A Distributed Resource Controller for QoS Applications*

(Appeared in NOMS 2000, Hawaii, April 2000)

Phil Y. Wang
Technology Center
Nortel Networks
pywang@nortelnetworks.com

Y. Yemini and D. Florissi
Computer Science Dept
Columbia University
{yemini,df}@cs.columbia.edu

John Zinky
BBN Technologies
GTE
jzinky@bbn.com

ABSTRACT

The Distributed Resource Controller (DRC) technology described in this paper provides a novel approach to interfacing applications with emerging network mechanisms to deliver Quality of Service (QoS) and controlling network resource utilization. DRC aims to unify network services (e.g., Diffserv, Intserv, and ATM) and application QoS provisioning by introducing a middleware system and a set of generic interfaces. DRC middleware is the core part that translates application requests for QoS delivery into respective access to underlying network systems in a manner that optimizes resource utilization and shelters applications from such a complexity. Applications can request QoS to the DRC middleware in two ways: either in-line using the DRC QoS API or off-line via the DRC utility. Application QoS requirements are specified in terms of two categories of parameters: Traffic Profile (quantitative) and User Expectation (qualitative). DRC currently uses the CORBA-based object-oriented technology to develop a CORBA-based Resource Controller (CRC). The working CRC prototype built to manage the Integrated Services is also presented.

KEYWORDS

QoS, quality of service, network management, resource control, CORBA, integrated service, RSVP, differentiated service, DRC, CRC

1. Introduction

Network services, such as Differentiated Services (Diffserv), Integrated Services (Intserv) and Asynchronous Transfer Mode (ATM), have been designed to offer diverse service levels rather than a single service that is the "Best-Effort" (BE) service offered by the current Internet. These service mechanisms intend to serve the

* This work is supported by the US DARPA under Contract No. F30602-96-C-0315 and done when Phil Wang was a research scientist in Computer Science Dept., Columbia University.

growing diversity of application needs for Quality of Services (QoS) delivery and to supply controlled network resource guarantees. One of the central task in making use of these mechanisms is for applications to be able to use QoS delivery capabilities in a manner that is simple and allocates efficiently the underlying resources among competing needs.

Each of these QoS provisioning services provides QoS using different paradigms. Diffserv and Intserv are proposed by the Internet Engineering Task Force (IETF), and built on the TCP/IP protocol suite. Diffserv is a packet-based priority service [8] deployed in intermediate systems such as routers, and provides Assured and Premium services with differentiated service priorities. Intserv is a flow-based reservation service [9] deployed in both end (e.g., hosts) and intermediate (e.g., routers) systems, and provides Controlled-Load and Guaranteed services to applications like real-time communications. ATM, proposed by the ATM Forum [10], uses a proprietary mechanism and provides Virtual Circuit (VC) services for end-end communications within the ATM network.

Each QoS provisioning service serves applications by providing their own custom user interfaces, which include Application Programming Interfaces (API) and QoS specifications. ATM provides the User Network Interface (UNI) [10], Diffserv uses the Bandwidth Broker (BB, still under development) [8] and its API, and Intserv uses the Resource ReSerVation Protocol (RSVP) [6] and its API (RAPI).

Applications therefore need to cope with the heterogeneity of user interfaces to acquire their QoS. In addition, the QoS requirements vary from application to application. For example, an Internet telephony application requires voice signals arriving within a tolerated delay variance (jitter); a video player requires a bandwidth guarantee to convey the images smoothly; and a real-time monitor requires a strictly assured delay of communication.

How can an application easily interact with the QoS provisioning services? This leads to several important challenges. First, a user may not have exact knowledge of a specific service offered by the network, and may not know how to customize a QoS request for its needs. Second, network QoS may be expressed using different service models and bring about inconsistent resource utilization. Third, legacy applications need to be extended or assisted to access and control QoS delivery. Fourth, if QoS control is only partially available to some network systems, how can it inter-work with systems and applications that are not equipped to handle QoS?

This paper describes the Distributed Resource Controller (DRC), a service architecture to meet these challenges. DRC shelters the diversity of both end applications and underlying provisioning services by placing a middleware between them. Applications use DRC to request QoS in a uniform approach independent of the particular provisioning service, service API, or QoS specification used. In addition, DRC controls applications by cautiously mapping their QoS requirements into underlying provisioning services.

