CoreGRID

12-13 JUNE 2007
Ηράκλειο, Κρήτη

Marco Aldinucci,
UNIPI
www.di.unipi.it/~aldinuc

BEHAVIOURAL SKELETONS FOR COMPONENT AUTONOMIC MANAGEMENT ON ~~GRIDS~~

CyberInfrastructure

Marco Aldinucci, M. Danelutto, S. Campa
University of Pisa, Italy
P. Kilpatrick
Queen's University Belfast, UK
D. Laforenza, N. Tonellotto, P.Dazzi
ISTI-CNR, Italy

Outline

- Motivation
  - Why behavioural (and autonomic management)
  - Why skeletons
- Behavioural Skeletons
  - parametric composite component with management
  - functional and non-functional description
  - families of behavioural skeletons
- GCM implementation
  - some running applications

CGM 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

Why Autonomic Computing

- // programming & the grid
  - concurrency exploitation, concurrent activities set up, mapping/scheduling, communication/synchronization 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, ...

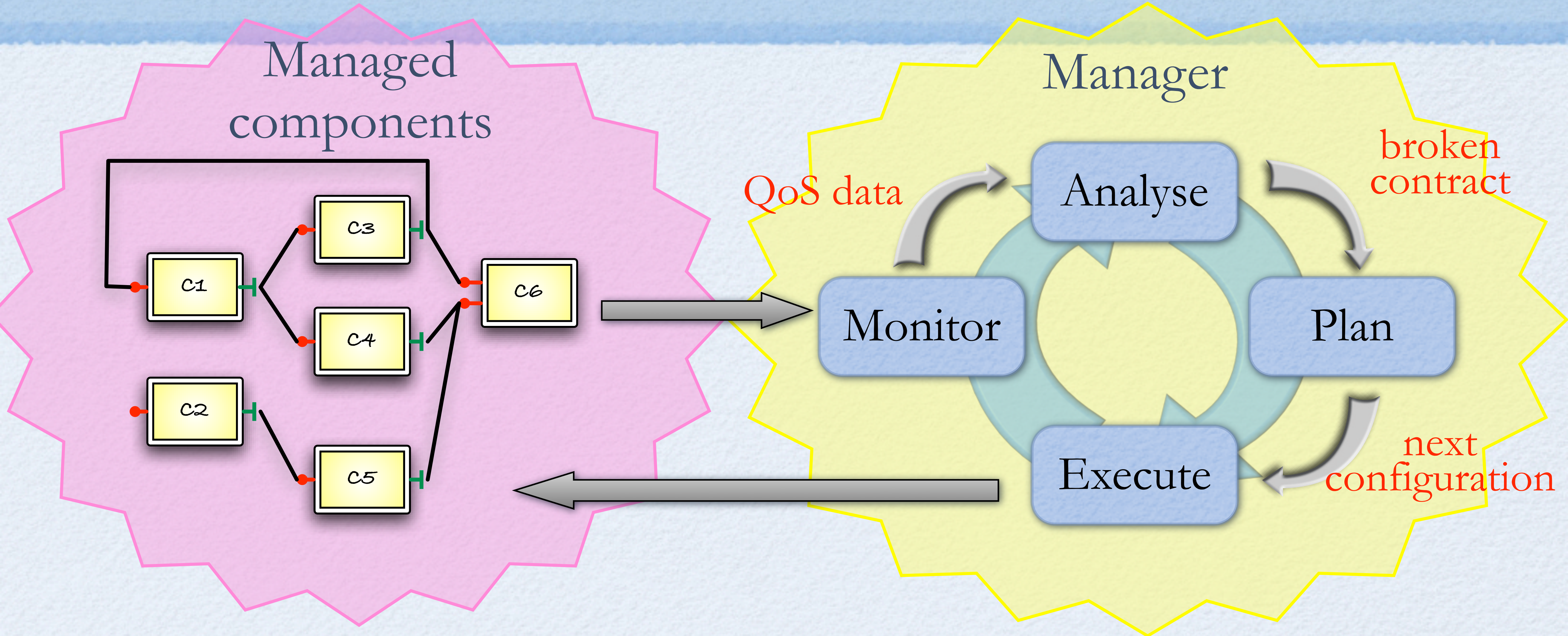
... and a non trivial QoS for **applications**

not easy leveraging only on middleware

GrADS@Rice, ASSIST, ...

