The Design of an Application Programming Interface for QoS-based Multimedia Middleware

Christian Conrad, Burkhard Stiller

Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich, Gloriastrasse 35, CH – 8092 Zürich, Switzerland, Phone: +41 1 632 [7015/7016], E-Mail: [conradstiller]@tik.ee.ethz.ch

Abstract

With the advent of networked and distributed multimedia applications, requirements of middleware for providing advanced communication services changed. The needs in well suited architectures for delivering unchanged high network performance to multimedia applications and flexible communication services increased. In any case, applications have to access services of multimedia middleware and they are required to be programmed on top of these systems. Therefore, the developed application programming interface takes into account service specifications in terms of Quality-of-Service and an easy-to-use and intuitively programmable communication service for multimedia applications. This interface for advanced multimedia middleware hides away communication-relevant information from applications and provides a minimal set of interface functions and operations.

1 Introduction

The extension of flexible multimedia middleware far beyond traditionally layered communication architectures has offered manifold opportunities for the provision of advanced multimedia communication services. Research at the boundary between multimedia middleware and the application has to follow two aspects. Firstly, a homogeneous and adjusted multimedia communication interface is required that offers unaltered performance characteristics with minimal overhead as the underlying communication subsystem delivered it from the network. Secondly, the interface is required to be independent of multimedia applications and as far as possible the communication subsystem. Therefore, it has to offer a set of abstract interface functions assisting an efficient exchange of control and user data between applications and the middleware.

Allowing for the provision of communication services between an arbitrary communication subsystem and multiple applications as well as a sufficient information hiding, two basic abstractions for application data streams have been introduced. Since data streams may vary according to their type, location, and origin of data, so-called Flows have been designed. Furthermore, the concept of a Session has been supplemented to take care of interdependencies between two or more flows. This is significant for, e.g., synchronization purposes between related flows. These abstractions allow for hiding all communication protocol specific features. In addition, basic operations for dealing with Quality-of-Service (QoS) have been introduced. Application requirements can be specified flexibly in a variety of details defining specific characteristics of a required communication service. An application programmer reveals advantages and drawbacks of an interface, while constructing complex application scenarios with a given set of functions. The developed powerful Application Programming Interface (API) offers for this step a minimized set of functions for communication purposes.

This paper is organized as follows. Section 2 introduces related work concerning application programming interfaces. Within Section 3, the communication subsystem environment utilized for implementing the Application Programming Interface is briefly discussed. Section 4 presents functionality, modeling, and structure of the developed Application Programming Interface as well as an example for its use and evaluations. Section 5 summarizes the work.

2 Related Work

Related work on application programming interfaces covers traditional operating system level interfaces and emerging middleware interfacing. As for handling devices, e.g., file input/output, a programming interface is required, networked send/receive operations have to be supported by a programming interface as well. In principle, there is no difference between handling devices and managing network accesses, except for the fact that networking introduces a number of additional details con-
cerning the type of associations, addressing multiple
application processes, and supporting various communica-
tion subsystems or protocols.

Within the distributed applications domain, the com-
plete architectural view of communication middleware,
such as of DCE, CORBA [8], or ANSA, covers not only
application programming interfaces (API), which are,
however, the focus throughout this paper, particularly.
The API for the Internet protocol family in the UNIX
environment is designed as a C-based BSD Socket inter-
facing [1]. Berkeley Sockets serve an asymmetrically clien-
t-server relationship and offer support for connectionless
and connection-oriented associations. In addition, address-
ing has been solved by combining process port numbers
and addresses of peer applications. Within a set of inter-
face functions, e.g., socket() to create an endpoint,
bind() to bind an address, or connect() for connecting
to a server, required functionality to create, manage,
and teardown connections and to transfer user data is provided.
A secure socket layer has been introduced as well [2]. The
Transport Layer Interface (TLI) has been introduced with
release 3.0 of System V UNIX. The TLI offers an interface
to the OSI transport layer — based on the ISO/OSI
Basic Reference Model [3] — including a small set of
library functions [4]. Release 4.0 contains a TCP/IP trans-
port provider interface. The STREAMS message I/O inter-
face has been used to implement TLI. STREAM pipes are
quite similar to UNIX domain sockets [5]. Similar Win-
dows-based socket interfaces are available and comprise,
e.g., of WinSock2 [6].