The DRC architecture has three major components. (1) A middleware system, consisting of distributed service managers and resource agents to deliver application QoS requests to and to control application utilization of network services and resources. (2) Generic QoS user interfaces for application QoS requests independent

of specific provisioning services, platforms and programming languages. (3) Application-independent utilities to request QoS on the behalf of applications using off-line commands.

DRC has employed the Common Object Request Broker Architecture (CORBA) technology to build the CORBA-based Resource Controller (CRC) object-oriented distributed system. A prototypical CRC has been implemented to support Intserv/RSVP. It uses RAPI to provisioning application QoS by setting up resource reservations.

The reminder of this paper is organized as follows. Section 2 analyzes current QoS provisioning approaches and related works. Section 3 motivates the goals of DRC. The DRC architecture is described in Section 4, which also describes how DRC assists QoS-demanding applications. Section 5 presents the DRC user interfaces, including the QoS API and the QoS specification. Section 6 presents CRC and its implementation. Finally, Section 7 concludes and presents future directions.

2. Challenges of QoS Provisioning and Related Works

In current proposed provisioning systems, applications can typically acquire QoS by directly interacting with the APIs of QoS provisioning services. An application must firstly determine its service quality, and select an available service (e.g., Diffserv, Intserv, or ATM). Then, it invokes the QoS API to map its quality requirements onto the QoS template of that service, and sends a request to a corresponding resource provider. After the resource provider confirms this request and allocates the demanded resource (e.g., a link bandwidth or a packet service priority), the application can hereafter be assured its requested QoS.

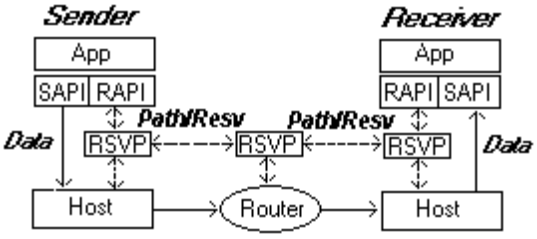

Figure 1: A QoS application with Intserv/RSVP

Figure 1 shows a simple example of how an application interacts with the Intserv/RSVP service to request QoS for its communication. This application consists of two programs, the *Sender* and the *Receiver*, each employing two types of APIs, SAPI and RAPI. SAPI stands for the Socket API and is used for the data communication that is a stream from *Sender* to *Receiver*; RAPI is the RSVP API, and is used to invoke RSVP routines for acquiring QoS (by setting up a resource reservation). Using RAPI, both *Sender* and *Receiver* communicate individually with their local RSVP daemons, and set up two RSVP sessions to the same destination

(i.e., the receiver host). Then, *Sender* passes the traffic requirements to its RSVP daemon, which creates and sends a PATH event signaling end hosts and intermediate routers along the routing path. After receiving the PATH event from the daemon at its side, *Receiver* generates the flow parameters for the reservation, and makes a reservation request to the daemon that signals an RESV event along the reverse path towards *Sender*. All end hosts and intermediate routers intercepting both PATH and RESVR events reserve their resources for the specified communication.

The interactions of applications with Diffserv and ATM are similar to the above Intserv example. Diffserv reconstructs routing service policies (e.g., packet forwarding priority and traffic conditioning) of the involved edge routers, while ATM sets up one or more VC links inside the ATM network. All of these services try to deliver QoS by conferring the application stream with appropriate amounts of resources.

As described in Section 1, there are important challenges for application development and network management in the current state of affairs. The following additional challenges further complicate the situation. (1) Applications become service-specific since they include code that is dependent on the APIs used. An application that includes the RAPI can only request QoS to Intserv, and cannot request directly another services like Diffserv or ATM. (2) Without an overall QoS control and management, an application does not have a global view of underlying service and resource consumption, which is crucial to enable an adaptive QoS request. (3) Legacy QoS interfaces or APIs may be available for a few platforms and programming languages. For example, RAPI is currently only available for C/C++ and under development for Java and other languages.

So far research efforts in the field do not provide a complete solution to encompass all these challenges. RFC 2382 [16] has proposed a framework for applying the Internet Intserv model to the ATM service model. It is based on the "Classical IP over ATM" technology [15] and has two key components. The first is the QoS translation, mapping the Intserv QoS into the corresponding ATM QoS. The second is the RSVP VC management, establishing a reservation through individual VCs or combined reservations through "aggregate" VCs. This work enables Intserv-based applications to use the ATM service.