high-level methodologies + tools

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 1/2

- 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
  - 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)

Why skeletons 2/2

- Skeletons represent patterns of parallel computations (expressed in GCM as graphs of components)
- Exploit the inherent skeleton semantics
  - thus, restrict the general case of skeleton assembly
  - graph of any component ➡ parametric networks of components exhibiting a given property
  - enough general to enable reuse
  - enough restricted to predetermine management strategies
- Can be enforced with additional requirements
  - E.g.: Any adaptation does not change the functional semantics

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/optimisation/healing/protection.
- 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

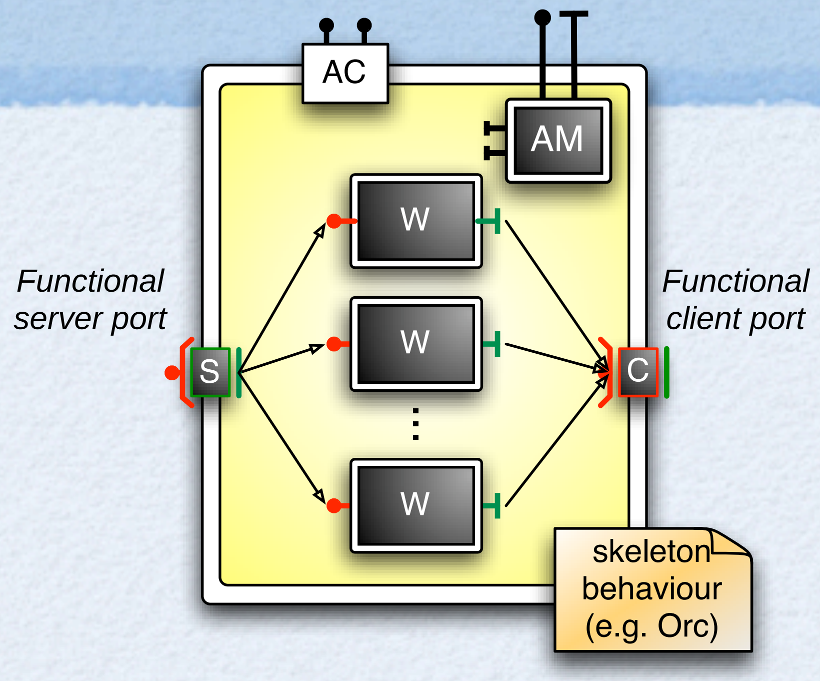
Behavioural Skeletons proprieties

- expose a description of its functional behaviour
- establish a parametric orchestration schema of inner components
- may carry constraints that inner components are required to comply with
- may carry a number of pre-defined plans aiming to cope with a given self-management goal
- carry an implementation (they are factories)