Operating systems offer interfaces at different levels of
abstraction. The approach of InterProcess Communication
Service Access Points (IPC_SAP) [5] provides a C++
class library, including object-oriented service access
points, encapsulating standard transport interfaces as men-
tioned above. Operating system dependent interprocess
communications are hidden behind C++ wrappers, shield-
ing applications from error-prone details, combining sev-
eral operations into single application programmer
statements, and parametrizing IPC mechanisms into appli-
cations. Subclasses are defined for bidirectional data-
grams/streams and active/passive connection setups, while
connectionless or connection-oriented traffic is supported.

The design of a uniform API for basic-level communi-
cations in [7] has followed independently a similar devel-
opment as the QoS-based approach presented here. A
detailed comparison of various API can be here as well.

The designed API for QoS-based communication mid-
dleware and flexible communication services is based on
object-oriented interfaces, including the important exten-
sion of dealing with specifications for application require-
ments in terms of Quality-of-Service (QoS) parameters
and offering configuration support for multiple data
streams. Goals encompass hiding of errors from applica-
tions, simplification operations to manage associations,
provision of an easy-to-use interface, and re-usability of
interface functions.

3 DaCaPo++ Communication Subsystem

The communication subsystem DaCaPo++ [9] pro-
vides the possibility to configure end-system communi-
cation protocols and has been used to implement the API.
The configuration process is based on application require-
ments, availability of local resources, and network prereq-
tuities expressed in QoS. In particular protocol functions
and mechanisms as well as their properties form the basis
for the configuration process, which is directly supported
by a number of components within the subsystem as de-
picted in Figure 1 [10]. Protocol tasks, e.g., checksum-
ing or flow-control, are processed by communication
modules (C-modules) that are located in the heart of the
protocol. Applications access required communication
services via data access points through access modules (A-
modules), located at the boundary to the application pro-
gramming interface.

The DaCaPo++ communication subsystem supports
a variety of network infrastructures, such as high-speed
ATM networks (Asynchronous Transfer Mode) [11] or tra-
ditional IP-based (Internet Protocol) networks including
specific features such as multicasting and security. Each of
these networks can be accessed via a dedicated transport
module (T-module) for transmitting or receiving user data.

An application requests services, while delivering a set
of QoS attributes and values. The best suitable protocol
configuration consisting of various protocol functions will
be determined applying the configuration. Local system
resources, such as memory, buffer space, and processing
time, and network resources, such as available bandwidth
or delay, are considered additionally to determine impor-
tant properties of the run-time environment or the infra-
structure access, e.g., in terms of adapters.

An appropriate protocol graph is calculated by the
Configuration and Resource Allocation, CoRA, delivering
a formal description of selected protocol functions [12].
Finally, the protocol graph is locally instantiated and dis-
tributed to peer systems by the Connection Manager. The
Security Manager validates users/applications and assures
that necessary modules are contained within the protocol
graph to comply with requested security requirements.
Standard security features [2] and advanced features, such
as security QoS parameter translation and a key database
are included in the Security Manager and a separated data
base. Finally, a monitor supervises the execution of com-
munication protocols within the end-system and issues
notifications if application requirements are violated.

275
Within the communication subsystem a communication protocol, represented by a protocol graph, acts as the basic execution unit, which in turn consists of a well-defined number of modules. On this module basis a specific run-time system supports protocol processing tasks. Modules form the basic element within the run-time environment to perform every type of processing, similar to STREAMS modules [4], but usually of a finer granularity.

