Quality Objects (QuO): Adaptive Management and Control Middleware for End-to-End QoS

Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz

BBN Technologies/GTE Technology Organization

Cambridge, Massachusetts, USA

http://www.dist-systems.bbn.com/tech/QuO/

OMG's First Workshop on Real-Time and Embedded Distributed Object Computing

July 24-27, 2000

Falls Church, Virginia, USA

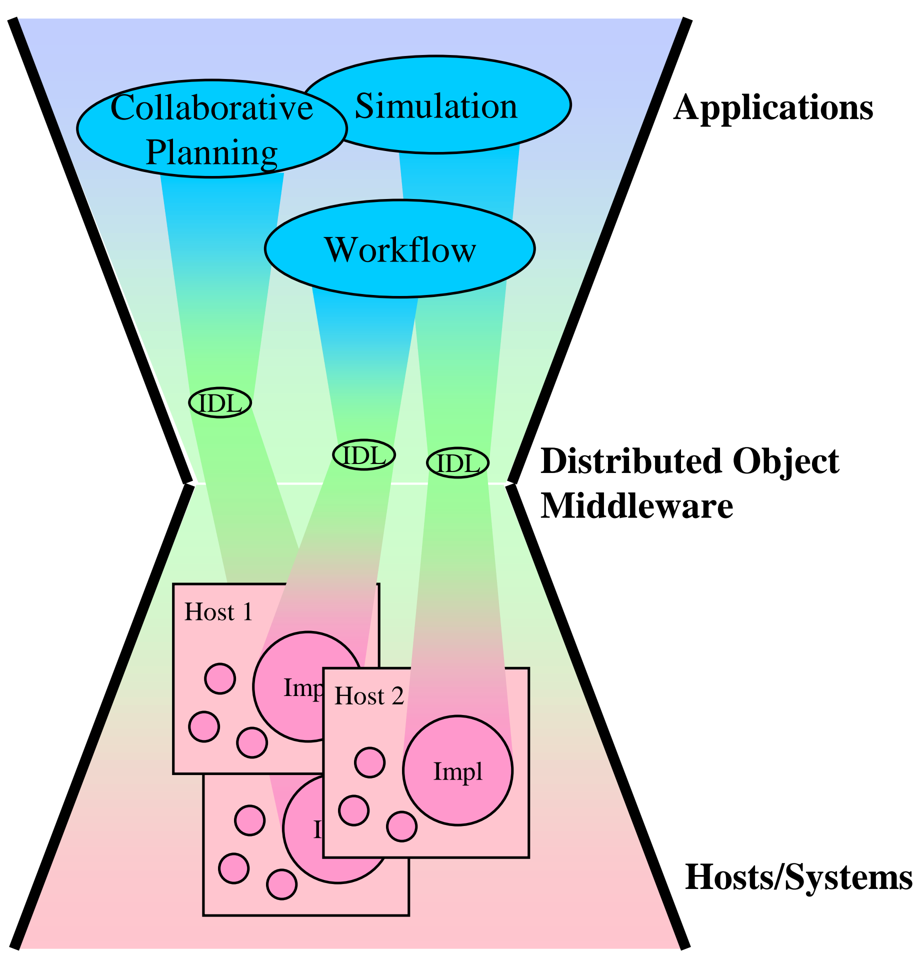
As real-time applications become more distributed and complex, their needs for knowledge and control increase.

- Real-time applications always require resource management and allocation.
- Traditionally, resources (CPU, memory, I/O) are controlled at run-time but allocated at design time.
- More complex applications and environments require adaptive behavior to deal with uncertainty.
- Distributed RT applications must deal with an unpredictable network, as well as control environment.

Resource Allocation and Control are Common to RT Applications - Solve Problem Once with Middleware.

- Adaptive resource allocation must be supported by all applications in a system.
- Resource management requires low-level control, which ties applications to a specific platform.
- Adaptive applications must support operation with variable available resources.
- Applications should monitor delivered QoS to determine current resource requirements.

