Reflective middleware
Reflektiiviset väliohjelmistot

Expectations

- Middleware able to cope with change
  - Applications are deployed in hostile and dynamic environments
    - Multimedia
    - Group communication
    - Realtime and embedded environments
    - Mobile and handheld computing
  - But there is a problem as conventional ORBs are static:
    - Fixed threading model
    - Fixed transport protocol: IIOP (over TCP/IP)
    - Fixed security strategy (typically no security)
    - Fixed scheduling

Expectations

- Applications need support for
  - Monitoring the system state = application environment
  - Making decisions on the preferred system state
  - Changing the system state
- No changes to the application
- As little "intelligent code" in the application as possible

Definition

- Reflective middleware is a middleware system that provides inspection and adaptation of its behaviour through an appropriate causally connected self-representation.

Examples

- Inspection
  - Reflection in Java
  - Finding out the current garbage collection policy
  - Finding and using new methods on an existing class
  - Monitoring throughput
- Adaptation
  - Changing grabage collection policy
  - Reallocating resources for a stream

Is CORBA reflective?

- CORBA ORB provides a meta object framework to connect clients with servers
- Metaprogramming mechanisms
  - DII (dynamic invocation interface), DSI (dynamic skeleton interface), and interface repositories provide an application with direct access to meta object system
  - MOF (meta object facility) provides a standard way to model the meta object in IDL definitions and standardizes the MOPs for accessing this metainformation
  - Smart proxies and CORBA portable interceptors can selectively alter object behaviour
  - Pluggable transport protocols control how ORBs exhange messages
  - Servant locators and containers direct a CORBA server's resource management

Terminology

- Reify/kuvata – to explicitly describe the system behaviour and rules
- Inspect/tutkia – to retrieve reified information
- Reflect/päättää – to make decisions as a result of inspection
- Adaptation/sopeutua – a change in system behaviour as a result of reflection
- Causally connected self-representation – a change in the self-representation causes immediate change in behaviour

dynamicTAO

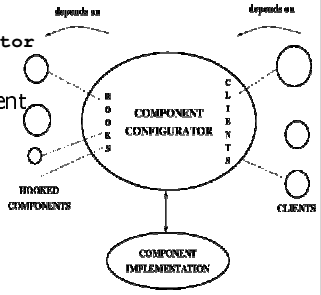
- reflective CORBA ORB with component support, built in C++
- Requirements
  - Reification of ORB internal components and interaction btw them
  - Change of strategies and policies
- extends TAO to support
  - on-the-fly reconfiguration
  - consistency of ORB engine
- TAO already supported startup configuration, configuration file specifies strategies for
  - concurrency (threading model)
  - request demultiplexing
  - scheduling
  - connection management

Adding Support for Dynamic Configuration

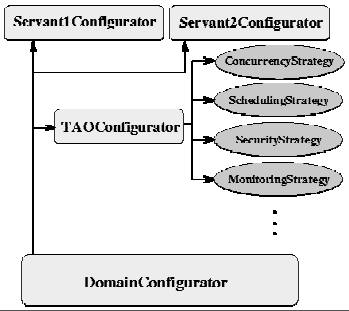
- *dynamicTAO* exports an interface called **`DynamicConfigurator`**, supporting
  - 1. Transfer of components across the distributed system
  - 2. Loading and unloading components
  - 3. Inspecting and modifying the configuration of the ORB (and of applications running on top it)

Reifying the ORB Structure

- **`ComponentConfigurator`** framework
- Stores inter-component dependencies
- Allows browsing, inspection, and reconfiguration
- Can be customized through inheritance



dynamicTAO Structure



`DynamicConfigurator` IDL Interface

```
interface DynamicConfigurator
{
 stringList list_categories ();
 stringList list_implementations  (in string categoryName);
 stringList list_loaded_implementations ()

 stringList list_hooks            (in string componentName);
 string     get_hooked_comp       (in string componentName,
                                    in string hookName);
 string     get_comp_info         (in string componentName);
 .
 .
 .
```

