A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience

Burkhard Stiller Christina Class Marcel Waldvogel Germano Caronni Daniel Bauer

Abstract—**Distributed multimedia applications require a variety of communication services. These services and different application requirements have to be provided and supported within (1) end-systems in an efficient and integrated manner, combining the precise specification of Quality-of-Service (QoS) requirements, application interfaces, multicast support, and security features, and within (2) the network. The Da CaPo++ system presented here provides an efficient end-system middleware for multimedia applications, capable of handling various types of applications in a modular fashion. Application needs and communication demands are specified by values in terms of QoS attributes and functional properties, such as encryption requirements or multicast support. Da CaPo++ automatically configures suitable communication protocols, provides for an efficient run-time support, and offers an easy-to-use, object-oriented application programming interface. While its applicability to real-life applications was shown by prototype implementations, performance evaluations have been carried out yielding practical experiences and numerical results.**

Keywords— **Flexible Middleware, Application Programming Interface, Quality-of-Service (QoS), Protocol Processing Support, Security.**

I. Introduction

WITHIN an environment of highly distributed systems sophisticated communication facilities are significant. A great number of distributed applications, most of them handling multimedia data, can be supported by tailored communication protocols and efficient middleware transparently hiding details of network technologies. However, in many cases communication middleware may create a performance and functional bottleneck, since communication protocol implementations available today do not offer proper protocol functions for handling continuous data adequately. Furthermore, standard run-time environments for protocol processing are not able to cope with high data rates.

Therefore, emerging multimedia applications require various communication features to be integrated and supported efficiently, which traditionally have been considered separately. For example, a video conference on public networks for confidential enterprise management meetings requires communication protocols providing appropriate encryption and authentication functionality in addition to multicast data transmission capabilities for audio and video. Many-to-many communication links between participants have to be established on demand. Moreover, scenarios involving financial transactions or confidential data require different degrees of security. Therefore, for real-world applications, an integrated solution for communication middleware has to provide security and multicasting functionality in addition to multimedia services. The developed middleware Da CaPo++ provides multimedia support within end-systems which is adaptable to application needs. This concept is applicable to standard communication processing environments. In addition, a number of re-usable middleware services for dealing with multimedia application communication services are defined and implemented. Da CaPo++ integrates many aspects of research results obtained so far, *e.g.*, Quality-of-Service architectures, object-oriented system development, efficient protocol run-time systems, protocol configuration, and advanced protocol function support. The Da CaPo++ middleware demonstrates within a powerful and efficient system that a general purpose end-system middleware for multimedia support is operational and interoperable with other end-systems applying Da CaPo++ as their choice of middleware.

B. Stiller, C. Class, and M. Waldvogel are with the Computer Engineering and Networks Laboratory, TIK of the Swiss Federal Institute of Technology, ETH Zürich, Switzerland (<last name>@tik.ee.ethz.ch); G. Caronni is now with Sun Microsystems Research Laboratory, Palo Alto, California, U.S.A. (gec@acm.org); Daniel Bauer is now with the IBM Zürich Research Laboratory, Zürich, Switzerland (dnb@zurich.ibm.com).

Extending a multimedia middleware far beyond traditionally layered communication architectures has offered manifold opportunities for the provision of tailored multimedia communication services. This avoids common design pitfalls for multimedia communications, such as low efficiency or dedicated functionality. Therefore, the following three classes of requirements hold for communication middleware in general. They imply major design goals for Da CaPo++ and its implementation specifically claim:

- Efficiency: Middleware has to provide an *efficient multimedia communication protocol processing support* which is applicable to standard workstations and operating systems. In addition, it has to support many specific protocol functionalities, *e.g.*, multicasting and security, in an integrated fashion. The Da CaPo++ run-time system and its protocol processing algorithm – called Lift – represent a flexible processing scheme for controlling modular protocol tasks based on a standard workstation's operating system.
- Usability: A homogeneous *Quality-of-Service-based (QoS) multimedia communication interface*, similar for all kinds of multimedia applications, is essential. The interface should be easy to use for application programmers and independent of specific multimedia applications. Therefore, a QoS-based Application Programming Interface (API) achieves application transparency by assisting the exchange of control and user data between applications and the middleware. Furthermore, It has to offer the unchanged performance of the underlying communication subsystem to applications. This allows for the provision of the following features:

1. The specification of various functional requirements, such as degrees of privacy or reliability, multicast group management, and addressing, in terms of QoS parameters.

2. The transfer of and agreement on application requirements in terms of traditional QoS attributes, including numerical values for, *e.g.*, bandwidth, delay, or bit error rates.
3. The enabling of application programmers to design reusable application components whenever possible or intended.

- Modularity: A *variety of communication protocols and network technologies* has to be supported in a modular fashion for a wide spectrum of traditional and multimedia applications. Based on QoS specifications, modular communication functions and specific protocols are selected flexibly, *e.g.*, for live audio, stored video, or plain data transfer, where protocols consist of building blocks. A series of various protocols and functions, particularly for security and multicast, has been implemented as prototypes and is integrated into Da CaPo++.

Da CaPo++'s real-life applicability including the list of above mentioned features, has been experienced and tested within an Application Framework offering by itself a modular structure. This framework has been implemented for real-life scenarios and applications, such as a tele-seminar or a picture phone. These applications and the middleware provides the basis specifically for performance evaluations under real-life conditions. A picture phone is discussed with respect to the above stated claims.

