Managed by

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

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

Outline

Activities held in
- CoreGRID Institute on Programming Models
- GridCOMP spin-off project (STREP)

- 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

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

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

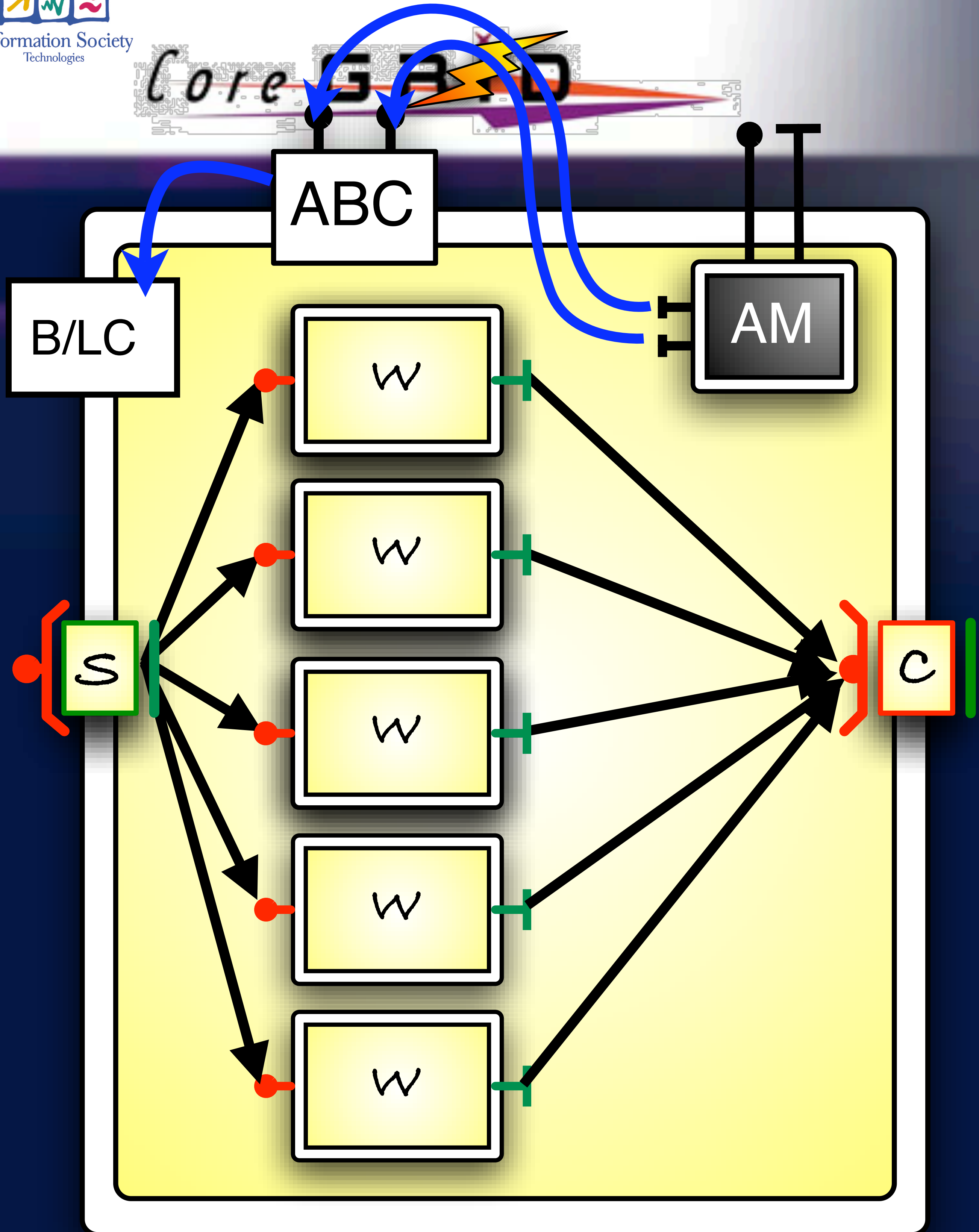
Why skeletons

- 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)

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 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

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

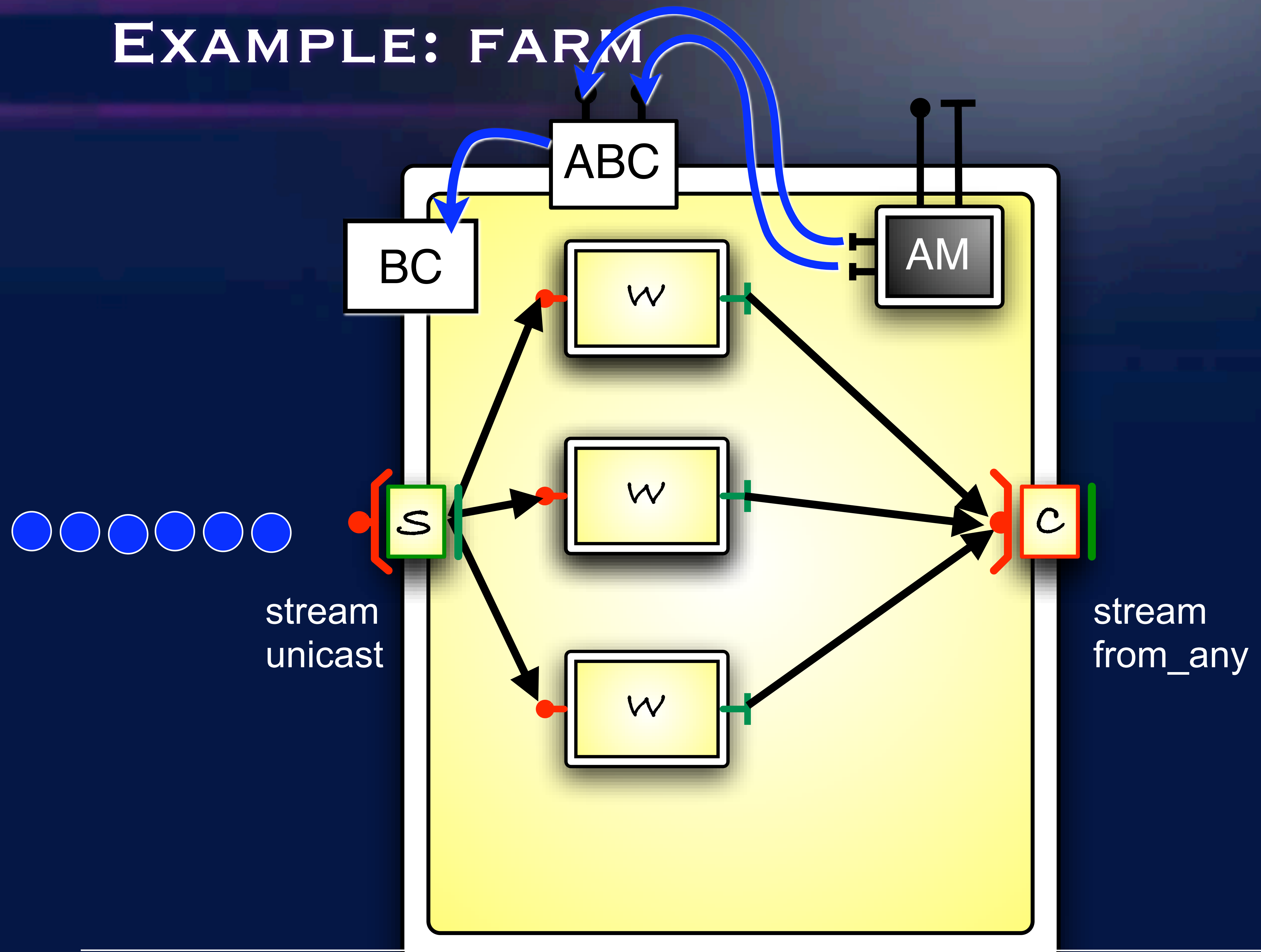
3. Choose behaviour of ports
   e.g. unicast/from_any, scatter/gather