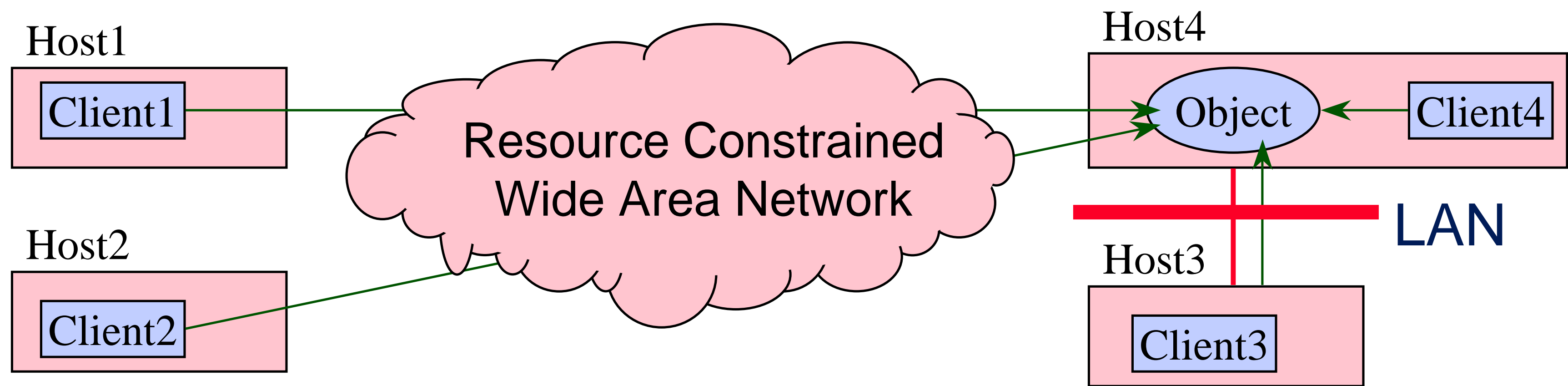
Distributed object middleware has emerged to solve heterogeneity and distribution problems



Middleware makes programming distributed applications easier

- Standard programming interfaces hide platform and system dependencies
- Standard protocols, e.g., message formats, allow applications on different systems to interoperate
- Middleware provides higher level, application oriented programming building blocks

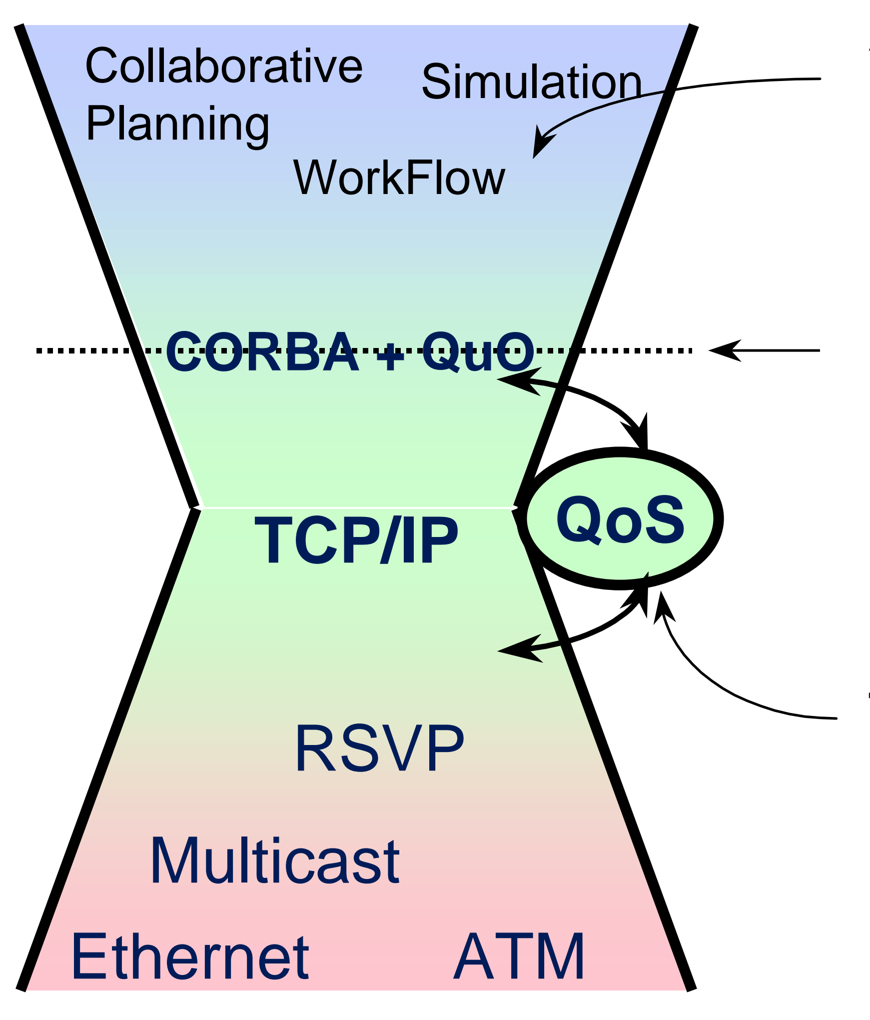
Wide-area distributed applications are (still) hard to build and maintain



