European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

A Network of Excellence funded by the European Commission

Autonomic Behavior of Grid Applications using Component Platforms

Ana-Maria Oprescu and Thilo Kielmann
{amo,kielmann}@cs.vu.nl
Dept. of Computer Science – Vrije Universiteit Amsterdam
De Boelelaan 1083, 1081HV Amsterdam, The Netherlands

Marco Danelutto and Marco Aldinucci
{marcod,aldinuc}@di.unipi.it
Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

CoreGRID Technical Report
Number TR-0156
July 22, 2008

Institute on Grid Systems, Tools and Environments
Institute on Programming Model

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Autonomic Behavior of Grid Applications using Component Platforms

Ana-Maria Oprescu and Thilo Kielmann
{amo,kielmann}@cs.vu.nl
Dept. of Computer Science – Vrije Universiteit Amsterdam
De Boelelaan 1083, 1081HV Amsterdam, The Netherlands

Marco Danelutto and Marco Aldinucci
{marcod,aldinuc}@di.unipi.it
Dept. of Computer Science – University of Pisa
Largo B. Pontecorvo 3, Pisa, Italy

CoreGRID TR-0156
July 22, 2008

Abstract

In this report, we present the results of a two-week research visit by the first author to the University of Pisa. We argue that problems like load-balancing or fault-tolerance can be addressed by integrating autonomic behavior in existing Grid applications. This can be achieved either by investigating novel approaches, such as multi-objective strategies, or by enhancing existing, specific solutions like cluster-aware random stealing [11] or empirically determined threshold of compute nodes [15]. We investigate both the JavaGAT [12] and Satin [14] systems to be augmented by autonomic behavior, based on autonomic managers from Muskel [7]. The main outcome of this report are items of future research work.

1 Introduction

Grid applications usually address a wide range of interdisciplinary problems using computational power spanning over multiple organizations, different types of resources and dynamic availability intervals. The dynamics of such heterogenous systems exceeds in many cases the natural limits of a human user in terms of steering a running application or adapting existing ones to ever-changing Grid environments. Autonomic behavior has emerged as a possible solution to these limitations, sustained in different projects by IBM and Microsoft. The human user chores would be mostly done by an autonomic manager, which follows a classical control-system loop. The main idea behind this new trend is mimicking the biological model of the (human) nervous system, which is self-*: adapting, optimizing, healing. We propose to take this metaphor a step further and to introduce component platforms as a counterpart of different functionalities offered by the diverse types of (human) cells. The main principle behind the component model is the *separation of concerns*, which is actually employed in the JavaGAT design.

The Java Grid Application Toolkit (JavaGAT) represents a virtual middleware, offering a set of consistent APIs for accessing numerous established Grid middleware systems (e.g. Globus [10], SGE [3] or Zorilla [8]). Given its modular design, it is extensible to support other Grid middleware systems, including customized ones.

The Satin programming environment provides Cilk-like [2], divide and conquer (D&C) primitives for Java-oriented developers. The runtime provides load-balancing, fault-tolerance, malleability and transparent migration.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

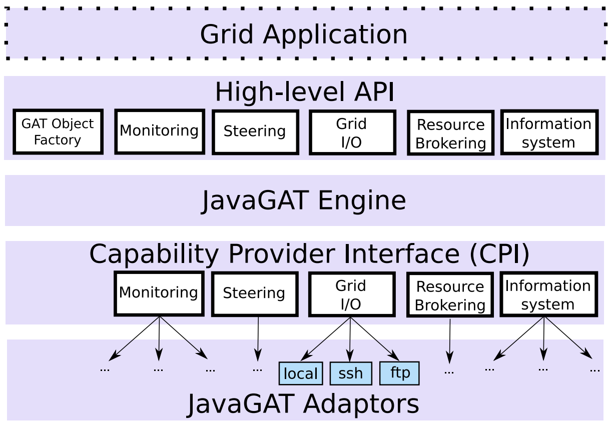


Figure 1: The structure of the JavaGAT implementation.