Manage Component Implementations loaded in memory

```
long load_implementation (in  string categoryName,
                          in  string impName,
                          in  string params, ...);

void hook_implementation (in string loadedImpName,
                          in string componentName,
                          in string hookName);

void suspend_implementation   (in string loadedImpName);
void resume_implementation    (in string loadedImpName);
void remove_implementation    (in string loadedImpName);
void configure_implementation (in string loadedImpName,
                               in string message);
```

Manage the ORB Persistent Component Repository

```
void upload_implementation   (in string categoryName,
                              in string impName,
                              in implCode binCode);

void download_implementation (in string categoryName,
                              inout string impName,
                              out implCode binCode);

void delete_implementation   (in string categoryName,
                              in string impName);
};
```

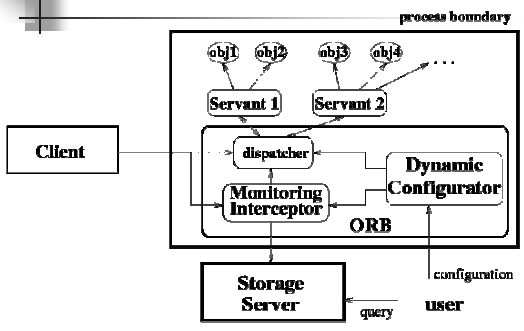
Example of Dynamic Configuration

```
1. myRemoteOrb->upload_implementation ("Security", "superSAFE",
      superSAFE_impl);
2. newSecurityStrategy =
      myRemoteOrb->load_implementation ("Security",
      "superSAFE");
3. oldSecurityStrategy =
      myRemoteOrb->get_hooked_comp ("dynamicTAO",
      "Security_Strategy");
4. myRemoteOrb->hook_implementation (newSecurityStrategy,
      "dynamicTAO", "Security_Strategy");
5. myRemoteOrb->remove_implementation (oldSecurityStrategy);
```

Consistency

- Dynamic reconfiguration may break the consistency of the internal ORB engine.
- Consistency must be ensured by the ORB developer and by the component developer.

Monitoring Service in dynamicTAO



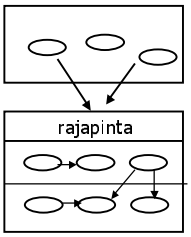
Selecting implementations and adapting strategies

- designing a reusable software module difficult
  - General-purpose implementation strategies never optimal for any specific client
  - Module interfaces should allow the client to assist or participate in the selection of the module's implementation strategy
  - "open implementation"

Kerrosarkkitehtuurien käytöstä

open box (vs. black/white box)
- rajapintaan liitetään ominaisuuksia
- järjestelmä tulkitsee ominaisuudet konfiguraationa ja toimintoina

Sopimus
- Ominaisuusarvot määrätään suoritusaikana
- sopimuksen mukaiset arvot uudelleenkonfiguroinnilla käyttöön



Open implementation interface styles

- Client provides the implementation strategy to use – e.g. an object that implements a custom strategy on top of e.g. cache management protocol
  - Module adopts the strategy provided
  - Easy to specify exact stragegy
  - Designing module to support traceable strategies might be difficult. For client, building new stategy might be expensive.
  - It is not feasible for a module to implement all implementation strategies that clients might need.

