Adaptive & Reflective Middleware

Andreas Rasche

Roadmap

- Quality of Service
- Adaptive Middleware
- Classification of Adaptive Middleware
- Reflective Middleware & Reflection & Metaprogramming Overview
- Adaptive Middleware Implementations
  - Patterns (Component Configurator, Virtual Component, Quality Connector)
  - BBN Quality Objects, TAO, DynamicTAO, OpenORB
- Adaptation-enabling vs. adaptive middleware

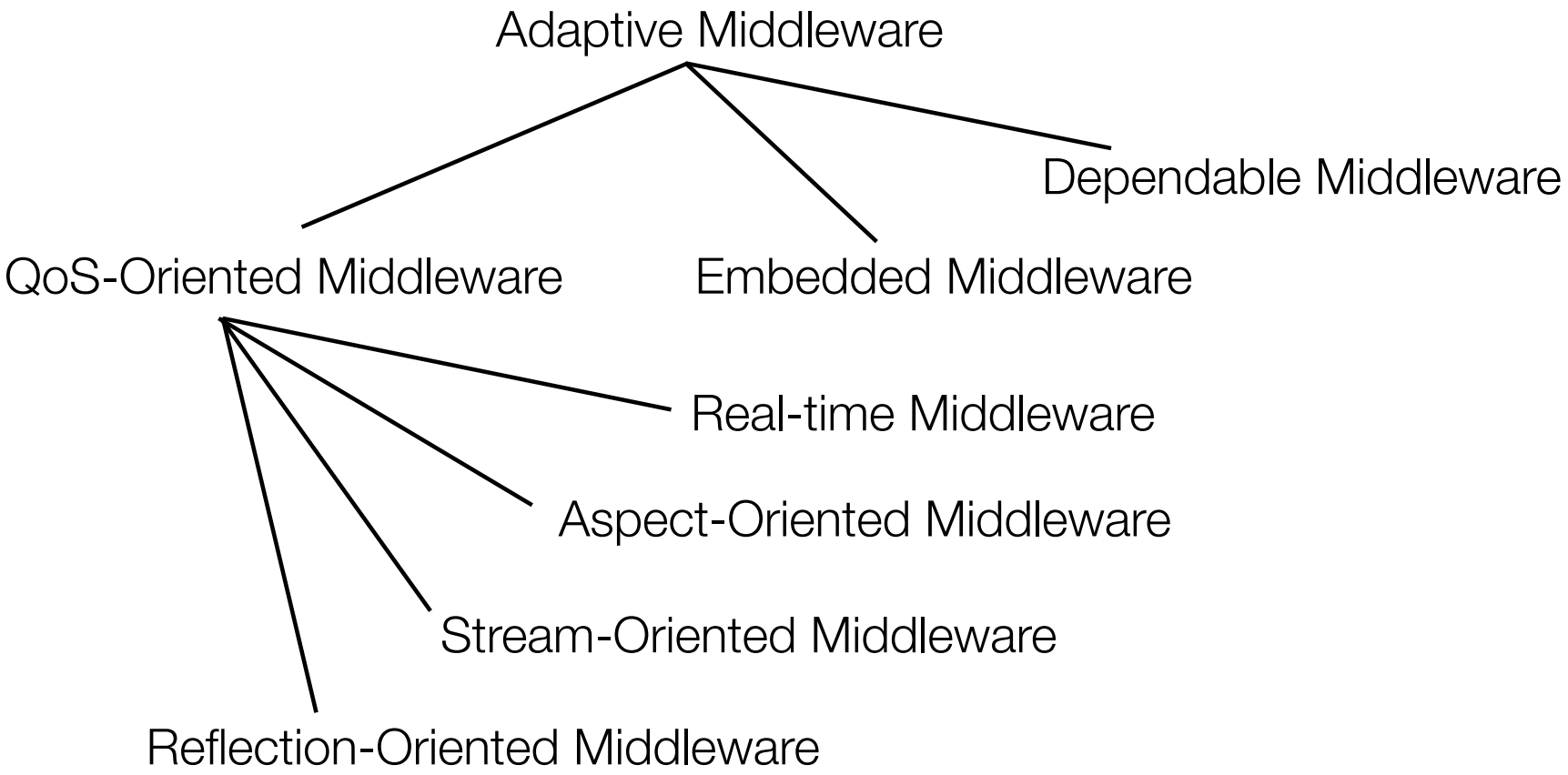
Quality of Service (QoS)

- QoS in field of telephony defined in ITU X.902 as “A set of quality requirements on the collective behavior of one or more objects”
- At the network level QoS refers to
  - control mechanisms that can provide different priority to different users
  - guarantee a certain level of performance to a data flow
- QoS is used as a general quality measure in the sence of “user perceived performance”, or “degree of satisfaction to the user”
- Application-level Quality of Service can be ensured by:
  - Ressource/QoS-reservations at all underlying levels
  - Adaptation of application to cope with changing resource availabilites

Adaptive Middleware

- Adapt: To alter or modify so as to fit for a new use
- Adaptive middleware is software whose functional behavior can be modified dynamically to optimize for a change in environmental conditions or requirements
- Adaptations can be triggered by:
  - changes to a configuration file by an administrator
  - instructions from another program
  - user requests
- Requirements of runtime adaptive system: measurement, reporting, control, feedback and stability
- Adaptive middleware concerned with adapting non-functional aspects of distributed appplications including QoS

Classification of Adaptive Middleware by Domain[Sadjadi]



