**Autonomic QoS Control with Behavioral Skeleton**

*Marco Aldinucci, Marco Danelutto, Sonia Campa*

Computer Science Dept. - University of Pisa - Italy

*Patrizio Dazzi, Nicola Tonellotto*

ISTI - CNR - Pisa - Italy

**GridCOMP WP3**

21 Oct 2008 - Sophia Antipolis - France

GridCOMP conference ProActive/GCM user group
Outline

- Motivation
  - why adaptive and autonomic management
  - why skeletons for high-level programming

- Behavioural Skeletons in GCM
  - parametric composite component with management
  - functional and non-functional description
  - families of behavioural skeletons
  - distributed overlay of management

- Demo
Why Autonomic Computing

- Scientific and industrial applications do require QoS control
  - QoS figures of a distributed application can hardly be predicted in static way
    - unstable platforms, irregular applications, dynamically changing requirements ...
  - QoS is often contractually specified; infringement of it may be fined
  - industry needs the dynamic sizing of applications (and their QoS) to expand market share while keeping design and tuning cost limited
    - design application once in a scalable way, sell it to many clients of different size
- QoS is a first-class concept of the emerging services/utility business
  - cloud, SaaS, PaaS, etc.
  - business/price may greatly depend by QoS, and vice-versa
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 ...
**Autonomic Computing paradigm**

- **monitor**: collect execution stats: machine load, service time, input/output queues lengths, ...
- **analyse**: instantiate performance models with monitored data, detect broken contract, in and in the case try to detect the cause of the problem
- **plan**: select a (predefined or user defined) strategy to re-convey the contract to validity. The strategy is actually a “program” using execute API
- **execute**: leverage on mechanism to apply the plan
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 development is already too difficult
  - how much of your time do you spend in run-time debugging and performance tuning?

- 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)
Behavioural 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 higher-order components (or 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
Be-Skeletons families

- Functional Replication
  - Farm/parameter sweep (self-optimization)
  - Stateless Data-Parallel (self-configuring map-reduce)
    - e.g. one server port (n of server ports is a parameter)
  - Stateful Data-Parallel (self-configuring stateful map-reduce)
    - e.g. two server ports: set_state and execute
  - Active/Passive Replication (self-healing)
- Proxy
  - Pipeline (coupled self-protecting proxies)
- Wrappers
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
Farm BeSke

Grid programming with components: an advanced COMPonent platform for an effective invisible grid
Farm BeSke (e.g. Mandelbrot)
Dicom demo: screen output

thumbnails of processed images

image detail

manager activity monitor & reaction

Grid programming with components: an advanced COMPonent platform for an effective invisible grid
rule "CheckInterArrivalRate"
  salience 5
  when
    $arrivalBean : ArrivalRateBean( value < ManagersConstants.LOW_PERF_LEVEL)
  then
    $arrivalBean.setData(ManagersConstants.notEnoughTasks_VIOL);
    $arrivalBean.fireOperation(ManagerOperation.RAISE_VIOLATION);
    System.out.println( "InterArrivalTime not enough - Raising a violation");
end

rule "CheckRateLow"
  when
    $departureBean : DepartureRateBean( value < ManagersConstants.LOW_PERF_LEVEL )
    $parDegree : NumWorkerBean(value <= ManagersConstants.MAX_NUM_WORKERS)
  then
    $departureBean.fireOperation(ManagerOperation.ADD_WORKER);
    $departureBean.fireOperation(ManagerOperation.BALANCE_LOAD);
    System.out.println( "Adding "+ManagersConstants.ADD_WORKERS+ "workers");
end

rule "CheckRateHigh"
  when
    $departureBean : DepartureRateBean( value > ManagersConstants.HIGH_PERF_LEVEL )
    $parDegree : NumWorkerBean(value > ManagersConstants.MIN_NUM_WORKERS)
  then
    $departureBean.fireOperation(ManagerOperation.DEL_WORKER);
    $departureBean.fireOperation(ManagerOperation.BALANCE_LOAD);
    System.out.println( "Rate "+$departureBean.getValue()+" (Removing 1 workers)");
end
Stateless Data Parallel BeSke
Stateful Data Parallel BeSke (e.g. IBM mockup)

1) references to DB slices are scattered
2) clients broadcast requests to all workers
3) each worker matches the fingerprint against its DB partition
4) clients get the answer OR(W1, W2, ...)
5) repeat 2-3-4 ... 2-3-4 ...
6) AM may sense a changed answer time (e.g. increased), due to a dataset size/kind and/or platform status change
7) AM reacts (e.g. increasing // degree): copying W1; bindings (external, AM, StorageComp) should be preserved; DB partitions (Wx state) should be redistributed via StorageComp

*Grid* programming with components: an advanced COMPonent platform for an effective invisible grid
Grid programming with components: an advanced COMPonent platform for an effective invisible grid
Overlay of management: motivation

1) push a QoS contract, e.g. low < Ts < high

2) run the application

3) suppose low > Ts

4) farm man react adding one or more workers to increase farm potential power

5) that is ok in many case, not always ...

6) if the farm is not receiving enough tasks the reaction is simply wrong
1) push a QoS contract, e.g. low < Ts < high

2) run the application

3) suppose low > Ts

4) farm man react as follows:
   if Tarrival > low then add_w
   else raiseup(not_enough_tasks)

5) man. now involve a global decision

6) as an example APP manager manager may ask
   lines gen manager to increase the task rate
Two tiers management demo (Mandelbrot)
Conclusions

- **Behavioural Skeletons in GCM**
  - 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
    - because once instanced, they are standard

- **Overlay of management**
  - relying on JBoss drools for manager policy
  - now supporting distributed overlay of management
    - e.g. hierarchical management