The Muskel [7] programming environment provides a Java skeleton approach to structured parallel programs. In its current state it does not support divide and conquer applications. Muskel implements a distributed macro data flow engine, based on RMI objects. Any kind of macro data flow instructions (MDFIs) can be computed remotely. A MDFI is a piece of code, which can be given some input data, run and obtain some output data. Fault-tolerance is guaranteed. A manager takes care of ensuring a user supplied performance contract (parallelism degree and/or (next release) service time).

Multi-objective optimization techniques have generated a plethora of research efforts since the nineteenth century, with cornerstone moments in the 1970s and 1990s [9]. The aim of such techniques is to optimize simultaneously a set of (possibly conflicting) objectives, including reaching a compromise. The similarity to human behavior in terms of decision making processes recommend these strategies as applicable to autonomic managers.

This report is structured as follows. Section 2 presents initial design issues of adding autonomic behavior to JavaGAT, including a discussion on multi-objective strategies adapted to autonomic managers. In Section 3 we analyze possible extensions of existing adaptation mechanisms to provide autonomic behavior, by looking at how Satin and Muskel could interact. We first analyze extending Muskel to support D&C applications by using Satin as a back-end; in this context, we look at how to extend Satin adaptation mechanisms. We then propose a D&C platform implemented with macro data flow technology, having Satin as a front-end. Section 4 outlines the relation of this work with the roadmaps of the relevant CoreGRID institutes.

2 Self-adaptive JavaGAT

The Java Grid Application Toolkit (JavaGAT) provides a high-level, middleware-independent and site-independent interface to the Grid. The global structure of the JavaGAT is shown in Figure 1. The system consists of four layers. The top layer is the high-level user API. The second layer is the JavaGAT engine, wich is responsible for delegating the API calls to the correct middleware. Because JavaGAT has to support multiple Grid middlewares, we use a “plug-in” architecture. The third layer of the JavaGAT is the interface that is used by these plug-ins. We call this the Capability Provider Interface (CPI). The bottom layer consists of the plug-ins, called *adaptors* in this context. The adaptors contain code that binds to a specific middleware platform.

JavaGAT is currently in an initial phase of self-adaptivity. It implements an elaborate mechanism, called *intelligent dispatching* [12] that, together with *nested exceptions*, allows to select at runtime an adaptor that is capable of implementing a given API call, based on the middleware available on the involved machines. While this mechanism solves a lot of practical user problems, it does not go further than checking adaptors until one of them does the given job. Further adaptation is not performed within JavaGAT right now.

2.1 Autonomic controllers within the JavaGAT engine

Here, we outline the design options for autonomic controllers within the JavaGAT engine. Basically, this design follows the range of possibilities for mediator components, as presented in [4]. Below, we investigate the opportunities for adaptation on the level of adaptors.

Active versus passive autonomic manager The autonomic manager of the engine can be an active element, working as a daemon. The passive paradigm assumes the managing code is run by the GATEngine instantiated as part of an application.

Stateful versus stateless adaptation policies The first take into account previous history and events, while the second only consider (rule, event, action) tuples; follows that the first type would require some ”warming” phase and may be more difficult to implement, but could perform better in cases where the (rule, event, action) tuple does not represent a good approximation of the addressed system.

On-demand versus proactive adaptation The first optimizes as service requests arrive, while the second is looking at a longer term plan. For instance, in a hierarchical structure of control managers and file operations as an example, the file open operation would be a good point for deciding which would be the best adaptor to use.

2.2 Autonomic control within adaptors

The GAT adaptors implement different functionalities according to the respective Capability Provider Interface (CPI's). Until now, selection between different adaptors implementing the same CPI takes into account a single constraint: whether it can be instantiated in the current context. We propose an extension of this policy to several metrics exported by each CPI. For instance, the GAT adaptors can be extended with the JBoss rule approach for a performance-targeted autonomic manager [6]; reliability could also come into play. As future research, we propose to use the JavaGAT adaptors as a vehicle to design a general framework for self-adaptation. The CPI might select which adaptor to use based on either user-provided hints or run-time optimization techniques, such as statistics or heuristics. The latter implies an intrinsic optimal state to achieve.