Current object-oriented middleware is not sufficiently transparent with respect to real-time, fault tolerance, and other non-functional issues, and does not sufficiently accommodate adaptive behavior

- WANs are unpredictable, dynamic environments
- Configuration for application changes over time
- Performance and system properties are buried under IDL's functional interface, so one can't easily
  - help application adapt to changing environment
  - reuse code for a new environment
- Programmers end up programming around the middleware to achieve real-time performance, predictability, security, etc.

Distributed object middleware with QoS extensions is a powerful abstraction layer on which to build applications



Applications for geographically dispersed, heterogeneous environments

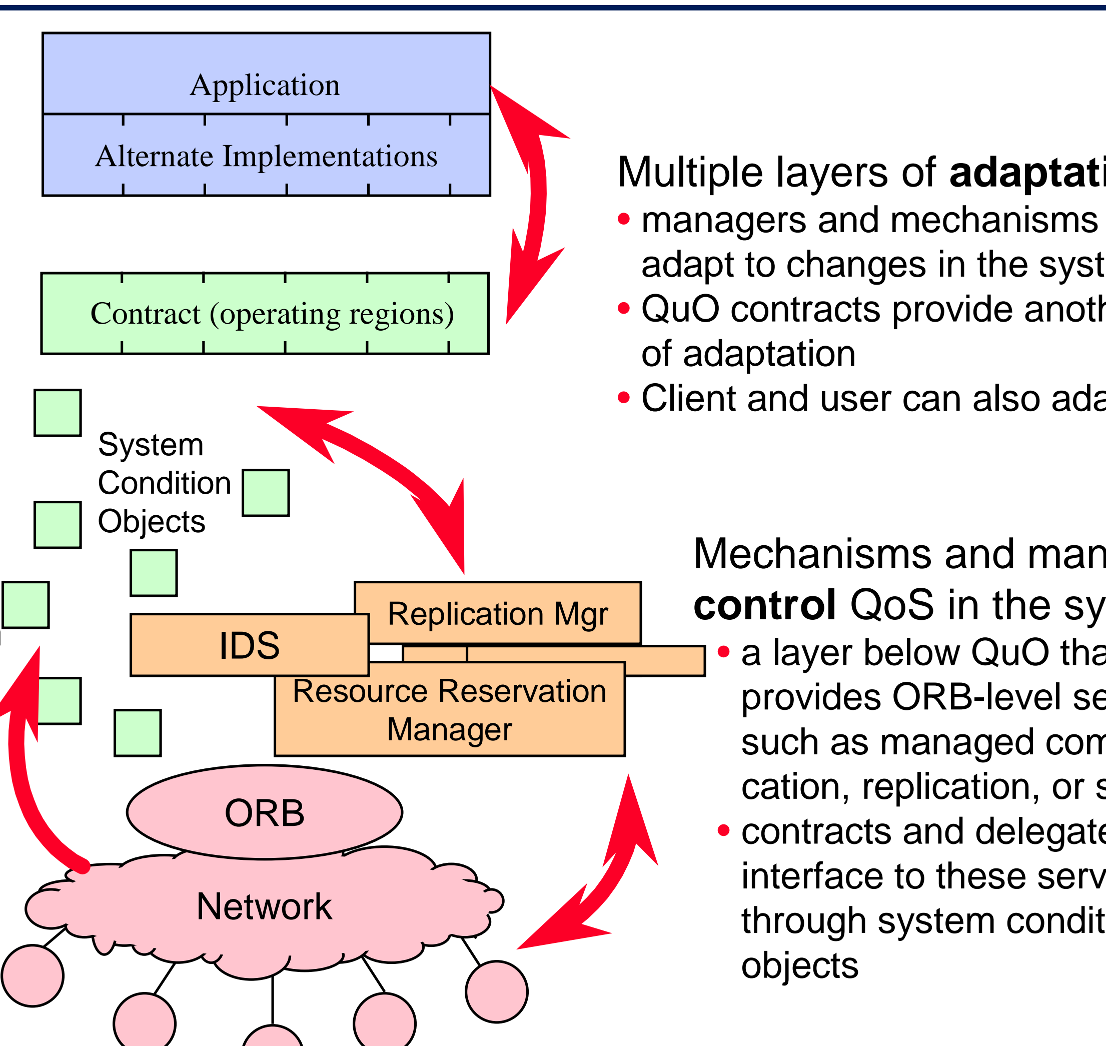
Distributed objects are the first abstraction layer that unifies CPU, storage, and communications