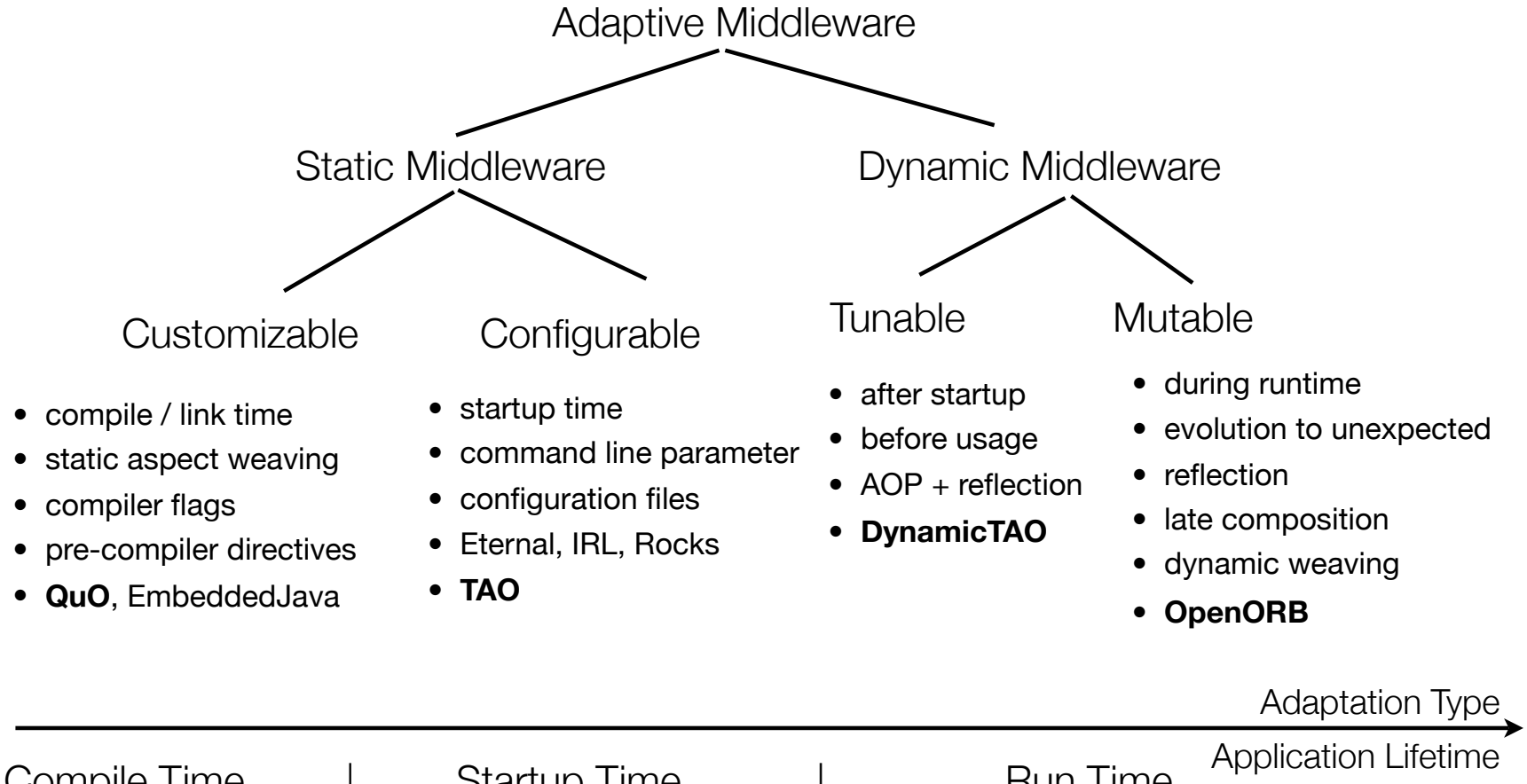
Reflective Middleware

- Reflection on programming languages started by Brian Smith at MIT
  - “Reflection is the integral ability for a program to observe or change its own code as well as all aspects of its programming language - even at runtime”
- Reflective middleware moves reflection to the middleware level
- Often implemented as a number of components that can be configured
- System and application code can use meta-interfaces to:
  - inspect internal configuration of the middleware
  - reconfigure it to adapt to changes in the environment
- Reflection is a technique to enable adaptation

Common Terms

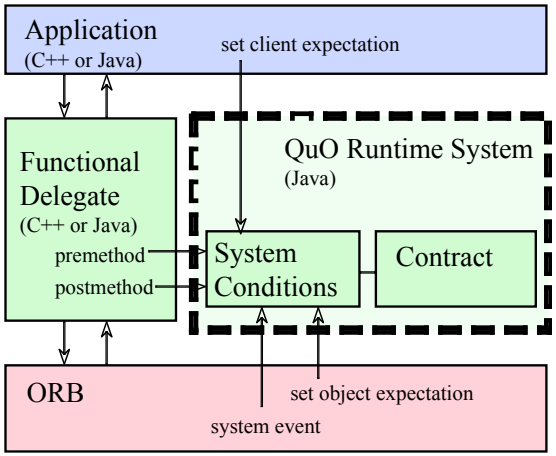
- **Reification**: Process of providing an external representation of the internals of a system. Representation allows for manipulation of system internals
- **Structural Reflection**: Provides the ability to alter the statically fixed internal data/funtional structures. Structural Reflection changes the internal makeup of a program.
- **Behavioral Reflection**: The ability to intercept an operation such as a method invokation and alter the behavior of that operation. Behavioral Reflection alters the actions of a program.
- **Introspection**: Read access to meta data (type information, classes, methods, members, inheritance)
- **Intercession**: Manipulation of meta data