Enabling Intserv applications to use Diffserv is also studied in [7]. It proposes a framework to support the delivery of end-end QoS by applying the Intserv model end-end across the Diffserv networks.

The Internet2 Bandwidth Broker (BB) Advisory Council (BBAC) [20] employs the BB to manage network resources for IP QoS services. It aims to automate the admission control decisions and network device configuration functionality in the context of the Internet QBone architecture [19]. A BB can be a type of policy server and manages the QoS resources within a given domain based on the Service Level Descriptions. BB also gathers and monitors the state of QoS resources within and at the edges of its domain. Two signaling mechanisms are proposed to set up the inter-domain communication for resource allocation. BB will support the BE and Diffserv Premium services. Its API allows applications to make BB service requests and to query for existing service commitments.

QoS negotiation based on the ATM service has been investigated in "the QoS Broker" [17]. The broker manages resources at the end-points and coordinates resource management across multi-layer boundaries. It provides services such as translation, admission, and negotiation. Configuration of the underlying system to application needs is achieved by QoS negotiation, resulting in one or more communication connections.

WinSock2 [21] is the version 2 of the Windows Sockets (WinSock) programming interfaces and provides QoS extension introducing a generic QoS interface. The interface aims to be independent of the underlying QoS provider (e.g. ATM and RSVP) by encapsulating respectively the existing QoS API such as ATM UNI and RAPI.

A QoS API proposal presented in [18] consists of five functions such as QoS binding, authentication and unbinding. The API is generic and hides the details of existing QoS API (e.g., RAPI). Application QoS is manually specified by knowledgeable experts and stored in service-dependent profiles.

Effective QoS performance of existing applications has been examined using QoSockets [2]. QoSockets extends Berkeley sockets with QoS and delivers end-to-end QoS (using renovated API) and monitors real-time network performances for QoS management (using SNMP-based QoS MIBs). Applications are tested with and without Intserv/RSVP reservations to compare their performances under normal, heavy, and overloaded traffic conditions.

3. Objective

The goal of DRC is to provide a novel and comprehensive architecture to meet the challenges of QoS provisioning. That is, DRC enables applications to obtain appropriate network service guarantees using uniform interfaces, independent of the underlying services. The DRC architecture offers QoS-based resource control in a distributed model, which is portable to different platforms, network services, programming languages and applications. It introduces three components, each facing different challenges set forth in the previous sections.

3.1 Middleware

The DRC middleware (introduced in Section 4) mediates between QoS demanding applications and resource provisioning systems. In simple words, it dispatches the application requests to the underlying systems, and returns status and feedback of the underlying systems to the applications.

Examining the application request and the available network resource, the middleware selects a provisioning service or service level, maps the application QoS to the service-specific QoS, and initiates resource allocation or service reconfiguration. In the mean time, DRC controls applications to utilize well network resources by reasonably mapping the application requests to available network resources.

DRC middleware uses the service managers, which do not rely on any platform and application, to accept application QoS requests. It also uses the resource agents,

which communicate with underlying provisioning mechanisms and enable the whole middleware system to be independent of any network service.

3.2 User Interfaces

The DRC user interfaces (introduced in Section 5) contain the generic QoS API and the QoS specification mechanisms and the QoS API are independent of any network service, Operating System (OS) platforms, or programming languages. Applications use these interfaces to communicate with the DRC middleware, and define their requirements in terms of DRC QoS specification. Then the DRC middleware does the mapping from DRC generic QoS to service-specific QoS.

3.3 Utilities

System administrators and QoS management systems use DRC utilities to perform two tasks. The first is to make off-line QoS requests on behalf of the applications that cannot request their own QoS. The second is to query and display DRC status and network information for QoS management.

DRC utilities (introduced in Section 4) are developed using the DRC user interfaces, and communicate with the DRC middleware.

4. DRC Architecture

DRC has a tiered architecture with three layers: *User*, *Middleware*, and *Resource*. Figure 2 depicts the basic DRC architecture with two network domains in the Internet frame.