This paper is organized as follows. Section II briefly compares related work on various aspects related to middleware for multimedia communications. Whereas Section III discusses the design of Da CaPo++, Section IV points out implementation issues. While Section V shows its practical use, Section VI evaluates obtained results. Finally, Section VII summarizes the work and draws conclusions.

II. Related Work

On one hand, related work on middleware covers approaches with a strong architecture-oriented focus. These approaches define the access level and degree of transparency for distributed applications to communication functionality. In general, they may cover transaction-based applications, directory services, location-independent services, or dynamic object invocation. Examples of some general purpose middleware comprise DCE [1], CORBA [2], TINA-C [3], COM [4], or ANSA [5]. The views of these middleware approaches focus mainly on the interoperability issue as well as the generic service provision, but they do not concentrate on efficient communication protocol processing or multimedia Quality-of-Service (QoS) support in the first place. The latter aspects started to be dealt by in recent work, however, have not been finished yet. The approach TAO [6] deals with investigations of CORBA-based middleware for high-speed networks and applications. The AQuA approach [7] develops adaptable, object-oriented, distributed computing systems while applying quality objects to manage system characteristics. The support of computational grids for applications is described in Globus [8], which defines the design of a special purpose middleware.

On the other hand, the support of diverse functionalities and the provision of adequate performance of the middleware is crucial for multimedia-capable approaches. Due to the wide range of relevant topics that are integrated to provide a flexible, multimedia middleware presented here, a number of different areas of related work is relevant. Four main groups of aspects are dealt by the Da CaPo++ middleware:

- Provision of advanced communication functionality,
- Flexible and multimedia middleware,
- Efficient run-time system and protocol configuration, and
- Application support by QoS specification.

Another set of important related work has been selected and categorized according to these main aspects. Table I depicts these aspects in addition to further comparison criteria, where a criterion not applicable is marked by N/A.

While multimedia middleware is intended to support a wide range of multimedia applications, flexible middleware introduces an orthogonal concept for communications to support adjustable protocol processing for high-performance applications and high-speed networks, as done within ADAPTIVE [9] or F-CSS [10]. Generally spoken, to facilitate a flexible approach requires to structure protocols in a modular fashion, where separate building blocks can inter-operate efficiently. Da CaPo++ offers a set of protocol functions implemented in terms of software modules that run in an efficient run-time system, the Lift algorithm.

Efficient run-time support for general protocol processing tasks has been investigated, *e.g.*, in the x-kernel for modular protocols [11], the Scout operating system for path-based module interconnections [12], and the Crossbow project supporting a high-performance toolkit for experimenting with IP (Internet Protocol) next generation protocols [13]. In particular, for middleware supporting tailored communication protocols accommodating the needs of communications, a suitable run-time system for fine-grained and interoperating modules is essential. In contrast to the Integrated Layer Processing approach (ILP) [14], Da CaPo++ favors a modular protocol processing approach which is integrated with the Application Level Framing (ALF) approach [14] to achieve good protocol and application performance.

Most existing approaches provide application knowledge to the middleware environment by offering an interface for the specification of QoS parameters. In OSI'95 [15], a QoS-based transport service including QoS parameter definitions was developed; the Lancaster QoS-Architecture (QoS-A) [16] defined a QoS concept for end-systems, and the QoS Broker [17] investigated QoS management issues which are continued in the QualMan approach [18]. A comparison of QoS specification and management as well as general QoS concepts may be found in [19] and [20]. Many of these approaches allow for the detailed characterization of applications and the specification of their communication requirements on different levels, such as the application-level, the transport-level, or the end-system-level. But they are lacking open, extensible, and efficient Application Programming Interfaces, *e.g.*, in support of multimedia applications. Therefore, object-oriented interfaces for stand-alone systems have been studied, *e.g.*, IPC-SAP [21] or Sockets++ [22] in addition to procedural ones, such as WinSock2 [23]. The Da CaPo++ middleware integrates an open, QoS-based object-oriented interface with the exploitation of many QoS attributes for configuring a specifically tailored communication protocol as well as the selection of an appropriate network technology, if at all applicable.

TABLE I
COMPARISON OF SELECTED RELATED WORK

Criteria	ADAPTIVE	F-CSS	QualMan	Scout	OSI'95	QoS-A	MCF	Da CaPo++
Multimedia Middleware	Medium	Medium	Yes	N/A	N/A	Yes	Yes	Yes
Flexible Middleware	High	High	Medium	N/A	N/A	No	High	High
Protocol Configuration	Flexible	Flexible	No	Flexible	No	No	Flexible	Flexible
Run-time System	No	No	Yes	Yes	No	No	Yes	Yes
Application Support	API	QoS-API	QoS-API	Path	Protocol	Interface	QoS-API	QoS-API
QoS Specification	Yes	Detailed	Yes	No	Yes	Detailed	Detailed	Detailed
QoS Management	No	Limited	Yes	Yes	No	Yes	Yes	Yes
QoS Exploitation	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Security Functionality	No	No	No	N/A	No	No	No	Yes
Application Framework	No	No	No	No	No	No	No	Yes

Multimedia middleware must integrate various functionalities, *e.g.*, encompassing security and multicasting capabilities. Security issues are dealt by a number of approaches, *e.g.*, the Globus approach [8], the work for high-level network protocols such as the Secure Socket Layer [24], and a number of specific security algorithms and protocols. A good overview of security relevant policies and solutions may be found in [25]. Many algorithms deal with multicast communications, such as for Audiocast [26] and multicast routing [27]. A feature-rich and efficient multicast framework for end-to-end QoS guarantees for multipoint communications (MCF) is presented in [28]. However, it has not been well understood how QoS requirements, security mechanisms, and multicast communication protocols inter-operate within one single middleware at the same time. Da CaPo++ offers a new approach for handling security requirements as QoS attributes, integrating multicasting independent of the underlying network technology, and for providing synchronization mechanisms for multimedia data streams. While this paper focuses on security issues, further details can be obtained from [29].

