The design and implementation of the A2QM3 System

Balázs Csizmazia
University Klagenfurt, Austria
csb@itec.uni-klu.ac.at

Hermann Hellwagner
University Klagenfurt, Austria
hellwagn@itec.uni-klu.ac.at

Abstract

In this paper we present the design, architecture and implementation of the A2QM3 System. It provides programmers re-usable QoS-aware Control Objects to enable building a complete middleware for adaptive applications over active networks. We introduce the programming model, the system architecture, and show the parts that make this system a full-featured middleware supporting QoS-aware reliable stream-oriented communication, communication using the request/reply-based CORBA model and real-time streaming for continuous multimedia contents.

1. Introduction

This paper presents the design and implementation of the A2QM3 (Active Adaptive QoS-aware Multimedia Middleware) System developed at the University Klagenfurt.

A2QM3 supports adaptive transfer of discrete and continuous multimedia data allowing the system designer to build QoS-based multimedia protocol frameworks from pre-defined and user-defined QoS-aware service objects. The whole system is supported by the ANTS active network system described in [13].

The system is destined to eventually support quality-adaptive MPEG-4 video transport over networks where active network nodes (routers and proxies) change video flows during transport in that they perform media scaling (and thus, quality variation) operations when required. Examples of such cases are: fluctuating network QoS, in particular congestion situations; delivery of video flows along heterogeneous network paths or to end systems with different playout capabilities; and multicast transfers. The network is best positioned to quickly and gracefully react to such situations by adapting the transported media data or giving hints to the application to adapt the streamed media data.

The architectural elements of the system were first presented at the DMMOS'01 Workshop held together with ECOOP 2001 in Budapest [4]. The lessons learned there led us to the current state of the system. Although the system itself is supported by an active network based infrastructure, the system can be easily adapted to any datagram-based network, like IP. The first version of the middleware – it didn't support any QoS-aware features – was developed for use with UDP/IP because of its simplicity. Afterwards it was moved to the ANTS system to allow a broader range of experiments with it.

Our current tests are running on the institute's parallel laboratory consisting of 6 Pentium II clones with 450 MHz CPUs connected by a switched 100 Mbps Ethernet LAN, with a simple media server and 3-4 clients. Our former experiments concentrated on implementing reliable multicast protocols to efficiently transfer multimedia data; the results were promising but the protocol used could be tested only on our local network testbed. Experiences in [7] showed that the chosen approach of using delayed acknowledgments with retransmission does not scale well even if using some performance-improvement attempts when the underlying multicast group management does not scale efficiently. Active networks enable us to replace parts of protocols that were designed based on the end-to-end approach of systems design by new parts based on the hop-by-hop approach (quite rare in the IP world). This problem is still a panacea for researchers to find a proper, well-scalable solution[1].

With the A2QM3 System, we are building an active network based infrastructure with pluggable, serializable and portable Java-based protocol objects supported by a scalable host and network monitoring layer. In this way, we extend the best-effort services provided by the ANTS active network system to provide QoS guarantees for adaptive multimedia streaming. Our work focuses on multimedia streaming, the protocol framework is general enough to support a broader range of middleware systems (like systems based on the request-reply paradigm, e.g. CORBA, or transactional middleware systems by treating fault-tolerance just as another QoS dimension). There are many research

[1]The advantage is that multimedia data does not need reliable multicasting when using the RTP protocol for data transfer. Like on a UDP/IP basis, RTP does not need a reliable multicast as a basis. Reliability is only necessary for RTCP, but not in a multicast fashion.

groups working on active network based middleware. By our approach, using the pluggable protocol framework, we provide a solid foundation for integrating a broader range of middleware systems, existing already over traditional best-effort "passive" networks, into active network based software infrastructures. Even more, we allow applications to use the best of both network phylisophies; we are not just porting existing middleware systems onto active networks.

The remainder of the paper is organized as follows. Section 2 introduces the key components of the A2QM3 system. Section 3 gives a brief overview of the implementation details and the use of the system. Section 4 describes the way we support QoS in the architecture both at the network level and the host level. Section 5 introduces the proposed internal architecture of the network monitor layer, responsible for the QoS awareness of the system. Section 6 presents related research in this field. Section 7 and 8 discuss how we plan to make A2QM3 to be a QoS-aware universal middleware over best-effort networks.

2. Key components of the system

Our solution is a protocol composition framework where QoS management is implemented in a QoS support layer. Atop this layer, protocol and control objects can use its services to achieve the data transfer quality they need. In the system, programs can re-use predefined (system-defined) service components and programmers can extend the system with new service objects they need. Thus, protocol stacks can be built from the defined service components, like network protocol stacks at lower layers.