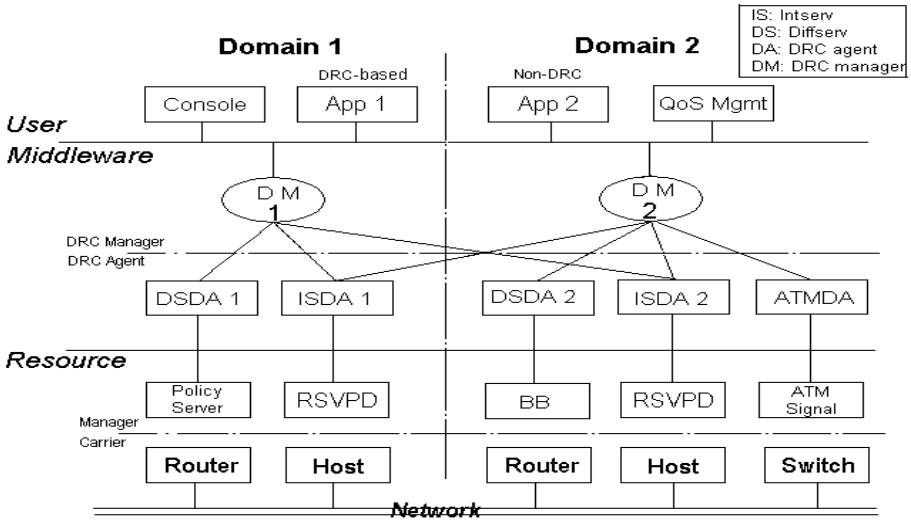

Figure 2: DRC Architecture

4.1 Three basic Layers

The *User layer* is composed of QoS demanding applications and DRC user-end utilities. These applications are divided into two types: DRC-based *(App1)* and non-DRC-based *(App2)*. *Console* is a DRC utility for off-line application QoS commands, and *QoS Mgmt* stands for the external QoS management system providing QoS control and management. Both *App1* and *Console* use the DRC QoS API to send QoS requests to the *Middleware layer*. *App1* requests QoS by itself while *Console* requests it on the behalf of applications like *App2*.

The *Middleware layer* is the DRC core and bridges applications (at the *User layer*) and underlying resource systems (at the *Resource Layer)*, consisting of DRC service managers (DM) and DRC resource agents (DA). DM is distributed in a network domain, accepts application QoS requests in its own or distant domains. It selects the provisioning services, generates the QoS requirements for specific services, and dispatches them to related DAs.

DA maps the QoS requirements to the service-specific QoS specification, and then interacts with a resource manager to make resource allocations or change routing policies. DA also monitors the utilization of network resources and services. Generally, each provisioning service has a particular DA. For example, *DSDA*, *ISDA*, and ATMDA are the DRC agents for Diffserv, Intserv, and ATM respectively.

The *Resource layer* includes underlying network resources and services. Hosts and routers are the resource carriers. Policy Server (*PS*) and Bandwidth Broker (*BB*) are the resource managers for Diffserv, while RSVP Daemon (*RSVPD*) is the one for Intserv.

4.2 How does DRC work?

DRC delivers QoS requests for both DRC-based (*App1)* and non-DRC-based (*App2*) applications, of which *App1* includes the DRC API while *App2* does not. If *App2* includes some QoS API (e.g. RAPI), it may request QoS by interacting with the underlying service (Intserv), bypassing DRC at the price of platform- and service-specific complexity. However, DRC can still assist *App2* in case *App2* does not include the QoS API for current available service or any QoS API at all.

App1 and *App2* have to specify their QoS using the DRC generic parameters (detailed in next section). *App1* invokes the DRC API, and directs a request to DM. *App*2, which does not invoke the DRC API, uses the DRC utility instead. The *App2* QoS requirements can be input (e.g., by a system administrator or using a script) to *Console*, which then make a QoS request to DM. Similarly to *App1*, *Console* also invokes the DRC API.

Suppose that *DM1* accepts the QoS request from either *App1* or *App2*. *DM1* determines the related DAs according to the traffic sources and/or destinations, and contacts these agents for available network services and resources. If the request specifies a network service, *DM1* queries DAs for the service and resource availability during the service period. If the request does not specify a service, *DM1* matches an available service close to the QoS requirements. Assume that the Intserv/RSVP is selected. Then, if both network service and resource are available,

DM1 converts the DRC QoS into the Intserv/RSVP QoS specification and initiates the *ISDA*s at both sender and receiver ends to make a resource reservation.