Open implementation interface styles

- client specifies the implementation strategy the module should use – e.g. LRU cache management
  - Module adopts the strategy specified by client
  - Easy to specify exact strategy, even if wrong
  - Difficult to choose automatically between a few candidate implementation strategies

Open implementation interface styles

- Client provides declarative information about its usage pattern – e.g. "sequential file scan"
  - Module selects strategy by matching usage pattern information from client to the best available strategy
  - Client provided information about its usage pattern doesn't constraint the implementation
  - Difficult for the client to know how it is influencing module strategy
  - Easy to choose an effetive implementation strategy when the client behaviour is known

Open implementation interface styles

- No implementation strategy control interface
  - Module selects implementation strategies by observing client's use of the black-box interface
  - One implementation strategy will satisfy all clients or the module determines a good strategy by itself

OpenORB

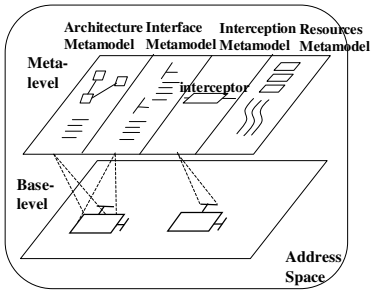
- designed to support multimedia programming
- combines components and reflection
- components are described in terms of sets of required and provided interfaces (streams)
- enables explicit bindings between compatible interfaces
- Offers a built-in event notification facility
- Every application component offers a meta-interface that provides access to metaspace
  - metaspace is support environment of the component
  - infinite towers of metaspaces for recursive use

OpenORB themes

- explicit bindings for multimedia transfer
  - End-to-end QoS contracts
  - Change requests from users
  - Changes in availability of resources
- Evolution vs environmental changes
  - Middleware heterogeneity
- restriction of changes to preserve consistency
  - Safe-states, architectural restrictions
- metainformation management

OpenORB Architecture

- RM-ODP computational model
- per-object (or interface) meta-space support
- meta-space structured as four orthogonal meta-models:



OpenORB architecture Structural reflection

- Interface metamodel for external view
  - Provided and required interfaces
  - Introspection facilities
- Architecture metamodel for internal construction (software architecture)
  - Component graph + architectural constraints
  - Local binding = mapping between required and provided interface in a single address space
  - Distributed binding components for distribution
  - Extensible set of binding types (architectures) for interaction models such as remote invocation, publish-subscribe, continuous media flows, group communication
  - Architectural constraints keep system consistent (type management system worries about substitutability)

OpenORB architecture Behavioural reflection

- deals with activity in the underlying system – actions and required resources
- Enables dynamic insertion of interceptors
  - Interceptors associated with interfaces
  - Enable insertion of pre- and post-behaviour (e.g. dynamically introducing monitoring or accounting into a running system; nonfunctional behaviour such as concurrency control or security checks)
- Resources metamodel per address space
  - Access to a set of components that represent resource management for inspection and adaptation
  - E.g. statistics for the effectiviness of thread scheduling

Design of the four metamodels

- architectural meta-object protocol
  - Retrieve base-level configuration
    - a component graph
    - List of component ids
    - List of binding object idsM
  - Find neighbours of an identified component
  - Retrieve/set architecture style or rules within it
  - Retrieve/set dependency constraints for adaptation tasks
  - Establish/terminate local binding
  - Create component and insert into a specified location in configuration
  - Remove and omit a component
  - Replace a component
  - Map interfaces onto composite component interfaces
  - Start/complete/rollback transaction

Design of the four metamodels

- interface meta-object protocol
  - Describe operation signature
  - Retrieve ids to supported interfaces
  - Interface style (stream, operational, signal)
  - Retrieve list of interaction details
  - get/set service offer values
  - Generic call_operation

Design of the four metamodels

- Interception meta-object protocol
  - Add interceptor to trap incoming messages
  - Remove interception
- Resources meta-object protocol
  - Get/set the lower/higher level abstraction
  - Create an abstract resource
  - Get resources created by this factory
  - Set manager's management policy
  - Associate a resource to a manager
  - Suspend/resume an abstract processing resource
  - Get/set manager/factory of a resource
  - Get/set scheduling parameters
  - Execute an operation

OpenORB and Xelha

- Xelha is an architecture description language
- descriptions in terms of componets, interfaces, connectors
- supports overlaying of task structure with associated QoS requirements
  - QoS requirements used to derive resource allocation policies for tasks
- Supports introduction of dynamic QoS management structures (monitoring and controlling)