The communication subsystem associates three processing tasks with each protocol, which deal in decreasing order of priority with monitoring, data movement from the network up to the application, and from the application down to the network. The monitoring task periodically checks the actual performance of the protocol against application requirements and triggers a reconfiguration if the latter are violated. Both data movement tasks are instances of the lift algorithm. This algorithm waits for a data packet (more precisely, its reference), either from the application or from the network, and requests the modules to do the protocol processing. Although the basic structure of a Da CaPo++ communication protocol looks similar to a sequence of STREAM modules, it differs in protocol construction and execution. While a STREAM has to be built by the application and is rather static, Da CaPo++ communication protocols are tailored to the actual application needs and may be adapted dynamically to changing network and end-system properties. Furthermore, Da CaPo++ knows the execution properties of protocols and has control over processing elements. This allows for selecting optimized strategies of data movement and buffer allocation.

Currently, the communication subsystem supports multiple protocols for different data streams at the same time, e.g., a picture phone handles outgoing and incoming audio and video data streams separately which are represented by four different uni-directional Flows (sending and receiving audio and video). In general, a number of flows – representing uni-directional transmissions of user data with similar QoS requirements – are coordinated within a Session. The Da CaPo++ communication subsystem is responsible for handling Flows and protocol processing, and the Application Programming Interface is responsible for handling Sessions. Since Da CaPo++ offers unicast- and multicast-services to applications, a three-leveled hierarchy of application components, applications, and application scenarios has been defined [13] providing a modular, re-usable architecture for the integration of user-space and process-based applications for communication purposes.

In general, Da CaPo++ is a framework for the support of distributed multimedia applications offering an advanced application and communication middleware [14] as depicted in Figure 2. It consists of low-level middleware – the Da CaPo++ Communication subsystem –, a
bipartite Application Programming Interface (API), and high-level middleware — a set of application components. The applications and scenarios belong to the highest domain in the architectural view, while network services belong to lowest domain.

![Middleware Architecture Diagram](image)

**FIGURE 2. Middleware Architecture**

4 The Application Programming Interface

The Application Programming Interface (API) for communication services is the only interface visible to application programmers in end-systems. A particular user interface for an application may be constructed while utilizing functions of the API. Therefore, special care must be taken for efficiency and simplicity when designing and implementing a general-purpose QoS-based communication API. An efficient API hardly is acceptable where the application programmer has to write cryptic code. Similarly, a functionally rich API including large delays for only sending some bytes is useless. Hence, the resulting API design is a trade-off between efficiency and simplicity, knowing that a certain overhead cannot be avoided, but has to be minimized.

The need for a QoS-based communication API is even stronger as today’s prevailing API is the BSD socket interface [1], or its variants based on the IP suite. It has been admitted that TCP or UDP are not the most suitable communication protocols to support multimedia applications with high throughput and stringent delay or jitter requirements. As the corresponding socket interface cannot enhance a service offered by the transport subsystem, only transport infrastructure limitations are considered here.

Firstly, TCP considers one type of user data only, namely a general stream of bytes, whereas a general-purpose QoS-based API distinguishes between several different data types. Transport protocol properties for audio and video data are different in terms of maximum acceptable delay, loss-rates, and required bandwidth according to the compression algorithm. Extreme differences are encountered between file transfer data and isochronous data. Unfortunately, TCP does not allow any QoS specifications or any guarantees, such as for delay, bandwidth, or error-rates. With the emergence of the ATM technology, a step is done in the right direction of resource reservation, currently applied to bandwidth only [11].

Secondly, to authorize applications allowing for the specification of their particular QoS requirements, an API explicitly must contain new features, such as access control and QoS handling.