DA serves the DM request, and maps it to the QoS template of the network service. DA then communicates with a resource manager for resource allocation by invoking the service-specific API. For Intserv, the *ISDA* at the sender end interacts firstly with the local resource manager (*RSVPD)* by invoking RAPI to send the traffic description. After getting the event, the *ISDA* at the receiver end interacts with its *RSVPD* to make a resource reservation. See the example in Section 2.

RSVPD, the resource reservation signaling process, works as the Intserv resource manager that checks whether the service is provided and the requested resource is available from the resource carrier. Then, it admits the resource reservation and signals the RSVP events. Resource carriers (routers) intercepting the events allocate the requested resource, and commit the QoS using traffic control.

At this stage, the final result of the resource allocation is returned from *RSVPD* to *App 1* or *Console* layer by layer. If the QoS request is confirmed, *App1* (or *App2*) is ready for its communication. If it is rejected, *App1* (or *App2*) may request *DM1* to provide information about available network services and resources, and then adjust their QoS demands and restart the reservation procedure outlined before.

Acquiring QoS through a DRC utility (*Console*) is a convenient way for some applications like *App*2. These applications want QoS, but do not include QoS API or cannot obtain QoS support from its network system. For example, legacy applications employing outdated technology cannot be extended to adopt QoS APIs, some network systems do not support QoS, and Diffserv-based applications cannot use the Intserv.

5. User Interfaces

DRC user interfaces include DRC QoS Specification and DRC QoS API. Both are generic so that an application demanding QoS is isolated from the complexities of provisioning services, OS platforms, and programming languages.

5.1 QoS Specification

DRC defines its generic QoS specifications in terms of two categories of parameters. The first category, "*Traffic Profile"*, describes quantitatively the QoS requirements. The other, "*User Expectation"*, represents qualitatively how the provisioning system should guarantee the QoS (if there are multiple services available).

- **Traffic Profile**

A traffic stream or session is specified uniquely by a *Traffic Profile* (TProfile), which is composed of two types of parameters.

The first type of parameters characterizes a traffic stream or session, and includes *source address* (e.g., an IP address and a port number), *destination address*, *transport protocol* (e.g., TCP or UDP), and auxiliary parameters. They are common for both QoS-demanding and non-QoS-demanding communications.

The second type represents the network performance requirements. These requirement parameters define quantitatively QoS of the traffic stream or session. They are *throughput* (transmitting rates and message size), *latency* (message delay), *jitter* (delay variance), and *reliability* (message loss and connection fail); and can be specified using three types of values: maximal, minimal, and average.

- **User Expectation**

Guaranteeing the requirements of a *TProfile* depends on the semantics of the resource provisioning mechanism used. To better map the application QoS and select a proper provisioning mechanism, the *User Expectation* (UExpect) is used by a user to describe his/her preferences and expectation of an application QoS, using more subjective concepts than TProfile.

UExpect uses the following parameters: *service type* (e.g., Intserv or Intserv/Controlled-Load), *service period*, *service priority*, *traffic conditioning* (shaping traffic or not), *authentication*, and *payment*. *UExpect* may include some extra information from a service contract, e.g., the Service Level Agreement (SLA) in Diffserv.

DRC uses *TProfile* to define an application QoS, and use *UExpect* to determine how to map this application QoS to a specific provisioning mechanism. From the user's prospective, *TProfile* parameters must be provided, but *UExpect* parameters are secondary and some of them are optional. A good *UExpect* helps DRC in allocating QoS, reducing the possibility of rejection, and delivering fewer QoS violations and disruptions.

5.2 QoS API

DRC-based applications invoke the DRC QoS API to make in-line QoS requests to the DRC middleware. The API set includes five key types of abstractions.

- **Open and close a DRC connection**

Drc_Handler ***open***(Drc_Name): create a new DRC handler or open an old one denoted by Drc_Name.

Drc_Status ***close***(Drc_Handler): close an existing DRC handler.

- **Resource query**

Drc_Info ***query***(Drc_Src, Drc_Dst): query available network service and resource. Drc_Src and Src_Dst are the source and destination of a given stream.

- **QoS specification**

Drc_Status ***specify***(Drc_Handler, Drc_QSpec): assign or modify the QoS specification of a DRC handler.

- **QoS request**

Drc_Status ***request***(Drc_Handler): commit or re-commit the QoS request of a DRC handler.

- **Error and status**

Drc_Status ***status***(Drc_Handler): check the status of a DRC handler.

Drc_Error ***error***(Drc_Handler): handle an error encountered by a DRC handler.

