Grid programming with components: an advanced COMPonent platform for an effective invisible grid

GCM Non-Functional Features Advances

(Palma Mix)

Marco Aldinucci

&

M. Danelutto, S. Campa, D. Laforenza, N. Tonellotto, P. Dazzi

Unipi & ISTI-CNR

e-mail: aldinuc@di.unipi.it

© 2006 GridCOMP Grids Programming with components. An advanced component platform for an effective invisible grid is a Specific Targeted Research Project supported by the IST programme of the European Commission (DG Information Society and Media, project no. 034442)
GridComp MODEL key points

- Hierarchic model
  - Expressiveness
  - Structured composition
- Interactions among components
  - Collective/group
  - Configurable/programmable
  - Not only RPC, but also stream/event
- NF aspects and QoS control
  - Autonomic computing paradigm
GridComp MODEL key points
(some further thoughts)

- Hierarchic model
  - Expressiveness, how to avoid push everything in the API?
  - Structured composition, how to exploit it?

- Interactions among components
  - Collective/group, not only DP scatter/gather ...
  - Configurable/programmable, how to introduce polices?
  - Not only RPC, but also stream/event, is it true?

- NF aspects and QoS control
  - Autonomic computing paradigm, how avoid to set-up a very complex machinery to deal with Grid complexity?
GCM implementation status

- GCM features under refinement
- My fat-free (underhanded) wishes
  - Avoid fat specification
    - Any implementation will hardly be compliant
    - Maybe already too fat
  - Avoid fat implementation
    - Nobody will use it, especially in the HPC community
- Trying to add a “dietetic” QoS control
  - less possible impact on the middleware, thus if the users don’t want it, they should not spend time avoiding it
- layered architecture
Monitor: collect execution stats: machine load, service time, input/output queues lengths, ...

Analyze: instantiate performance models with monitored data, detect broken contract, in and in the case try to individuate the problem

Plan: select a (predefined or user defined) strategy to re-convey the contract to valid status. The strategy is actually a list of mechanism to apply.

Execute: leverage on mechanism to apply the plan
**Autonomic Components**

- **Management is difficult**
  - Application change along time (ADL not enough)
  - How “describe” functional, non-functional features and their inter-relations?
  - The low-level programming of component and its management is simply too complex

- **Component reuse is already a problem**
  - Specializing component yet more with management strategy would just worsen the problem
  - Especially if the component should be reverse engineered to be used (its behavior may change along the run)
**Behavioral Skeletons (BeSke)**

- Exploit skeleton idea for management
- Common parallel programming paradigms which management can be pre-determined
  - In a parametric way
  - Capturing several aspects of management
    - optimization, healing, configuration, protection
- Can carry an implementation
- Carry an explicit semantics
  - described via standard GCM ADL hook
- Implementation cannot automatically derived from the description
  - Description is useful to reason about management
BeSke Advantages

- Each skeleton carries a semantics
  - Restrict the orchestration of composite components
    - I.e. contextualize components with respect to nesting
  - are Higher-Order functions
  - Management may be parametric and pre-determined

- Behavior description
  - Parametric functional and non-functional behavior
  - Functional behavior should be invariant with respect to parameter
  - Non-functional behavior is not invariant
    - E.g. performance, robustness, healing likely, ...
More on description

- Aims to enable the designer to reason about management
  - functional description enumerate the possible evolutions of composite component
  - should comply with the intentional skeleton semantics
  - the management follows a path in this search space
  - the exploration is driven by evaluation of monitoring variables, through QoS formulas
    - some variables come from the membrane
    - some from inner components, in this case they should be required in the inner components
Fill the holes, in two steps

1. Scatter (S), Gather (G), AC & AM [skeleton designer]
2. Worker (W) & AM [application designer]
1) Specialize the skeleton with the behavior

- **Server port type (S)**
  - Broadcast, DP scattercast, **Unicast**
    - Unicast: One-to-One_in_a_Set, scheduling is done across different calls
    - not in GCM-proactive, we developed our own version

- **Client port type (G)**
  - From-any, GCM gathercast, reduce

- **Inner component pre-requisites**
  - E.g. stateless, one func. server and one func. client port

- **Describe functional behavior**
  - Currently in Orc (to be present CoreGRID@Heraklion)
2) **Use it**

- Instantiate the behavioral skeleton with inner components
- Select (statically or dynamically) the management goal and its parameters
Example: Farm

- S = Unicast, G = From-any, W is stateless
- Self-optimizing
  - goal = sustain at least K transaction/sec with minimal resource usage
- AC can
  - Monitor: length of the queue of requests, W load status
  - Execute: add/destroy an instance of W
- AM can
  - Heuristically keeps a low/high water mark, raise contract violation, accept new bounds
**Example: Data Parallel**

- S = Scatter, G = Gather, W is stateless
- Self-configuring
  - reconfiguring on new request
  - goal = keep resource balance (e.g. load, memory, disk ...)
- AC can
  - Monitor: resource usage on Ws
  - Execute: add/destroy an instance of W, change scatter/gather policy
- AM can
  - Compute new policies, recruit fresh resources
Example: Active Replication

- $S = \text{Broadcast}$, $G = \text{Reduce}$, $W$ is stateless
  - Reduce examples: average, vote, ...
- Self-healing
  - goal = tolerate fault, tolerate Byzantine workers, ...
- $AC$ can
  - Monitor: fault detectors
  - Execute: add/destroy an instance of $W$
- $AM$ can
  - Exclude workers from the, recruit fresh ones
Much More Under the Hood

- Other cases can be covered with the same skeleton
  - Gracefully extendible to stateful components
    - state serialization

- Other skeletons under design
  - Inspired to software engineering literature
    - proxy, wrapper, superimposition, ...
    - will cover self-protection and self-configuration mostly
Conclusions

- Work is going on
  - Theory consolidation
  - Implementation and user experience

- Current GCM status: mileage may vary
  - Exploring new formalization, e.g. behavioral skeletons
  - Development and learning curve
    - and consider we already implemented a similar system in C++ (ASSIST)
    - in many case we know what we would like to do, but we should find a suitable trick to avoid a middleware “feature”

- Middleware appears already a bit too fat?
  - Where is the error when the application does not work?
  - Performances non always satisfactory (experiments follows, tomorrow?)
Grid programming with components: an advanced COMPONENT platform for an effective invisible grid

CoreGRID: The European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and P2P Technologies

Communication Time (Int)

Communication time

Communication Bandwidth (Theoretical 12800 KB/s)
Communication Time (Integer)

Communication time

Communication Bandwidth (Theoretical 12800 KB/s)
Farm SpeedUp 1

Speedup vs n. of workers (Tw=40)

Tw(jobsizestar)=40 ms Tc(jobsizestar)=30 ms
Farm SpeedUp 2

Speedup vs msg size (8 nodes)

- 40 ms
- 80 ms
- 160 ms
- 320 ms
- Ideal

Speedup vs msg size (8 nodes)