Be-Skeletons families

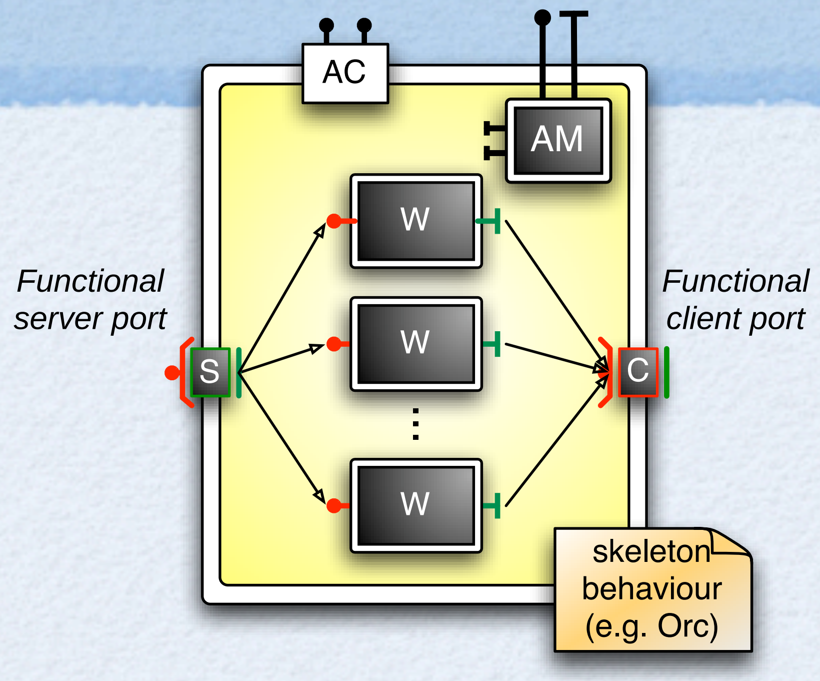
- Functional Replication
  - Farm/parameter sweep (self-optimization)
  - Simple Data-Parallel (self-configuring map-reduce)
  - Active/Passive Replication (self-healing)
- Proxy
  - Pipeline (coupled self-protecting proxies)
- Wrappers
  - Facade (self-protection)
- Many others can be borrowed from Design Patterns

Functional replication



- Farm
  - S = unicast, C = from_any, W = stateless inner component
- Data Patallel
  - S = scatter, C = gather, W = stateless inner component
- Fault-tolerant Active Replication
  - S = broadcast, C = get_one_in_a_set, W= stateless inner ...