4. **Run** your application
   then trigger adaptations

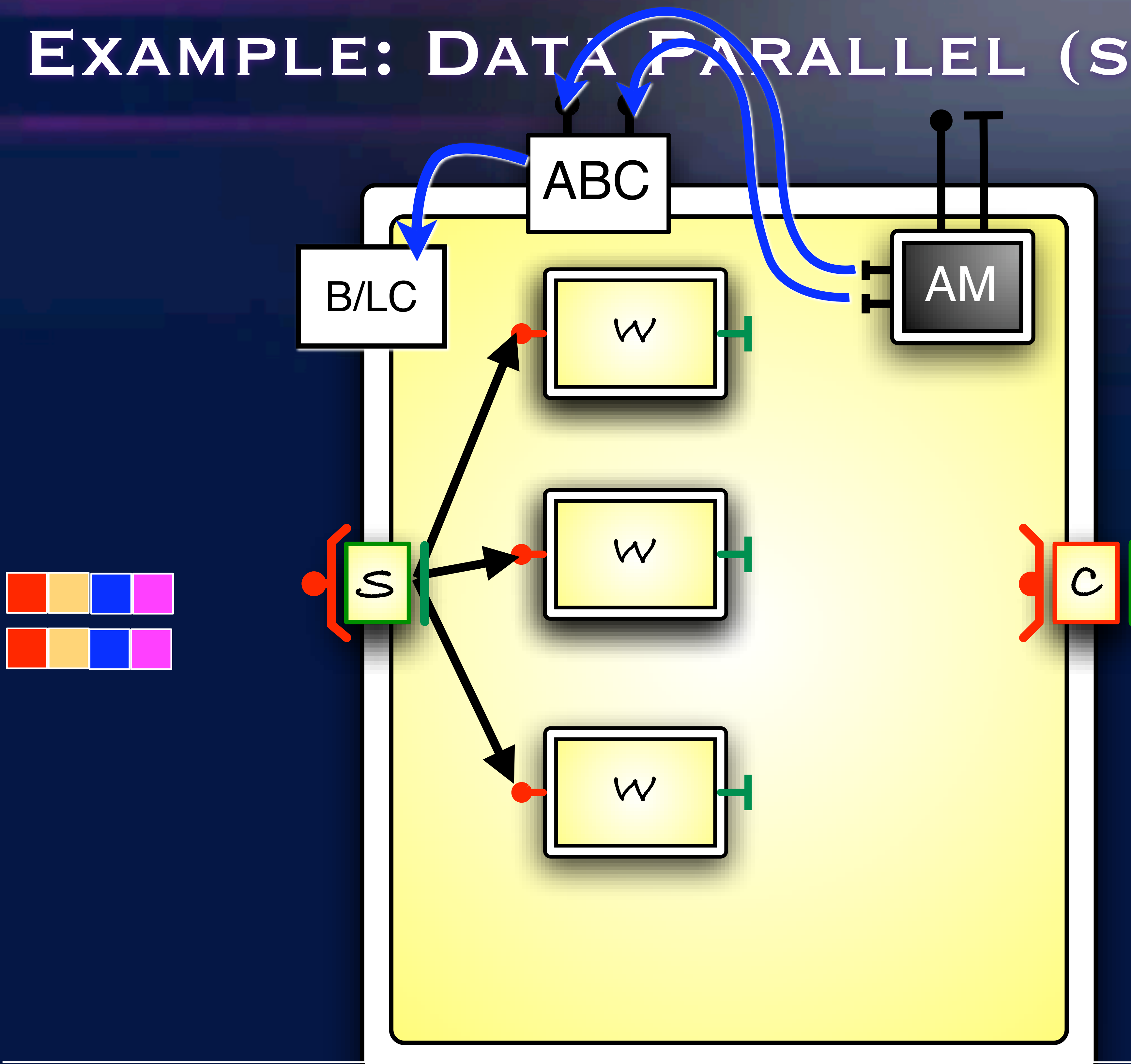
5. Automatise the process
   with a Manager

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

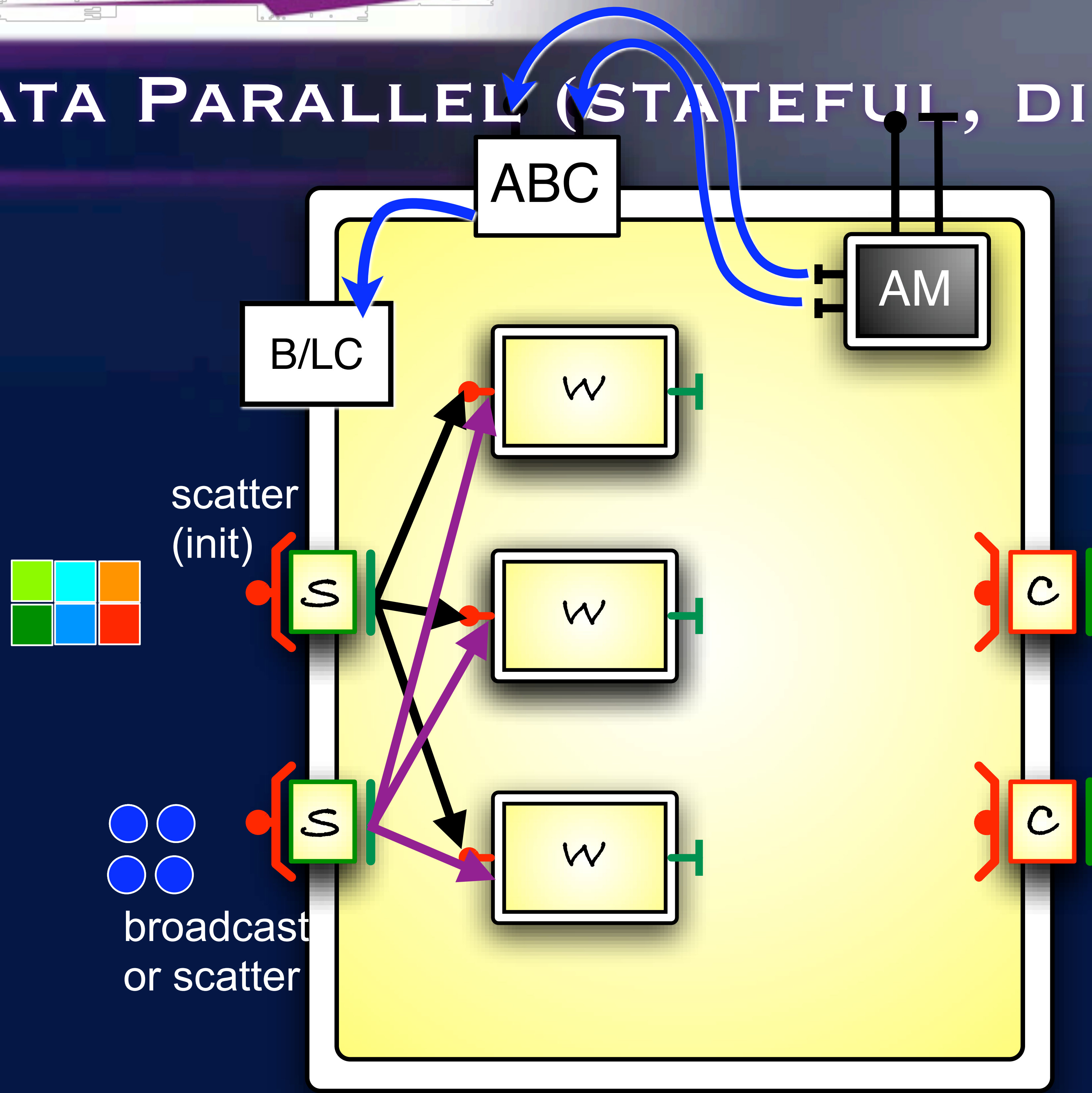
Example: farm



Example: Data Parallel (stateless)