In the interest of being independent on any programming language, the whole DRC API is defined using the CORBA Interface Definition Language (IDL) [1] and encapsulated as CORBA objects. Applications can use the DRC API as long as CORBA is available on their system environments. This is the DRC in-line mode.

In contrast, in the DRC off-line mode, an application that does not use the above DRC API can make QoS requests to the DRC middleware with the help of the DRC utilities.

6. CRC: A CORBA-based implementation

Common Object Request Broker Architecture (CORBA) [1] is increasingly accepted as an object-oriented distributed technology interoperable among heterogeneous hardware and software platforms. The CORBA-based Resource Controller (CRC) that is depicted in Figure 3 is the DRC system that uses CORBA to gain the platform portability.

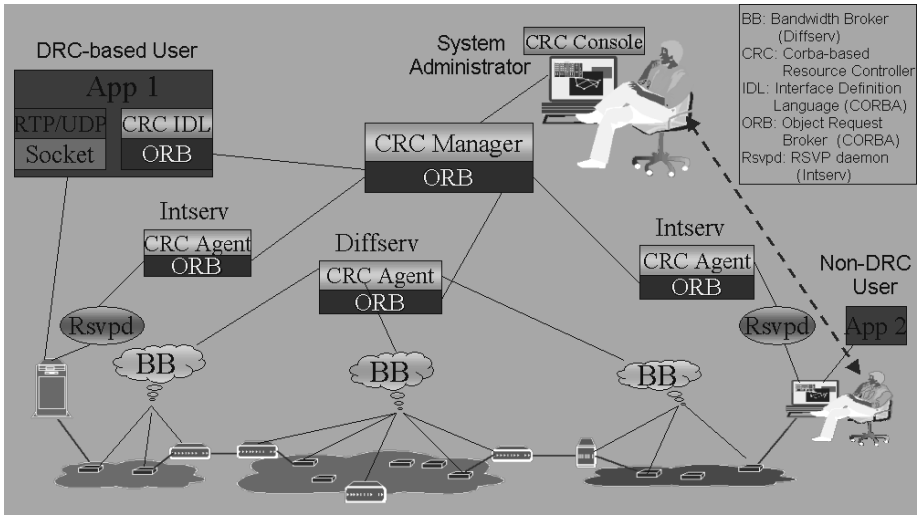

Figure 3: CORBA-based DRC scenario (CRC)

The CRC prototype system is implemented to bridge Intserv and QoS-demanding applications. This system also provides IDL interfaces, and has been used to make RSVP/Inserv resource reservations..

6.1 Prototype system

Figure 4 shows a CRC prototype implemented with a simplified structure, where Intserv is the only QoS provisioning service. CRC ORB servers are placed at the *Middleware layer*, and at the *User layer* are the CRC ORB clients (i.e., applications). Between these two layers is the ORB communication. CRC middleware interfaces with the *Underlying layer* through the RSVP API (RAPI).

Reservation Manager (*RMgr*) and Resource Agent (*RAgt*) are the ORB servers, and *RMgr* is the DM while *RAgt* is the DA (i.e., ISDA). *RMgr* converts application requests to QoS specifications of Intserv, and dispatches them to *RAgt* at the sender and receiver ends of applications. *RAgt* maps the specifications to Intserv QoS templates, and communicates then with the local *rsvpd* for resource reservations. Inside the middleware, *RMgr* is the ORB client of *RAgt*.

Console, *App 1*, and *N* are ORB clients, and send QoS requests to *RMgr* using CRC object interfaces. *App 1* and *N* are CRC-based applications requesting QoS by themselves while *Console* is the CRC utility, specialized to request QoS for other applications.