Thirdly, the BSD socket interface considers data only, which are directly generated or consumed by the application. Inherent inefficiencies when moving data from user to kernel space and vice-versa are significantly alleviated, if data do not transit always through applications. This is not suitable for every type of application, but for a video conference application, video and audio data may traverse directly from their associated device (camera and microphone) to the corresponding remote device (monitor and speakers), without having to transit through the application [15]. Only the less expensive control of devices in terms of the amount of data still remains under responsibility of applications.

4.1 API Design Objectives

Due to this introductory discussion the following design objectives have been achieved for the designed and implemented API for QoS-based multimedia middleware:

1. Simplicity and ease to use;
2. Handling QoS issues by configuration files;
3. Efficient data transfer process to process;
4. Efficient data transfer process to device;
5. General purpose and multimedia capable; and
6. Communication subsystem independence.

4.2 API Architecture and Functionality

The design of a general-purpose QoS-based communication API implies two different steps, which are independent of the underlying middleware. Firstly, within a static process resources are locally allocated and configured according to application needs. This process is similar to opening and binding a BSD socket with options. Secondly, a dynamic process is involved to establish a connection between two or more end-points and to exchange user data. Finally, user data are transferred via the API.

The designed API has to enforce this two-phase process by offering the application programmer a maximum degree of flexibility, while keeping in mind issues, such as
application QoS requirements, that play an important role not only during the establishment phase (including configuration and reservation), but also during run-time (QoS renegotiation). A central issue in the API is concerned with the definition of an association. If security mechanisms are supported, the application and the user must authenticate themselves during the establishment phase. Succeeding the authorization, an association between two or more applications must be defined in terms of user data streams and QoS requirements.

At the very beginning of the run-time phase, applications have to connect to each other via the communication subsystem. Once connected, they start sending or receiving data. In addition to the data transport to be supported, specific components of the communication subsystem also have to be addressed individually. Examples may comprise a configuration and security manager. For tearing down an association, proper functions are needed to deallocate resources. Moreover, for QoS-based communication subsystems, additional facilities must be available for the run-time management of QoS requirements, e.g., QoS monitoring and QoS renegotiation. Finally, it is of interest to extend or reduce the number of application flows during run-time, e.g., adding of a shared white board facility during a video conference application.

Concerning the general API functionality as depicted in Figure 3, the application itself is considered as the “API-Client” and the communication subsystem as the “API-Server”. The API-Client generates a request followed by a response from the API-Server, which is defined in the Main Control Protocol, MCP (1). However this scheme is not sufficient to meet every requirement of a communication subsystem, as it does not allow for the generation of asynchronous server messages. Nevertheless, it is compulsory for a communication subsystem to handle asynchronous requests. Therefore, so-called events (4) are directed towards the application.

Finally, a process boundary between the API-Client and the API-Server exists always, since the communication subsystem with the tied up lower API resides in a multithreaded process on the workstation and the applications including an upper API each, reside in separate processes on the same workstation. Therefore, a bisection of the API has been done: the lower API tied to the communication subsystem and the upper API linked to applications.

![Application Components Diagram](image)

**FIGURE 3. The Architecture of the Application Programming Interface**

### 4.2.1 Control and Data Delivery Protocols

Basically two types of channels have been designed for the API as depicted in Figure 3. Firstly, as mentioned above, the Main Control Protocol, MCP (1) between the components API-Client and API-Server via the Control Access Point, CAP has been defined. Secondly, the Data Delivery Protocol, DDP (2) and the Control Delivery Protocol, CDP (3) between each flow and its corresponding Data Access Point, DAP are used. These protocols need to be reliable, however, their characteristics are different in terms of efficiency, amounts of data, and transfer rates. During the application registration, session setup, and connection release phases, the amount of data sent over MCP is significant in comparison with control data exchanged.
during run-time. On the other hand, requirements for DDP are much more demanding, as the amount of data transmitted may vary according to the type of application. Hence, efficiency is a key parameter to be considered for selecting a mechanism to implement the two protocols. Transmitting asynchronous events (4) belongs logically to MCP, however it includes identical properties as DDP and CDP. The efficiency considerations for the transfer of user data have to be analyzed in closer detail, since data transits through the API only when it is directly generated or processed in the application. Typically for continuous data, such as live audio and video, the application does neither “see” nor “touch” data, as it is generated or consumed by the appropriate device directly within the communication subsystem process. The application manages this continuous flow of data by issuing control requests directly to the corresponding Access Module. Thus, it is not critical data that transits via CDP, but control requests only.