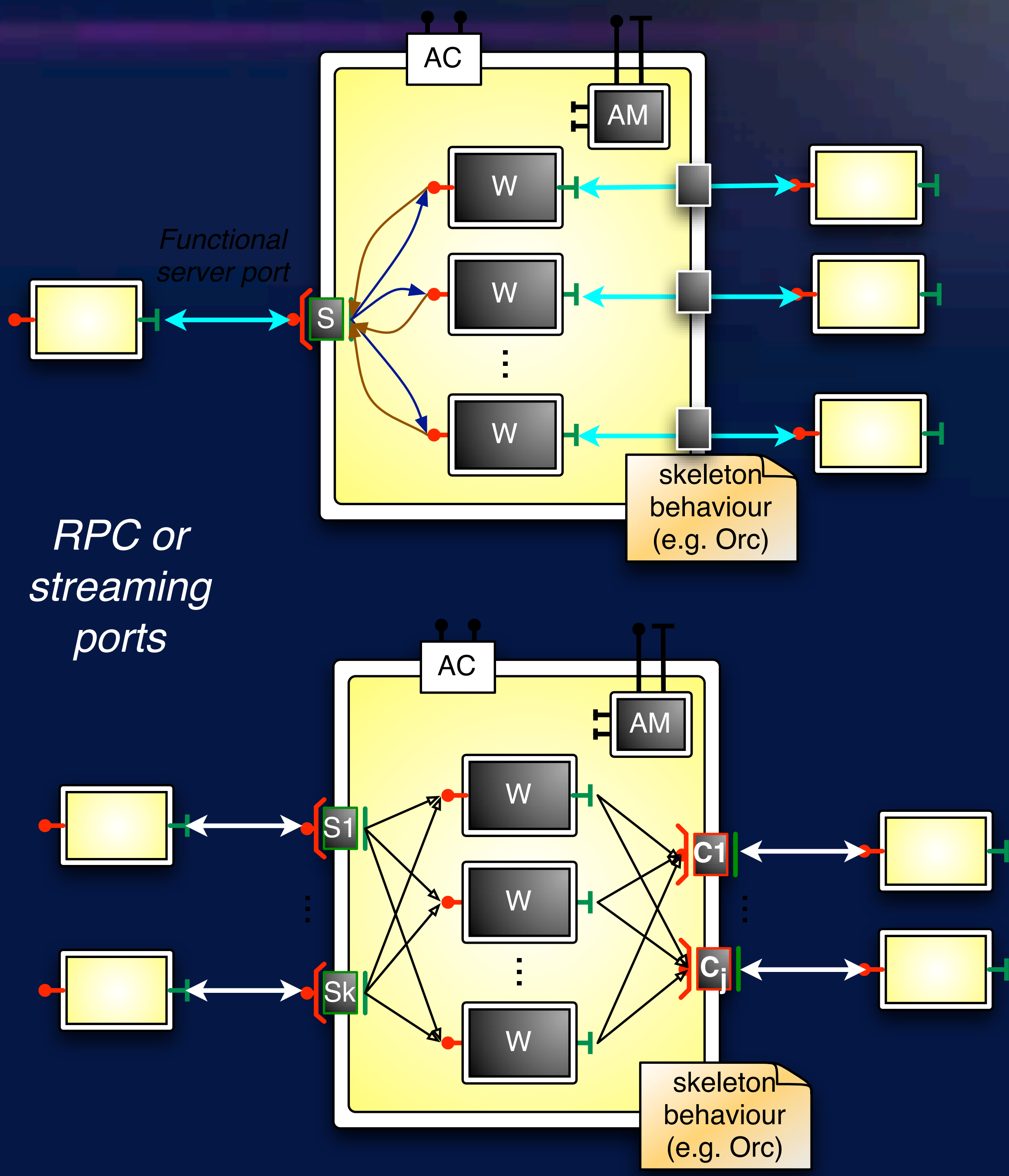
Data Parallel (stateful, distr. state)



Notes

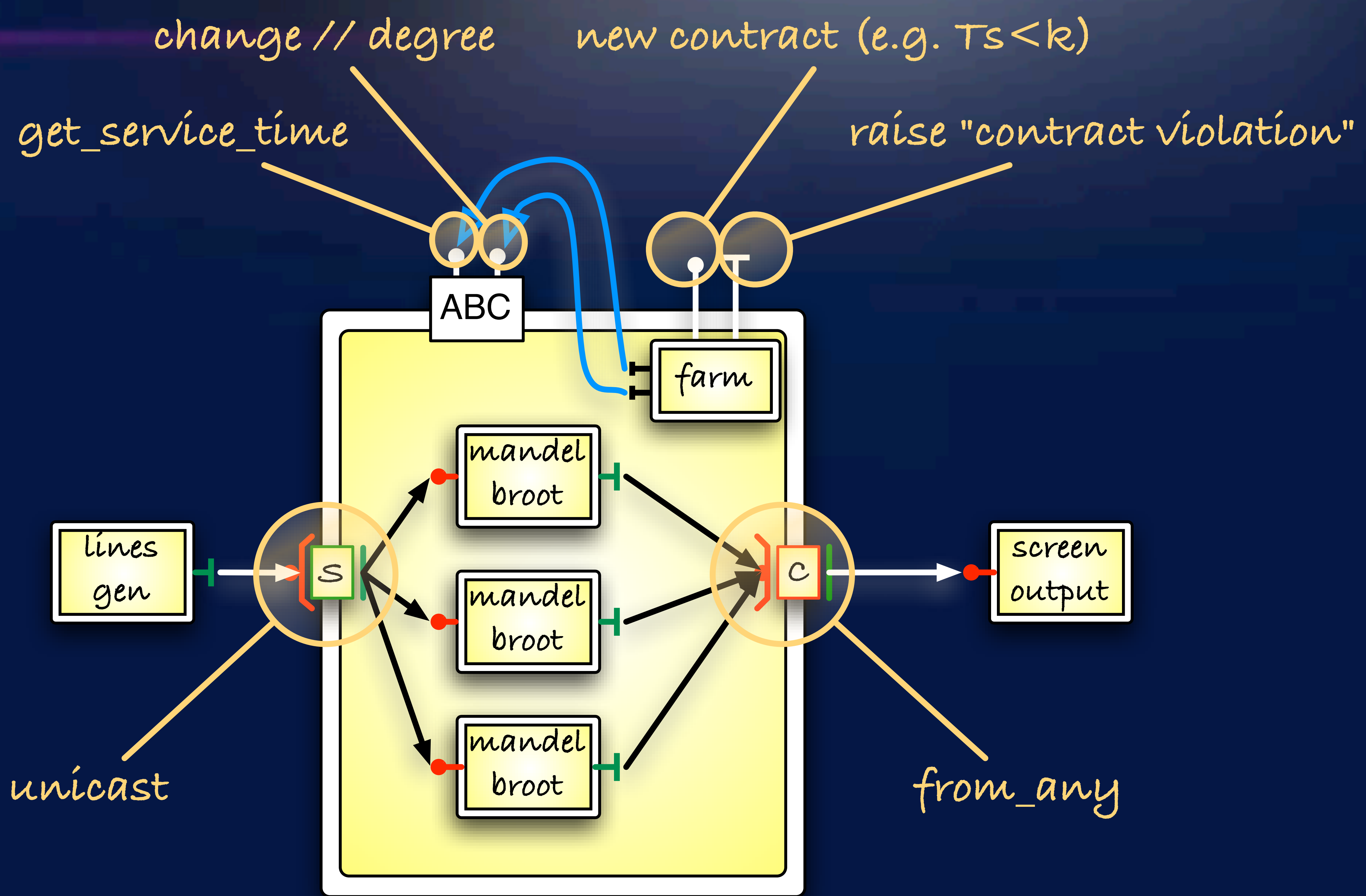
- any number of server and client ports (either RPC or stream, in theory)
- the model cannot (structurally) enforce init happens before requests on other ports
- port reconfiguration and data redistribution should be atomic (no tasks should be distributed in the middle.
- data can be reconfigured in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

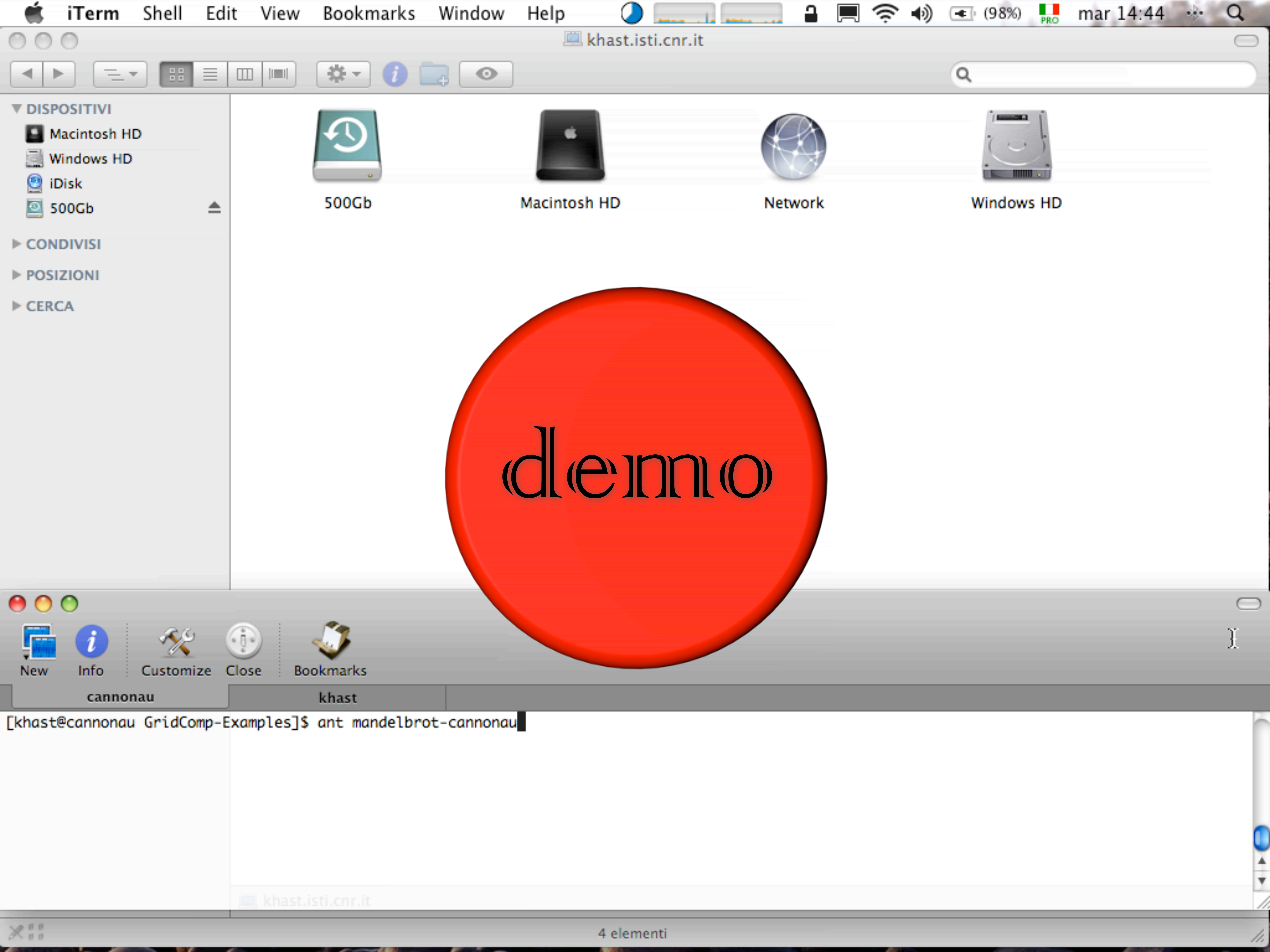
Variations and Flavours (examples)



