# DotQoS – QoS in .NET

## Integration of non-functional properties in .NET

**Andreas Ulbrich, Torben Weis**
Intelligent Networks and Management of Distributed Systems (iVS)
TU Berlin

# Topics

Motivation and Objectives
DotQoS Inside
The Road Ahead

# Quality of Service

Non-functional properties
Separation of concerns
Existing Approaches

- Mostly built around CORBA (MAQS, OpenORB, Quo, TAO…)
- Aspects, modified languages (QIDL, …)

Design, Implementation, Runtime

# Objectives of DotQoS

Simple, useable
Runtime Adaptation
Support multi-category QoS
Allow reuse of QoS mechanisms
Do we really want extended IDL or aspect languages?
Do not modify underlying middleware!
No real-time support!

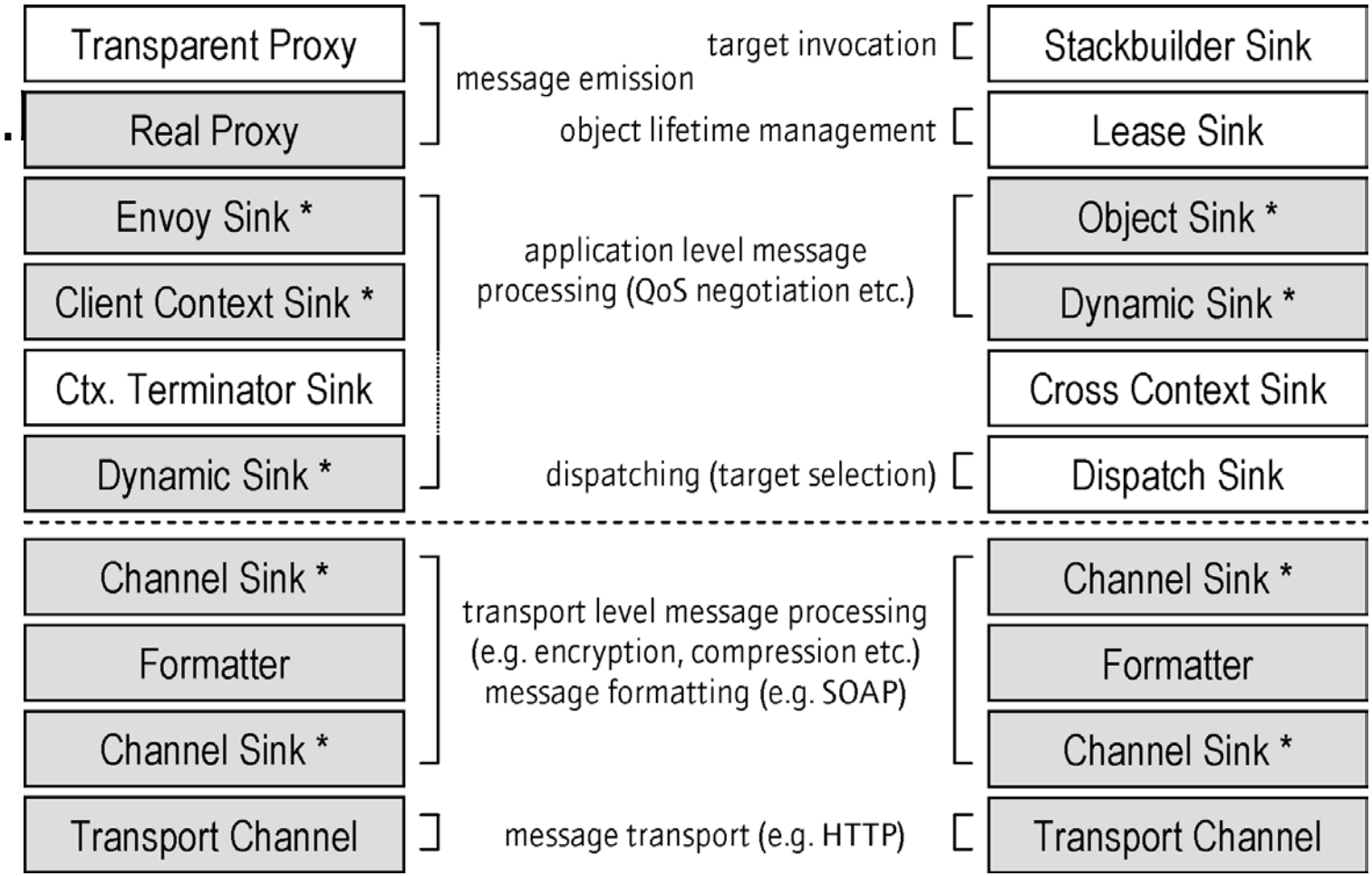
# The way it works

What do we need?