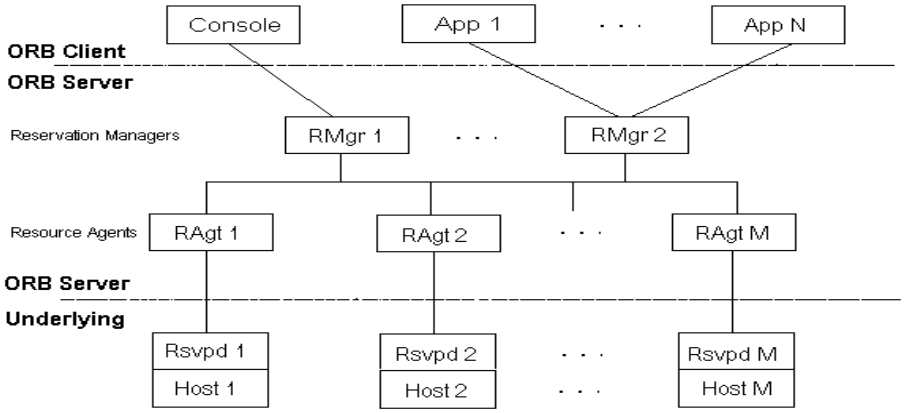

Figure 4: CRC prototype system

For a unicast application, a traffic flow has one destination (receiver) and one or more senders, and CRC uses one *RMgr* and multiple *RAgts* (one for the receiver and one for each sender). For multicast, a flow involves more than one send and receiver, so CRC uses one *RMgr* and multiple *RAgts* for every sender and receiver.

6.2 Interfaces

RMgr and *RAgt* are designed as multi-objects to avoid blocking the respective ORB servant when processing multiple requests. Both use two CORBA interfaces (implemented as C++ classes): *Factory* and *Object.*

Applications use the *RMgr* object interfaces written in IDL (see below) to communicate with CRC middleware. Rfct (*RMgr Factory*) and Rmgr (*RMgr Object*) are key interfaces encapsulated in the module RMM. Factory *Rfct* is invoked by an application to duplicate an instance of the Object *Rmgr*. The meanings of the operations, types, and declarations are included in the comments.

```
// @(#) Rmgr.idl 1.8 99/07/08 Copyright (C) Columbia University
// Corba-based Resource Controller (CRC)
// Version: 1.0

module RMM  // Reservation Manager Module
{
 enum STATUS
 {
  UNKNOWN,   // No such flow?
  INITOK,    // The flow initialization is OK
  …
  RESVOK,    // The reservation is made
 };

 enum ERROR
 {
  NOFLOW,    // No such a flow
  …
  RSVPRESVERR    // Caused by a RAPI_RESV_ERROR event from rsvpd
 };

 interface Rmgr // an Rmanager Object
 {
  STATUS specify(…); // assign or modify the QoS specification
  STATUS request(…); // commit or update QoS request
  STATUS status(…); // return the status of the flow
  ERROR  error(…);  // return the error of the flow
 };

 interface Rfct  // RManager factory
 {
  long init(…); // init a flow, return the flow ID
  Rmgr create(…); // create an Rmgr object
  STATUS stop(…);   // stop an Rmgr service
 };

};
```

Figure 5: CRC IDL interfaces

6.3 Results

CRC 1.0 has been developed for Linux 2.0 and 2.2, using the TAO real-time CORBA ORB [4] and the Linux port of the RSVP package [14]. It is written in C++, and available at the URL http://www.cs.columbia.edu/dcc/crc.

The CRC Console, an off-line application reservation tool, has been successfully used to make the Intserv/RSVP reservations for applications that do not use RAPI or CRC API. These applications can use their own programming languages (e.g., C, C++, Perl and Java), and run on their own platforms (e.g., Solaris, Linux).

Applications may also invoke the CRC API (i.e., IDL interfaces in Figure 5) to make Intserv reservations, using CORBA-based programming. CRC has been used for the bandwidth management of QuO 2.0 [5] and for applications that are coded in Java and C/C++.

7. Concluding Remarks

This paper has presented the DRC distributed resource control architecture. The implemented CRC prototype system shows that the DRC architecture is feasible for applications requesting QoS either in-line by invoking the DRC interfaces or off-line by using the DRC utilities. Moreover, in addition to its generic interfaces (QoS specifications and APIs), DRC is independent to any platform, provisioning service, programming language or application.

There are still many potential challenges in the DRC architecture. The first challenge is to incorporate Diffserv in CRC, which implies implementing the DSDA. Diffserv is a sender-initiated service that offers service levels at different granularities. Similarly to the ISDA for Intserv, the DSDA communicates with a

policy server (PS) or bandwidth broker (BB) for Diffserv admission and resource allocation. CRC will use fewer DSDAs since they are only required at the sender end, which can determine QoS without participation of the receiver end. Moreover, Diffserv comes usually with an SLA that includes the detailed service contract, which can be used by CRC.