Classification (II) of Adaptive Middleware

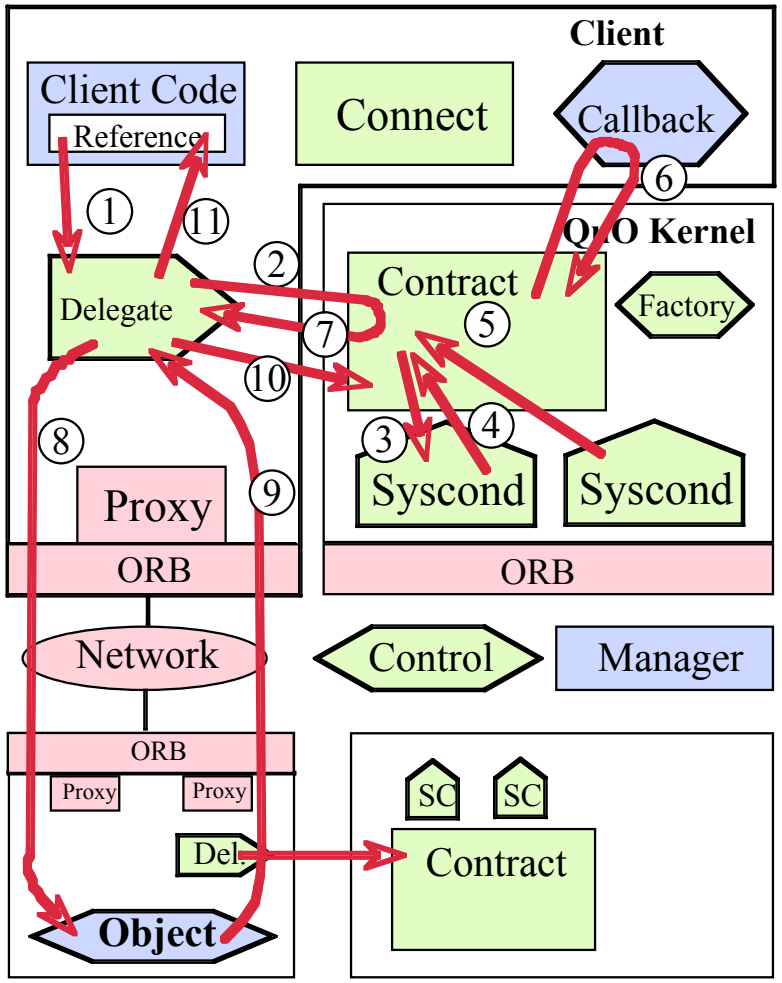


BBN Quality Objects (QuO)

- **Contracts** summarize possible states of QoS and behavior to trigger when QoS changes
  - Defined as regions in form of predicates over system condition objects
- **System Condition Objects** are used to measure and control QoS
- **Delegates** provide local state for remote objects
  - Upon method call/return, delegates can check current contract state and choose behavior based on the current state of QoS
  - Delegates can choose between alternate methods, alternate remote object bindings, perform local processing of data, or simply pass the method call or return through



BBN Quality Objects (QuO) - Adaptive Behavior



1. Client calls delegate
2. Delegate evaluates contract
3. Measurement system conditions are signaled
4. Contract snapshots value of system conditions
5. Contract is re-evaluated
6. Region transitions trigger callbacks
7. Current region is returned
8. If QoS is acceptable, delegate passes the call to the remote object
9. Remote object returns value
10. Contract is re-evaluated...
11. Return value given to client

BBN - Quality Objects - QDL Example

```
typedef sequence<long> LongSeq;
interface Targeting {
  long calculate_distance_to_target(in long xcoord, in long ycoord);
  long identify_target(in long xcoord, in long ycoord);
};
```

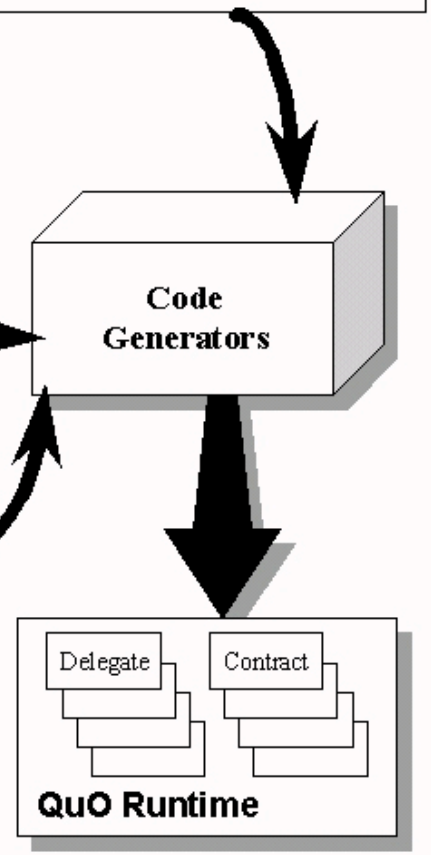
CORBA IDL

```
delegate behavior for Targeting and repl_contract is
  obj   : bind Targeting with name SingleTargetingObject;
  group : bind Targeting with characteristics { Replicated = True };

  call calculate_distance_to_target :
    region Available.Normal :
      pass to group;
    region Low_Cost.Normal :
      pass to obj;
    region Available.TooLow :
      throw AvailabilityDegraded;
  return calculate_distance_to_target :
    pass_through;
  default : pass_through
end delegate behavior;
```

