Reflective Middleware: The Open ORB approach and some future directions

Fábio M. Costa

Instituto de Informática

Universidade Federal de Goiás

UFG

Roadmap

- Reflective systems
- Reflection in middleware
- The Open ORB architecture
- Prototypes
  – Meta-ORB
  – OpenORB v2 / OpenCOM
- A “brief” look into future trends
- Concluding remarks

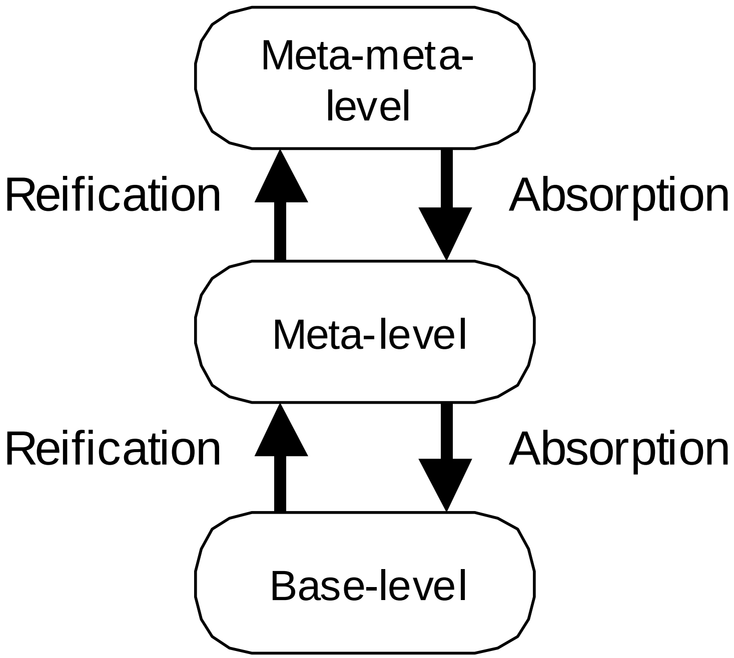
Reflective systems

"A system that is capable of manipulating representations of itself in the same way as it manipulates representations of its domain of application" *(adapted from B.C. Smith, 82)*

- A reflective system maintains a self-representaion
  - causally connected with the system´s own implementation
  - inspection and adaptation at runtime

Meta-level architectures

- Base-level
  - usual functionality of the system
- Meta-level
  - reflective functionality
  - self-representation
- Object-oriented concepts:
  - base-objects
  - meta-objects
  - Meta-object protocol (MOP)



Reflective middleware

Motivation

- A standard meta-object protocol for accessing reflective functionality (overcoming heterogeneity, etc.)
- A consistent and comprehensive approach to *open up* the platform implementation
- Greater flexibility

- Base-level: usual middleware services
  - as found, e.g., in CORBA
  - accessed through the platform APIs
- Meta-level:
  - Meta-objects that reify the platform implementation
  - accessed through a MOP (meta-interface)

Principles of reflective middleware

- Modular platform infrastructure
  - based on component models
- OS and language independence
- Pervasiveness of the reflective mechanisms
- A unified approach for (static) configuration and (dynamic) re-configuration of the platform
- Managing the complexity of the meta-level

The Open ORB approach

Lancaster University: Blair et al

- The platform is built in terms of a component model (modularity)
  - all middleware funcionality is realised in terms of components
  - same component model as used for applications
- Components exist at runtime
- Explicit binding to connect the interfaces of remote components
- Runtime adaptation through comprehensive reflective meta-interfaces

Open ORB: meta-level

- Split into multiple meta-space models
  - Each one dealing with the reification of a different aspect of the platform implementation