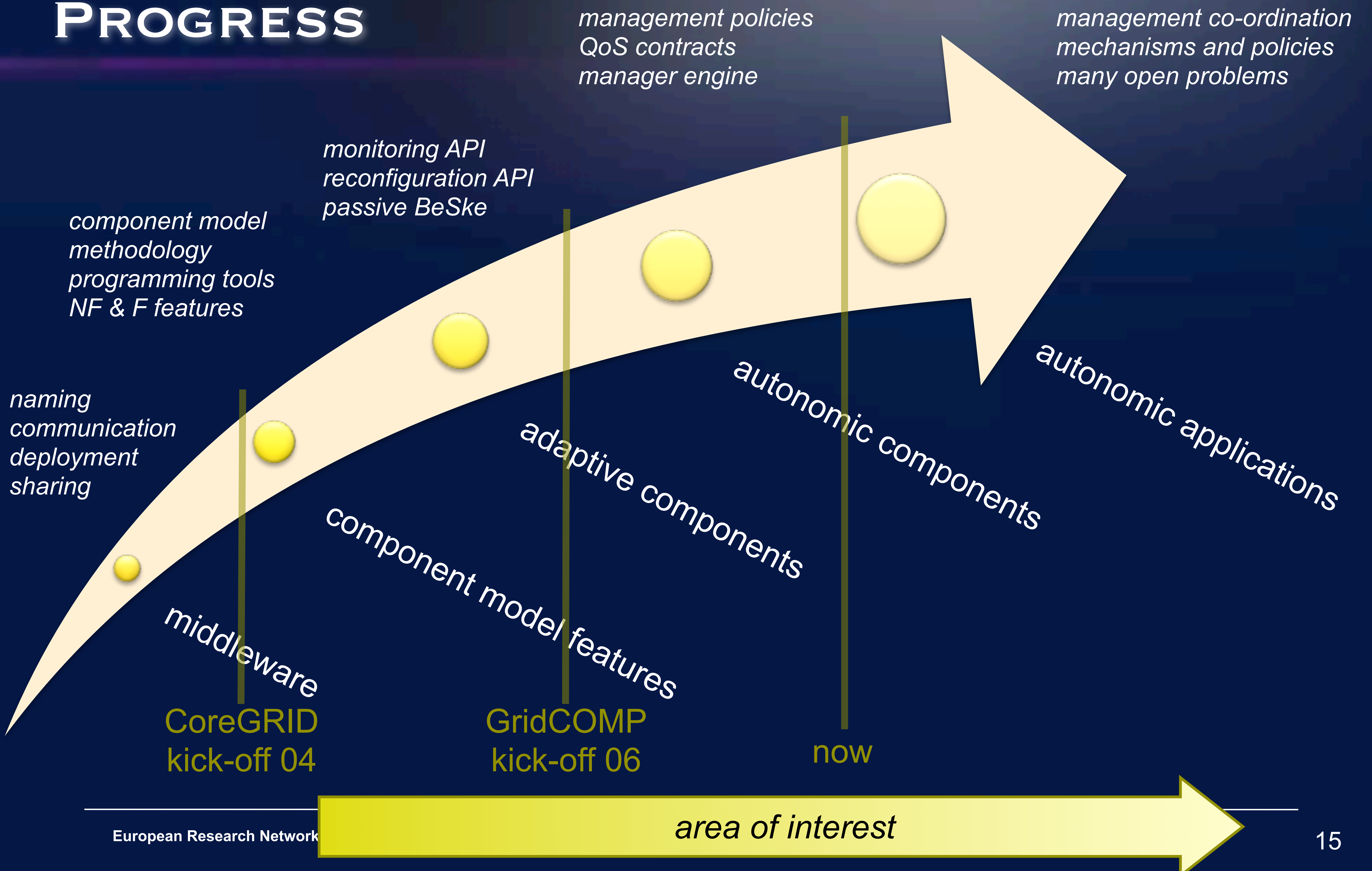
- Functional Replication
  - Farm/parameter sweep (self-optimization)
  - Data-Parallel (self-configuring map-reduce)
  - Active/Passive Replication (self-healing)
- Proxy
  - Pipeline (coupled self-protecting proxies)
- Wrappers
  - Facade (self-protection)

Farm example (Mandelbrot)

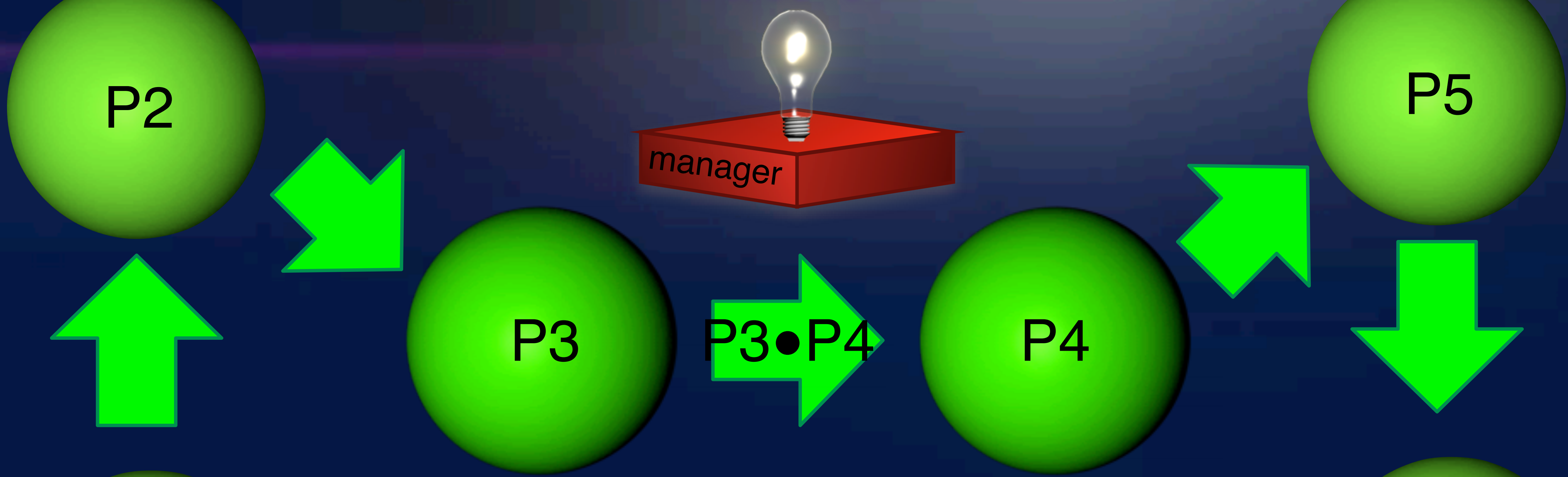


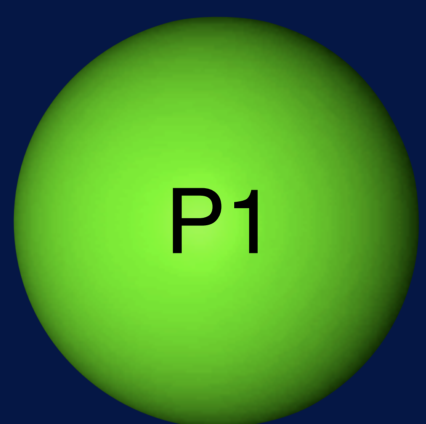

iTerm Shell Edit View Bookmarks Window Help
(98%) mar 14:44
khast.isti.cnr.it
DISPOSITIVI
Macintosh HD
Windows HD
iDisk
500Gb
CONDIVISI
POSIZIONI
CERCA
500Gb
Macintosh HD
Network
Windows HD
demo
New
Info
Customize
Close
Bookmarks
cannonau
khast
[khast@cannonau GridComp-Examples]$ ant mandelbrot-cannonau
4 elementi