Xelha example

```
Def connector<stream> AudioConnector_V1 (string
  srcCapsule, string sinkCapsule):
  components:
     srcStub: SrcStub, srcCapsule
     sinkStub: SinkStub, sinkCapsule
  connectors:
     streamConn:
     StreamConnector(srcCapsule,sinkCapsule)
  interfaces:
```

```
interfaces:
   interaction:
      IN: SrcStubIN, (srcStub, IN)
      OUT: SinkStubOUT, (sinkStub, OUT)
   control:
      CTRL:StreamConnCTRL,(streamConn, CTRL)
composition graph:
   interfaces:
       OUT: (srcStub, OUT)
       streamConnIN: (streamConn, IN)
       streamConnOUT: (streamConn, OUT)
       IN: (sinkStub, IN)
```

```
     edges:
          (OUT, streamConnIN)
               (streamConnOUT, IN)
     tasks:
          Def task transmitAu.marshall:
               switching points:
                srcStub:CTRL:start [if taskx]
     qos specifications:
          delay (srcStub:IN:read,
                 streamConn:IN:put) = 5
          troughput (srcStub:OUT:put) = 64
```

```
Def task transmitAu includes
transmit.Au.marshall, transmitAu.unmarshall:
   importance: 5
   qos specifications:
        delay(streamConn:IN:put,
             streamConn:OUT:put) =10
        packet_loss(streamConn:IN:put, ..
        jitter ...
        qos management structure ...
```

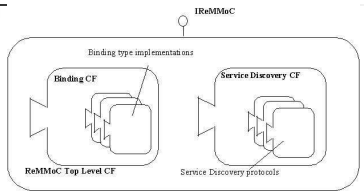
Metainformation management

- binding types
  - Architecture descriptions
- Interface types
  - Stream interfaces with QoS annotations
- Component types
  - Primitive and composite component types
- Each component has an interface for accessing metaspace
  - Structural meta-information provided by type repository
  - Type repository is an extended interface repository

The ReMMoC Project: Overcoming Heterogeneity

- The characteristics of the mobile environment present a number of challenges for middleware
  - Weak connection, poor network QoS, frequently changing device & environmental context, discovering services …
- Emerging middleware technologies that solve these issues
  - Asynchronous middleware paradigms (publish-subscribe, tuple spaces)
  - Existing platform (e.g. CORBA) fixes (ALICE, Dolmen, RAPP)
  - Adaptive middleware that alters its behaviour based upon environmental context (CHARISMA, OpenORB)
  - Service Discovery protocols (SLP, UPnP, JESA, Centaurus)
- Explosion of technologies creates *middleware heterogeneity*

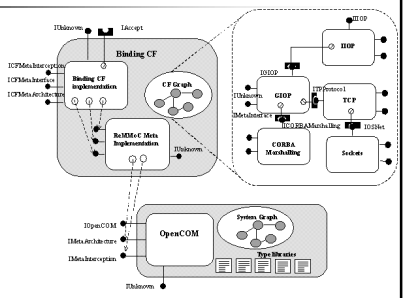
Overview of ReMMoC



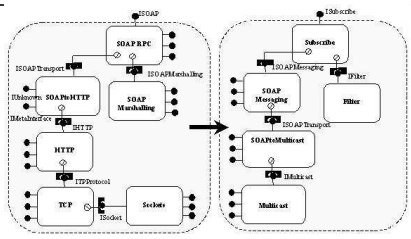
- Organised as two CFs: binding and resource discovery
  - find mobile services irrespective of the service discovery protocol
  - interoperate with services implemented by different middleware types
- Top Level CF abstracts reconfiguration and provides interface for use of middleware (based on *web services* abstractions)