This interface needs to be hidden from applications
- It is too complicated
- It is changing too quickly

QuO applications specify, control, monitor, and adapt to QoS in the system

Specification of operating regions, alternate implementations, and adaptation strategies using QuO's QDL

System condition objects **monitor** QoS in the system
- system condition objects recognize changes in the system and notify the contracts that observe them
- QuO contracts notify client programs, users, managers, and other system condition objects through transition behavior



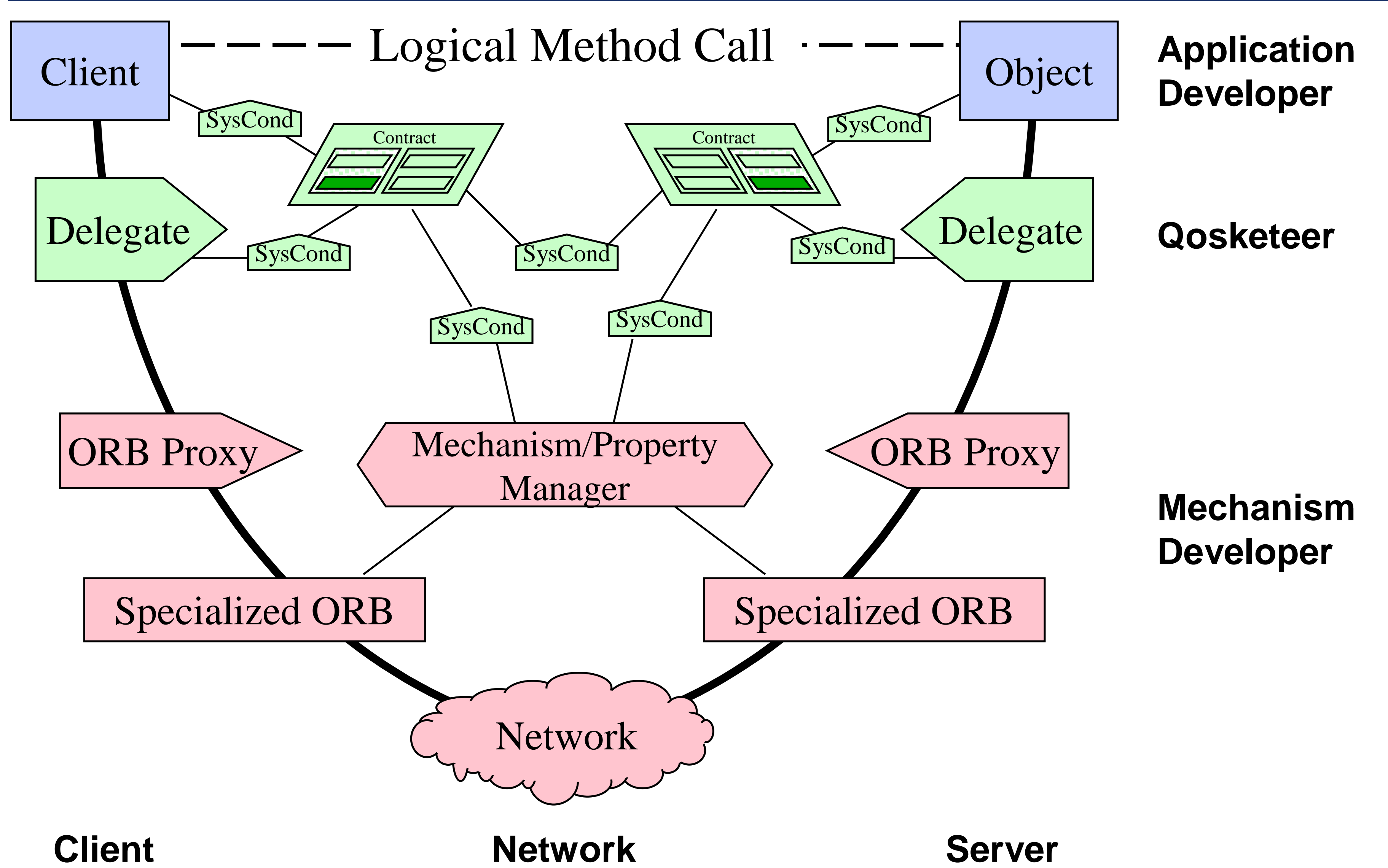
Multiple layers of **adaptation**
- managers and mechanisms can adapt to changes in the system
- QuO contracts provide another layer of adaptation
- Client and user can also adapt