Functional replication

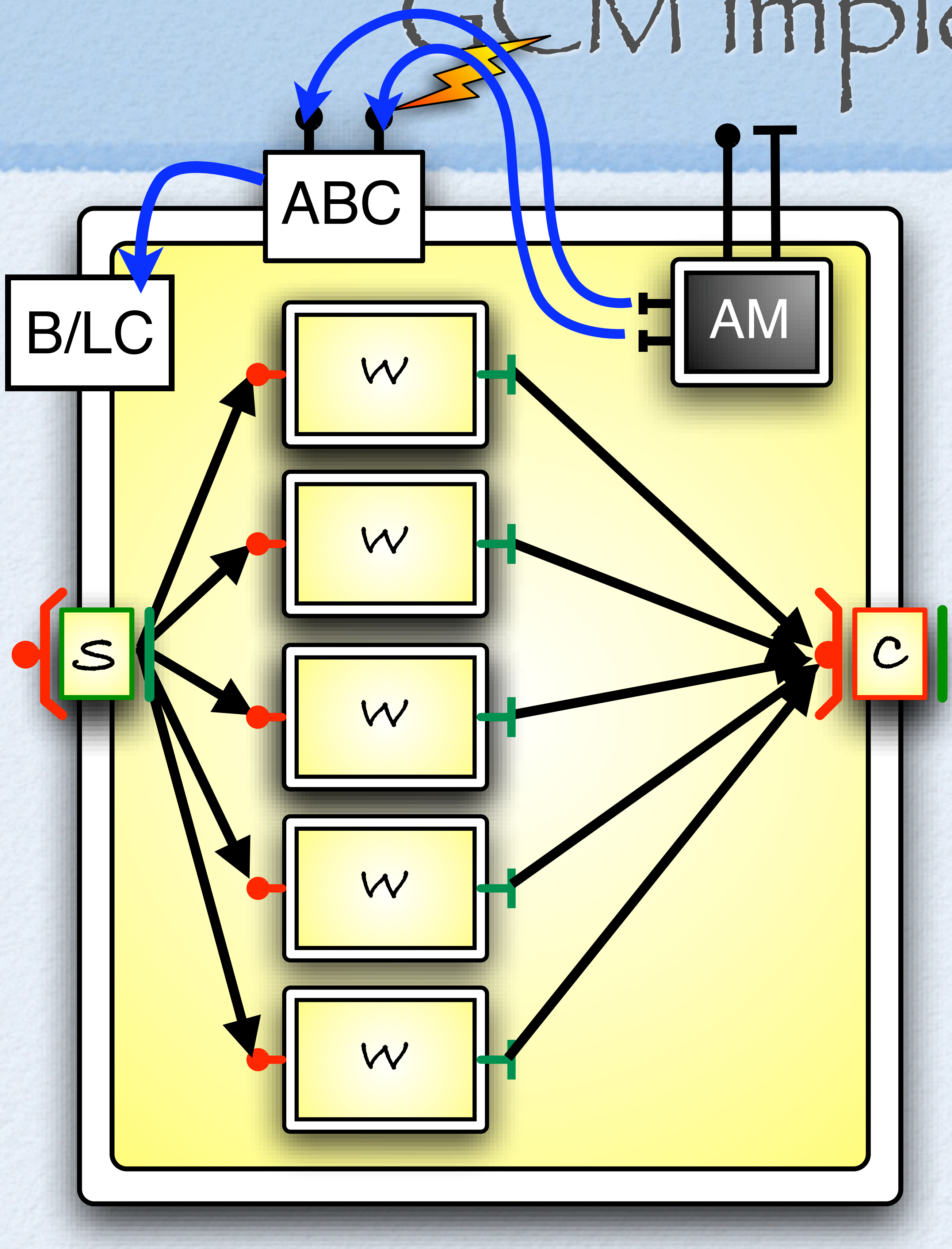


Functional behaviour description (orchestration)

$$system(data, S, G, W, in, out, N) \triangleq$$
$$S(data, in) \mid (\mid i : 1 \leq i \leq N : W_i(in_i, out_i)) \mid C(out)$$
$$W_i(in_i, out_i) \triangleq$$
$$in_i.get > tk > process(tk) > r > (out_i.put(r) \mid W_i(in_i, out_i))$$

- Meant to parametrically expose all allowed adaptation
  - Any AM policy that does not change this semantics is *correct*
  - As an example changing *i* in this schema is correct
  - Functional semantics is invariant from *i*, non-functional one is not (and changing *i* means changing the number of Ws for self-* purpose