- Interfaces
- Architecture

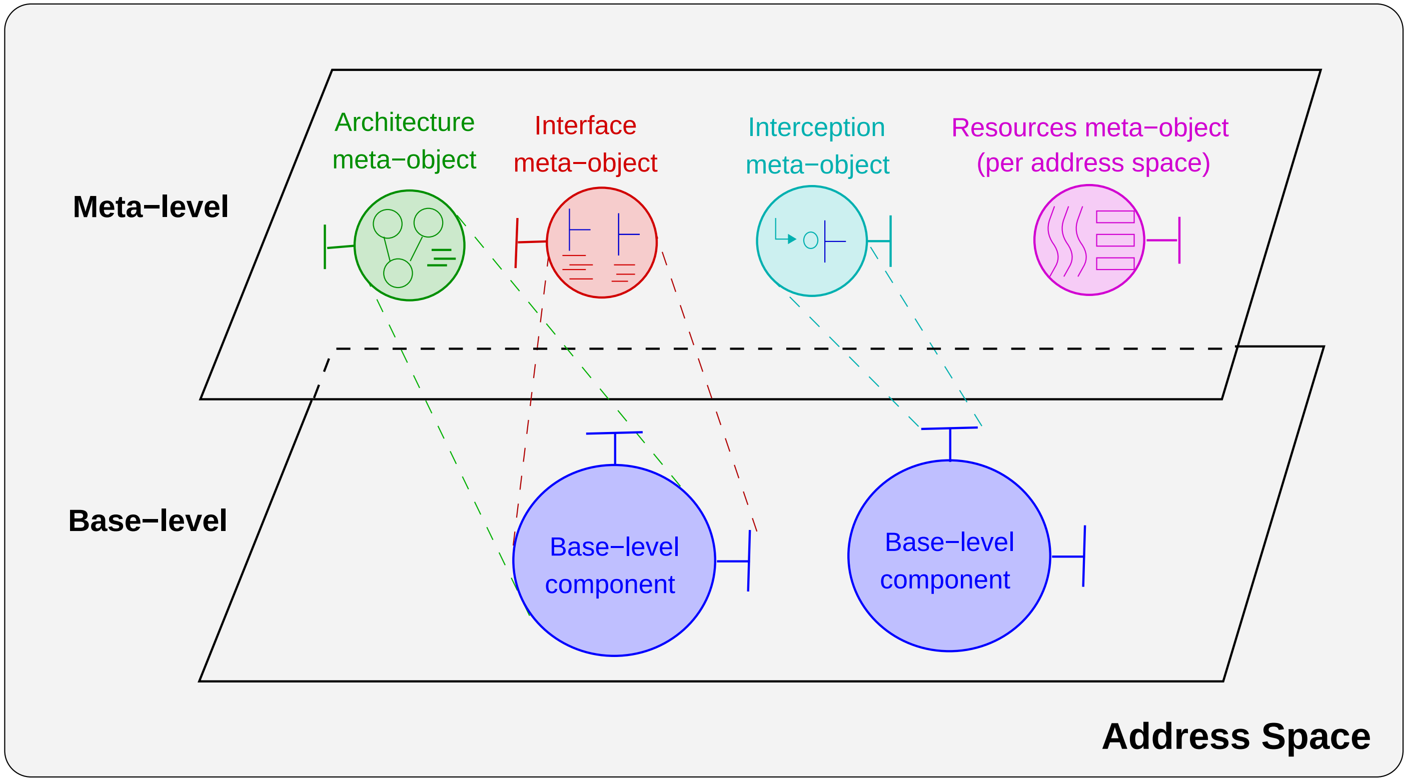
Structural meta-space models

- Interception
- Resources

Behavioural meta-space models

Fábio M. Costa

Open ORB: meta-level



Fábio M. Costa

Open ORB prototypes

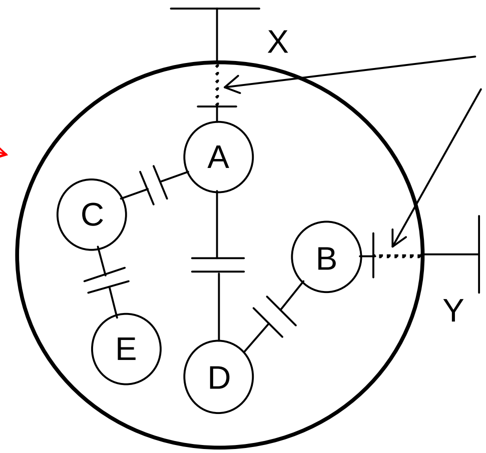
- OOPP: Open ORB Python Prototype
  - proof-of-concept implementation
- GOORB – Group Support for Open ORB
  - flexible object group service
- Xelha
  - reifies resource management in the platform
- **Meta-ORB**
  - integration with meta-information management
- **OpenORB v2**
  - underlying component model (OpenCOM) + component frameworks to build concrete configurations of middleware

Meta-ORB

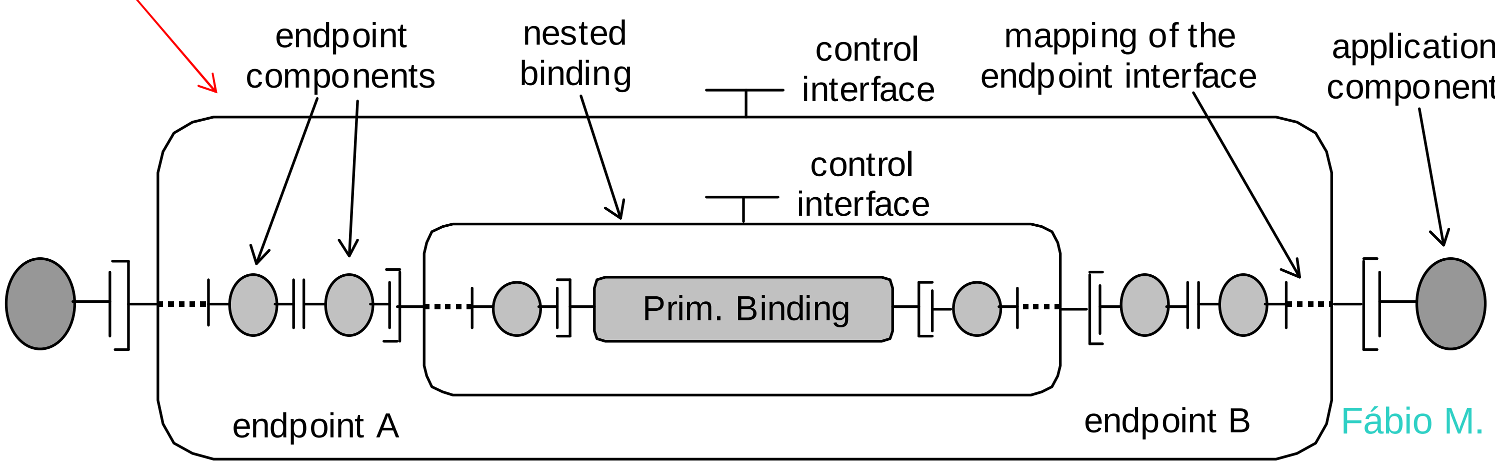
- Implemented in Python, for rapid prototyping
- Main constructs of the programming model:
  - interfaces, components, and explicit bindings
- Configuration based on type and template definitions
  - definition: interactive GUI or definition language
  - stored and managed in a repository
- Re-configuration through reflective meta-interfaces
- Key points:
  - reflection based on runtime available meta-information
  - reflective adaptation causes type evolution
  - constrained by type evolution rules
  - focus on structural reflection