Mechanisms and managers **control** QoS in the system
- a layer below QuO that provides ORB-level services, such as managed communi-cation, replication, or security
- contracts and delegates interface to these services through system condition objects

The QuO designer provides contracts, system condition objects, and delegates in the application

- **Contracts** summarize the possible states of QoS in the system and behavior to trigger when QoS changes
  - Regions can be nested, representing different epochs at which QoS information becomes available, e.g., *negotiated regions* represent the levels of service a client expects to receive and a server expects to provide, while *reality regions* represent observed levels of service
  - Regions are defined by *predicates* over system condition objects
  - *Transitions* specify behavior to trigger when the active regions change
- **System condition objects** are used to measure and control QoS
  - Provide interfaces to system resources, client and object expectations, mechanisms, managers, and specialized ORB functions
  - Changes in system condition objects observed by contracts can cause region transitions
  - Methods on system condition objects can be used to access QoS controls provided by resources, mechanisms, managers, and ORBs
- **Delegates** provide local state for remote objects
  - Upon method call/return, delegate can check the current contract state and choose behavior based upon the current state of QoS
  - For example, delegate can choose between alternate methods, alternate remote object bindings, perform local processing of data, or simply pass the method call or return through

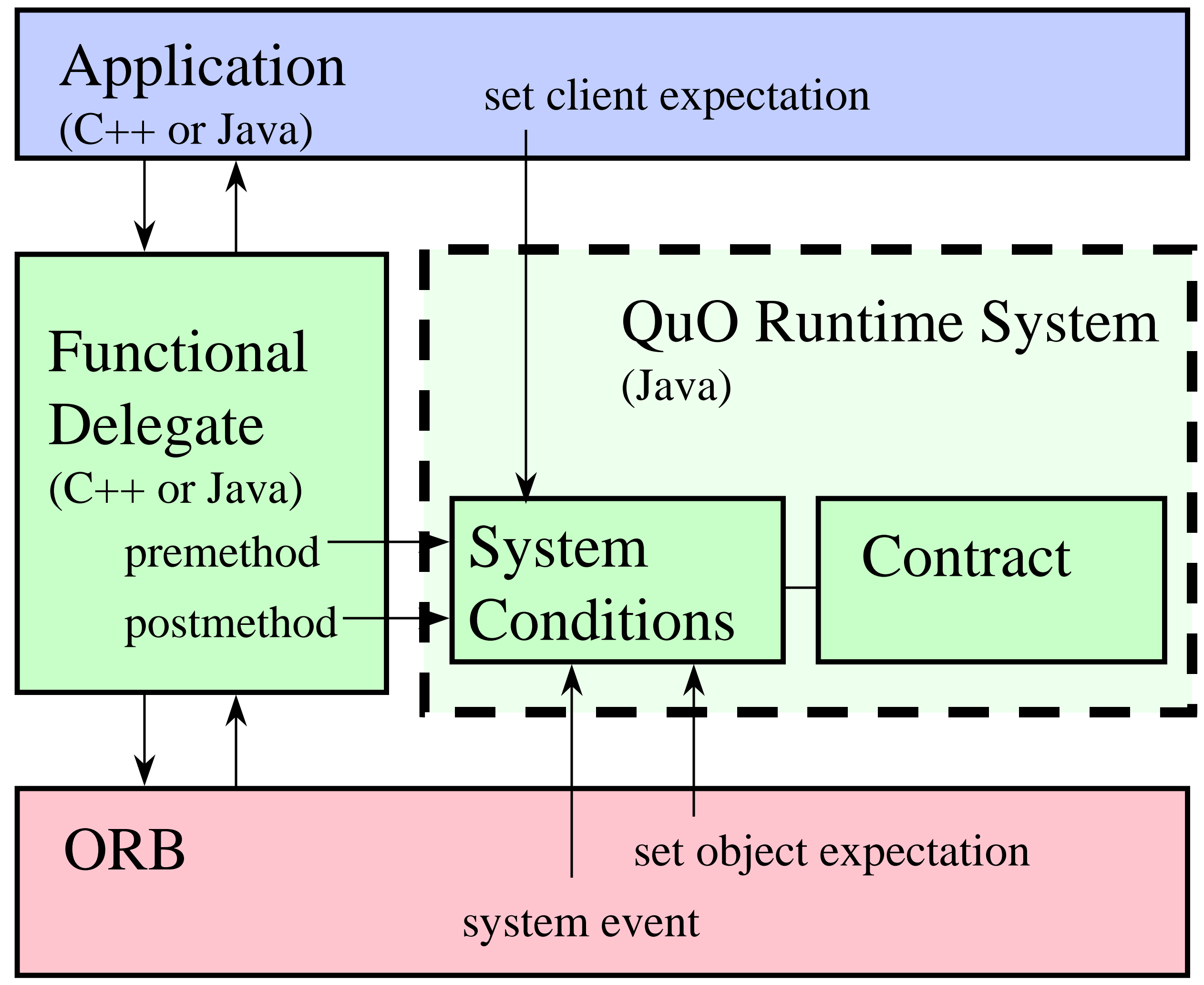
QuO adds QoS control and measurement into the DOC remote method call