2.3 Multi-objective strategies

Multi-objective strategies can be employed at the JavaGAT engine level. Different CPIs have usually different metrics to determine best behavior, hence multiple objectives need to be optimized; this manager would live somewhere between the application and the GATEngine. Though a conflict in constraint fulfilling might not arise at the adaptors level, scenarios at the engine level can easily be imagined. An example would be an application with job submission and file staging activities which has to choose from a pool of resources either slow computational-wise, but optimal w.r.t. file access time, or having fast CPUs, but too slow w.r.t. file access time; a trade-off decision must be taken by the application-level autonomic manager and then imposed on adaptor-level autonomic managers.

Existing extensive literature on multi-objective optimization can provide possible approaches to apply to Grid systems. We propose to exclude evolutionary algorithms as too heavyweight for the dynamic setting of Grid systems. We would need to focus on more pragmatic approaches.

Figure 2 shows the proposed, hierarchical structure of autonomic managers for the JavaGAT. The actual autonomic managers are affiliated with the individual CPI packages (one for files, one for jobs, etc.). A multi-objective manager integrates the individual, per-CPI package, managers.

3 Satin and Muskel interaction

In this section, we investigate how the Satin [14] system can be augmented by autonomic behavior, based on autonomic managers from Muskel [7]. Satin's programming model is an extension of the single-threaded Java model. To achieve parallel execution, Satin programs use simple divide-and-conquer (D&C) primitives. Figure 3 shows the complete Satin application to compute Fibonacci numbers in parallel, referred to as Satin's "Hello, world" application.

Satin is implemented as an API to the Ibis [13] system. Satin uses Java's marker interfaces to let the programmer mark which recursive methods should be executed in parallel. A Satin-specific byte code rewriter inserts all code that is necessary for parallel execution. Load balancing in Satin is performed via cluster-aware work stealing [11], allowing Satin to run highly efficiently in cluster-based Grid environments. Currently, Satin allows performance adaptation

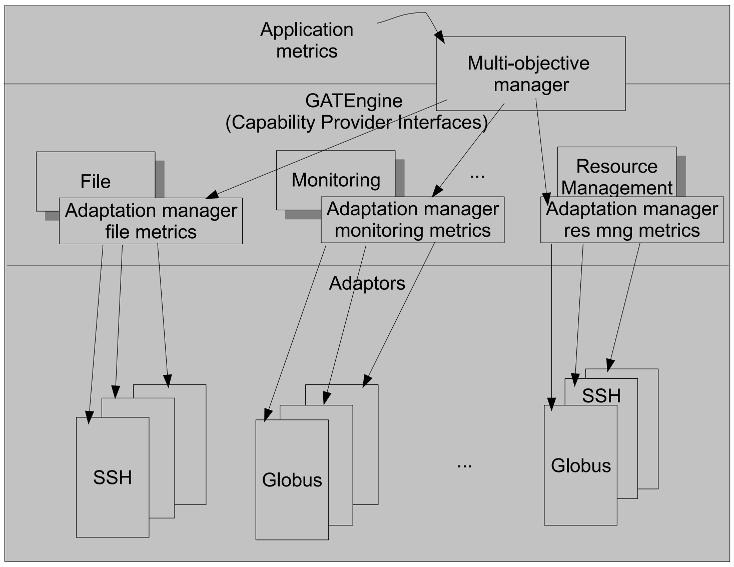

Application metrics
Multi-objective manager
GATEngine (Capability Provider Interfaces)
File
Monitoring
...
Resource Management
Adaptation manager file metrics
Adaptation manager monitoring metrics
Adaptation manager res mng metrics
Adaptors
SSH
Globus
...
SSH
Globus

Figure 2: Initial design of an adaptive JavaGAT

only by means of dynamically adding or removing compute nodes at runtime [15], irrespective of other factors or objectives.

Muskel is an expandable, skeleton-based parallel programming environment [1], implemented as a full Java library, targeting workstation clusters, networks and Grids. Muskel is implemented exploiting the (macro) data flow technology, rather than the more classical implementation templates. Using macro data flow instructions (MDFIs), Muskel easily and efficiently implements both classical, predefined skeletons, and user-defined parallelism exploitation patterns. Figure 4 shows the code needed to set up and execute a two-stage pipeline with parallel stages (farms), except the sequential code.

The current status of Satin and Muskel programming environments is shown in Table 1. In the following, we analyze their features of interest with respect to each other.