Meta-ORB: configuration examples

Component (composite)



Binding



Fábio M. Costa

Meta-ORB: component definition example

```
module Example {
   primitive component AudioDeviceComp {
      implementation: AudioDeviceImpl;
      interfaces: AudioDevice audio_interf;
   };
   primitive component VideoDeviceComp {
      implementation: VideoDeviceImpl;
      interfaces: VideoDevice video_interf;
   };
   interface <stream> AVDevice : AudioDevice, VideoDevice {};
   primitive component MixerComp {
      implementation: MixerCompImpl;
      interfaces: AudioDevice audio_interf;
                  VideoDevice video_interf;
                  AVDevice av_interf;
   };
   component AVDeviceComp {
      internal components: AudioDeviceComp audio_comp;
                           VideoDeviceComp video_comp;
                           MixerComp mixer_comp;

      object graph: (audio_comp, audio_interf):(mixer_comp, audio_interf);
                    (video_comp, video_interf):(mixer_comp, video_interf);

      interfaces: AVDevice av is (mixer_comp, av_interf);
   };
};
```

Meta-ORB: binding definition example

```
module Example {
   binding AVBinding {
      control interfaces: CtrlInterf ctrl is (CtrlComp, ctrl_interf);
      internal bindings: AudioBinding audio_binding;
                         VideoBinding video_binding;

      role AVBindingPartic {
         components: AVStubComp stub;
                     AudioFilterComp audio_filter;
                     VideoFilterComp video_filter;
         target interface: AVDevice is (stub, av_interf);
         cardinality: 2;
         configuration:
            (stub, audio_interf):(audio_filter, audio_interf);
            (stub, video_interf):(audio_filter, video_interf);
            (audio_filter, forward_interf):(audio_binding, audio_role);
            (video_filter, forward_interf):(video_binding, video_role);
      };
   };
};
```