- QoS Specification
- QoS Negotiation
- QoS Mechanisms

What do we use?

- .NET Remoting
- Highly reflective, custom meta-data
- Flexible middleware (not really adaptive yet)


Transparent Proxy
Real Proxy
Envoy Sink *
Client Context Sink *
Ctx. Terminator Sink
Dynamic Sink *
Channel Sink *
Formatter
Channel Sink *
Transport Channel
message emission
application level message processing (QoS negotiation etc.)
dispatching (target selection)
transport level message processing (e.g. encryption, compression etc.)
message formatting (e.g. SOAP)
message transport (e.g. HTTP)
target invocation
object lifetime management
Stackbuilder Sink
Lease Sink
Object Sink *
Dynamic Sink *
Cross Context Sink
Dispatch Sink
Channel Sink *
Formatter
Channel Sink *
Transport Channel

# QoS Specification in DotQoS

A QoS category is defined by its contract scheme

- Defines the QoS dimension that hold parameters of the QoS level

DotQoS

- Classes derived from DotQoS.Contracts.QoSCategorySchemeBase
- Dimensions are properties decorated with QoSDimension attribute

Specialisation by inheritance

Highly reflective stuff

# QoS Specification

```
public class Encryption : QoSCategorySchemeBase {

    [QoSDimension(QoSDirection.Ascending,
    QoSUnit.Amount)]
    public int KeyLength {
            get { return (int)this.Parameters["KeyLength"]; }
            set { this.Parameters["KeyLength"] = value; }
    }
}
```

# Interfaces with QoS support

Need to declare interfaces to support QoS
Costum attribute QoSContractClass
QoS valid for all **public** methods of this interface

- Including property accessors

Declaration only!!!

- Does say nothing about the QoS mechanism to use.

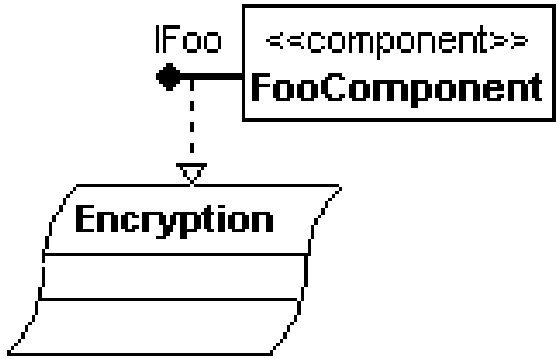
## Declaring QoS Support

```
[QoSContractClass(typeof(Encryption))]
public interface IFoo {

    public void DoSomething() {...}
    public int DoSomethingDifferent(int
    whatever) {...}

}
```

# Multicategory QoS

Multiple contracts
Interface inheritance
Objects support more than one QoS
Need compound contracts

# Component Contract




Idea based on the concept of software components
Components have ports

- Port provides an interface (functional contract)
- Ports may have non-functional contracts, too (QoS)

Interfaces implemented by a class represent ports
QoS of a port

- Union over all QoS on the interfaces in the ports inheritance hierarchy

Component contract maps ports to their QoS categories

- Keep a mapping of methods to QoS at runtime

# Plugging QoS into .NET Remoting

Contract negotiation
Contract enforcement
QoS mechanisms

# QoS aware Components

All component impl. inherit from DotQoS.RemoteObject

Defines common QoS functions

All objects are context bound

Cross context invocations are QoS aware (even in-process)

Every component (i.e. object) executes within own context

Must be decorated with DotQoS.QoSContextAttribute

Defines the context to execute in

# Negotiation

FrameContract for each client/server interaction

- Contains component contract for the server component
- Has a unique ID (actually a GUID)

Client can negotiate elements of component contract

- Either in a defined order (i.e. security first) or entire component contract

Server throws exception if client wish is not acceptable

Contract scope must map to component

- Server side: connection

# Client Example

```
Foo foo = Activator.GetObject(...);

DotQoS.FrameContract fc =
    foo.CreateFrameContract(DotQoS.ContractServices.Default
    Observer);

// specify required QoS level
Encryption encr = new Encryption();
encr.KeyLength = 1024;
encr.IsActive = true;
// configure elements of component contract
fc.ConfigureCategory(typeof(IFoo), encr);
```