The key components of the system are the ANTS layer [13], an object-based active network infrastructure that sits below the layer supporting various QoS policies, the network load monitoring layer (NLML). This layer provides the necessary information and control mechanisms with respect to QoS for the protocol and service components of the upper layers (we will show more details about it in section 5). Other important components of the system are the active routers, involved in multicast data delivery, media stream filtering, or caching. The components are connected by well-defined interfaces (each multimedia device has to adhere to these interfaces), with standardized up- and downstream interfaces between the components and additional extra interfaces between the QoS-aware protocol components and the NLM layer. Besides modularity, the design builds on the portability of Java as the implementation language. Today's Java compiling techniques (like just-in-time compiling) provide the necessary performance for these applications.

Figure 1 illustrates a possible protocol stack for transmitting MPEG-4 multimedia streams. The protocol stack configuration presented there is just an example, how a multi-

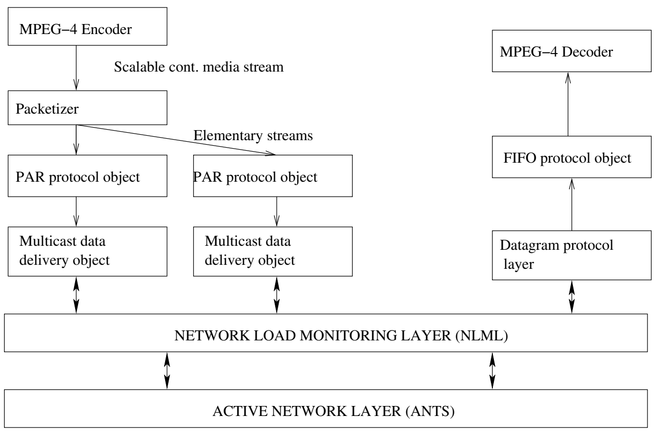
MPEG−4 Encoder
Scalable cont. media stream
Packetizer
Elementary streams
PAR protocol object
PAR protocol object
Multicast data delivery object
Multicast data delivery object
MPEG−4 Decoder
FIFO protocol object
Datagram protocol layer
NETWORK LOAD MONITORING LAYER (NLML)
ACTIVE NETWORK LAYER (ANTS)

Figure 1. Proposed protocol architecture

media stream generated by an MPEG-4 codec can be transferred over the network without losing packets in the network. The multimedia stream generated by the MPEG-4 encoder is packetized and sent to the client. The reliability of this transfer is achieved by the use of positive acknowledgment with retransmission (PAR) of lost packets. A similar configuration and its implementation in Java is shown in section 3. The figure is for illustrational purposes only with easy to understand concepts behind it. It does not mean that we intend to stream multimedia data with positive acknowledgment protocols. First, it is unnecessary because the decoders can tolerate packet losses. Second, it is not scalable at all.

The NLML itself is only partially implemented: global information such as network load saturation and congestion status in remote regions is not available in the current implementation. The interfaces are defined by which applications can receive information about the local network node like CPU utilization, total and available network throughput, total and available memory. The global information – eventually an ANTS-based service – collects and distributes information acquired locally. To do so, it is necessary to maintain a connectivity graph of the hosts participating in the active network environment. To make the system scalable we also need an intermediate grouping mechanism allowing regions of nodes to be treated as a single unit with respect to network and host quality parameters (the quality parameters of the regions may be characterized by calculating values from the region's local hosts' performance parameters). Active networks can be used to collect and carry local information to the distributed information service for further processing and make available for clients in a well-defined manner (for example in a CORBA Trader built upon an ORB equipped with the ability to communicate over ANTS[2]).

[2]While it is not implemented, a simple piece of software is enough

Finally, we do not want to re-invent the wheel for manipulating multimedia streams. At the client and server nodes we use the Java Multimedia Framework [5] which provides a set of easy-to-use APIs for manipulating and streaming multimedia. The JMF RTP API allows us to work with real-time streaming. Aside the original goals of RTCP – a part of the RTP protocol suite [12] – RTCP packets are good candidates to collect multimedia-specific real-time information from the packets transmitted in ANTS capsules with a specific ANTS protocol handler code; see section 5 about monitoring information we intend to use.

3. Implementation of the components

All components described here are implemented in Java.