Progress


management policies
QoS contracts
manager engine
management co-ordination
mechanisms and policies
many open problems
monitoring API
reconfiguration API
passive BeSke
component model
methodology
programming tools
NF & F features
naming
communication
deployment
sharing
middleware
component model features
adaptive components
autonomic components
autonomic applications
CoreGRID
kick-off 04
GridCOMP
kick-off 06
now
area of interest

Part II





- 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 component 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

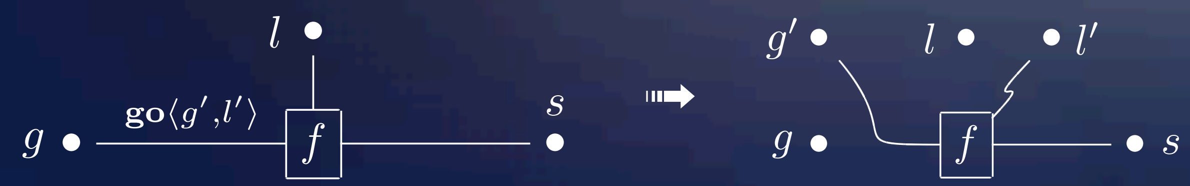
- Graphs + graph rewriting
  - rewriting rules represent possible adaptation patterns
    - enough expressive ... even too much
    - some formalisation do not capture important concepts for // computing such as locality of the rewriting, context-dependence correctness, ...
      - e.g. double push-out, 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)

Rules

move component
f from l to l'
(keep state)

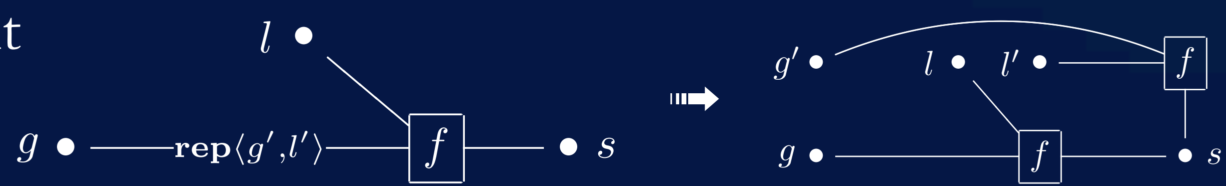


e.g. g = AM, g'=g,
s = external state

move component
f from l to l'
(fresh state)

e.g. g = AM, g'=g,
s, s' = external states

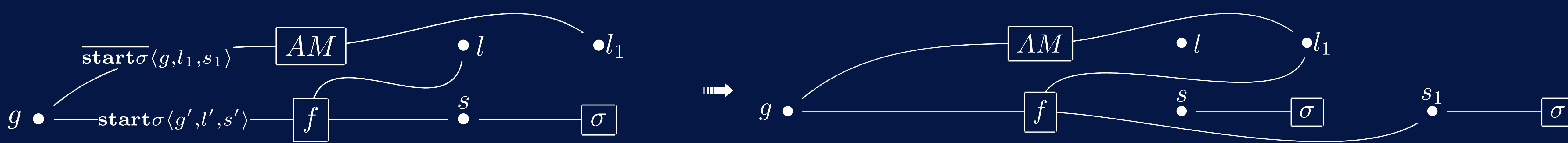
replicate component
(keep state,
change location)