# QoS Enforcement

Contract ID (Guid) is passed with **every** message

- Sinks take care of this

QoS mechanisms retrieve ID, fetch contract and do whatever is necessary

# QoS Mechanisms

Request level vs. Message level
Must be adaptable (i.e. exchange them at runtime)
Build on existing concepts

Sink chains

# QoS Mechanism

Default sinks that are always present

Request level: Object sinks (server side) and envoy sinks (client side)

- Requires objects to be context bound

Message level: Channel sinks, formatters

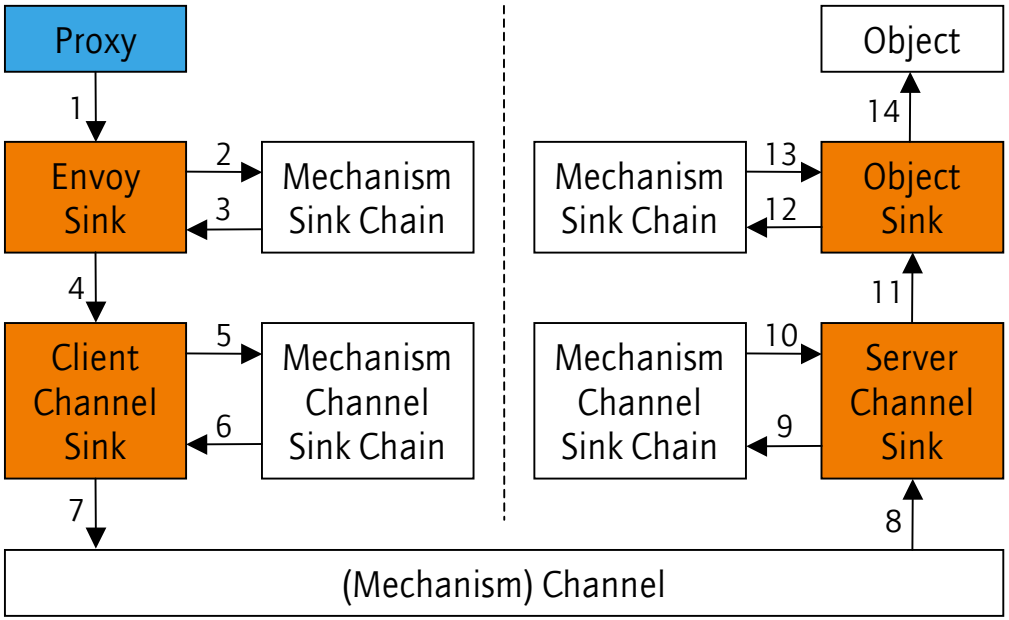
Inject contract IDs into message or message headers

Problem: Sink chains set once, then fixed

Won't be adaptable

Solution: Sink loops (not really nice but work well)

# Sink loops


Proxy
1
Envoy Sink
2
3
Mechanism Sink Chain
4
Client Channel Sink
5
6
Mechanism Channel Sink Chain
7
(Mechanism) Channel
8
Server Channel Sink
9
10
Mechanism Channel Sink Chain
11
Object Sink
12
13
Mechanism Sink Chain
14
Object

This slide was inserted after the talk in response to audience questions.

# Mapping Categories to Mechanisms

So far we have

- Categories for QoS specification
- Mechanisms to enforce categories a runtime

Configuration-information maps categories to mechanisms

- May be application/component specific
- Multiple mappings may exist

Mechanism installed at frame contract activation

Still under investigation

- Combination of mechanisms for different categories
- Selection of most suitable mechanism if more than one

# So far so good

We have

- Flexible, adaptable infrastructure
- QoS specification, negotiation, enforcement
- Pure add-on (a couple of DLLs), **No changes to .NET or Remoting!!!**
- No IDL and No Aspect language!

Still work on

- Safe combination of mechanisms
- Resource management, policies, and runtime adaptation

# Experience

Fun to work with .NET
VS.NET is very decent
Remoting is cool but does not always behave as expected

- Contexts and related stuff is poorly documented
- Current implementation is a bit of a hack
- Need to read the source (SSICLI) to get things done
- Conclusion: Some concepts are very cool but MS seemed to have worries about exposing them to the programmer.