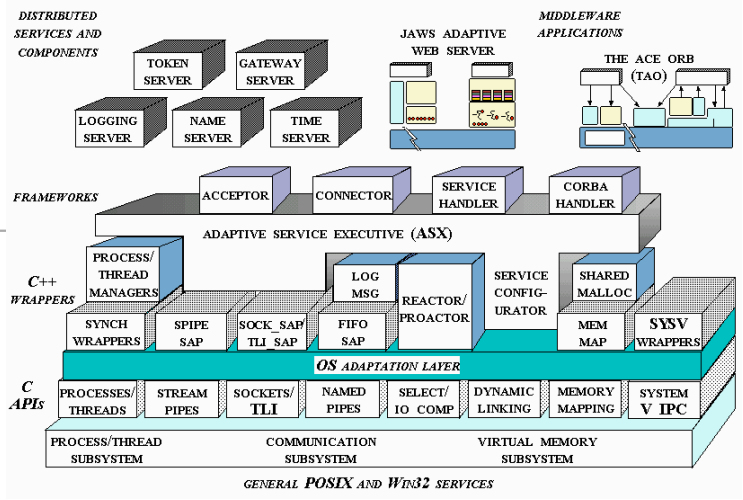
SDL

```
contract Replication( object client, object server ) is ...
 negotiated regions are
  region Low_Cost :   ...
  region Available :
   when client.expectations.requested > 1 =>
    reality regions are
     region Too_Low  : when measured <= 1 =>
     region Normal   : when measured > 1  =>
     transitions are
      transition any->Too_Low : client.callbacks.availability_degraded();
      transition any->Normal : client.callbacks.availability_back_to_normal();
      ...
  transitions are ...
end Replication;
```

CDL

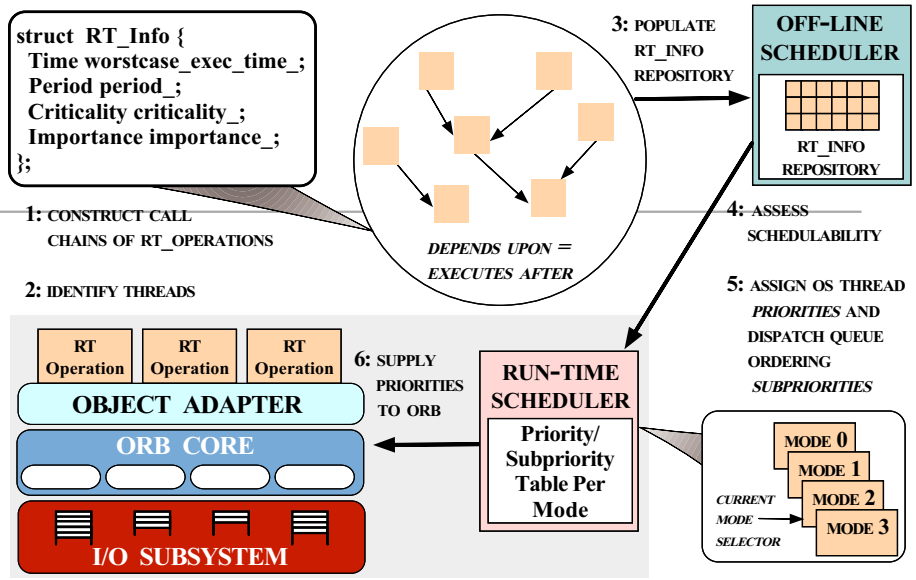


The ACE ORB (TAO)



- Implemented at Washington University St. Louis (D. Schmidt)
- ACE: Adaptive Communication Environment
  - Object-oriented framework that implements many core patterns for concurrent communication software
- TAO - Real-Time CORBA ORB developed on top of the ACE framework
- Uses the strategy design pattern for the encapsulation of ORB internals
  - IIOP pluggable protocols, concurrency, request demultiplexing, scheduling, connection management
  - Strategies defined in configuration files, which are evaluation at start-up (dynamic configuration possible in DynamicTAO and later versions of TAO)

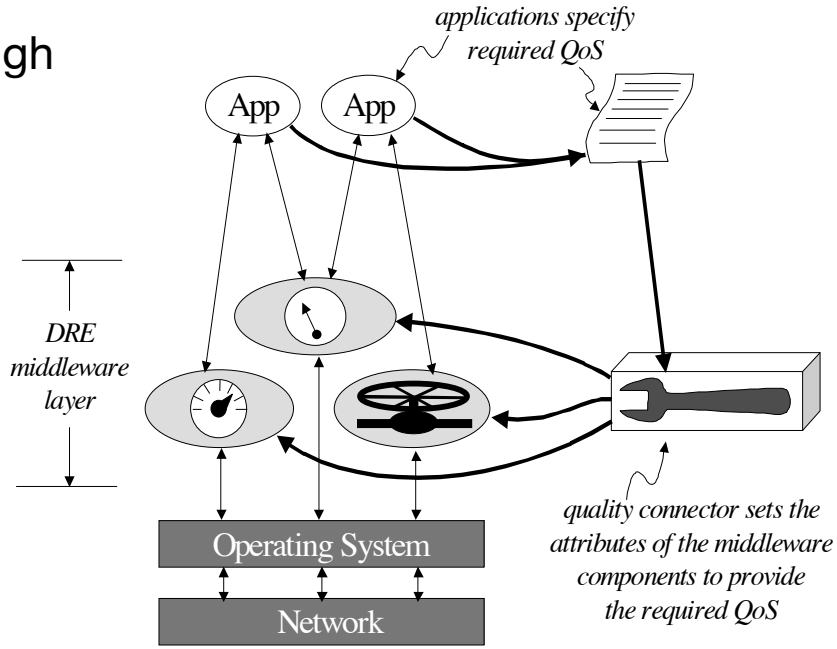
TAO - QoS Specification



- Application specifies QoS using an RT_Info interface
- For every RT_OPERATION the application specifies
  - WCET, period, importance
- During a configuration run the off-line scheduler extracts QoS specification
- During the configuration run a RT_OPERATION dependency graph is created, which can be used for configuration of the middleware
- Scheduler calculates threads, priority dispatch tables, thread priorities for the application including feasibility analysis
- Approach also allows for performance optimization for non-real-time task, by recording task runtimes during configuration runs and calculate optimal prio's