GCM implementation



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

2. Choose an inner component (compliant to Be-Ske constraints)

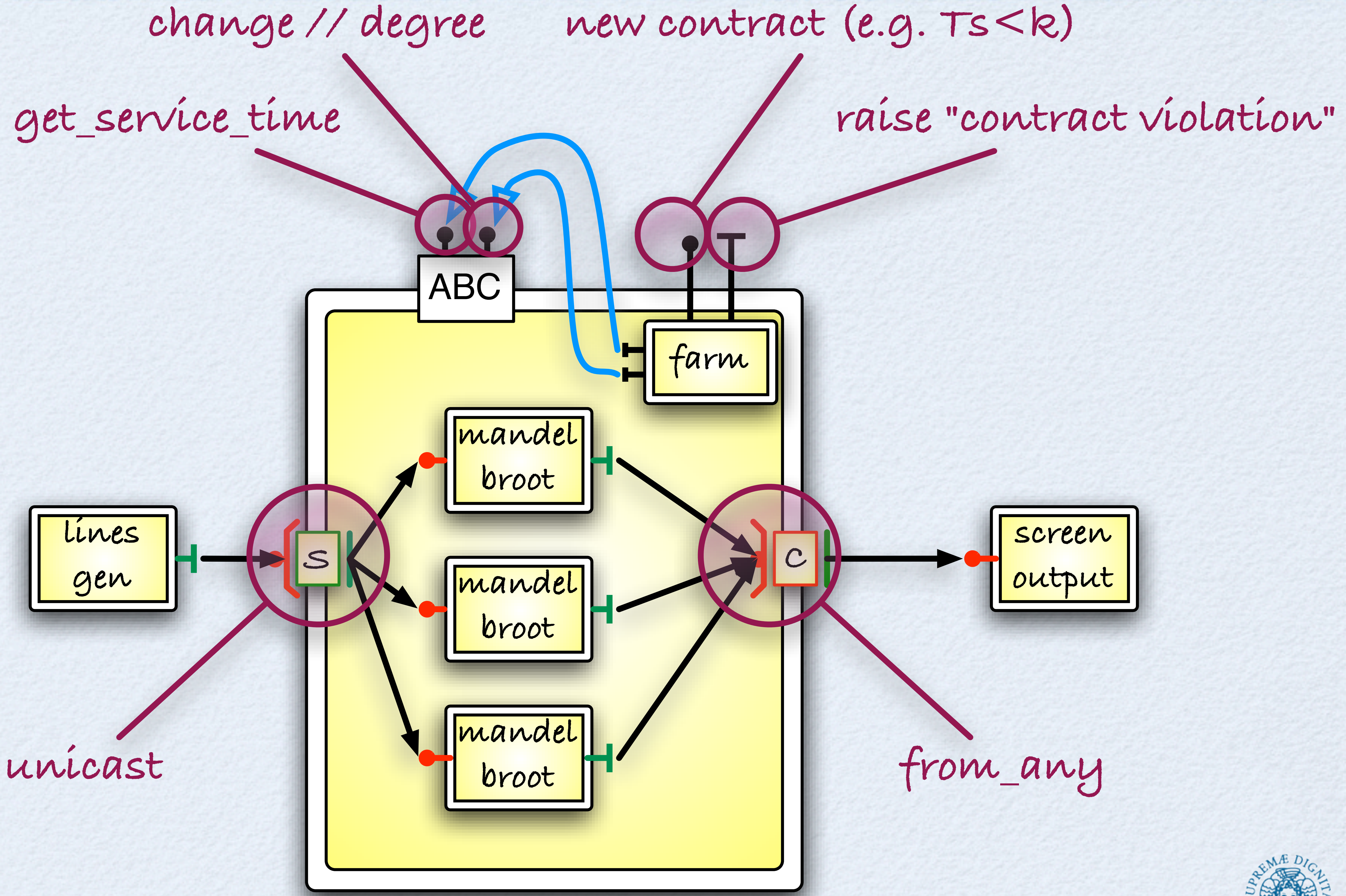
3. Choose behaviour of ports (e.g. unicast/from_any, scatter/gather)

4. Wire it in your application. Run it, then trigger adaptations

5. Possibly, automatise the process with a Manager

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

Farm example (Mandelbroot)