4.3 API Abstractions - Object-oriented Modeling and Implementation

Fulfilling the design goal of simplicity, abstractions have been introduced to encapsulate the complex behavior of communication subsystems and to provide accessibility via a small, but exhaustive interface. Hence, communication subsystem inherent complexity is hidden from the application programmer. The abstractions lead to an object-oriented model including three high-level abstractions, namely API-Client, Flow, and Session.

4.3.1 The API-Client Abstraction and Implementation

The API-Client abstraction is responsible for registering applications and users employing a communication subsystem. Thus, it encompasses all aspects of security, e.g., an authentication process. These tasks have to be performed once during an association setup. An API-Client exists only once in every application and is instantiated directly by the application programmer. The required three public client methods are included in Table 1. Additional private methods handle and maintain sockets to support the MCP and access via the Control Access Point (CAP).

4.3.2 The Flow Abstraction and Implementation

The Flow abstraction represents on one hand a communication protocol used between two applications, or between one and several applications in case of multicasting. It is dedicated to a data-type, e.g., audio, video, or raw data, to an origin, the sink or source of a device, a file, or directly the application, and finally to the sending or receiving direction. The Flow abstraction easily may be extended if new data types are required. On the other hand, the Flow abstraction offers a set of internal functions only to deal with the communication protocol, e.g., sending or receiving either data or control-data or setting new QoS requirements, since the Flow functionality can be accessed only via the Session abstraction. For that reason public Flow methods do not exist.

<table>
<thead>
<tr>
<th>Table 1. Public Client Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>API_Client()</td>
</tr>
<tr>
<td>Close()</td>
</tr>
<tr>
<td>SendSecMgrRequest()</td>
</tr>
</tbody>
</table>

4.3.3 The Session Abstraction and Implementation

The Session abstraction provides the facility to group and manage several Flows as a single unit. Flows that belong to a Session may have interdependencies, certain synchronization requirements. The Session abstraction is directly visible to the application programmer. Relevant public methods concerned with the maintenance of a Session are presented in Table 2. Hence, a Flow that belongs to a given Session is accessible only by its logical name via the corresponding Session. This trade-off is due to the fact that Sessions contain all relevant information for a single or multiple associations to be managed jointly.

<table>
<thead>
<tr>
<th>Table 2. Public Session Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Session()</td>
</tr>
<tr>
<td>Listen()</td>
</tr>
<tr>
<td>Connect()</td>
</tr>
<tr>
<td>Configure()</td>
</tr>
<tr>
<td>Activate()</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Deactivate()</td>
</tr>
<tr>
<td>Close()</td>
</tr>
<tr>
<td>GetFlow-Descriptor()</td>
</tr>
<tr>
<td>SetReq-Flow()</td>
</tr>
<tr>
<td>GetReq-Flow()</td>
</tr>
<tr>
<td>Send-Data-Flow()</td>
</tr>
<tr>
<td>Recv-Data-Flow()</td>
</tr>
<tr>
<td>SendCtrl-1Flow()</td>
</tr>
<tr>
<td>RecvCtrl-1Flow()</td>
</tr>
</tbody>
</table>

### 4.4 API Interface Structure and Objects

To support simultaneously several applications dealing with the communication subsystem and thus facilitating resource management, e.g., port numbers, devices, and memory, a client-server approach has been chosen. Hence, as mentioned in Subsection 4.2, the architecture of applications and communication subsystem has been split into at least two different processes, one for the communication subsystem and others for applications. As depicted within Figure 3, the resulting API is divided into two parts as well, namely the upper API linked to applications and the lower API defining the front-end of a communication subsystem. The upper API contains API-Client, Session, and Flow abstractions, whereas the lower API consists of a well-known Control Access Point (CAP) and of several dynamically configurable Data Access Points (DAP).