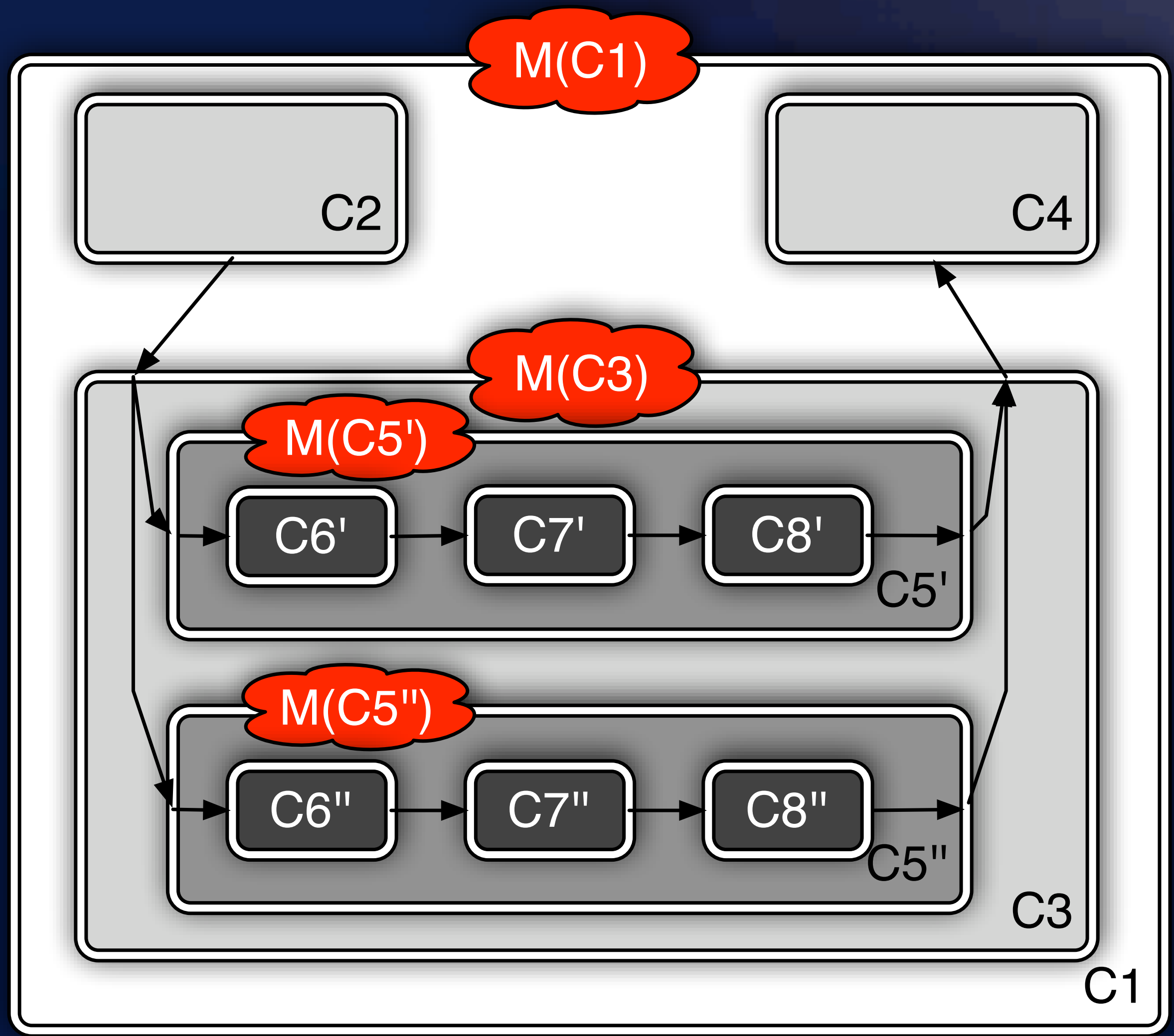
e.g. g = AM, g'=g,
s= external state

…

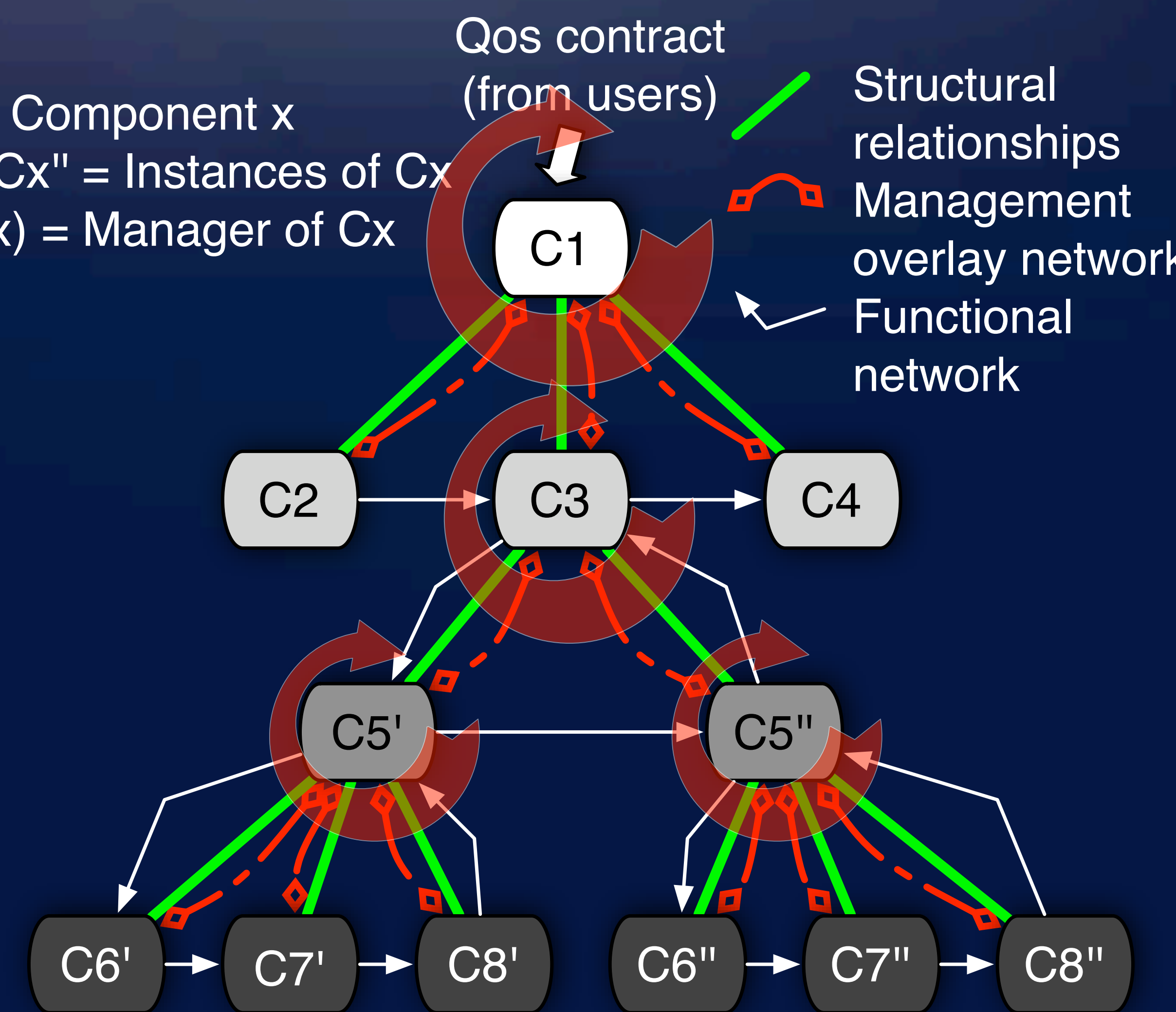
Example: AM ask component f to change location and attach to a new external state (application of 2nd rule - Aldinucci, Tuosto)



Orchestration of Management



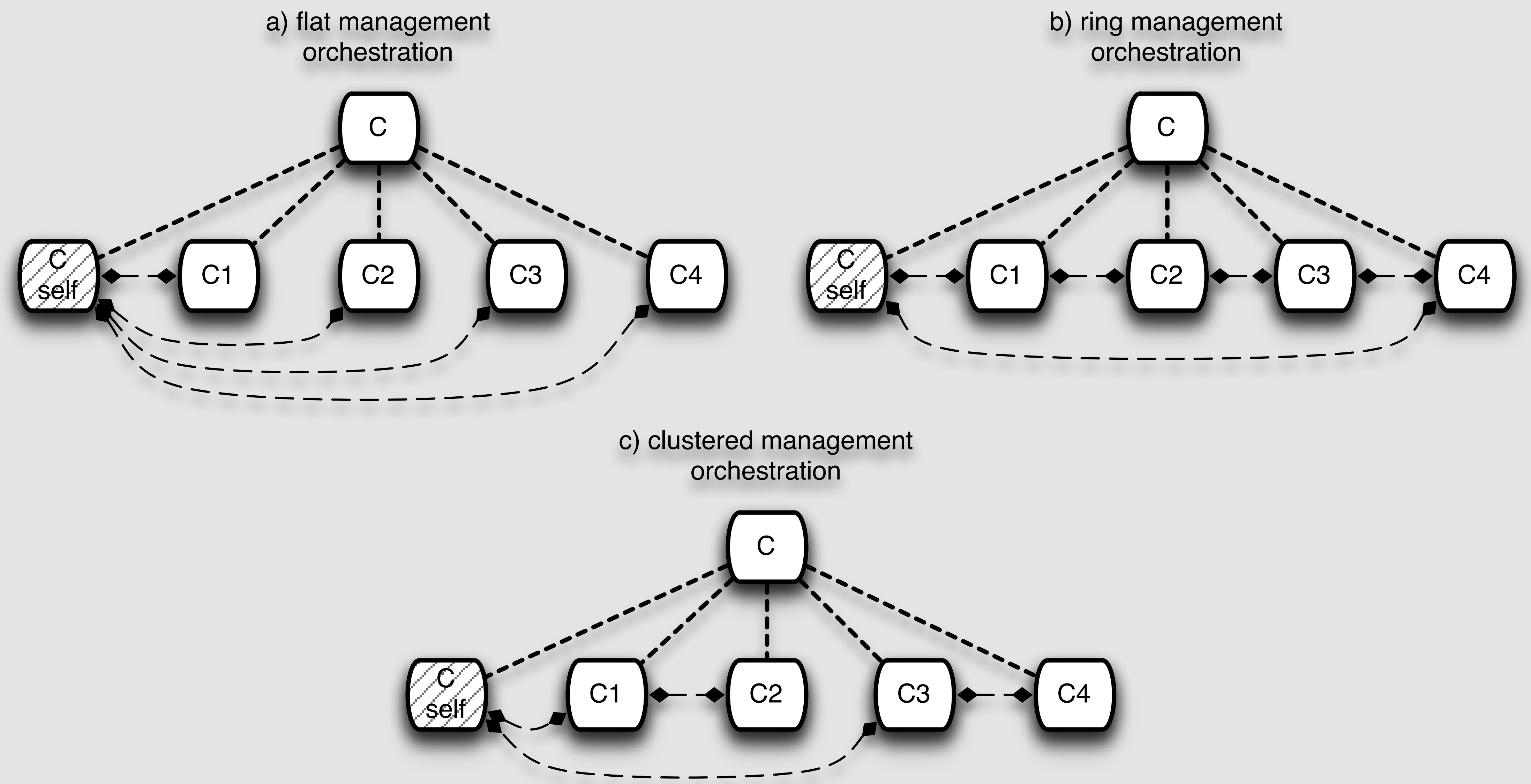
Cx = Component x
Cx', Cx" = Instances of Cx
M(Cx) = Manager of Cx