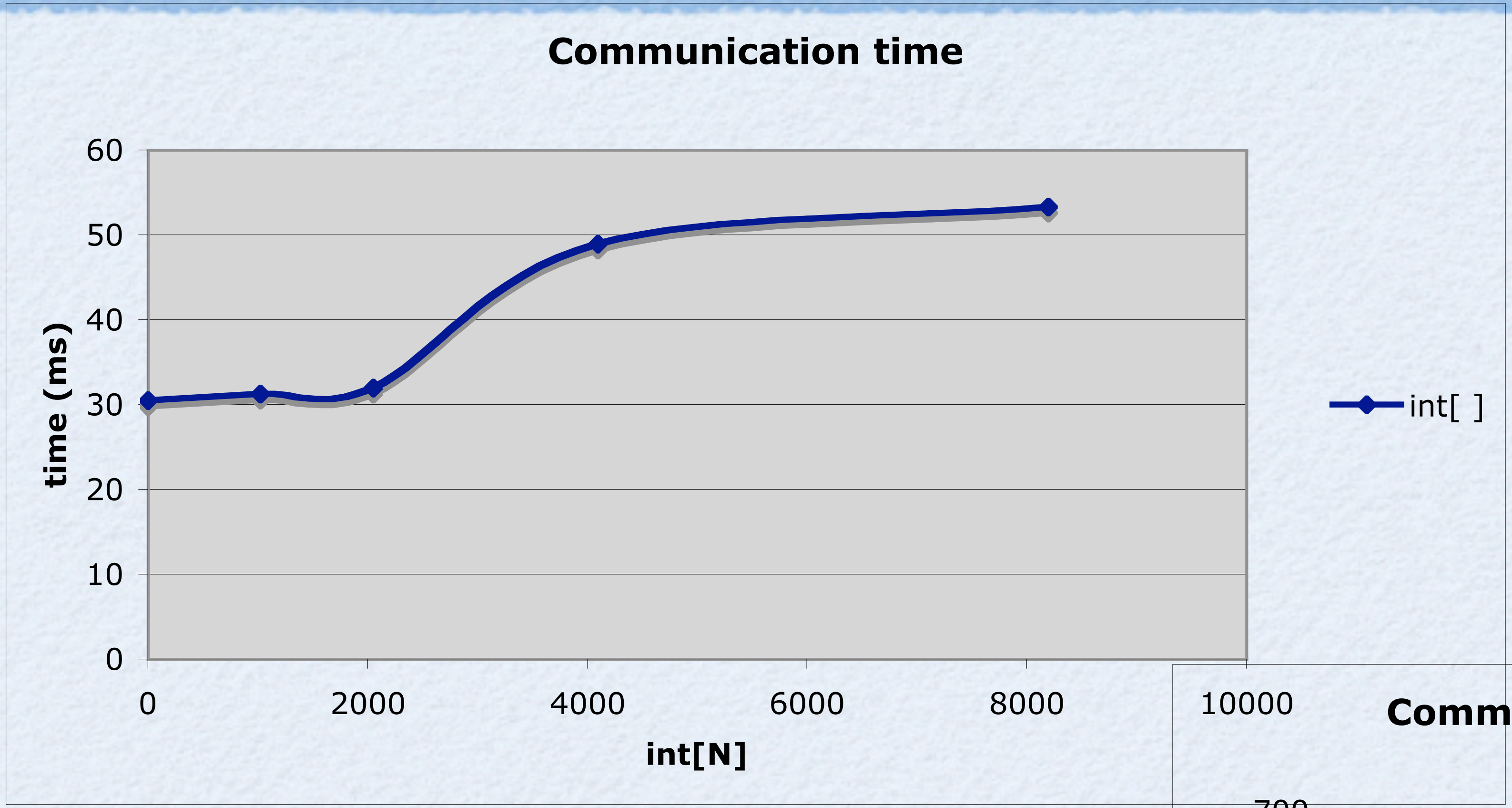
Not just farm (i.e. param sweep)