The Binding CF

- Dynamically change between IIOP client, IIOP Server, SOAP, Event Publish & Event Subscribe
- Manage reconfiguration – only valid personalities

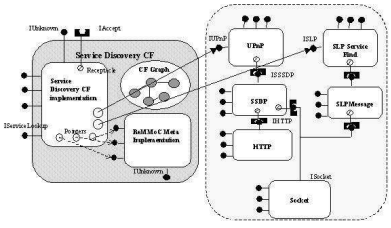


The Binding CF (2)



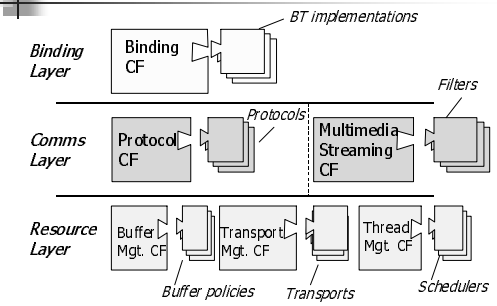
- Dynamic replacement of binding through meta-interface (ICFMetaArchitecture)
- Changes can be also be made at component level – context changes, application requirements

Service Discovery Framework



- Change discovery protocol - Service Location Protocol Lookup, SLP Register, UPnP Lookup …& multiples
- IServiceLookup interface to abstract over lookup requests

A Brief Look at Open ORB (Generation 2)



Use of reflective middleware

- ubiquitous computing
  - Limited resources
    - What is kept on PDA?
  - Heterogeneity
    - Data representation? Processing personality?
  - Dynamism
    - Type of communication network? Protocols? Properties?
    - Discovery of available services?
    - Security policies?

Use of reflective middleware

- Self-healing systems
  - Fault-tolerance, replication groups, ...
  - Self-monitoring by injected monitors and separate meta-level management services
  - Monitoring:
    - Event collector observes behaviour of functional components and generates relevant QoS events
    - Monitor collects QoS events and reports abnormal behaviour to interested parties
  - Control:
    - Strategy selectors select an appropriate adaptation strategy based on feedback from monitors
    - Strategy activators implement a particular strategy by manipulating component graph (scripts)

Use of reflective middleware

- Context-aware applications
  - Device awareness
    - Memory, battery power, screen size
    - Internal resources
  - Environment awareness
    - Bandwidth, network connection, location, services in reach
    - External resources
  - User and application profiles
    - Metadata guidance for context formulation
    - User profile instructs how application should behave in different situations
    - Application profile instructs how middleware support should change in different situations

Use of reflective middleware

- QoS management in multimedia applications
  - End-to-end
  - Monitors
  - Resource managers
  - Service levels: best effort, ... quaranteed

Kurssikirjallisuus

- Geoff Coulson, What is Reflective Middleware? IEEE Distributed Systems ONLINE http://dsonline.computer.org/middleware/RMarticle1.htm
- G. Kiczales, J. Lamping, C V. Lopes, C. Maeda, A. Mendhekar, G. Murphy: Open Implementation Design Guidelines.
- F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, R. H. Campbell: Monitoring, Security and Dynamic Configuration with the dynamicTAO Reflective ORB. Middleware2000, LNCS 1795, 121-143. http://choices.cs.uiuc.edu/2k/dynamicTAO/
- Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski: The Design and Implementation of Open ORB 2. DSONLINE 2, 6. http://dsonline.computer.org/0106/features/bla0106_print.htm

Muu kirjallisuus

- N. Wang, D. C. Schmidt, M. Kircher, K. Parameswaran: Adaptive and Reflective Middleware for QoS-Enabled CCM Applications. DSONLINE 2, 5, 2001. http://dsonline.computer.org/0105/features/wan0105_print.htm
- Projekteja:
  - http://dsonline.computer.org/middleware/Mwprojects.htm
  - Saha, Mukherjee, Pervasive computing: a paradigm for the 21st century.IEEE Computer, march 2003