Quality Connector Architectural Pattern - Problem

- Decouples application components from the QoS configuration mechanisms provided by the infrastructure
- Mediates between application and non-standard middleware configuration and control interfaces
- Implementation of service available through standardized functional interfaces provide only non-standard mechanisms for controlling quality of the service
- A quality sensitive application should be able to monitor and control the quality of its supporting services
- [J. Cross and D. Schmidt '02]

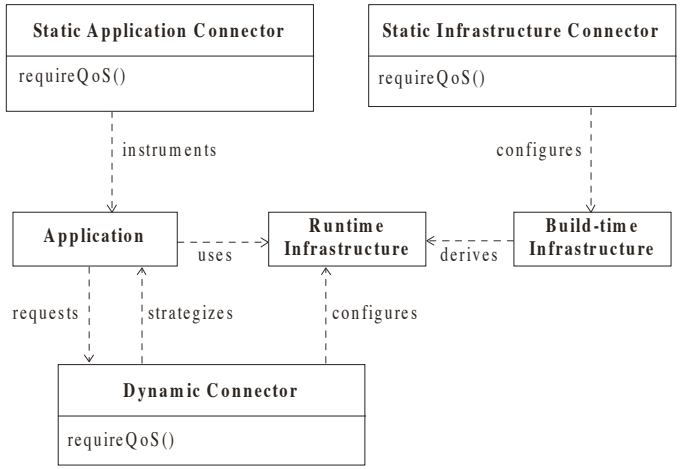


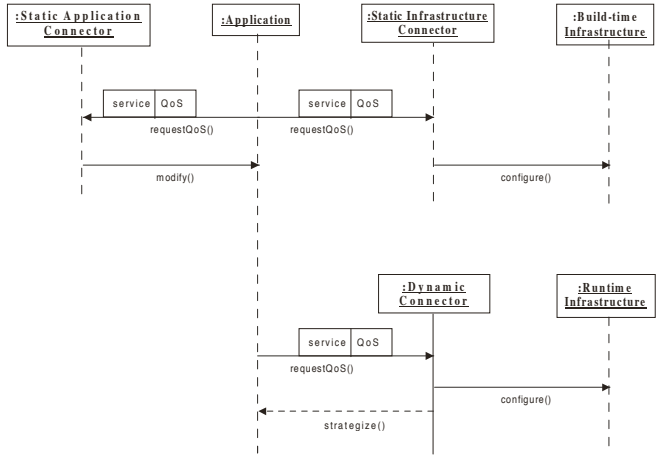
Quality Connector - Solution

- Implemenation of a Quality Connector object for each infrastructure component
- Platform independend interface between application and infrastructure
  - Concerned only with: quality of service provided, modes of the system, load that will be imposed on the service
- Implementation strategy:
  - Define a small language (definition of acceptable values, depending on system mode, use XML ...)
  - Provide configuration tools (to check feasibility and consistency of requested quality values, to set properties of runtime components)
  - Implement the dynamic connector (runtime allocation of resources)

Quality Connector - Structure & Dynamics

- **Static application connector** - integrates hooks for dynamic connector configuration (request of quality values)
  - applied at source code level e.g. using AOP
- **Static Infrastructure Connector** - acts on underlying middleware before linking (selection of implementations, recompilation of middleware using chosen values for configuration parameters)
- **Dynamic Connector** - linked with the application - allocates infrastructure resources to data flows





Quality Connector - QoS Language

```
<proposal>
   <mode>
      <or>
         <ci name="radioVHF" state="onLine"/>
         <ci name="radioUHF" state="onLine"/>
      </or>
   </mode>
   <QoS type="latency">
      <upperPoint secs="1.0" prob="0.99"/>
      <upperPoint secs="4.0" prob="0.9999"/>
   </QoS>
   <load type="interMessageTime">
      <upperPoint secs="1.0" prob="0.0001"/>
      <lowerPoint secs="1.0" prob="0.9999"/>
   </load>
   <load type="messageSize">
      <upperPoint bytes="256" prob="1.0"/>
      <upperPoint bytes="32"  prob="0.5"/>
   </load> <load type="priority">
      <urgency val="10"/>
      <importance val="2"/>
   </load>

</proposal>
```

Proposal applies in this mode

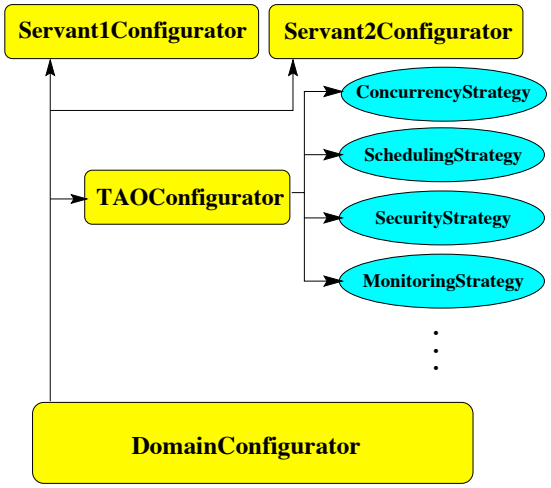
There are QoS types other than latency -- e.g., jitter

