:  <component name="Grm"/>
 7:
 8:  <reconfiguration-strategy on-event="MTBFChanged">
 9:       <action type="parameter" target="MTBF" value="setMTBF(Event.MTBFChanged.MTBF)"/>
10:  </reconfiguration-strategy>
11: </reconfiguration>
```

Fig. 6. Reconfiguration language

B. Integrade Self-optimization Mechanism

Opportunistic Grid environments are highly dynamic due to several reasons, such as: resources can fail or become unavailable instantly, executions can be interrupted by higher priority internal jobs, new resources can be added on the fly. Grid environments also experience a variable rate of application submissions. In this situation, the scheduling activity should also be made dynamic by altering algorithms and parameters used to take scheduling decisions accordingly to the current and/or future resource status [1]. Adaptive scheduling can optimize the overall Grid performance and minimize application response time.

We are currently altering InteGrade scheduling mechanism, allowing it to dynamically switch between three distinct algorithms: the MCT (Minimum Completion Time), max-min and min-min. MCT is a on-line scheduling algorithm, while max-min and min-min are batch ones. On-line scheduling algorithms are appropriate to environments where the application income rate is high; therefore, decisions should be taken quickly. Batch algorithms, however, enqueue application submissions and decide, based on environmental conditions, which schedule is better. Min-min and max-min take decisions based on a set of applications and a set of resources. While min-min schedules the fastest application (with minimum execution time) to the resource that can finish it quicker, the max-min approach schedules the longest application to the resource that can finish it quicker. Min-min is profitable when mean response time should be taken into consideration and resource availability is low. On the other hand, max-min maximizes concurrency, and is better suited when resource availability is high. Adapta framework provides the means for dynamic replacement of scheduling algorithms with the necessary state transfer among them.

V. Related Work

QuO [11] helps the development of distributed applications adaptable to changing quality of service (QoS). QuO primary

goal was to support QoS-based adaptation while our work followed a more general approach. QuO runtime kernel is the main component responsible for monitoring the executing environment, controlling QoS changes and providing support for adaptation. QuO relies on indirection layers (provided by Delegate objects) to evaluate system conditions and trigger adaptive behaviors and callbacks. In contrast, Adapta avoids the overhead during methods invocations by performing environment monitoring and event detection concurrently to the application execution. QuO, as Adapta, supports the addition of new adaptation mechanisms. Since our work was strongly influenced by computational reflection, we also provide mechanisms to represent explicitly the application structure, describing the inter-component dependencies, which is not supported in QuO. This meta-object representation is used to propagate reconfiguration actions responsible for adapting the application to a new environmental state.

Accord [6] assists users to develop autonomic applications. Accord manages behavioral and compositional (organization, interaction, and coordination between components) aspects of an application using high-level rules, injected on runtime, and enforced by an agent infrastructure. Instead, Adapta relies on a synchronization protocol to assure the correct execution of adaptation actions. In Accord, each autonomic element is augmented with an Element Manager that monitors its execution and context, and fires adaptation rules. On the other hand, Adapta decouples monitoring concerns from event management, services common to every component executing on the same host. This approach makes those services easier to manage, debug and maintain.

CASA [8] enables the development and operation of autonomic applications. In CASA, dynamic adaptation is carried according to a contract, stored externally to the application and written using a XML-based application. The contract is very similar to AdaptaML, both describe how the application should be reconfigured according to changes in the executing environment. CASA includes a set of four adaptation mechanisms: change in lower-level services, weave and unweave of aspects, change of application attributes and recomposition of application components. Adapta's set of actions is smaller, but its adaptation engine is extensible, allowing advanced users to develop new adaptation mechanisms and introduce them into the framework.

VI. Conclusions and Future Work

As dynamism and heterogeneity increase in today's modern computing systems, application development must shift to a self-adaptive paradigm, instead of a static-based one. Self-adaptive applications can reconfigure themselves transparently according to changes on the executing environment, without service interruption and code recompiling.

This paper presented Adapta, a framework that enables the development of distributed self-adaptive applications. Adapta is also a runtime system, monitoring resource availability and notifying interested components when changes are detected on resource usage. Adapta set of adaptation mechanisms is not restricted and can be easily extended by experienced users through a flexible architecture. Each Adapta component can be dynamically altered using *AdaptaML*, a XML-based reconfiguration language used to describe the component object model that can be modified and loaded at runtime. Finally, Adapta is based on computational reflection, promoting a clear separation of concerns between the adaptable part of the application from its core (business rules), simplifying the application implementation, debugging, and maintenance.

Adapta framework was evaluated through a concrete case study. The Integrade Grid middleware was modified in order to incorporate autonomic mechanisms towards a self-healing and self-optimization infrastructure. The new application fault tolerance mechanism automatically decides the amount of replicas to be generated for a given application submission based on the currently execution environment MTBF and the application mean execution time. The benefits of our approach were measured through a set of experiments.

We are currently altering InteGrade scheduling mechanism, allowing it to dynamically switch between distinct algorithms. We argue that adaptive scheduling can optimize the overall Grid performance and minimize application response time. We are also extending Adapta with inter-component reconfiguration mechanisms, such as component replacement, addition, removal, migration, and replication.

References

[1] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Report CW 504, School of Computing, Queen's University, Kingston, Ontario, January 2006.

[2] J. Y. R. J. Francisco Jose da Silva e Silva, Fabio Kon. A pattern language for adaptive distributed systems. *SugarLoafPLoP*, 2005.

[3] H. R. P. S. M. S. Frank Buschmann, Regine Meunier. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. Wiley, 1996.

[4] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[5] F. Kon. *Automatic Configuration of Component-Based Distributed Systems*. PhD thesis, University of Illinois, Urbana, Illinois, 2000.

[6] H. Liu and M. Parashar. Accord: A programming framework for autonomic applications. *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, 36(3):341–352, May 2006.

[7] D. Moreto. Monitoramento de eventos compostos em sistemas distribudos. Master's thesis, Instituto de Matemtica e Estatstica, Universidade de So Paulo (USP), Setembro 1998.

[8] A. Mukhija and M. Glinz. The casa approach to autonomic applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, Paris, France, 2005. IEEE Computer Society.

[9] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS*, pages 124–138, 2005.

[10] Object Management Group (OMG). *Event Service Specification, version 1.2*, October 2004.

[11] R. Vanegas, J. Zinky, D. Karr, J. Loyall, R. Schantzand, and D. Bakken. Quos runtime support for quality of service in distributed objects. In *in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98)*, pages 207–223, England, September, 1998.

[12] J. W. Yoder and R. E. Johnson. The adaptive object-model architectural style. In *Proceedings of the IFIP 17th World Computer Congress TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.