	Target architecture	Impl. Language	Skeletons	Status	RTS
Satin	COW, NOW, Grid	Java	Divide&Conquer	Well engineered Java code	Fork/Join model over distributed JVMs with job stealing
Muskel	COW, NOW, POSIX/ssh Grids	Java	Pipe, Farm	Prototype Java code, new (stable) version forthcoming	Macro data flow distributed interpreter with centralized instruction repository

Table 1: Current status of Satin and Muskel

3.1 A Muskel front-end to Satin

In this subsection we discuss different aspects of providing a Satin D&C service to Muskel, which currently does not have a type of skeleton to support D&C applications. Wrapping Satin with a WSDL interface has the advantage of generality compared to hand-crafted glue between Satin and Muskel. Different experiments to evaluate the performance of Satin and Muskel have to be done (e.g. D&C in pure Muskel as a farm executing dynamically created tasks).

```
interface FibInter extends satin.Spawnable {
    public long fib(long n);
}

class Fib extends satin.SatinObject
    implements FibInter {
    public long fib(long n) {
        if(n < 2) return n;

        long x = fib(n-1); // spawned
        long y = fib(n-2); // spawned, too
        sync();

        return x + y;
    }

    public static void main(String[] args) {
        Fib f = new Fib();
        long res = f.fib(10);
        f.sync();
        System.out.println("Fib␣10␣=␣" + res);
    }
}
```

Figure 3: *Fib*: an example divide-and-conquer program in Satin.

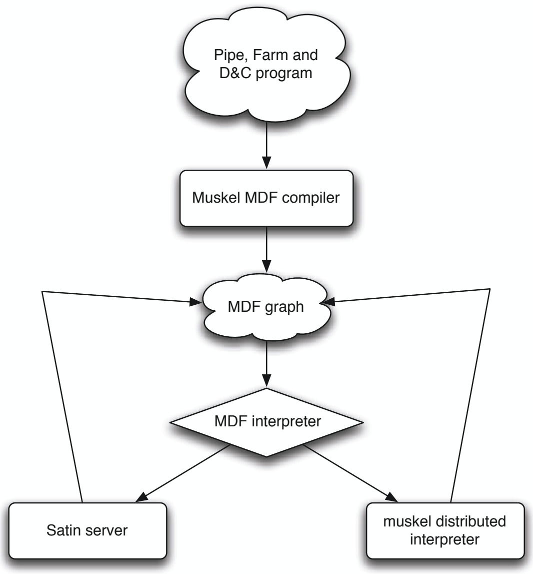
```
...
    Skeleton main =
        new Pipeline(new Farm(f),
                     new Farm(g));
    Manager manager = new Manager();
    manager.setProgram(main);
    manager.setContract(new ParDegree(10));
    manager.setInputManager(inputManager);
    manager.setOutputManager(outputManager);
    manager.eval();
...
```

Figure 4: `Muskel` code for a two-stage pipeline with parallel stages (farms).

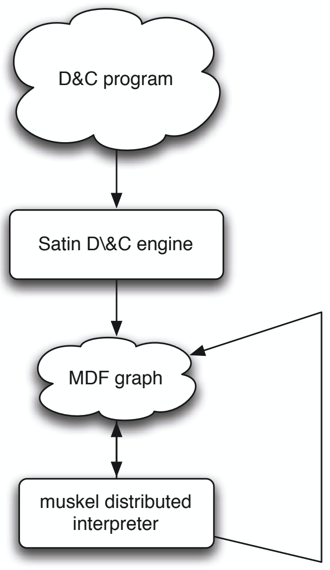
Autonomic management related aspects Satin as a D&C computational server could be equipped with a QoS manager to optimize some parameters relative to the requests either coming from the same user or coming from different users. As an example, the (autonomic) manager may ensure that an user supplied performance contract (e.g. execute the request with a parallelism degree not inferior to k) is fulfilled. It could also guarantee tolerance to node/network faults (on top of the one already provided by Satin). The manager may optimize throughput w.r.t., the overall service requests (those coming from different users) and could balance the load (or priorities) of service requests coming from different users. The general self-adaptation framework mentioned in Section 2.2 can be employed here as well.