Summarizing, the Da CaPo++ approach combines most of the advantages mentioned above for related work and handles multimedia applications and advanced functionality in an integrated and efficient manner as not performed before. This includes the Da CaPo++ middleware provision on end-systems for standard workstations, showing a close cooperation between applications, the API, security and multicast capabilities, QoS concepts, and the communication middleware itself.

III. DA CAPO++ DESIGN

The Da CaPo++ middleware is end-system-based and located between the network access and the Application Programming Interface (API) (cf. Figure 1). The middleware as well as the API support multimedia communications, since multiple time-dependent media flows in addition to native data flows being part of a single or multiple flow session can be processed on standard workstations. This is due to the middleware's good performance and its provision of appropriate protocol functions.

Da CaPo++ provides on end-systems communication protocols in support of application flows. In addition, it covers the possibility to flexibly configure these communication protocols built out of protocol functions according to application requirements expressed in terms of QoS parameters [29]. A configuration process to perform this application-driven adaptation is directly supported by a number of internal Da CaPo++ components (cf. Figure 2) and a component to configure the required protocol. In addition, this configuration is based on application requirements, availability of local resources, and network prerequisites as well as protocol functions and mechanisms including their properties [30]. Relevant protocol functions, *e.g.*, checksumming or flow-control, are processed during run-time by individual communication modules (later referred to as C-modules) and are located in the heart of the protocol. Applications access any type of communication service of a configured communication protocol via the API through application support modules (A-modules) including a direct multimedia device support. This integration is achieved by combining the physical end-system architecture in terms of data producing or consuming devices into the Da CaPo++ design, *e.g.*, for multimedia devices cameras, microphones, or speaker boxes.

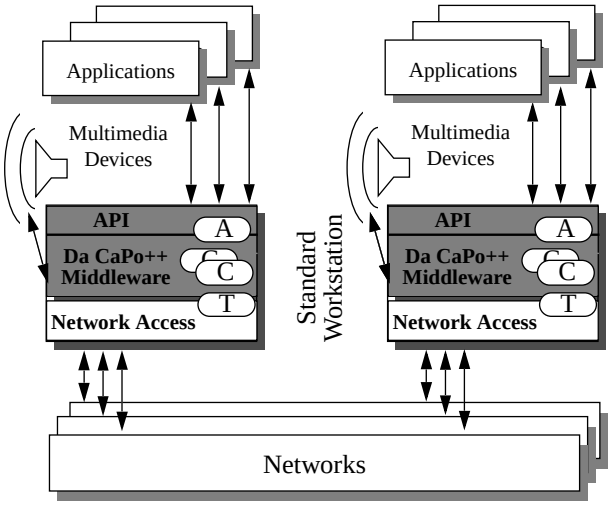


Fig. 1. Overall Da CaPo++ Middleware Architecture

On the network access side of Da CaPo++ available networks in terms of ATM (Asynchronous Transfer Mode) and an Ethernet-based Internet are utilized, particularly offering different levels of guarantees for network performance, such as bandwidth guarantees or no guarantees at all. As an application does not have to care about differences in network mechanisms used, properties and especially semantics of different networks are hidden. This level of abstraction is provided by transport modules (T-modules) being part of the configured communication protocol. Summarizing, every Da CaPo++ protocol consists of one A- and T-module each and up to multiple C-modules.

For design purposes the Da CaPo++ middleware covers end-system issues on standard workstations, common multimedia devices, and applications on top (cf. Figure 1). The Da CaPo++ core – determining an instance of the middleware on one end-system – and an API reside once per workstation in end-systems, while multiple applications may utilize the same middleware core at the same time [31]. To accommodate diverse networks, QoS specifications are used in the API and in the core as powerful abstractions, enabling application programmers to ignore specific properties. Also most applications and protocol modules of the core do not have to care how end-system-internal security services or unicast or multicast are internally implemented.

All important details of Da CaPo++ tasks and additional internals of the middleware core are discussed below. The following introduces how sessions are configured and set up. Afterwards, the data transport mechanism is explained including the module concepts used the resource management. The designed security features are discussed before an overview of the API developed is presented.

A. Da CaPo++ Tasks and Components

The Da CaPo++ core determines the heart of the middleware, it performs all functions related to session management and data transfer, and it specifies an evolution of the original Da CaPo system [30]. Its central goal is to take as much as possible burden off the application and the programmer yet still give them a maximum of freedom. To show how Da CaPo++ achieves these properties, the following issues describe main functionalities and tasks from an application viewpoint for setting up a communication association:

- The application names the source or sink for data and specifies communication requirements. It may choose among predefined protocols offered by the protocol database, instead of specifying a list of parameters itself.
- The application identifies the communication peer, requests the establishment of an association, and starts the data transfer for sessions consisting of a single or multiple flows.
- Afterwards, data transport is performed independently of the application. Instead of caring about each individual packet that is transmitted, the application is free to return to its main task, *e.g.*, perform user interaction.
- Whenever important communication events happen, *e.g.*, alarms or change requests of QoS specifications, the application is notified to take appropriate measures. The application can also query and modify the state of flows at any time.
- When the transmission of user data is finished, the application requests a session tear-down.