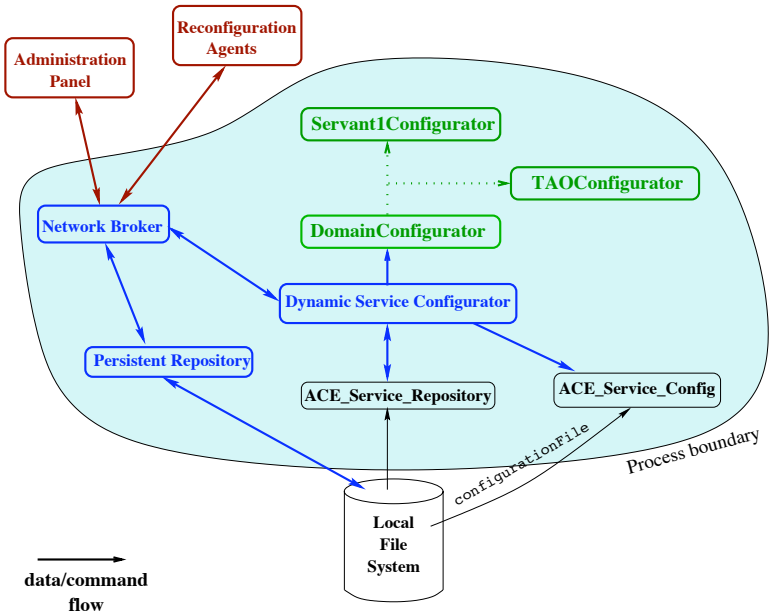
Flow is periodic

Priority determines how this request will compete with others for resources

DynamicTAO - Real-Time CORBA

- Developed at University of Illinois (Campbell, F. Kon)
- Adds dynamic reconfiguration features to the TAO ORB implementation
- ORB strategies can be changed/adapted during runtime
- Uses the Service Configurator pattern for strategy configuration





DynamicTAO

- Component implementations are organized in categories representing different aspects of the TAO ORB
- Components are packaged as dynamically loadable libraries that can be linked to the ORB at runtime
- For example the category “Concurrency” contains:
  - Reactive_Strategy
  - Thread_Strategy
  - Thread_Pool_Strategy

```
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);
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);

 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);
};
```

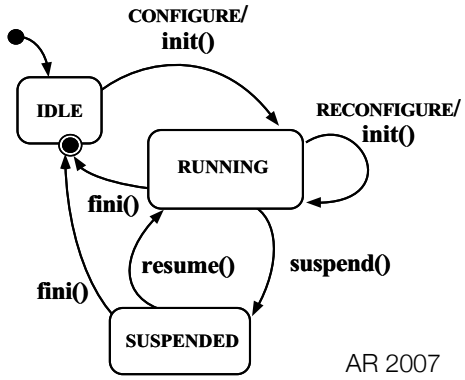
DynamicTAO - Configuration Example

```
CORBA::Object_var           dcObj;
DynamicConfigurator_var     dynConf;
CORBA::ORB_var              orb;

orb     = CORBA::ORB_init(argc, argv);
dcObj   = orb->resolve_initial_references
                                ("DynamicConfigurator");
dynConf = DynamicConfigurator::_narrow(dcObj.in());

stringList *list
        = dynConf->list_implementations ("Concurrency");

printf ("Available concurrency strategies:");
printStringList (list);

char *ret
     = dynConf->get_hooked_comp ("TAO",
                                "Concurrency_Strategy");

printf("Now, using the <%s> concurrency strategy.", ret);
```

```
myRemoteOrb->upload_implementation(“Security”, “superSAFE”,superSAFE_impl);

newSecurityStrategy =  myRemoteOrb->load_implementation (“Security”,“superSAFE”);

oldSecurityStrategy = myRemoteOrb->get_hooked_comp (“dynamicTAO”,“Security_Strategy”);

myRemoteOrb->hook_implementation (newSecurityStrategy, “dynamicTAO”,“Security_Strategy”);

myRemoteOrb->remove_implementation (oldSecurityStrategy);
```

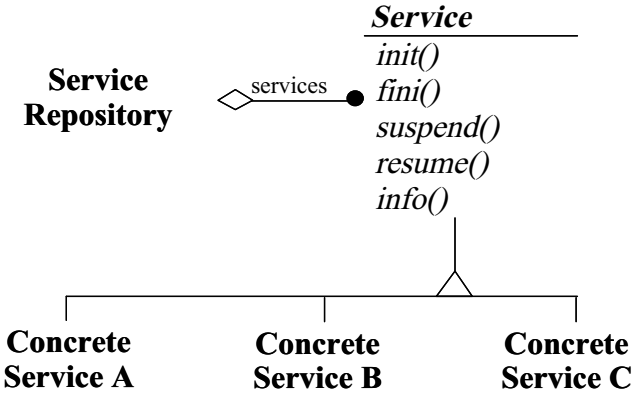
Service Configurator Pattern [P.Jain, D. Schmidt '96]

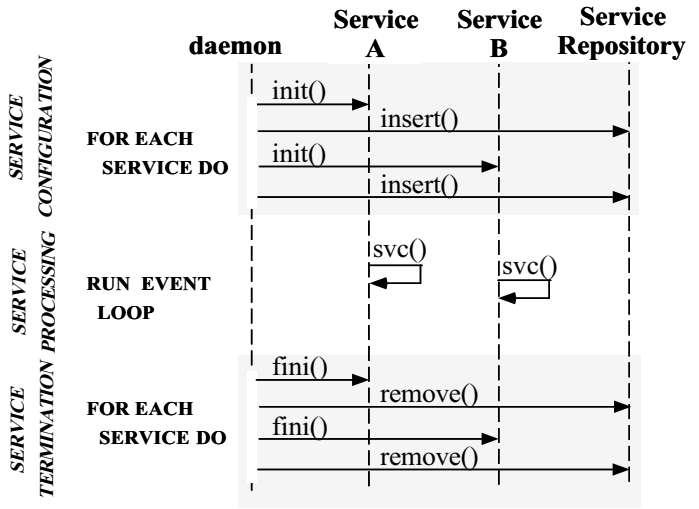
- Also known as Component Configurator pattern
- Decouples the implementation of services from the time when they are configured
- The service configurator pattern should be applied when:
  - a service needs to be initiated, suspended, resumed, and terminated dynamically
  - a service configuration decision must be deferred until runtime
  - depending service implementation must be configured independently at runtime
- Already used in device drivers architecture in Windows NT and Solaris, inetd, Java applets im WWW-Browsers, Linux modules