API components provide three distinct tasks. Firstly, security issues are linked with the corresponding API-Client abstraction. Secondly, several alternatives are proposed on how to setup a session object and manage it during runtime. Thirdly, lower API issues comprise related API-Server and Data Management components.

#### 4.4.1 Communication Subsystem Access and Security

Before being authorized to use the communication subsystem, an application and, therefore, implicitly a user, has to be registered properly by a security manager, which is in the current implementation part of the communication subsystem. This authentication process is encapsulated in the API-Client abstraction, thus, only one instance of the API-Client exists at a time per application. Moreover, the API must guarantee that an application is only granted access to its individual Flow objects. This separation principle is compulsory, if several applications use the communication subsystem simultaneously. Firstly, a check is performed to verify that the application is authentic and has not been altered, since it has been made available by a trustworthy user [2] on the system. A security manager has to ensure that the user is authenticated. In addition to a compulsory user name, e.g., a unique global name (email address), the user may provide a public key and either the corresponding secret key or a passphrase, unlocking a previously stored secret key. In every case, a generic security manager component decides to accept or reject the user.

#### 4.4.2 Programmers’ Session and Flow Specifications

The specification of Sessions, Flows, and QoS requirements is a crucial point in the API design as it has a direct influence on the design goal of simplicity.

The first considered approach contains a purely procedural interface via ioctl() similar to the BSD socket interface [1]. This solution has been rejected due to many reasons. Firstly, the BSD socket interface is based on IP and considers raw data transfers only, whereas a modern communication subsystem can use as well ATM, IP, or any other network infrastructure and distinguishes between several types of data, e.g., audio, video, or raw data. This significant step in complexity discouraged a further explo-
ration in this direction. A second disadvantage of this solution comprises its cryptic aspect. The amount of system calls necessary to setup a "simple" connection under the BSD socket interface is not scalable for a feature-rich communication subsystem [5]. The pure procedural approach did not take advantage of the object-oriented facilities that can be combined with API abstractions.

Therefore, the next natural step consists in declaring an object of class Session that contains all desired Flows including their QoS requirements. This solution is a significant improvement compared to the procedural interface, however, it still suffers from an important disadvantage as it leads to purely static object definitions being part of the program text. For example, it would be difficult to simply add a new Flow to a Session during runtime, or similarly to append new QoS requirements to a given flow without manipulating the program. For these reasons, a human-readable script-based approach including a configuration file is supported by an automated configuration parser as defined below, hiding object-oriented configuration details from the application programmer.

The selected script-based approach consists of a configuration file and of a set of functions based on the object model as introduced in Subsection 4.3. Initially, static properties of a Session are defined in a human-readable manner within a configuration file (ASCII text) as depicted in Table 3, where all flows are listed including their QoS application requirements.

A dedicated parser checks for the validity of the configuration file and relevant objects are instantiated automatically. Therefore, the application programmer only has to invoke the constructor of a Session containing the configuration file as a parameter (cf. Subsection 4.5). Hence, the complexity of setting up a Session is hidden completely within the parser and its tools. Naturally, the configuration file obeys a well-defined syntax, which is independent of the communication subsystem used. The upper API transparently passes these QoS parameters, along to the lower API, which in turn hands them to the communication subsystem. Besides these static Session definitions, a set of functions is required to dynamically act on the generated objects, e.g., during the connect/listen process, during protocol configuration, or for sending or receiving data. These functions comprise the functionality of the object model, are provided as methods, and already have been presented in Table 2.