Orchestration of Management

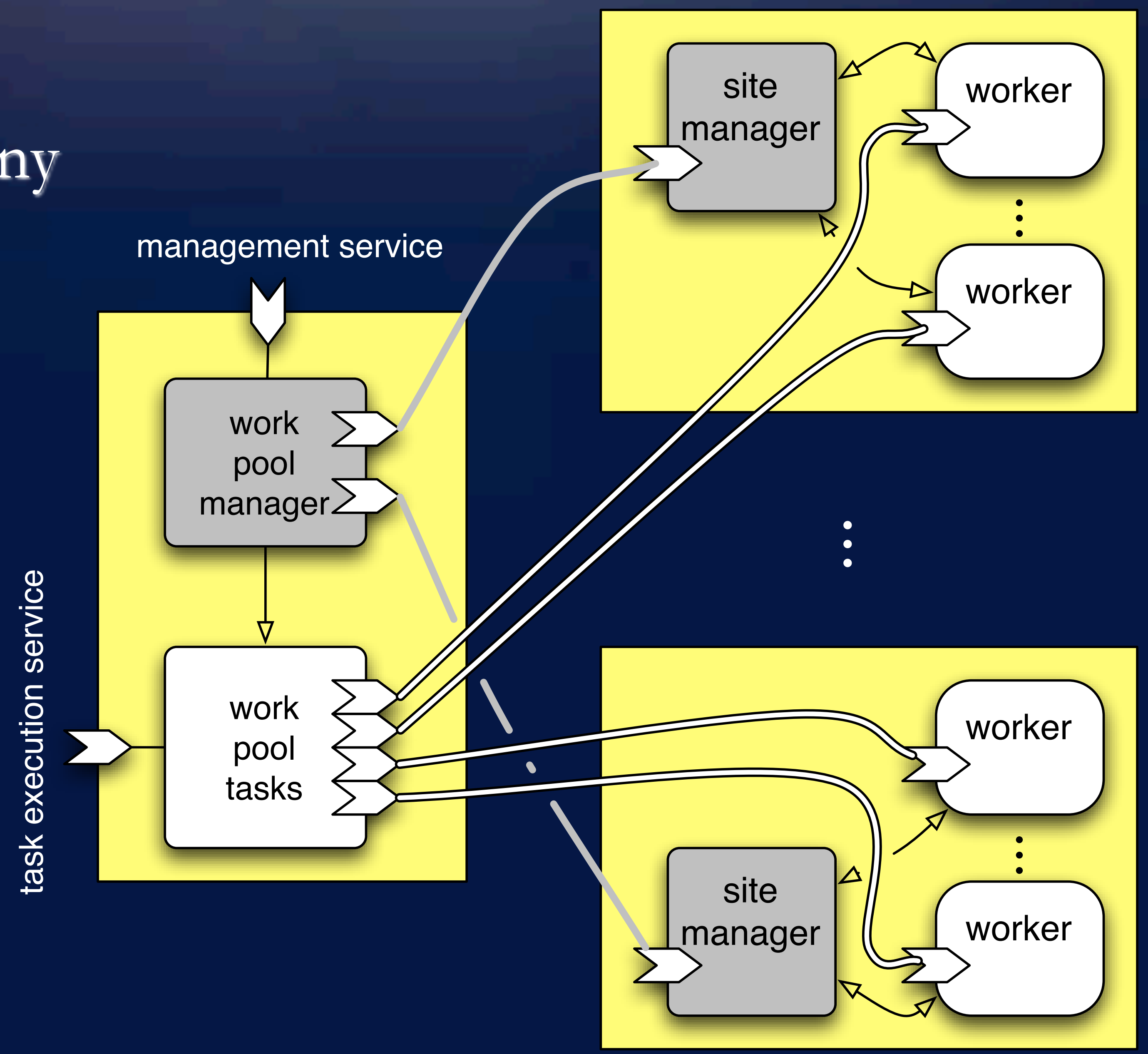
- Managers are orchestrated via an overlay network
  - in GCM naturally hierarchic (sort of "synch fat-tree")
    - however, the orchestration between children of the same node is not restricted and can be set up according to a user-defined goal
  - in general, no restrictions
- methodologies to reason 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)



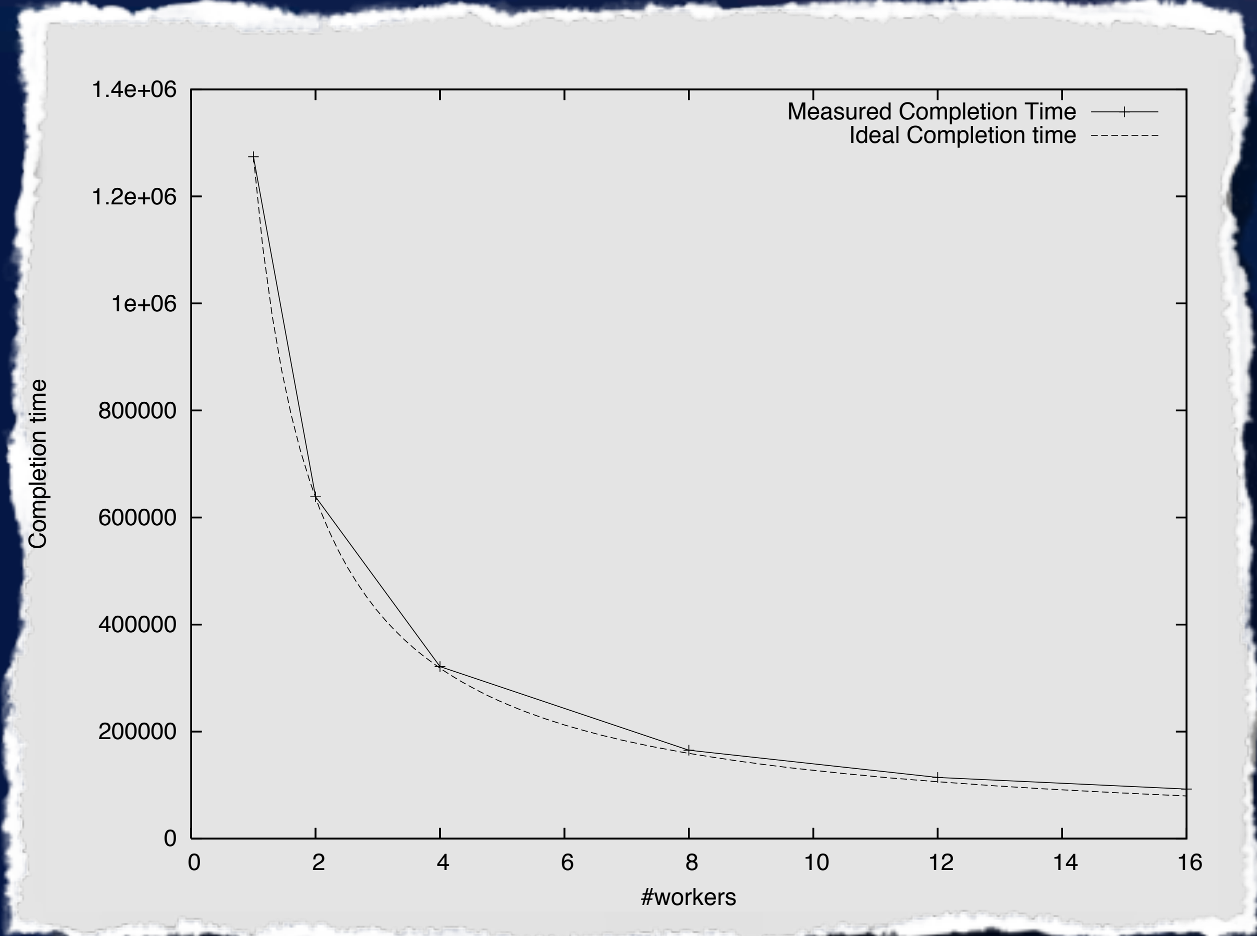
Component, services or both?

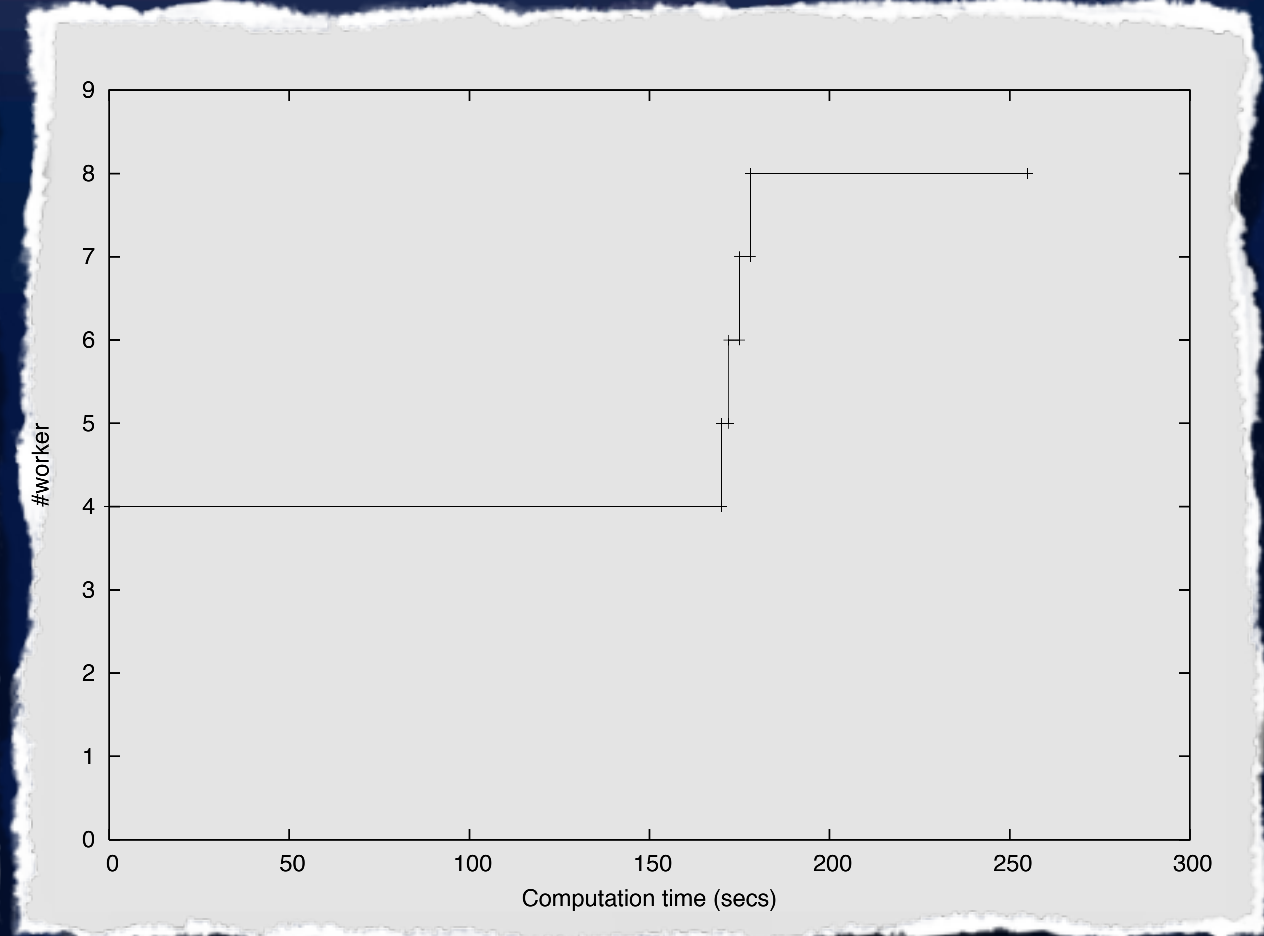
- we re-defined and implemented autonomic BeSke in SCA/Tuscany
  - proof-of-concept implementation
  - JBoss rule-based manager
- few differences
  - manager: JBoss rules vs POJO code
  - protocols: standard XML/SOAP vs Proactive
  - binding: static vs dynamic
- proposal for standard extension
  - dynamic binding of components
  - Tuscany people shown interest