Another future challenge for DRC is to use adaptive QoS support, which will be handled by the DM. Most applications can adapt to dynamic oscillations of QoS under light and heavy network conditions. DRC service mangers (DM) may downgrade or upgrade their QoS based on the real network conditions. Ordinary applications are incapable of performing the adaptation since they do not have a dynamic overall view of network resource consumption, but DMs do. As a result, by adapting the application QoS needs, an application may be able to get better-than-requested QoS when networks are idle or lower QoS violations when networks are stressed. In this respect, DM performs more intelligent allocation decisions than a Bandwidth Broker (BB) does.

Finally, DRC can be extended to work with policy-based services, in which packet routing, traffic shaping, and network configuration and management are under controls of policy servers. Some research work has been pursued on policy-based QoS, such as COPS [12] and RAP [13]. In fact, a policy server acts just like a resource manager at the *Underlying layer* of the DRC architecture, so one could apply DRC to such service by simply introducing new DRC agents and managers.

References

[1] Object Management Group, "The Common Object Request Broker: Architecture and Specification", Rev. 2.2, Feb. 1998

[2] Wang, P.Y., Yemini, Y., Florissi, D., Zinky, J. and Florissi P., "Experimental QoS Performances of Multimedia Applications", submitted in publication, Infocom 2000, March 2000

[3] Yemini, Y., Konstantinou, A. and Florissi, D., "NESTOR: An Architecture for Network Self-Management and Organziation", submitted in publication, JSAC, Nov. 1999

[4] Schmidt, D.C., Levine, D.L. and Mungee, S., "The Design and Performance of Real-Time Object Request Brokers", Computer Communications, V21, Apr. 1998

[5] Zinky, J., Bakken, D. and Schantz, R., "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems, 1997

[6] Zhang, L., Berson, S., Herzog, S. and Jamin, S., "Resource ReSerVation Protocol (RSVP) – Version 1 Function Specification", Internet RFC 2205, 1997

[7] Bernet, Y., Yavatkar, R., Ford, P., Baker, F., Zhang, L., Speer, M., Braden, R., Davie, B., Wroclawsky, J. and Felstaine, E., "A Framework for Integrated Services Operation over Diffserv Networks", draft-ietf-issll-diffserv-rsvp-03.txt, Sept. 1999

[8] Blake, S., Black D., Carlson, M. Davies, E., Wang, Z. and Weiss, W., "An Architecture for Differentiated Services", Internet RFC 2475, Dec. 1998

[9] Braden, R., Clark, D. and Shenker, S., “Integrated Services in the Internet Architecture: Overview”, Internet RFC 1633, June 1994
[10] ATM Forum, “ATM User-Network Interface Specification”, Version 3.1, 1994
[11] Nichols, K., Blake, S., Baker, F. and Black, D., “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, RFC2475, Dec. 1998
[12] Boyle, J., Cohen, R., Durham, D., Herzog, S., Raja, R. and Sastry, A., “The COPS (Common Open Policy Service) Protocol”, Internet Draft, draft-ietf-rap-cops-05.txt, Jan. 1999
[13] Yavatkar, R., “A Framework for Policy-based Admission Control”, draft-ietf-rap-framwork-01.txt, Nov. 1998
[14] Wang, P., “Linux Port Of RSVP r4.2a3”, http://www.cs.columbia.edu/~yhwang/ftp/qos/rsvftwo-p, Aug. 1998
[15] Laubach, M., “Classical IP and ARP over ATM”, RFC 2225, April 1998
[16] Crawley, E., Berger, L., Berson, S., Baker, F., Borden, M. and Krawczy, J., “A Framework for Integrated Service and RSVP over ATM”, RFC 2382, Aug. 1998
[17] Nahrstedt, K. and Smith, J.M., “The QoS Broker”, IEEE Multimedia, V2, N1, 1995
[18] Riddle, B. and Adamson, A., “A Quality of Service API Proposal”, http://apps.internet2.edu/qosapi.htm, Aug. 1998
[19] Tietelbaum, B., ed., “Draft Qbone Architecture”, Internet2 QoS WG Draft, June 1999
[20] Neilson, R., Wheeler, J., Reichmeyer, F., and Hares, S., “A Discussion of Bandwidth Broker Requirements for Interneet2 Qbone Deployment”, Version 0.6, Interent2 Qbone BB Advisory Council, March 1999
[21] The WinSock Group, “Windows Sockets 2 Protocol Specific-Annex”, http://apps.internet2.edu/winsock2.htm, May 1996