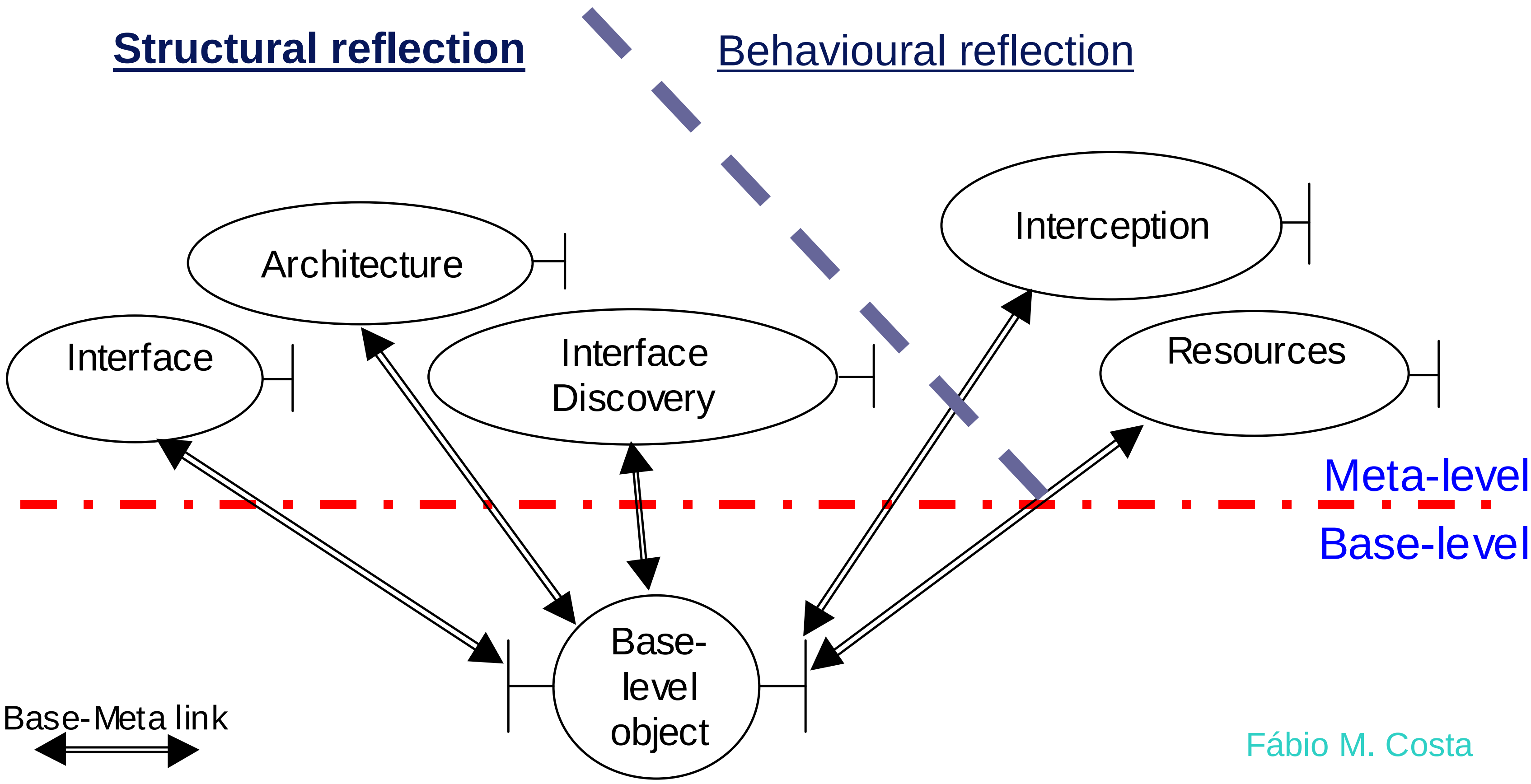
Meta-information management

Type and configuration repository

- Manages type and template definitions
- Provides an interface for accessing such meta-information at runtime
  - inspection of
    - interface types
    - component definitions and compositions
    - binding configurations
  - type and template evolution: dynamic definition
- Structure derived from the CORBA IR
- Implementation using the Meta-Object Facility (MOF)

Fábio M. Costa

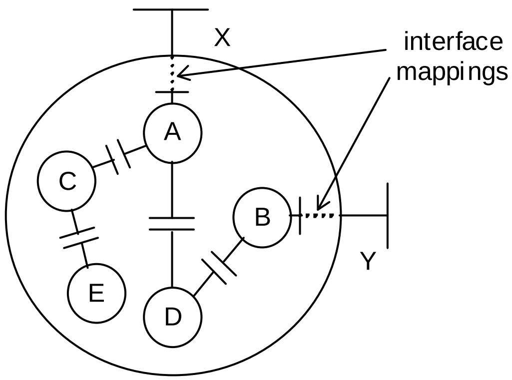
Multiple meta-space models



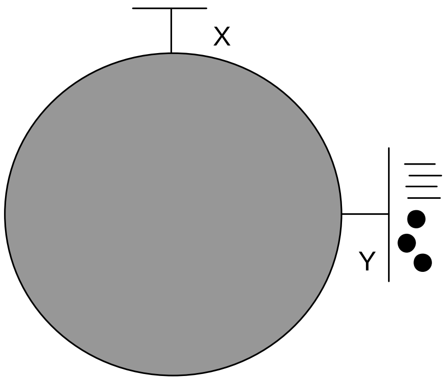
Fábio M. Costa

Structural reflection

Architecture: configuration of internal components plus composition rules
→ inspection and adaptation



Interface Discovery: set of interfaces of a component
→ inspection only



Interface: operations and attributes of an interface
→ inspection

Example of architectural adaptation