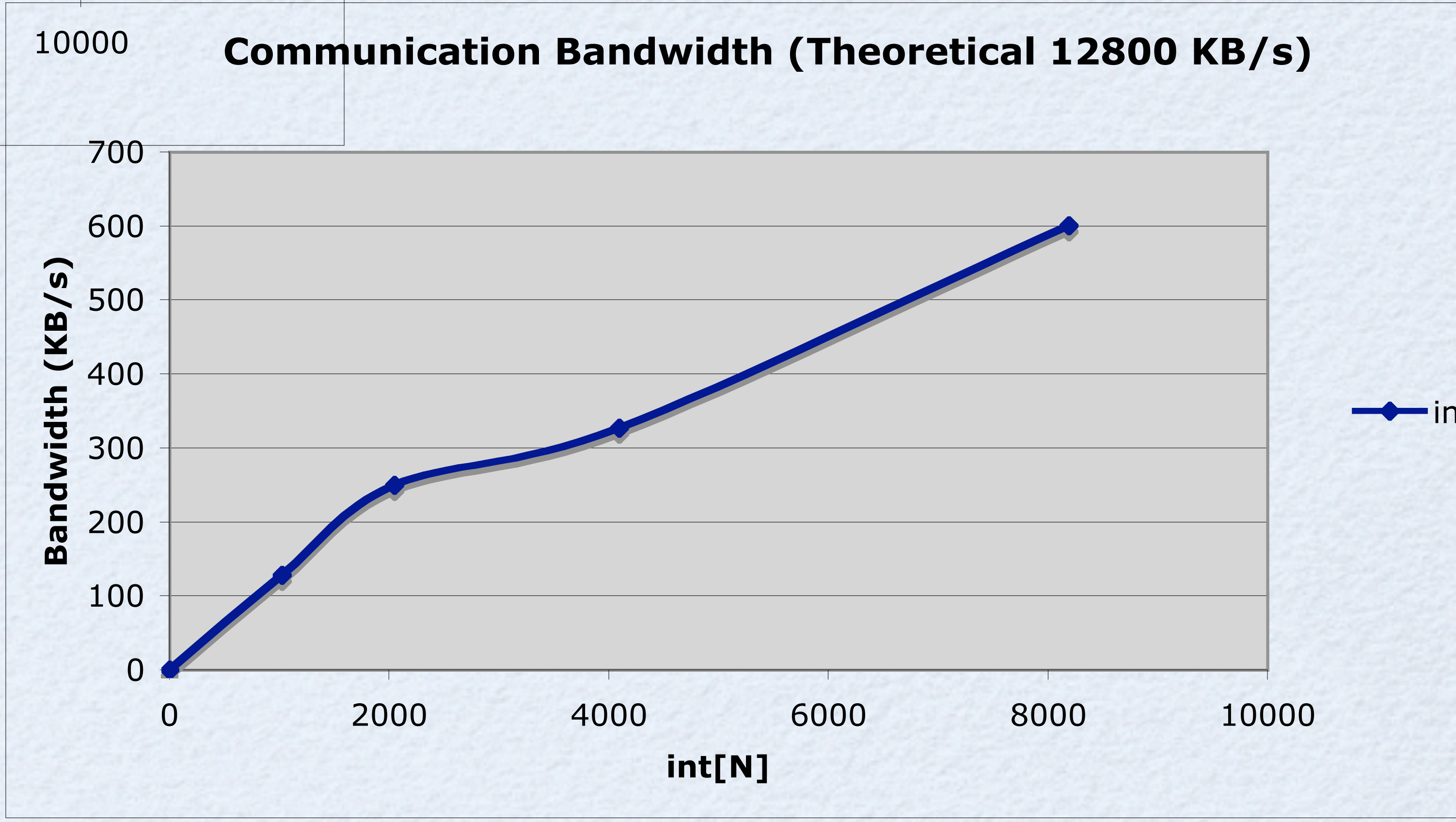
- Many other skeletons already developed for GCM
  - some mentioned before
- Easy extendible to stateful variants
  - imposing inner component expose NF ports for state access
- Policies not discussed here
  - expressed with a `when-event-if-cond-then-action` list of rules
  - some exist, work ongoing ...