The main workhorse is the Da CaPo++ core. As shown in Figure 2, it consists of several main components, which interact closely. Applications send requests through the API to the Session Manager, which performs the necessary session management functions. It receives help for setting up protocols – configured out of modules – from information stored in the Protocol Database. It also assigns resources and buffers to the protocols themselves, which perform the actual data transfer. The Security Manager will be discussed below (cf. Section III-E).

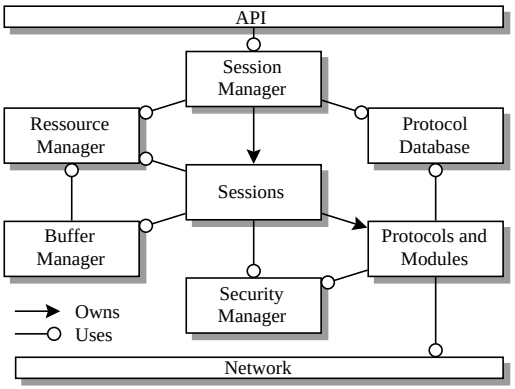


Fig. 2. Da CaPo++ Component Relations

B. Session Specification

Recall that Da CaPo++'s main design goals were to provide a modular and efficient middleware. It should offer applications the flexibility of specifying their communication needs in detail where desired, but also remain oblivious about elements applications do not want to specify and still receive reasonable service. To achieve this, a two-layer model was chosen: the application can (a) specify the types of flows needed, *e.g.*, audio transmission, and (b) mention any specific QoS requirements it has for each of these flows.

To allow this, sessions have to be created in a modular fashion. They form bundles of unidirectional data flows which form the basic data transport entities; *e.g.*, a picture phone session would consist of an audio flow in one direction, an audio flow in the reverse direction, and a corresponding pair of video flows.

To achieve this modularity, a single session needs to be split into a hierarchy of elements, which are selected according to application-specified parameters and then combined into the final protocol. In the picture phone example (cf. Section V-C), the application would specify its need for a session consisting of the above-mentioned audio and video flows. The session hierarchy is depicted in Figure 3. Since many protocols will require feed-back mechanisms, *e.g.* retransmission requests or camera control, the data flows are split into two data paths, a "forward" main path, consisting of the data and some protocol control information, and into a "return" path, transmitting this feed-back information.

Every data paths can be implemented by a protocol stack, which can be built according to the application's requirements. Each flow definition can specify the set of functions the underlying protocol may need to fulfill. For a video flow, this might include frame grabbing, compression, encryption, and transmission on the sending side, and the corresponding inverse functions on the receiving side. These generic protocol functions can be implemented each by any number of modules, tuned to a specific environment and making use of the existing hardware. Such modules could include a SunVideo frame grabber, DES encryption, support for a specific ATM networking card, among others.

This approach for abstraction allows for a high flexibility in that the application does not need to know any details of data transfer from the source to the sink, but still can influence whatever is needed. For example, an application might request com-

pression and does not care about how the video stream is compressed, as long as a specified compression factor providing a specified minimum quality is met, but it might request a specific encryption scheme with specified parameters.

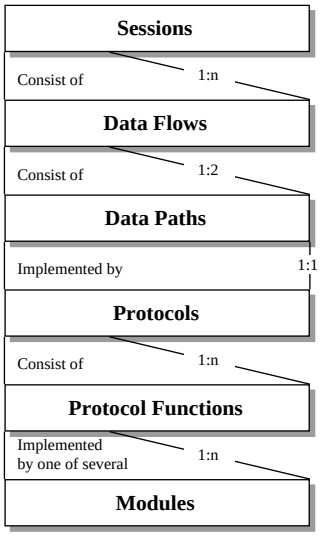


Fig. 3. Session Hierarchy

Besides the flows specified by the application, each session requires a reliable flow, used internally by Da CaPo++ for session-wide management information. It is used at session setup time to inform the joining participant of protocols and parameters being used. Later on, it is used to send out-of-band session control information between individual modules, the middleware core per se, or between applications.

At configuration time, all modules performing necessary services are selected and configured according to a set of requirements specified by applications. These requirements are grouped into several categories, *e.g.*, peer to connect to, throughput, security parameters, and levels, *e.g.*, high-level/abstract requirements, low-level requirements.

High-level requirements specify parameters in an abstract manner and do not provide necessarily complete determinism with respect to modules selected and parameters tuned, as long as the result meets the requirements. In contrast, low-level requirements select a specific module to use or a specific parameter of a given module. Requirements usually do not specify fixed values, but a (possibly weighted) range, so the Da CaPo++ middleware has flexibility in fulfilling requests. Since the requirements specified may conflict with or have influences on each other, a precedence hierarchy has been set up. Low-level requirements have precedence over abstract requirements which in turn override system-specified default parameters. After configuration, the application is informed of the configuration success and values selected.