The heart of the protocol stack implementation is the Protocol Object. Every protocol object has a configuration information: references to the protocol objects atop and below it and a reference to the containing protocol stack instance. Additionally, there are push methods available for the protocol objects atop/below it to call when a protocol data unit arrives. Also an event-monitoring method is available to get the necessary information from the NLML (the NLML can also be actively monitored, a callback interface is provided for convenience). This is a very simplistic approach to define a protocol object: a protocol object (for example one on the server side providing Positive Acknowledged delivery of packets) contains a queue of – out of order – arrived but not yet delivered packets (a window) and a thread of control allowing asynchronous processing of protocol data arrived.

Programmers rarely instantiate a protocol object. Instead they instantiate protocols stacks. A Protocol Stack is another central abstraction: it allows the programmer to specify protocol instances by means of their names and required QoS-parameters.

A Positive Acknowledgment object contains a cache of packets sent on the server side. It stores packets until they are acknowledged by the client side PAR (positive acknowledgment with retransmission) object (an own thread of control is used to maintain a retransmission timer). A FIFO object is used to give each packet a unique packet number enabling an ordered delivery of packets at the receiver side.

We emphasize that the most important part of the system with respect to QoS is the network load monitor. All of the above mentioned components can be implemented – without QoS support – both over IP and ANTS datagrams.

At the lowest layer of a stack is a protocol object interfacing with the network. Now we support UDP/IP and ANTS as a datagram layer, the bottommost object in a protocol stack.

The following code fragment illustrates the use of the concepts in practice:

```java
import java.util.Hashtable;
import java.io.Serializable;
import a2qm3.*;

public class SimpleDemo {

 public static void main (String[] args)
 {
  ProtocolStack s = new Stack
    ("FRAGMENT(size=512):FIFO:"
    +"PAR(buffer=64):"
    +"UNRELIABLE(reliability=30):"
    +"ANTSDATAGRAM(capsuleclass="
    +"ants.Capsule)");
  PARReceiver r = new PARReceiver ();
  // attach to default interface:
  s.registerReceiver (r);
  // attach to NLML:
  s.registerQoSListener (r);

  while (true) {
    try {
      Message m;
      ByteArrayOutputStream barr=
            new ByteArrayOutputStream(200);
      DataOutputStream dos=
            new DataOutputStream(barr);
      // write some user data
      dos.writeByte(c);
      dos.writeLong(position);
      dos.writeLong(filesize_initially);
      dos.write(file_data);
      dos.flush();
      barr.flush();
      m.setContent(barr.toByteArray());
      m.send(s);
    } catch (Exception e) {
      System.err.println ("Send failed:"
                          +e.toString());
    }
  } // end while
 } // end main
} // end class

class PARReceiver
       implements MessageReceiver,
                  QoSListener
{
 public void processMessage (Message m)
```

because of the fact that ANTS is entirely written in Java and Java is IIOP-enabled: the collected information can be stored in a CORBA Trader [8] using IIOP. This enables a quick integration to the Adaptive Multimedia Server's supporting infrastructure developed at the University Klagenfurt. The Adaptive Multimedia Server needs to know where it is worth to allocate new network nodes in the system to make the streaming as efficient as possible. The communication with the servers's supporting mobile agent is done via a CORBA trader.

```
    // from MessageReceiver
  {
   // Processing of received
   // packet goes here
   System.err.println ("Got message from "
                       +p.getSender () );
  }

 public void qosViolation(QoSEvent e)
   // from QoSListener
  {
   // QoS-violation processing
   // code goes here
  }
}
```

The above code-fragment instantiates a protocol stack that fragments the stream sent through it with a maximum payload fragment size of 512 bytes. Below this, there are the FIFO and PAR protocol objects. The UNRELIABLE protocol object simulates a lossy network: the given percentage of packets are lost or duplicated so that we can test how our protocol behaves over lossy networks.

Afterwards, we instantiate a `PARReceiver` object and register it with the local NLML for packet listening and listening to QoS-related events. The `PARReceiver` class implements the `MessageReceiver` and `QoSListener` interfaces for receiving packets on the registered protocol stacks and receiving QoS notification events from the NLML layer. The `MessageReceiver` interface defines the `processMessage()` method. This method will be called on the receipt of user data giving the user data as a parameter. The `QoSListener` interface defines the `qosViolation()` method. Applications register this method so that the NLML can notify them about violating the QoS contract between the application and the NLML. It is up to the application how it deals with these events.

Sending data can be done simply by the message's `send()` method giving it the reference to the stack we want to use for sending data.