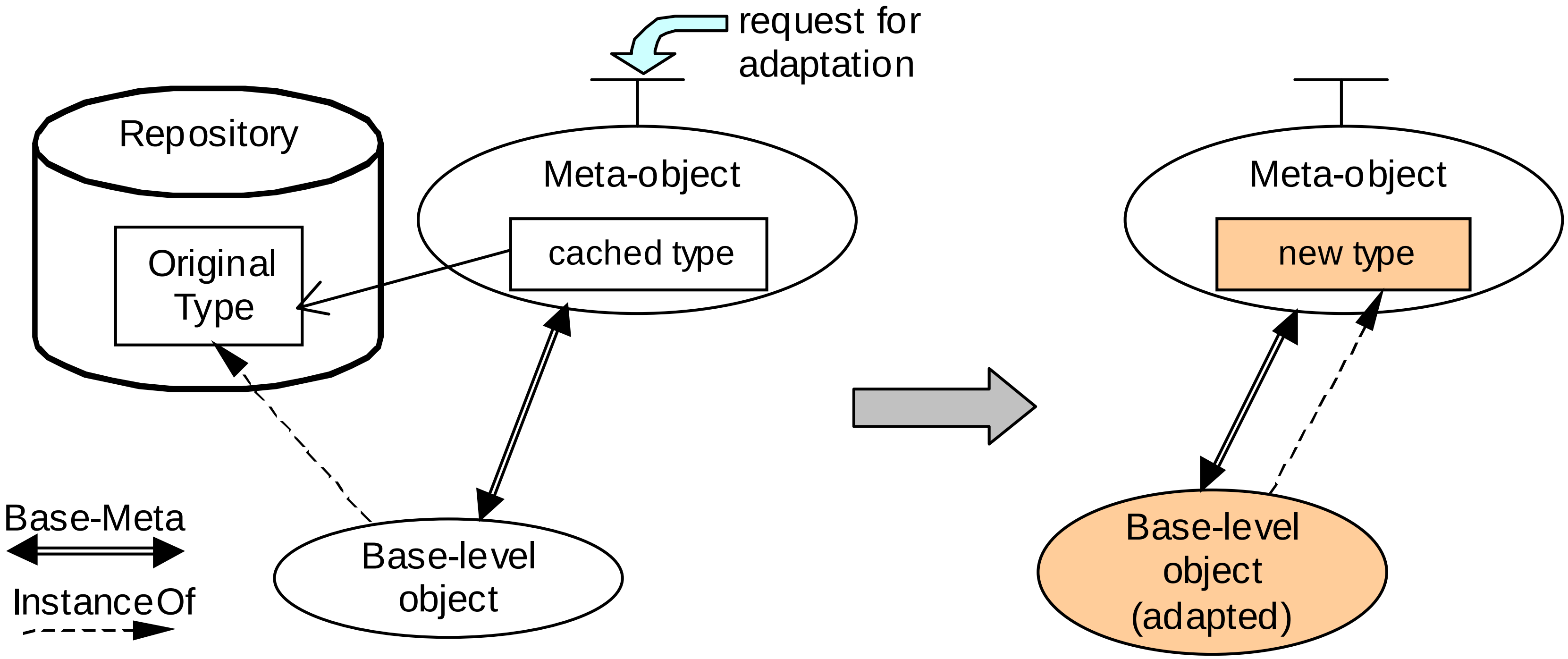
```
import MetaORB
# Obtain a reference to the Architecture meta-object
arch_mobj = MetaORB.get_arch_mobj(bind_ctrl.get_binding_name())

# Obtain the type of the new component from the Type Repository
new_video_fiter_type = MetaORB.TypeRep.lookup_name('LowBandwidthVFilter',
                                                    dk_Binding)

# Pause the binding, so that reconfiguration can be performed without
# breaking its consistence
bind_ctrl.pause()

# Invoke the appropriate operation of the Architecture MOP to replace all
# occurrences of the video filter component (in all endpoints conforming
# to the AVBindingPartic role) with components instantiated from the new
# component type
arch_mobj.role_replace_component(AVBindingPartic, video_filter,
                                 new_video_filter_type)
# Resume normal operation of the binding
bind_ctrl.resume()
```