A single module is not of much use, it needs at least a corresponding peer at the other end of an association. Often, the receiving module also needs to provide feedback to the sending module to function properly. This shows that many operations classically considered as one function indeed consist of up to four parts (cf. Figure 4). In Da CaPo++, these four parts are treated independently. A forward path consists of a "down"

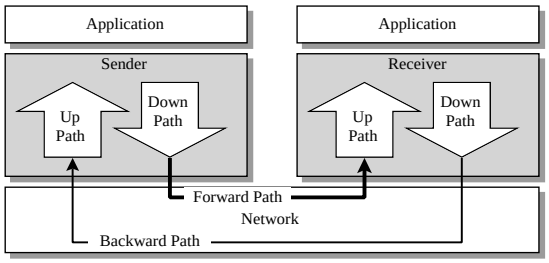


Fig. 4. Module Relations

part in the sender transporting (usually a lot of) data towards the network and a matching "up" part in the receiver in addition to a corresponding backward path with comparatively little control information. The forward and backward data paths may have different module configurations, either because only some modules need to have access to the backward path or because the modules in the backward path themselves need some protocol processing, *e.g.*, authenticated acknowledgments. Each of the data paths had to be completely separate. To provide efficient use of the backward (feedback) channel, communication among the parts of the module performing operations in either the "up" or "down" path have to be simple and fast.

C. Lift

As described above, protocols determine the middleware's view of application flows. Flows are split into two data paths (cf. Figure 3) for the forward (data and control) and backward (feedback) direction. User data transfer only occurs in one direction (forward path), where resource reservation based on the requirements may be applied. The backward path is used for control information only, *e.g.*, acknowledgments, quality feedback, which encompasses usually small amounts of data. Thus, flows are uni-directional from an application point of view, but they are bi-directional for control.

Although Figure 3 may suggest that there is a lot of hierarchical overhead involved, this overhead is negligible for protocol processing (cf. Section VI). A limited amount of overhead occurs at session setup and almost no penalties are to be paid at run-time, where all protocols' selected module instances are directly accessed, making the layering a conceptual tool only.

In each data path, data is transported by an algorithm called *Lift*, an active transport mechanism, originally developed in a first version in the predecessor project Da CaPo [32]. Once started, the Lift works autonomously calling in turn modules' processing functions, according to the sequence set out at protocol configuration time. The Lift goes on to transfer data from the network to the user or vice versa, until it receives new instructions from the application or one of the modules it passes by. The Lift passes a packet along all modules within a protocol and each module performs appropriate changes and may request the Lift to pause, bring another packet, or discard the packet. The independence of the Lift – every Lift responsible for a single protocol runs in a separate thread – frees other system parts from duties (cf. Section IV-C for Lift/module interaction). It also makes a protocol easy to trace and schedule.

Compared to most other flexible protocol architectures, this scheme does not cause each module to be stacked on top of each

other on the function call stack, possibly requiring a large stack for local variables. Compared to traditional stacking architectures, after a module returns the control back to the Lift, only minimal module state is present, making this an ideal point for efficient context switching. An additional advantage is that module implementation can be simplified. They do not need to care for special cases, such as errors returned from called modules. Instead, the Lift determines the decision-making mechanism. To relieve the programmer from a burden, generally, a module's handler function will be called. This is achieved by requesting a module for its requirements at protocol set-up time. Knowing all the requirements in advance enables for further optimizations. This approach eases configuration changes, since there is only one location knowing about the protocol chaining.

The actions a module can control include the following:

- Communication between forward and backward path: The corresponding module in the other path received information which it will need to send out with the next data packet. This schedules a Lift run in the other path.
- Out-of-band information has to be transmitted to the module implemented in the communication peer.
- The module has remaining data to be transmitted. Do not turn idle after finishing protocol processing for the current packet.
- The module is currently busy, wait for a mutex to be cleared.

Normally, the Lift passes a packet through all modules in only one direction, according to the direction of the data path. It starts with an empty packet, obtained from a buffer list maintained by the Buffer Manager, which is being filled with data by the first module. Possibly the packet is modified by intermediate modules and emptied by the last module in the chain. Under some circumstances, *e.g.*, for segmentation and reassembly or reliable data transmission, modules may not only have data to fill in, but entire packets to send. In this case, the module will signal the Lift that its next run only should be a partial run to pick up the remaining data. This partial run merely covers the signaling module and the modules beyond it.

Concerning memory requirements of modules, buffers containing packets show a packet structure including a fixed header, where each module owns a pre-arranged number of bytes at a known offset. The packet also contains a variable-sized data block, for use by A-modules to transport end-to-end data. Since protocol details are known at initialization time, the size of the maximum data block can be deduced in advance; growing and shrinking data blocks within these limits is just a matter of telling which part of the allocated block contains the data (cf. Section III-D).

Probably one of the main points of interest for the application programmer concerns the control he has over running protocols. It is possible to create or destroy sessions and to pause or restart flows in a session, where the control flow is used to transmit changes in the session's state and, thus, is always active (cf. Section III-F). The A-module directly can be controlled by applications and provides feedback to applications using request/response messages and asynchronous events. This scheme can be used to request fast forward or rewind functions for a remote video server and could be extended to control a remote camera, *e.g.*, zooming and panning.

D. Resource Management

Another of the main goals for designing the Da CaPo++ core was to provide efficiency by reducing data copying. Therefore, the Lift only transports a packet descriptor. A packet consists of three buffers whose sizes are determined at configuration time. One of the buffers holds data of variable-length to be transmitted between A-modules located at peers. Another buffer holds the constant-sized header enabling all modules to communicate information to its partner module located at the peer. A third buffer is used for communication between modules in a single data path within one end-system and is not transmitted over the network. Packets and buffers are managed by the Buffer Manager also keeping track of reference counts for each buffer. This allows modules to keep a copy of a buffer by a referencing pointer without actually copying the data. Segmentation and reassembly modules can also pass on partial buffers to avoid the creation of partial copies. The possibility to pass partial buffers together with separate, constant-sized headers gives processing advantages. As far as Da CaPo++ is concerned, it also allows for a zero-copy operation using Application Layer Framing [14] even for segmentation/reassembly.

