Autonomic Components in GCM

Marco Aldinucci, M. Danelutto, M. Vanneschi, D. Laforenza, N. Tonellotto, S. Campa, P. Dazzi, G. Zoppi, P. Kilpatrick

University of Pisa Italy, ISTI-CNR Italy, QUB Belfast UK

CoreGRID Institute on Programming Model

Scientific Advisory Board meeting, Amsterdam, Apr 29th, 2008

aldinuc@di.unipi.it
http://www.coregrid.net
Part I (assessed work)

- Motivations
  - GCM (coreGrid Component Model)
  - why adaptive and autonomic management, why skeletons
  - behavioural skeletons (in insulation)
  - demo

Part II (ongoing and future work)

- formalisation of component and services
- component and service is not a dichotomy
- static and dynamic adaptation should not be a dichotomy

Activities held in
- CoreGRID Institute on Programming Models
- GridCOMP spin-off project (STREP)
**CGM model key points**

- Hierarchic model
  - expressiveness
  - structured composition

- Interactions among components
  - collective/group
  - configurable/programmable
  - not only RPC/RMI, but also stream/event

- Non-Functional aspects and QoS control
  - autonomic computing paradigm
  - adaptive and autonomic components

---

Part I: Motivations
Why Autonomic Computing

// programming & the grid

- concurrency exploitation, concurrent activities set up, mapping/scheduling, communication/synchronisation handling and data allocation, ...
- manage resources heterogeneity and unreliability, networks latency and bandwidth unsteadiness, resources topology and availability changes, firewalls, private networks, reservation and jobs schedulers, ...

... and a non trivial **user-defined** QoS for applications

not easy leveraging only on middleware

our approach: high-level methodologies + tools
Why Autonomic Computing
(User-defined QoS requirements for Apps)

- **Performance**
  - the app should sustain $x$ transactions per second
  - the app should complete each transaction in $t$ seconds

- **Security**
  - the link between $P1$ and $P2$ should be secured with $k$-strong encryption
  - the DB service is exposed by platform $P3$

- **Fault-tolerance**
  - the parallel server should survive to the failure of $y$ platforms

... then consider that $x$, $t$, $P1$, $P2$, $P3$, $k$, $y$ can dynamically change as may dynamically change the performance and the state of the running environment
**Why skeletons**

- **Management is difficult**
  - application change along time (ADL not enough)
    - how “describe” functional, non-functional features?
  - the low-level programming of component and its management is simply too complex

- **Component reuse is already a problem**
  - specialising component yet more with management strategy would just worsen the problem
  - especially if the component should be reverse engineered to be used (its behaviour may change along the run)
Behavourial Skeletons Idea

- Represent an evolution of the algorithmic skeleton concept for component management
  - abstract parametric paradigms of component assembly
  - specialised to solve one or more management goals
    - self-configuration/optimization/healing/protection
  - carry a semi-formal/formal description and an implementation
    - they are component factories, actually

- Are higher-order components

- Are not exclusive
  - can be composed with non-skeletal assemblies via standard components connectors
  - overcome a classic limitation of skeletal systems
Part I: BeSke (in insulation)

Functional replication (GCM implementation)

1. Choose a schema
   e.g. functional replication
   ABC API is chosen accordingly

2. Choose an inner component
   compliant to BeSke constraints

3. Choose behaviour of ports
   e.g. unicast/from_any, scatter/gather

4. Run your application
   then trigger adaptations

5. Automatise the process
   with a Manager

ABC = Autonomic Behaviour Controller (implements mechanisms)
AM = Autonomic Manager (implements policies)
B/LC = Binding + Lifecycle Controller
Example: farm

Part I: BeSke (in insulation)
Example: Data Parallel (Stateless)
**Notes**

- any number of server and client ports (either RPC or stream, in theory)
- the model cannot (structurally) enforce init happens before requests on other ports
- port reconfiguration and data redistribution should be atomic (no tasks should be distributed in the middle)
- data can be reconfigured in a distributed way (provided a suitable data port abstraction is defined)
Variations and Flavours (examples)