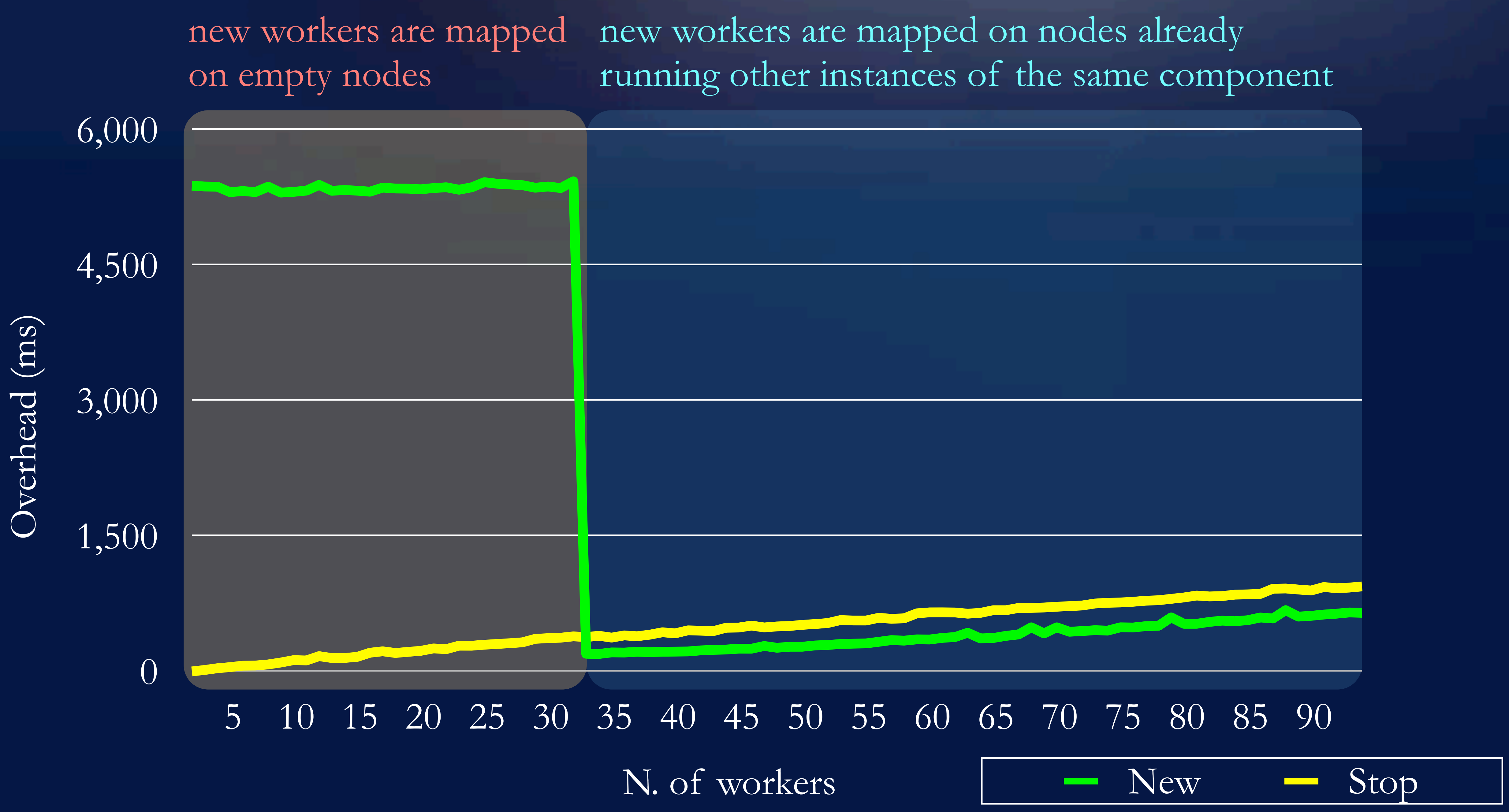
SCA/Tuscany farm performances

half of the tasks ⟷ half of the tasks





Analysis: Overheads (GCM/ProActive)



Analysis: Overhead (Alternative Impl)

ASSIST/C++ overheads (ms)

M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo.
Dynamic reconfiguration of grid-aware applications in ASSIST.
Euro-Par 2005, vol. 3648 of LNCS, Lisboa, Portugal. Springer Verlag, August 2005.

parmod kind	Data-parallel (with shared state)						Farm (without shared state)					
reconf. kind	add PEs			remove PEs			add PEs			remove PEs		
# of PEs involved	1→2	2→4	4→8	2→1	4→2	8→4	1→2	2→4	4→8	2→1	4→2	8→4
R_l on-barrier	1.2	1.6	2.3	0.8	1.4	3.7	–	–	–	–	–	–
R_l on-stream-item	4.7	12.0	33.9	3.9	6.5	19.1	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0
R_t	24.4	30.5	36.6	21.2	35.3	43.5	24.0	32.7	48.6	17.1	21.6	31.9

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

Information Society Technologies

CoreGRID

THANK YOU

related CoreGRID TR

1. M. Aldinucci, M. Danelutto, and P. Kilpatrick.
   Hierarchical autonomic management: a case study with skeletal systems.
   CoreGRID Technical Report TR-0127, February 2008.
2. P. Kilpatrick, M. Danelutto, M. Aldinucci.
   Prototyping and reasoning about distributed systems: an Orc based framework.
   CoreGRID Technical Report TR-0102, August 2007.
3. P. Kilpatrick, M. Danelutto, M. Aldinucci.
   Deriving Grid Applications from Abstract Models.
   Technical Report TR-0085, April 2007.
4. M. Aldinucci, G. Antoniu, M. Danelutto, M. Jan.
   Fault-tolerant data sharing for high-level grid programming: a hierarchical storage architecture.
   CoreGRID Technical Report TR-0058, August 2005.
5. M, Aldinucci, A. Benoit.
   Automatic mapping of ASSIST applications using process algebra.
   CoreGRID Technical Report **TR-0016**, October 2005.
6. M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, C. Zoccolo.
   Parallel program/component adaptivity management
   CoreGRID Technical Report TR-0014, September 2005.
7. J. Dünnweber, S. Gorlatch, M. Aldinucci, S. Campa, M. Danelutto.
   Behavior Customization of Parallel Components for Grid Application Programming.
   CoreGRID Technical Report TR-0002, April 2005.
8. M. Aldinucci, M. Danelutto, J. Dünnweber, S. Gorlatch.
   Optimization Techniques for Implementing Parallel Skeletons in Distributed Environments.
   CoreGRID Technical Report **TR-0001**, January 2005.