Effect of reflection on type meta-information: Type evolution



Fábio M. Costa

Current work

- Implementation of Meta-ORB for handheld devices
  - PalmOS implementation
  - Written in Java (J2ME / MIDP 1.0 and J2SE)
  - Preserving the high-level programming model
  - Minimal core mechanisms
  - Configuration facilities allow for the definition of minimal versions of the platform
- Objective: verify the effects of limited resource environments and mobility

Fábio M. Costa

Meta-ORB: Java version

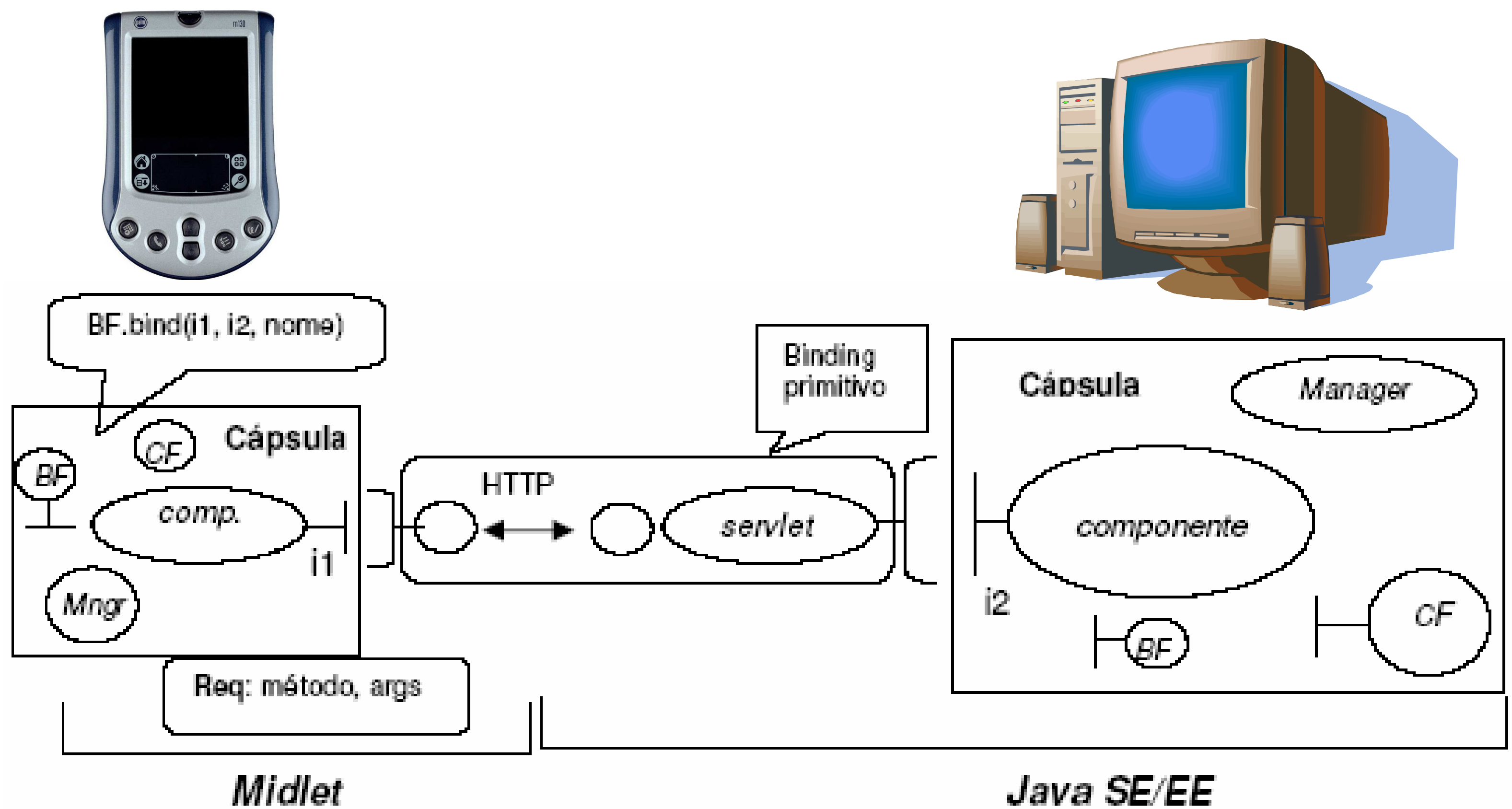


Fig. 4 - Comunicação entre as partes da plataforma, geradas dinamicamente via *servlet*

OpenORB v2

- A lightweight component model, based on Microsoft's COM: OpenCOM

+

- Component Frameworks
  - for each major aspect of the platform implementation (e.g., binding, resource management)
  - guide the (static) configuration of middleware
  - constrain runtime re-configuration
  - → a focus on ensuring the integrity of the platform

→ Emphasis on eficiency and adherence to standards
  - CORBA, COM

OpenCOM component model

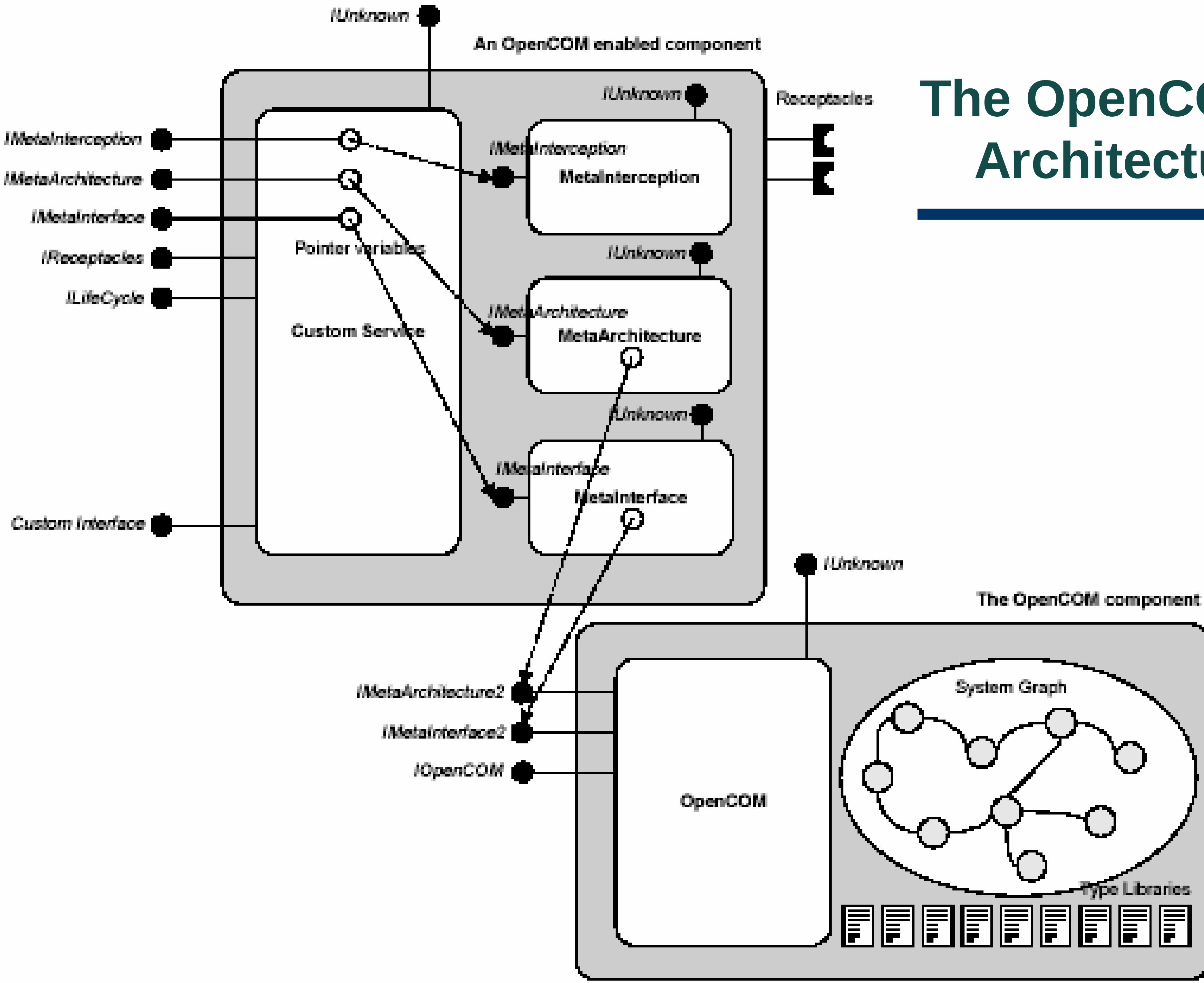
- Based on a subset of COM
  - without distribution, persistence, security and transactions – such aspects are built atop the component model
- Core features
  - binary-level interoperability standard (*vtable*)
  - Microsoft's IDL
  - COM's Globally Unique Identifiers (GUIDs)
  - IUnknown interface (for interface discovery)