The Buffer Manager – and all other components requiring system resources – request their resources from the Resource Manager (cf. Figure 2). It provides an abstraction layer for memory, CPU resources (in the form of threads), and timers. This simplifies portability, reduces memory management overhead and minimizes memory copying.

E. Security Functionality

In Da CaPo++ encryption and authentication functions are not only available as an integral part of the middleware, but the security degrees (amount of privacy and authenticity required for messages) are also treated as QoS parameters. In an environment handling multimedia data streams of high data volume, computational resources required to provide the highest level of security usually exceed available CPU power. For this reason, the provided amount of security (the strength and, therefore, computational requirements of employed protocols) is variable, depending on user demands.

Privacy and authenticity of communication are as much considered a basic service quality parameter of the network as packet loss rate, bandwidth, and delay. Together with C-modules providing the actual data security, the Security Manager (cf. Figure 2) represents all security functionality within Da CaPo++. Special C-modules encrypt or authenticate arbitrary data streams and the Security Manager provides peer authentication services. It is fed with application requirements and translates them into low-level security parameters, collects randomness from system state to provide keying material, and assures security at runtime. Additionally, it includes the key database and cryptographic routines of PGP (Pretty Good Privacy) [33]. For integration purposes, PGP has been changed into a library and linked into the middleware.

E.1 Authentication Services

In Da CaPo++, communication peers and local applications connecting to the middleware are authenticated. The application

or the user must authenticate itself when the application first requires services from Da CaPo++ through the Application Programming Interface (cf. Section III-F). During the delegation of identity, the application indicates the keys in the database to be used and provides a passphrase to unlock them. This information (cf. Figure 5) is passed from the application via the API to the Security Manager. Figure 5 shows details on data structures shared between the Security Manager and the Da CaPo++ Application Programming Interface. In practice, only the provision of a key ID for a public/private key pair and the corresponding passphrase are required to prove one's identity to the middleware. The middleware then utilizes the public/private key pair that is provided by the user and its own public/private key in the proof of authenticity to the peer system. The administrator of a system must be trusted, because he can impersonate any user accessing the middleware by multiple means, one of them is the capture of the passphrase as it is transferred from the user to the middleware.

```
struct auth_data {
    process_id;
    // simplified RFC822 address, e.g. <joe@doe.com>
    user_id;
    // String with separate fields, e.g. name = netscape_navigator;
    // ver = 2.0b; platform = sun4u; os = sunos5.5.1; author = foo@bar.
    application_id;
    // Hash of application file(s), signed by trusted entity.
    application_certificate;
    // Given as PGP key ID, RFC822 or hex style.
    user_public_key_id;
    // Unlocking the key in the PGP DB.
    user_private_key_passphrase;
    // Alternative to giving the passphrase.
    user_private_key;
    // String passed on to the user for confirmation.
    query_string;
};
```

Fig. 5. Application Authentication Data Structure

As an additional means of control, the user may choose to terminate any or all applications using the middleware on his behalf. This is done by accessing a separate user interface that directly connects to the Security Manager.

Admission control of communicating peers is done by applications, because evaluations have shown that criteria for admission control are too varied to be efficiently delegated to the middleware. Upon arrival of a new peer, the Security Managers exchange (1) certified keys and (2) additional authenticating material forming a authentication hierarchy. So an application could specify whether it would only accept a particular remote user, or that it would also trust the remote application or the remote middleware. If the application accepts the association, communication begins.

E.2 Security QoS Translation

Security parameters can be controlled as application requirements. Security parameters in Da CaPo++ express requirements on four separate layers: (1) user requirements, (2) abstract application requirements, (3) low-level requirements, and (4) infrastructure requirements. Depending on requirements put upon a layer, certain costs result. Requirements posed on the infrastructure are, *e.g.*, necessary CPU seconds per real second for a flow transmission, memory consumption, or network bandwidth.

Low-level requirements within the middleware cover parameters, such as key length, choice of algorithm, and key change rate, and are easily understood and directly adhered to by modules. As it is the goal of Da CaPo++ to provide a comfortable environment for application programmers, these parameters may be well-known and straightforward, nevertheless the average application programmer or even user cannot be expected to fully understand their security implications. Additionally, it might be undesirable to pre-select encryption and authentication algorithms and their parameters in detail. Whenever advances in cryptology indicate the insufficient safety of such an algorithm, all applications statically demanding the algorithm would require changes.

To address this problem, low-level requirements are derived from application and user level requirements, as outlined in Figure 6. The application can specify the required strength of security algorithms to be employed, defined as the amount of time a communicated information is supposed to stay unreadable or authenticated against a predefined class of potential enemies. The model employed in Da CaPo++ to specify security requirements on the user level is the threat model. Users specify the most likely attacker, *e.g.*, casual hacker, determined group, competing enterprise, multinational corporation, or rogue government agency, as well as the presumed value of the information. Therefore, the system must provide security mechanisms whose breaking costs are higher than this value. Additionally, a probability specifies how likely these promises should be met.