Conclusions

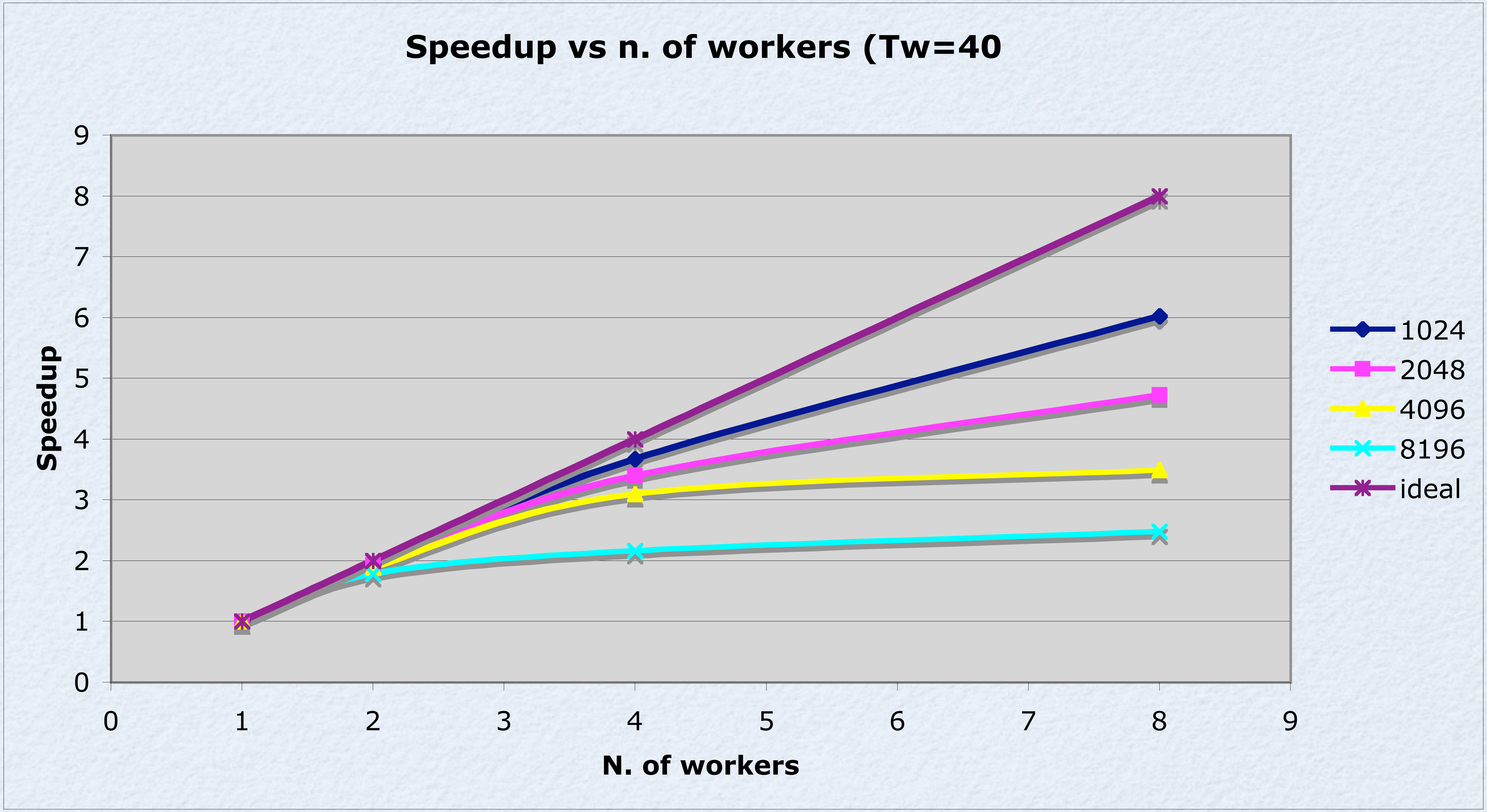
- Behavioural Skeletons
  - templates with built-in management for the App designer
  - methodology for the skeleton designer
    - management can be changed/refined
    - just prove your own management is correct against skeleton functional description
  - can be freely mixed with standard GCM components
    - because once instanced, they are standard
  - actually what Gannon called "application factories"
- Already implemented on GCM
  - not happy about GCM runtime perf. (can be improved)
    - We also implemented in ASSIST with different performances

Communication Time (Int)





Farm SpeedUp



Tw(jobsize=*)=40 ms Tc(jobsize=1)=30 ms

Thank you

P.S. the
COREGRILLEDFISH
effect
is written in Quartz,
which is a hierarchical
component model based on
streams

Less than one hour of
development time, because
of code reuse.
ALL the PARAMETRIC
components was already
available (sprite, oscillators,
streams mixer, ...)