4. QoS support by the protocol architecture

The protocol stacks - they are protocol objects, too, so they allow building of nested protocol objects - forward messages upwards or downwards on client or server nodes. Messages are extended by a message ID that identifies the type of the message (e.g. user data, high priority user data, acknowledgment messages or subscription messages to a given multicast group). User data is normally sent over the network to its destination, whether or not it contains high priority information. Administrative messages, like subscription to a multicast group is not sent over the network. These messages trigger protocol-specific operations at protocol objects below the sending object; it is up to the protocol implementation to process these messages, like building a multicast distribution tree.

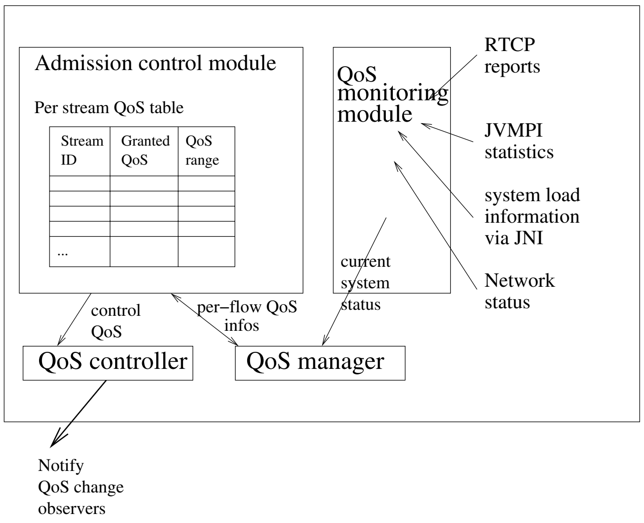

Figure 2. Structure of the NLML

It is up to the lowest layer protocol object whether or not it sends message type IDs to the peer.

The use of typed messages enables us to provide multiple queues to send low- and high-priority user data in independent message queues thus supporting QoS implementation at the protocol object implementation level.

Protocol stacks can be installed as ANTS extensions on any intermediate node thus allowing intermediate nodes to process messages traveling through them not only on a per-packet basis but depending on the user data in a given context (examples are caching on active network nodes or stream content transcoding). With ANTS extensions it is also possible to intercept non-active network traffic for any purpose, for example to support adaptation of RTP/IP streams sent over a hybrid ANTS/IP network or use information sent in RTCP packet flows unaware of active network technology.

The use of typed messages with separate message queues for high priority messages supports the QoS-awareness in the implementation of the protocol objects. The distributed QoS support is based on the information collected and distributed by the NLML presented in section 5.

5. Architecture of the NLML

The NLML contains many parts that are here described briefly.

Figure 2 illustrates the main components of the NLML.

The *QoS monitoring module* collects information on the status of the resources on the local network node. It can extract information from RTCP sender/receiver reports to get a picture about the media presentation quality. These packets contain information about the sender's RTP timestamps, the number of packets and bytes already transmitted. The receiver's statistics about the highest packet number received, inter-arrival jitter and packet loss ratio is in the RTCP receiver report packets. It uses the Java Virtual Machine Profiler Interface [6] to collect status indicators describing resource allocation statistics for a Java thread. Performance indicators at the operating system level are accessed on ANTS nodes with a special extension installed on the specific node using the Java Native Interface.

The *admission control module* is responsible for maintaining information about all streams currently served. The information stored contains a flow ID and the requested and admitted QoS requirements for a specific flow. This module decides whether a new stream can be admitted. Its decision is based on the QoS provided for other, already admitted streams and the information collected by the QoS monitoring module.

The *QoS manager module* is responsible for adapting provided QoS guarantees for an admitted stream according to the stream's requirements (required QoS range) and the current system load.

The *QoS controller* is responsible for detecting QoS-agreement violations and notifying the registered components in such situations.

The design of the NLML is scalable because of its management-domain based architecture. Domains, typically a set of active nodes in a LAN, work with common QoS-management policies. Management information is collected within a domain by active nodes and they are exchanged between domains to enable inter-domain QoS-management. The architecture is modeled after the IntServ [3] and DiffServ [2] model of the IETF. Although ANTS does not support separately managed domains (it knows the whole network), in a test environment we can separate domains by the use of multiple ANTS networks connected by tunneling packets between networks.

6. Related work

Similar research is going on at BBN in the QuO framework [11]. This work does not address the features provided by Active Networks and the current aim is to support IIOP/CORBA-based applications, although a new application area, multimedia systems, are being incorporated into the system [10]. They hide the QoS monitoring functionalities from the programmer. Our solution is independent of the underlying network layer, but enables to use active network technology efficiently and allows programmers to customize their solutions at the deepest level.