- **Functional Replication**
  - Farm/parameter sweep (**self-optimization**)
  - Data-Parallel (**self-configuring** map-reduce)
  - Active/Passive Replication (**self-healing**)

- **Proxy**
  - Pipeline (**coupled** self-protecting proxies)

- **Wrappers**
  - Facade (**self-protection**)

Part I: BeSke (in insulation)
**Farm Example (Mandelbrot)**

- change // degree
- new contract (e.g. $T_s < k$)
- get_service_time
- raise "contract violation"
- unicast
- from_any

[Diagram of a farm example involving Mandelbrot, with nodes labeled as ABC, farm, mandelbrot, lines gen, mandelbrot, and screen output.]
demo
European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

Part I: BeSke (in insulation)

Progress

- component model
- methodology
- programming tools
- NF & F features

- middleware
- CoreGRID kick-off 04

- naming
- communication
- deployment
- sharing

- monitoring API
- reconfiguration API
- passive BeSke

- management policies
- QoS contracts
- manager engine

- adaptive components
- GridCOMP kick-off 06

- autonomic components
- management co-ordination mechanisms and policies
- many open problems

- component model features
- now

CoreGRID

Area of interest
formalisation of component and services
adaptations, QoS contracts, orchestration of managers (as services)
component and service is not a dichotomy
from GCM/Proactive to SCA/Tuscany
static and dynamic adaptation should not be a dichotomy
if we care about performance
Manager formalisation & design

- Hierarchic assemblies of components that may structurally change at run-time. Issues:
  - Formally represent adaptations
    - they should be described in the AM and automatically applied
    - the ADL give just a static view of the assembly
  - Formally represent QoS contracts
    - they should be described in the AM and automatically evaluated
    - they should be projected and joint (almost automatically)
  - Describe the interaction/orchestration among managers
    - Globally, managers describe a distributed algorithm

Some hints presented here

... but still many open problems (just a few discussed here)
Formalising Adaptations

- Graphs + graph rewriting
  - rewriting rules represent possible adaptation patterns
    - enough expressive ... even too much
    - some formalisation do not capture important concepts for computing such as locality of the rewriting, context-dependence correctness, ... 
    - e.g. double push-out, Milner’s bi-graphs
  - restricting general graph rewriting
    - Synchronised Hyperhedge Replacement (SHR, from Sensoria IP-FP6)
    - Architectural Design Rewriting (ADR, forthcoming)

Implementing concepts in GCM

- *when-event-if-cond-then-act* list of rules
  - where *act* either an adaptation or a message to a set of companion managers
  - as JBoss beans
Example: SHR (Synchronised Hyperedge Replacement)

- move component \( f \) from \( l \) to \( l' \) (keep state)
  
  
- move component \( f \) from \( l \) to \( l' \) (fresh state)
  
- replicate component (keep state, change location)
  
  Example: AM ask component \( f \) to change location and attach to a new external state (application of 2nd rule - Aldinucci, Tuosto)

---

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

- $C_x = \text{Component } x$
- $C_x', C_x'' = \text{Instances of } C_x$
- $M(C_x) = \text{Manager of } C_x$

Qos contract (from users)

Structural relationships
Management overlay network
Functional network

Part II: BeSke (orchestration)
**Orchestration of Management**

- Managers are orchestrated via an overlay network
  - in GCM naturally hierarchic (sort of “synch fat-tree”)
    - however, the orchestration between children of the same node is not restricted and can be set up according to a user-defined goal
  - in general, no restrictions
- Methodologies to reason about management
  - e.g. manager orchestration as service orchestration
    - Orc to describe their orchestration (Misra, Cook, Hoare, ...)
    - reason on Orc programs to prove management global properties
      - semi-formal reasoning for Orc (Aldinucci, Danelutto, Kilpatrick)
      - papers at Europar 07, CoreGRID Symposium 07, IEEE PDP 08, ...
Different Orchestrations (Examples)

a) flat management orchestration

b) ring management orchestration

c) clustered management orchestration
Component, services or both?