OpenCOM

- Makes explicit the dependencies among components
- Basic support for reconfiguration
  - mechanism for connecting components
    - interfaces, receptacles and connections
  - mutex locks to serialise concurrent adaptations
- Pre- and post-methods (interception)
  - lightweight means of adding new behaviour
  - does not require reconfiguration of the existing component architecture

Fábio M. Costa

The OpenCOM Architecture

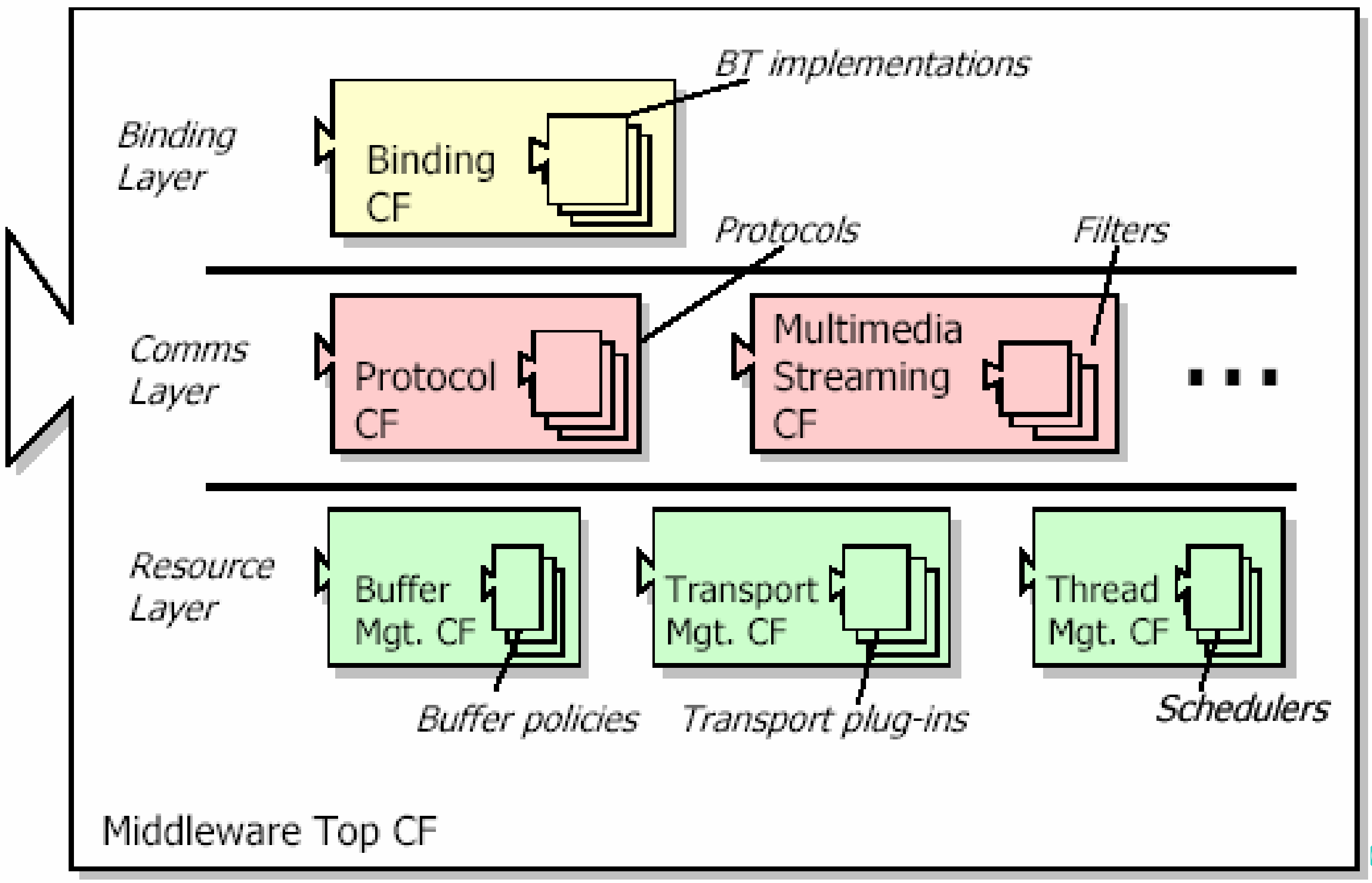


Component frameworks

"Collection of rules and interfaces that govern the interaction of a set of components plugged into them."
[Szyperski,98]

- CFs reified at runtime
  - Meta-information to represent the configuration of components
  - Meta-interfaces for manipulating
  - Rules and policies that constrain adaptation
- Hierachically structured, e.g.:
  - root CF: the ORB itself
  - lower level CFs realise internal aspects of the ORB
  - manager / managed pattern

An example middleware component framework for OpenORB v2



Granularity of adaptation

- Fine-grained
  - component adaptation through low-level OpenCOM API
- Coarser-grained
  - replacement of CF implementations
    - e.g., replace a standard RMI binding type with one that includes security
  - change the component framework
  - using the top level CF's meta-interface to introduce new low-level CFs or change existing ones
  - allow for the definition of different middleware personalities
  - e.g., the ReMMoC approach for adaptation to different service environments