A similar research is going on at the UCLA in the PANDA project. PANDA is a middleware support system for active networks that allows both aware and unaware applications to benefit from better adaptability and performance of active networks (see [1] for more details). The PANDA project addresses security issues in active networks together with selected Artificial Intelligence algorithms to provide good data transfer and network resource use [9]. In our project we do not address security issues and support of active network unaware applications is only an option. Our main goal is to support multimedia content streaming.

7. Future work

Up to now we have seen many parts of an active middleware. Some parts are already ready to use, other parts are being developed.

We have discussed the following components in the paper which represent the main middleware support part of the A2QM3 system:

- A "common" protocol stack framework letting the programmer specify QoS-constraints and capabilities and providing a simple and uniform messaging and data transport interface to the application.
- A network load monitor layer augmenting the protocol stack implementation. It enables building QoS-aware protocol objects by means of providing them the static and dynamic system parameters they need and a callback interface for event notification.

The following components were mentioned in the paper as a part of the protocol infrastructure:

- A FIFO-ordered reliable data transfer protocol is already up and running. This is already in use to transfer still images and, when necessary, adapting them in the active network nodes during transmission. This protocol works with positive acknowledgments.
- A JMF/RTP socket adapter implementation for use with ANTS. This is still under development. Our goal is to get the necessary know-how from both the ANTS's point of view and JMF's point of view about how to extend JMF and how JMF does perform over ANTS. Here, a simple video chat application is under development that can run both over ANTS and UDP/IP under RTP. We use H.263/RTP for video coding and GSM/RTP for audio.

We plan an extension to a freely available CORBA ORB using an active network interoperability protocol (AN-

IOP) instead of the TCP/IP-based IIOP. ORBacus[3] or the OpenORB[4] are good candidates for these experiments.

8. Conclusions

The components introduced in this paper will be integrated into one "universal" QoS-aware middleware, the A2QM3. The protocol stack and the NLML implementation will serve as the basis for QoS-aware protocol stacks. The JMF/RTP and CORBA/ANIOP solutions will be integrated into the system by replacing the intermediate protocol adapters with active network based ones from the "common" protocol stack framework. The mentioned projects serve also a good way to learn more about the QoS requirements and QoS tolerance of real-world applications and for making further research on application-specific topics like a bypass mechanism for CORBA messaging where a separate QoS-layer might be unnecessary under certain conditions. This is possible because CORBA messaging already provides some kind of QoS.

Although CORBA is based on the request-reply communication paradigm, a unification of the distributed object's world with the multimedia streaming world is foreseeable by treating multimedia objects such that their content is accessed using RTP and the state of the object (i.e. the content) changes rapidly over time.

References

[1] The PANDA Project Home Page. http://lasr.cs.ucla.edu/panda/welcome.html.

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

[3] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture (RFC 1633), June 1994.

[4] B. Csizmazia and H. Hellwagner. Proposal for a QoS-aware middleware for adaptive multimedia data transfer. *ECOOP 2001 Workshop Reader (to appear in Springer LNCS 2323)*, 2002.

[5] R. Gordon and S. Talley. *Essential JMF - Java Media Framework*. Prentice Hall, 1999.

[6] JavaSoft. Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html.

[7] S. K. Kasera, J. F. Kurose, and D. F. Towsley. Scalable reliable multicast using multiple multicast groups. In *Measurement and Modeling of Computer Systems*, pages 64–74, 1997.

[8] OMG. Trading Object Service Specification Version 1.0, May 2000. OMG document number: formal/00-06-27.

[9] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated planning for open architectures. In *Proceedings of OpenArch 2000*, Mar. 2000.

[10] C. Rodrigues, J. Loyall, R. E. Schantz, and D. A. Karr. Controlling quality-of-service in a distributed video application by an adaptive middleware framework. In *Proceedings of ACM Multimedia 2001*.

[11] R. Schnatz, J. Loyall, M. Atighetchi, and P. Pal. Packaging Quality of Service Control Behaviors for Reuse. Submitted to the 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, ISORC 2002. BBN Technologies.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications (RFC 1889), Jan. 1996.

[13] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings IEEE OPENARCH'98, San Francisco, CA*, Apr. 1998.

[3]ORBacus is available from IONA Technologies. URL: `http://www.ooc.com/`

[4]Available under the URL `http://www.openorb.org/`.