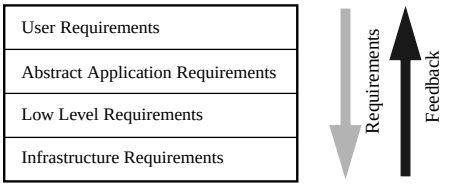


Fig. 6. Abstraction of Requirements

This type of parameters can be determined easily by the user than the low-level requirements. These parameters are evaluated based on a database containing strengths and weaknesses of different algorithms, together with their likeliness to be broken or weakened in the years to come. This likeliness based on current and expected cryptanalytic results. Creating and maintaining this database is not an easy task, but is only marginally more complex than directly specifying *well chosen* low-level security parameters in the first place. One advantage of this database is that it only needs to be defined once by the developer or administrator of the middleware and not by every application programmer or user. Additionally whenever necessary due to advances in cryptology, the strength of security mechanisms offered to applications can be increased transparently by updating the database. This mechanism even allows for adding new and improved encryption algorithms without user or application programmer involvement.

E.3 Security Assurance

The Da CaPo++ middleware allows for modification of security parameters in an ongoing communication. This reconfiguration can switch off or change cryptographic algorithms without interrupting the flow of data. This admits users to tune system performance in a fine-grained manner, *e.g.*, receiving better quality in video transmissions, when security is not required. At the same time, if underlying infrastructure offers security functionality by itself or if it is considered to be secure, *e.g.*, a leased line or an office LAN are usually considered much more private and authentic than packet radio or the Internet, security functionality employed in the middleware can be reduced. As an additional consideration, the middleware administrator may enforce certain minimal security requirements which can not be circumvented by applications relying on the Da CaPo++ middleware.

The security assurance component in the Security Manager also monitors the usage of keying material and keeps track on the amount of encrypted data and period the key was used. Whenever the user or the systems determines the necessity, a change of keying material is initiated. To economize costly asymmetric cryptographic operations, multiple data encryption keys are transferred as one asymmetrically encrypted data packet and containing keys are consumed as needed.

Within this novel approach of the Da CaPo++ middleware, security functionality is integrated tightly into the Da CaPo++ core and protocol processing. This provides key management and a variety of encryption and authentication functions to flows and sessions, which are selectable be users in a similar fashion as they request for reliable transfer of data.

F. Application Programming Interface

The Application Programming Interface (API) for communication services is the only interface visible to application programmers in end-systems from the middleware. Since data streams may vary according to their type, location, and origin of data, two basic abstractions for application data streams, called flows and sessions have been designed (cf. Section III-B). This allows for hiding all communication protocol specific features [34]. In addition, basic operations for dealing with Quality-of-Service (QoS) have been introduced [30]. Although, *e.g.*, TCP considers one type of user data only, namely a general stream of bytes, a general-purpose QoS-based API needs to distinguish between several different data types. Transport protocol properties for audio, video, and user data are different in terms of maximum acceptable delay, loss-rates, required bandwidth, security-levels, and multicasting features. This is formalized in a configuration file as depicted and discussed in Section V. However, the application always handles communications in an association-based manner, where the API handles association context information, *e.g.*, including session identifiers, and the underlying protocols may provide a connection-oriented or connectionless service. In general, the API utilizes an object model, where the base class of flows consists of three subclasses for an audio-flow, a video-flow, and a data-flow, each of them containing the respectively required functionality. As every flow may receive or sent data only, separate classes encompass the required functionality. Therefore, applying the concept of multiple inheritance to these classes, the requested instance will be automatically generated based on the application requirement specification and it contains the functionality for, *e.g.*, sending user data which is termed `SendDataFlow`.

An important difference is encountered for data from applications and live data, originating from multimedia devices. As the BSD socket interface [35] considers data only being directly generated or consumed by the application, inefficiencies when moving data from user to kernel space and vice-versa are significant. Since this is not suitable for every type of application, *e.g.*, for a video conference application, video and audio data may traverse directly from their associated device (camera and microphone) or file to the corresponding remote device (monitor and speakers), without having to transit through the application. In general for any application, only the less expensive control of devices – in terms of the amount of data – still remains under the responsibility of applications which may include control commands such as fast forward or fast rewind for video. Multimedia user data per se are directly handled by the appropriate multimedia device.

The design of a general-purpose QoS-based communication API implies the provision of three different steps, which are independent of the underlying middleware. Firstly, within an application process resources are locally allocated and configured according to application needs using available API functions (cf. Section IV-E and Table IV). This process is similar to opening and binding a BSD socket with options. Secondly, a set-up process is involved to establish an association between two or more end-points and to exchange user data. Thirdly, user data are transferred via the API, if they do not originate from multimedia devices, otherwise, they are handled by the corresponding A-module directly. The designed API has to enforce phase one and two to offer the application programmer a maximum degree of flexibility. This takes into consideration that application QoS requirements play an important role not only during the establishment phase (including configuration and reservation), but also during run-time (QoS re-negotiation).

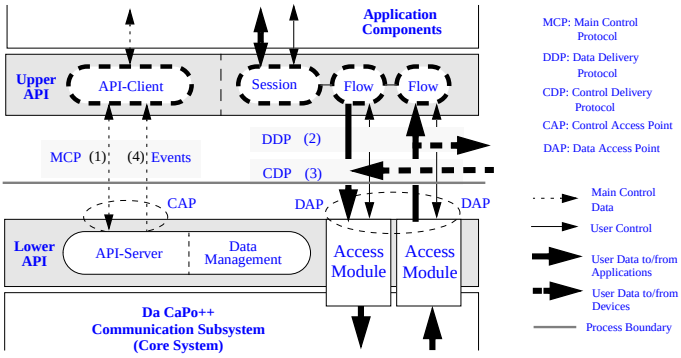


Fig. 7. API Architecture

To support several applications on top of the Da CaPo++ middleware a client-server approach has been designed. This facilitates the resource management tasks for port numbers, devices, and memory (cf. Figure 7). The upper API is linked to the application, while the lower API defines the front-end of the Da CaPo++ core. Functionalities and tasks are accessible by ways of the Control Access Point and Data Access Points.