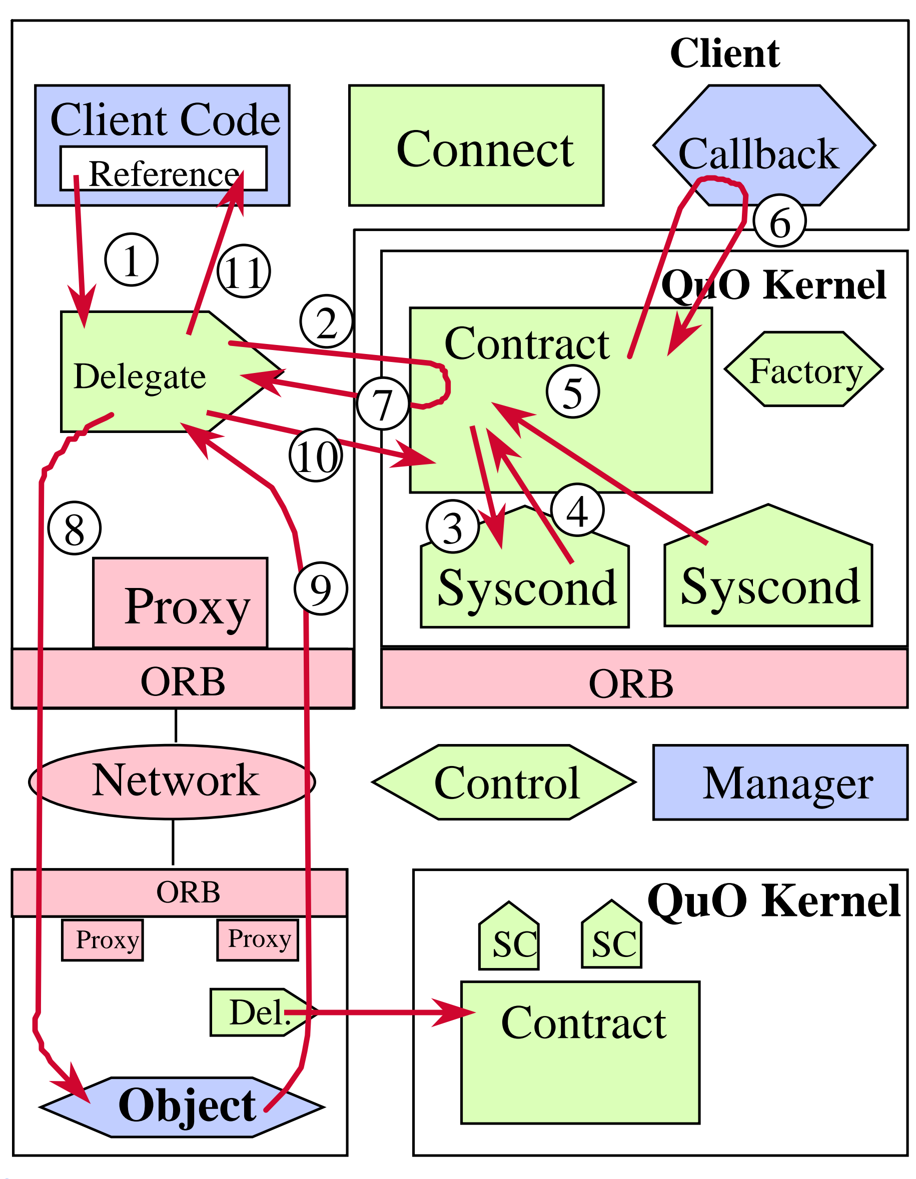
Major components of the QuO framework

- Quality Description Languages (QDL)
  - Analogous to CORBA's Interface Description Language (IDL)
  - Support the specification of
    - QoS contracts
    - delegates and their adaptive behaviors
    - connection, creation, and initialization of QuO application components
  - QuO includes code generators that parse QDL descriptions and generates Java and C++ code for contracts, delegates, creation, and initialization