Service Configurator Pattern

- **Service** - Specifies the interface that contains the abstract hook methods
- **Concrete service** - Implements hook methods and other service specific functionality (event processing, communication with clients)
- **Service repository** - maintains a repository of all services offered by a Service Configurator-based application



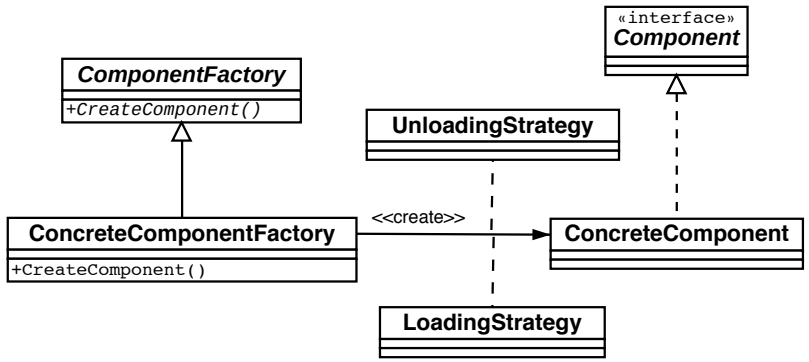


Virtual Component Pattern [Corsaro '02]

- Provides an application transparent way of loading and unloading components
- Reduction of static and dynamic memory footprint for embedded applications
- Ensures that middleware provides a rich and configurable set of functionality, yet occupies main memory only for components that are actually used
- One example are compliant implementations of CORBA having many features not needed by all applications
  - A server application may not use all versions of the CORBA IIOP protocol
  - A client application may not use all collocation optimizations, interceptors, or smart proxy mechanisms
  - "Pure client" applications do not require a POA
  - Applications may not use all common middleware services (naming, security, transactions ...)

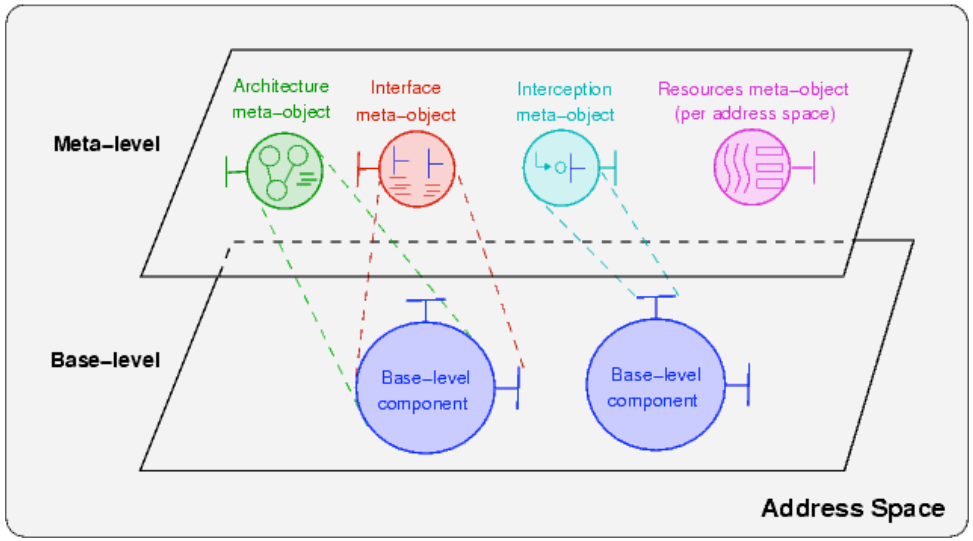
Virtual Component Pattern - Solution

- Identify components whose interfaces represent the building blocks of the middleware
- Define concrete components that implement the middleware capabilities
- Define factories that create concrete components using a set of loading/ unloading strategies
  - Lazy loading: e.g. when memory gets low
  - Eagerly loading: e.g. as soon as the instance reference count goes to 0

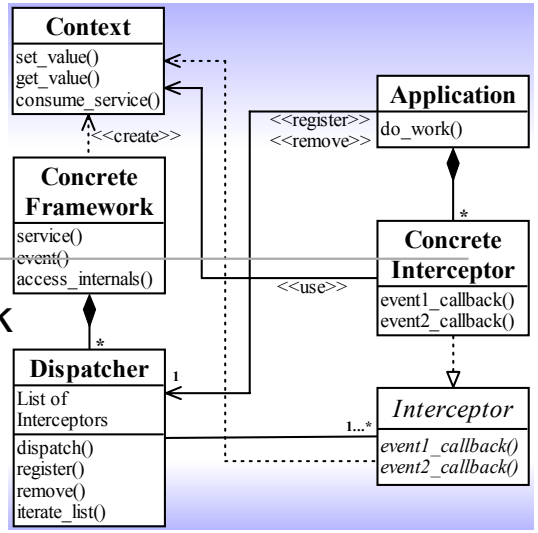


Open ORB - Metaspace Models

- **Interface metamodel** provides access to the external representation of a component (provided and required interfaces) (introspection)
- **Architectural metamodel** provides access to the implementation in form of a software architecture (component graph and architectural constraints)
- **Interception metamodel** allows for dynamic insertion of interceptors (insertion of pre- and post-behavior)
- **Resource metamodel** provides access to underlying resource management (memory usage, OS threads, buffers...)