Fábio M. Costa

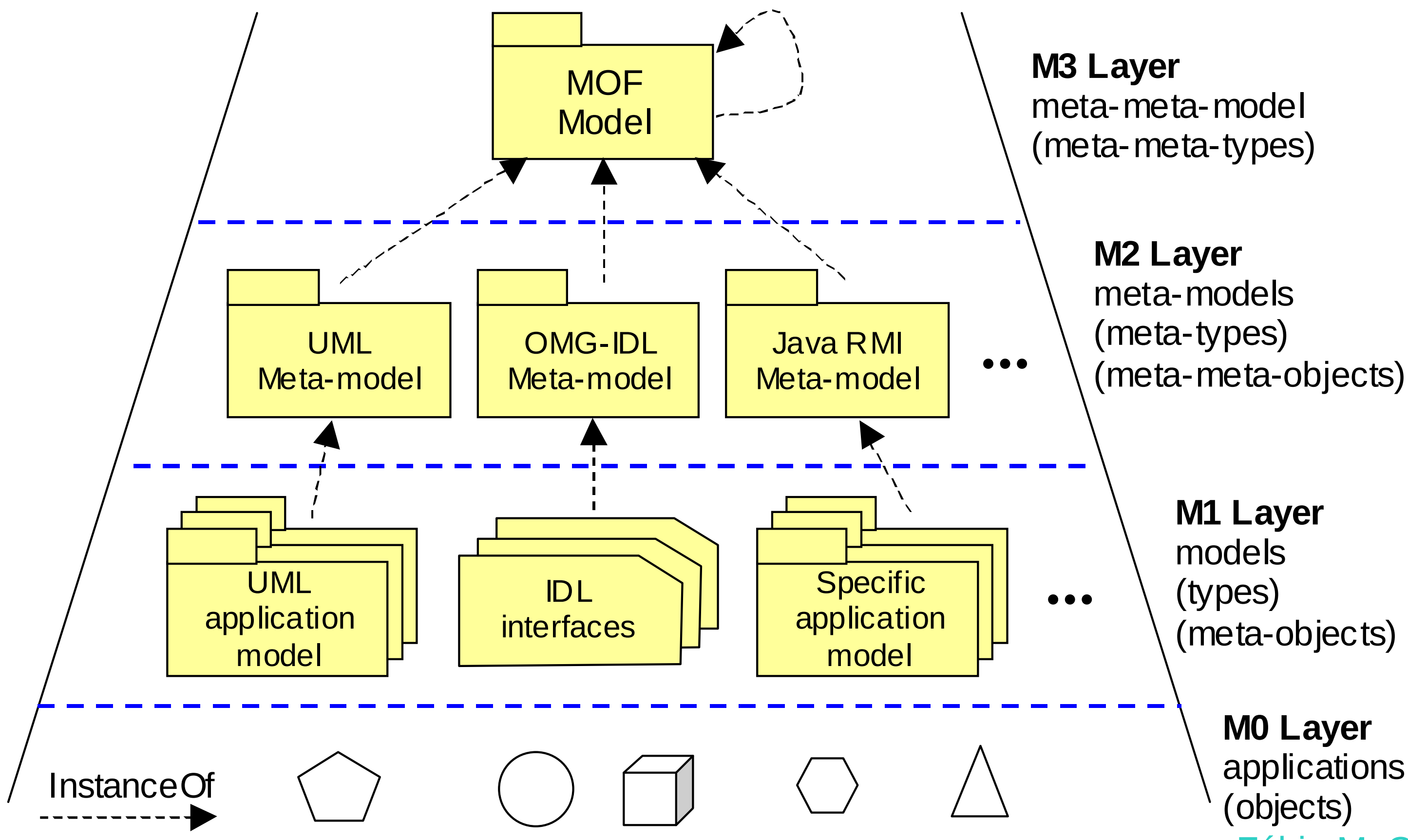
Some future trends

- Current approach to interoperability
  - middleware mandates a common programming model
  - Problem: there are multiple such “common” programming models
- 1st generation solution: *ad hoc* bridges
- 2nd generation solution: adaptation of the whole programming model
  - make the platform adopt different personalities in each context (e.g., as in the ReMMoC and UIC approaches)
  - limitation: one personality at a time
- A 3rd generation solution? One that is more flexible?

A more flexible solution to interoperability

- Deal with the problem at a higher level
  - Programming model = meta-model
- Handle programming model constructs as first-class entities
  - Through a meta-modelling architecture
  - The component model can be interpreted at runtime, if need be
    - e.g., when interacting with a different services environment
    - in order to “learn” how to interpret another platform’s constructs

A meta-modelling architecture



Fábio M. Costa

Some interesting consequences

- Makes the platform truly independent of any particular component model
  - Choose your favourite component model
    - native component model: optimise for it
  - Other component models can be seamlessly accommodated
- Issues
  - performance?
  - how to express the semantics of constructs?
  - complete mapping between component models?

Overall Remarks

- We have several architectures for reflective middleware
- Reflective facilities in current off-the-shelf middleware
- Standardise on meta-interfaces
  - in the same way as for the usual middleware service APIs
- Derive common patterns for fully reflective middleware
- Recognise the value of structured meta-information
- A roadmap for the (far) future

fmc@inf.ufg.br