- QuO Runtime Kernel
  - Contract evaluator
  - Factory object which instantiates contract and system condition objects
- System Condition Objects
  - Implemented as CORBA objects
  - We have a growing library of system condition objects for reuse

Overview of the QuO architecture



Client calls delegate, which triggers contract evaluation and chooses behavior based upon current regions



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

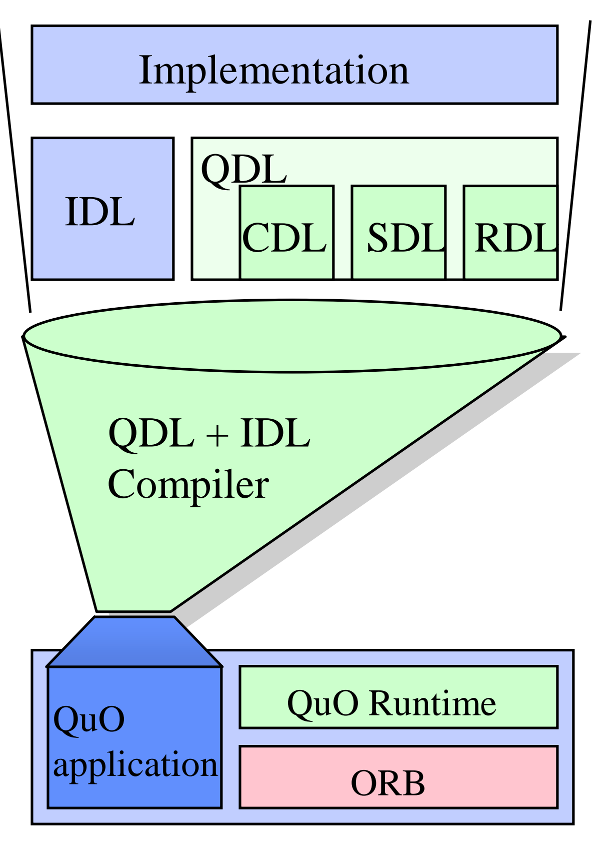
10) Contract is re-evaluated...