- We re-defined and implemented autonomic BeSke in SCA/Tuscany
  - Proof-of-concept implementation
  - JBoss rule-based manager

- Few differences
  - Manager: JBoss rules vs POJO code
  - Protocols: standard XML/SOAP vs Proactive
  - Binding: static vs dynamic

- Proposal for standard extension
  - Dynamic binding of components
  - Tuscany people shown interest
SCA/Tuscany Farm Performances

Part II: Component & Services

Measured Completion Time vs. Ideal Completion Time

Computation time (secs) vs. #workers

Half of the tasks vs. half of the tasks

European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
new workers are mapped on empty nodes

new workers are mapped on nodes already running other instances of the same component

**Analysis: Overheads (GCM/Proactive)**

Overhead (ms) vs. N. of workers

- **New**
- **Stop**
### Analysis: Overhead (Alternative Impl)

**ASSIST/C++ overheads (ms)**


<table>
<thead>
<tr>
<th>parmod kind</th>
<th>Data-parallel (with shared state)</th>
<th>Farm (without shared state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reconf. kind</td>
<td>add PEs</td>
<td>remove PEs</td>
</tr>
<tr>
<td># of PEs involved</td>
<td>1→2  2→4  4→8</td>
<td>2→1  4→2  8→4</td>
</tr>
<tr>
<td>$R_l$ on-barrier</td>
<td>1.2  1.6  2.3</td>
<td>0.8  1.4  3.7</td>
</tr>
<tr>
<td>$R_l$ on-stream-item</td>
<td>4.7  12.0  33.9</td>
<td>3.9  6.5  19.1</td>
</tr>
<tr>
<td>$R_t$</td>
<td>24.4  30.5  36.6</td>
<td>21.2  35.3  43.5</td>
</tr>
</tbody>
</table>
It is just C++ against Java?

No, unfortunately it is not so simple ...

- dynamic class loading (red vs blue zone of the previous chart)
- dynamic introspection
- dynamic binding
- generic data serialisation, shared data alignment
- JIT, code factories, etc.
- non optimised protocols
  - look-ahead resource recruiting
  - pre-deployment
  - atomic multicast (replica management)
  - consensus (reconf-safe-points)
- and at the end ... C++ is usually a bit faster than Java
SUMMING UP ...

- exploit both static and dynamic techniques
  - represent adaptations as graph transformations
    - in such a way only correct configuration can be generated (e.g. as types)
    - QoS constraints with free variables
  - bound free variables with values
    - free variables can be bound at compile, launch time with constant or non-constant values
    - manage adaptation accordingly

- uniformly define static and dynamic adaptations
  - apply them the earlier is possible
    - compile/deploy/launch/run-time
  - here abstraction (e.g. high-level BeSke) become crucial
  - idiom recognition and generative approach
Conclusions

* Behavioural Skeletons
  - templates with built-in management for the App designer
  - methodology for the skeleton designer
    - management can be changed/refined
    - just prove your own management is correct against skeleton functional description
  - can be freely mixed with standard GCM components
  - already implemented on GCM (GridCOMP STREP)

* Future work
  - many interesting open problems
    - irrespectively of buzzwords (e.g. grid/cyber-infrastructure)
    - irrespectively specific technologies (e.g. component/services)
  - this might mean we are trying to address the core of the problems
related CoreGRID TR

   Hierarchical autonomic management: a case study with skeletal systems.

   Prototyping and reasoning about distributed systems: an Orc based framework.

   Deriving Grid Applications from Abstract Models.

   Fault-tolerant data sharing for high-level grid programming: a hierarchical storage architecture.

5. M. Aldinucci, A. Benoit.
   Automatic mapping of ASSIST applications using process algebra.

   Parallel program/component adaptivity management

7. J. Dünnweber, S. Gorlatch, M. Aldinucci, S. Campa, M. Danelutto.
   Behavior Customization of Parallel Components for Grid Application Programming.

   Optimization Techniques for Implementing Parallel Skeletons in Distributed Environments.