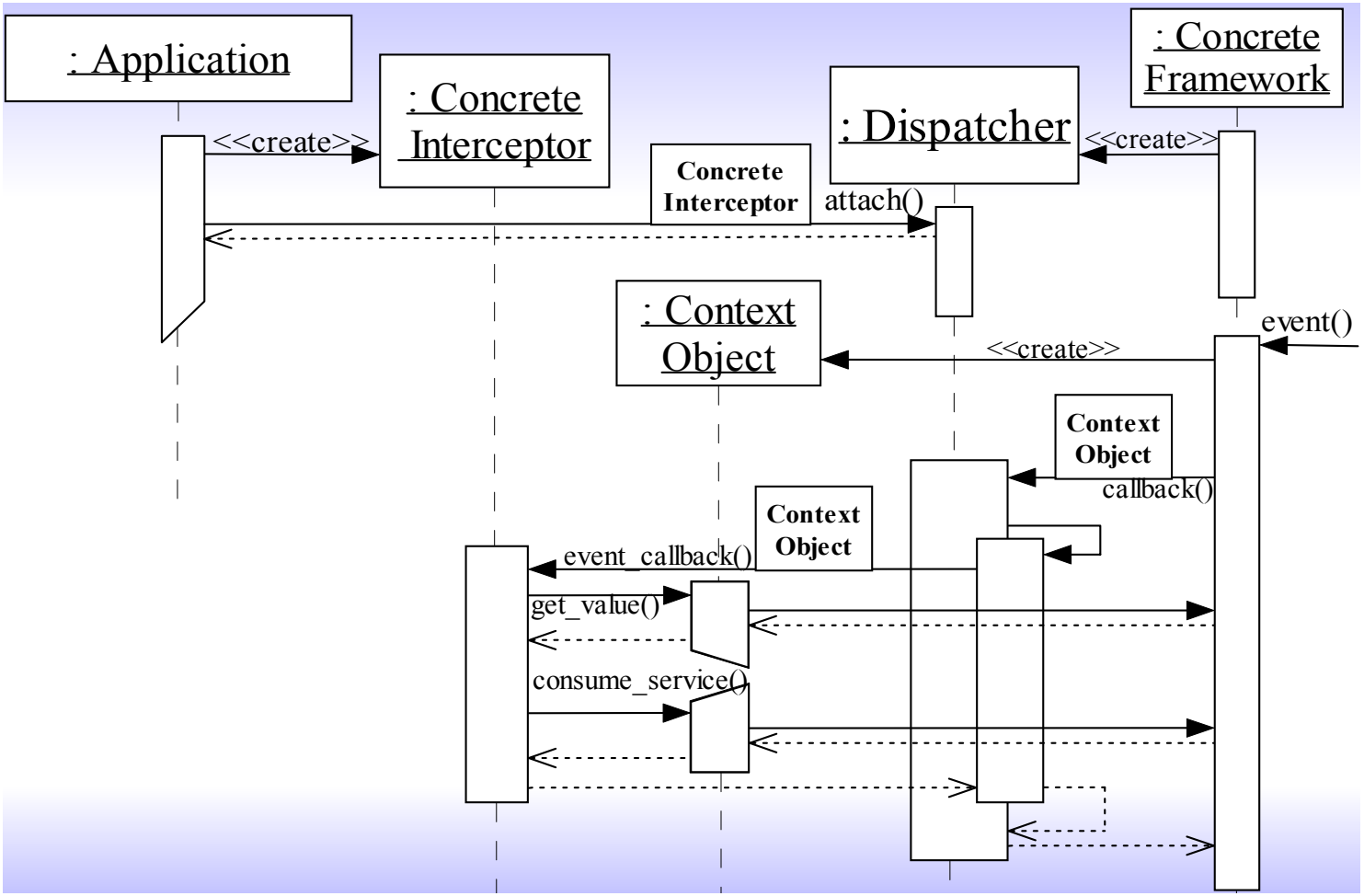


Interceptor Pattern [POSA II]



- Allows services to be added transparently to a framework
- Problem: Late integration of new services
- Solution:
  - For a set of framework events (outgoing call, incoming request ...) processed by a framework specify and expose an interceptor callback interface
  - Applications can derive concrete interceptors from this interface to implement out-of-band services
  - Applications can register concrete interceptors at dispatchers
  - Context objects allow concrete interceptors to introspect and control certain aspects of the framework's internal state and behavior
- Also a mechanism to change behavior and application-level QoS parameters

The Interceptor Pattern - Dynamics



Further Readings

- J.Malenfant et. al. “A Tutorial on Behavioral Reflection and its Implemtation”
- P. Pal. et. al. “Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration”
- S.M.Sadjadi “A Survey of Adaptive Middleware”
- J. Cross and D. Schmidt “Quality Connector - An Architectural Pattern to Enhance QoS and Alleviate Dependencies in Distributed Real-time and Embedded Middleware”, 2007
- D. Schmidt, M. Stal, H. Rohnert, F. Buschmann “Pattern-Oriented Software Architecture Volume 2, Pattern for Concurrent and Networked Objects”, Wiley 2001(Interceptor, Component Configurator, Leader/Follower, Half-Asynch/Half-Synch Pattern)