11) Return value given to client

Overview of Aspect-Oriented Programming

- A program is decomposed functionally into modules, objects, procedures, etc. as usual
- *Aspects* are programmed separately in special purpose languages
  - Aspects cross-cut the functional components
  - Traditionally are programmed by inserting code in many modules, throughout the system, resulting in a tangled mess of code
  - Examples of aspects are synchronization, instrumentation, data structure, QoS
- Code generators *weave* aspects into the functional modules

QuO's aspect languages, aka Quality Description Languages (QDL)



- Contract Description Language (CDL)
  - expected regions of QoS
  - reality regions of QoS
  - transitions for adapting to changing levels of service
- Structure Description Language (SDL)
  - behavior alternatives for remote objects and their delegates
  - alternate bindings and connection strategies
- Resource Description Language (RDL)
  - available system resources and their status

Contracts summarize system conditions into negotiated and reality regions and define transitions between them

- *Negotiated* regions represent the expected behavior of client and server objects, and *reality* regions represent observed system behaviors
- Predicates using system condition objects determine which regions are valid
- Transitions occur when a region becomes invalid and another becomes valid
- Transitions might trigger adaptation by the client, object, ORB, or system

Normal:
Expected capacity >= 10

- As_expected: Measured capacity >= 10
- Insufficient_resources: Measured capacity < 10

Degraded:
Expected capacity < 10
Expected capacity >= 2

- As_expected: Measured capacity < 10, Measured capacity >= 2
- Extra_resources: Measured capacity >= 10
- Insufficient_resources: Measured capacity < 2

Unusable:
Expected capacity < 2

- As_expected: Measured capacity < 2
- Extra_resources: Measured capacity >= 2

= Expected Region
= Reality Region

Quality Description Languages for specifying operating regions and adaptive behaviors

CORBA IDL

```
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);
};
```

SDL

```
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;
```

CDL

```
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;
```

Code Generators

Delegate

Contract

QuO Runtime

CDL contract to control object replication

```
contract Replication( object client, object server ) is
  var measured  : measured_replication init();
  negotiated regions are
    region Low_Cost :
      when client.expectations.requested == 1 =>
        reality regions are
          region Too_Low  : when measured == 0 =>
          region Normal   : when measured == 1 =>
          region Too_High : when measured > 1 =>
          transitions are
            transition any->Too_Low : client.callbacks.availability_degraded();
            transition any->Normal : client.callbacks.availability_back_to_normal();
          end transitions;
        end reality regions;
    region High_Avail :
      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();
          end transitions;
        end reality regions;
    transitions are
      transition Low_Cost->High_Avail :
        adjust_degree_of_replication(client.expectations.requested);
      transition High_Avail->Low_Cost :
        adjust_degree_of_replication(client.expectations.requested);
    end transitions;
  end negotiated regions;
end Replication;
```

SDL code that supports choosing between replicated and non-replicated server objects

```
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 supports choosing between objects, choosing between methods, run-time binding, exceptions, and embedded Java or C++ code.

Goals

- Continuing development of the QuO framework
  - Multiple contracts, layering of delegates, contracts on the client and the server side
  - More example applications, libraries of system condition objects
  - applying QuO concepts to other frameworks such as Java RMI

Conclusions

- We have developed middleware that supports adaptable software that can measure and control changing system properties
  - Supports software that has critical resource needs and can be deployed in unpredictable WAN environments
  - Instrumentation, triggers and effects can be defined and employed in an application in a structured manner as opposed to ad-hoc exception handling
  - Enables the combination and trade-off of multiple QoS dimensions, e.g., real-time, dependability and security
- This is useful for distributed RT applications
  - Supports applications that can adapt to changing external environments
  - Supports application-level interfacing of resource managers

Further information

- QuO

  http://www.dist-systems.bbn.com/tech/QuO

- To get the QuO Toolkit software, send e-mail to

  quo-help@bbn.com