The main goal of the configuration file is to transfer application requirements to the communication subsystem in a human-readable manner, directly supporting the application programmer and users. An application has not to be rewritten by a programmer, since the user himself may change the number of flows or its requirements in the configuration file. These features can be of a very general advantage, since the specification of QoS requirements and interdependencies of flows are more specifically adapted to the user's demands. In addition, the communication subsystem used may offer further specification possibilities, e.g., weight values associated to each QoS requirement. Therefore, the syntax of the configuration file has to be highly flexible and easily extendable for specific needs.

In general, the syntax is composed out of a set of keywords, which are either fixed to reflect the overall structure of the configuration file, e.g., SESSION, FLOW, ENDFLOW, SYNCHRONIZE, and ENDSESSION, or they are extendable to characterize the object to be generated at a finer granularity, such as the type of the flow VIDEO_SEND_DEVICE, AUDIO_RECV_DEVICE, or such as the specification of related QoS requirements THROUGHPUT, FPS, DELAY, or JITTER.

### Table 3. Configuration File Example

<table>
<thead>
<tr>
<th>Picture Phone Including 4 Flows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION CREATOR UNICAST PicturePhone</td>
</tr>
<tr>
<td>FLOW VIDEO_SEND_DEVICE VideoSendFlow</td>
</tr>
<tr>
<td>THROUGHPUT 4.5 2.0</td>
</tr>
<tr>
<td>FPS 5</td>
</tr>
<tr>
<td>ENDFLOW</td>
</tr>
<tr>
<td>FLOW AUDIO_SEND_DEVICE AudioSendFlow</td>
</tr>
<tr>
<td>THROUGHPUT 1.41 1.41</td>
</tr>
<tr>
<td>DELAY 0.1</td>
</tr>
<tr>
<td>JITTER 0.001</td>
</tr>
<tr>
<td>ENDFLOW</td>
</tr>
<tr>
<td>FLOW VIDEO_RECV_DEVICE VideoRecvFlow</td>
</tr>
<tr>
<td>THROUGHPUT 4.5 2.0</td>
</tr>
<tr>
<td>FPS 5</td>
</tr>
<tr>
<td>ENDFLOW</td>
</tr>
<tr>
<td>FLOW AUDIO_RECV_DEVICE AudioRecvFlow</td>
</tr>
<tr>
<td>THROUGHPUT 1.41 1.41</td>
</tr>
<tr>
<td>DELAY 0.1</td>
</tr>
<tr>
<td>JITTER 0.001</td>
</tr>
<tr>
<td>ENDFLOW</td>
</tr>
</tbody>
</table>

4.4.3 Lower API

The main component of the lower API is the API-Server reachable via the Control Access Point (CAP) as depicted in Figure 3. It has a well-known address and may be accessed by any application on the workstation. The API-Server is responsible for registering applications by the communication subsystem (cf. Paragraph 4.4.1), for setting-up Sessions including all Flows, and for the run-
time control of the association, such as starting and stopping of the data transport, setting of new QoS parameters, and closing the connection by deallocating reserved resources. The API-Server consists of a finite state machine (FSM) to process various application requests.

As several applications may access the API-Server simultaneously, two different approaches have been considered. (1) A single FSM deals with all applications or (2) a separate instance of this FSM exists for each application. There are inherent risks of “blocking” or unfairness with the first solution, as one application may block all others, either maliciously or due to a crash. Moreover, an application should re-authenticate itself towards the API-Server for each transaction. Both limitations can be circumvented by using timers and application passwords. The second solution provides fairness and the application authentication has to be done only once at the beginning. This approach provides greater flexibility at the expense of a slight programming overhead due to concurrency. Nevertheless, it has been preferred to the first one [16].

The API forwards QoS requirements from the application to the communication subsystem in a transparent manner. The task of mapping them into relevant network QoS parameters is left to either a mapping component inside the communication subsystem or to the corresponding Access Module. Therefore, facilities are provided by the lower API to retrieve QoS requirements as they have been specified by the application programmer in the configuration file and hand them to internal components.