3.2 A Satin front-end to Muskel

In this subsection we look at adapting Satin in such a way it is used to feed fireable macro data flow instructions (i.e. MDFI whose input data is available) to the Muskel engine. Here, Satin takes care of the functional aspects, while Muskel takes care of the non-functional ones. Muskel integrates a performance-oriented manager which takes care of user-supplied contracts (parallelism degree and/or (forthcoming version) service time). This setting allows to evaluate pure macro data flow vs. fork/join with stealing.



(a) Muskel on top of Satin



(b) Satin on top Muskel

Autonomic management related aspects The Satin engine should be able to adapt the *granularity* of the computations "passed" to Muskel. A new form of a D&C skeleton can thought such that a code fragment like

```
if(isBaseCase(X))
   then baseCase(X)
   else combine(applyToAll(recCall, split(X)))
```

is changed to

```
if(isWorthSeqComputation(X))
   then seqMuskelCall(X)
   else combine(applyToAll(recCall, split(X)))
```

Then the amount of splitting required is adapted to the computation *and* to the efficiency of the Muskel subsystem to execute that computation. Assume that each time we delegate a computation to Muskel on one hand we measure the time needed to get back an answer and on the other hand, Muskel reports on how much time it spent in the delegated computation. Using this information, the Satin/Muskel manager decides whether the granularity of the computation is adequate (i.e. comparison between T_{muskel} and $T_{roundtriptime}$). Let us assume further that more splitting leads to overhead, then a message (e.g. "not worth splitting any more if…") is propagated through the D&C tree. Again, the general self-adaptation framework mentioned in Section 2.2 can be employed here as well.

4 Relation to the roadmap of CoreGRID institutes

The topic of this work is at the junction between the Institute on Grid Systems, Tools, and Environments (WP)7), and the Programming model institute (WP3). Here, we briefly outline the relationships of this work to the roadmaps of the two institutes.

Section 2 proposes to introduce autonomic managers within the engine of the JavaGAT. Systems like the JavaGAT have been identified by WP7 as a *service and resource abstraction layer* for the generic components-based Grid platform [5]. Introducing autonomic behaviour in this layer clearly improves the overall functionality.

Section 3 suggests different approaches to program single D&C components (i.e. to wrap Muskel/Satin (or Satin/Muskel) interpreters in a GCM component) which addresses Task 3.1 of the WP3 activities in CoreGRID. It also addresses Task 3.3 as both D&C components and skeletons in general are advanced programming models. Section 2 discusses different design issues when enabling an existing Grid system with autonomic behavior, therefore addressing Task 3.3 of WP3.

References

[1] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[3] Goncalo Borges, Mario David, J Gomes, Carlos Fernandez, Javier Lopez Cacheiro, Pablo Rey Mayo, Alvaro Simon Garcia, Dave Kant, and Keith Sephton. Sun Grid Engine, a new scheduler for EGEE middleware. In *IBERGRID - Iberian Grid Infrastructure Conference*, May 2007.

[4] CoreGrid NoE, Institute on Grid Systems, Tools, and Environments. Design of the Integrated Toolkit with Supporting Mediator Components. Deliverable Report D.STE.05, 2006.

[5] CoreGrid NoE, Institute on Grid Systems, Tools, and Environments. Design Methodology of the Generic Components-based Grid Platform. Deliverable Report D.STE.07, 2008.

[6] M. Danelutto and G. Zoppi. Behavioural skeletons meeting services. In Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M.A. Sloot, editors, *Computational Science – ICCS 2008*, volume 5101 of *LNCS*, pages 146–153. Springer, 2008.

[7] Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in muskel. In *International Conference on Computational Science (2)*, pages 937–944, 2006.

[8] Niels Drost, Rob V. van Nieuwpoort, and Henri E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing*, Singapore, May 2006.

[9] Matthias Ehrgott and Xavier Gandibleux, editors. *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*, volume 52 of *International Series in Operations Research & Management Science*. Springer, 2002.

[10] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[11] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM.

[12] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. User-friendly and reliable grid computing based on imperfect middleware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)*, nov 2007. Online at http://www.supercomp.org.

[13] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an efficient Java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

[14] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[15] Gosia Wrzesinska, Jason Maassen, and Henri E. Bal. Self-adaptive applications on the grid. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, San Jose, CA, USA, March 2007.