4.5 An Example for Using the API

Providing an example for writing an application based on the API, a bidirectional picture phone application is outlined shortly. Firstly, the application and thus the user must register themselves towards the communication subsystem. This is programmed by instantiating an appropriate API-Client object with required security parameters:

\[
\text{API\_Client::API\_Client(security\_Info).}
\]

Afterwards, the session and its flows must be locally created. This is programmed by instantiating a Session object with the configuration file as shown in Table 3 and a reference onto the corresponding API-Client instance as parameters:

\[
\text{Session::Session(configuration\_file, APIClient).}
\]

At this point, the session is ready to either connect to a peer or to listen for incoming connections. Once connected, the session can be activated and data transfers can take place. During run-time, parameters of the access modules may be adjusted via the flow control interface or QoS requirements may be updated. This example demonstrates how straightforward the API can be used to set-up easily different Flows forming an association between two or more applications.

4.6 Implementation and Initial Evaluations

As the API design is independent of an underlying communication subsystem, the prototypical implementation is based on the DaCaPo++ communication subsystem. The designed API has been implemented on Sun SPARCSStations running Solaris 2.5.1; the implementation is described in closer detail in [16]. The DaCaPo++ core is written in C [17]. To take advantages of high-level abstractions of the API and to ensure compatibility with components of DaCaPo++, C++ [18] has been chosen for implementing the application framework. Hence, the upper API is written in C++ and the lower API in C. The parser for the configuration file has been generated by bison/flex tools [19] and is integrated in the upper API.

| TABLE 4. Saturated Throughput for the API |
|------------------|---------|---------|
|                  | UltraSPARC 170E | SPARC 20 |
| Sending          | 50.02    | (not available) |
| Receiving        | 70.93    | 31.93    |

To demonstrate the efficiency of the implemented API, it is necessary to consider at least two classes of applications. Mainly those that cope with continuous data only, such as live audio and video. For these applications the overhead due to the API is negligible concerning the exchange of control data only via the API. User data are handled with the device directly. The application only may send control data from time to time, which does not affect the performance of data transfer. However, other applications may exchange amounts of data directly over the API. This is handled by shared memory and inter process control mechanisms, depending on the current application in use. Table 4 lists these process to process throughput numbers in Mbps that can be expected when either sending or receiving user data over the API, while sending and receiving 1000 packets of 1000 Byte size.

| TABLE 5. Instantiation of the API-Client |
|------------------|---------|---------|
| Response         | 14.88   | 31.70   |
| IPC for MCP      | 20-35%  | 50-65%  |

Table 5 shows results for the overall instantiation time of an API-Client object to register an application process over an Ethernet (LAN). The different processing times in ms between creators (the initiator of an application setup)
and participants (responders) are presented additionally, including an approximation of the time spent in the IPC for the MCP as described in Subsection 4.2 in comparison to the response time. A number of further performance evaluation details have been analyzed and may be obtained from [16] in more detail.

5 Summary

The developed and implemented Application Programming Interface (API) supports flexible communication services handling QoS specifications, transparently hiding communication-relevant features from applications. The easy-to-use upper API from a programmer’s perspective achieves the required degree of simplicity. Application QoS requirements are passed by a highly flexible configuration file via the API from applications to the communication subsystem. Efficient process to process data transfers are supported by the data delivery protocol. By bypassing isochronous data to devices directly, the API satisfies multimedia applications causing significant improvements of data management efficiency during the data transfer phase. The API is independent of the underlying communication and multimedia middleware which results in its portability on other communication architectures. Finally, the architecture provides a feature-rich functionality compared to existing APIs.

Acknowledgments

The authors would like to express many thanks to their colleagues, namely to D. Bauer, G. Caronni, C. Class, and M. Waldvogel for technical discussions, and B. Plattner for providing an excellent research environment.

References