During the set up procedure of associations the Main Control Protocol assures that the appropriate number of resources is allocated. While user data is in transit the Control Delivery Protocol is applied. The Data Delivery Protocol ensures that common shared memory or appropriate Inter-process Communication (IPC) schemes are utilized to optimize the communication performance.

A central issue in the API is concerned with the definition of an end-to-end association between peers. Besides middleware-internal encryption and decryption functionalities being 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, which is additionally supported by separate memory segments for every session.

IV. Implementation

A. Object-oriented Module Implementation

The entire Da CaPo++ core is object-based, in order to achieve the desired modularity of the middleware. This is most noticeable with modules: neither the module selection and configuration components, nor the setup components, nor the Lift algorithm needs any adaptation, when a new module is added. In order to have all the flexibility of an object-based approach, yet still full control over everything that happens, and the speed available, the core and every module was written in the C programming language. Since creating classes and instances is not supported by the C run-time environment, a special run-time support was created. Modules form basic building blocks and behave like classes in an object-oriented environment: They have a descriptor structure which contains key elements identifying them, *e.g.*, their name, and a list of function pointers to call and perform well-known functions. Additionally, they have an assigned partner module to be used at peers.

Individual instances of a module can be created using a function in the Da CaPo++ run-time system. Instances do also have a descriptor structure, containing identifying elements, information about which module they stem from and which protocol they are used in. Unlike most other object-oriented systems, it also contains function pointers. As each module gets to know application requirements at configuration time, it can and must adapt to these parameters, *i.e.*, an audio module may configure the sampling rate and input device according to specified requirements. Although most modules are capable of handling different media types, each individual instance will process only a single data type during its lifetime. Therefore, some modules go even further and change their instance's function pointers to point to functions which are optimized for a number of special cases, at initialization or even run-time. This turned out to be especially handy in implementing protocol state machines. A module that has adapted itself to its environment is called a virtual module. For example, instances of transport modules know, whether the configured protocol will ever use header fields or what the maximum size of a data block can be and do replace their generic function. Also, when audio receiver modules are instantiated, they can determine whether they are the second instance and can make sure that the first instance (and any further instance to be created) will use the audio mixer. Using a mixer is required since the used audio device only supports a single reader and a single writer.

Instances can not only find out about their class or other instances of the same class, they can also determine information about any module within their session, both on the local and remote site. After having found the desired module, they can also communicate with them; locally using method invocations and remotely by sending them a control packet. Although the class concept is being used, no inheritance is currently provided, but modules providing similar functionality may share code providing common functionality by putting it to a code library.

Besides internal test, protocol trace/debug, measurement, and traffic generation modules, a number of multi-media communication protocol processing modules have been implemented. Among A-modules are modules to transmit application-to-application data (RawData) and to receive and transmit from the audio and video ports or from stored files including the usual rewind and fast forward functionality, *e.g.*, SunVideo, VideoFile, SunAudio, AudioFile. T-modules include unicast and multicast transport support for ATM, UDP, and TCP, where multicast for TCP is emulated by opening multiple ordinary TCP connections. C-modules for different groups have been implemented, such as flow control and reliable transfer (Alternating Bit Protocol, Idle Repeat Request, Multicast Error Control), segmentation and reassembly, encryption (DES, Triple DES, IDEA, and RC5 in both Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes, Diffie-Hellman and RC4), and authentication (H-MAC MD5, RSA signature). All these modules are designed for multimedia communications. They are capable of handling different data types at performances required by high-quality streaming media.

B. Protocol Database

As we have seen, each data path is implemented as a series of individual modules in Da CaPo++. Although the modules are independent of each other, the corresponding modules in the forward and backward path usually share their instance variables to simplify state updates. Since these modules are tailored to be used together, they are combined into one *mechanism*. A mechanism usually has a natural way to be integrated into a protocol, *e.g.*, video compression should be done in the down path on the sending side and the corresponding decompression step on the receiving side. Reconsider the authentication of acknowledgments; sometimes it would be useful to use a mechanism a little bit differently, *e.g.*, decompress stored video in the sender, because the receiver is only able to handle uncompressed video or use the segmentation/reassembly module to assemble tiny packets from the source into suitably large network packets. To fulfill these demands, it is possible to individually swap each mechanism in a protocol (specified by flags in the protocol database) along all its symmetry axes: Swap sender and receiver side, up and down direction, or forward and backward path.

During the development and testing of the Da CaPo++ middleware, it turned out that the level of flexibility mentioned in Section III-B and prototypically implemented in [36] is seldom needed. It results in indeterministic behavior and requires a lot

of effort on the side of the module designer to fully specify the configuration dependencies (each module may specify pre- and postconditions as requirements, *e.g.*, a reliable transport below). Last but not least, it also introduces a very high evaluation overhead at session setup time. Therefore, the middleware-internal table of modules has been augmented by a database of pre-configured protocols. Each of these named protocol definitions consists of a sequence of modules to use and configuration parameters to these modules. Still, the application retains complete control and can override any of the parameters, yet it was possible to greatly simplify the configuration algorithm during session setup both in code and run-time overhead.

```
{
    {"pfVideo",          NULL, "mcVideo", 1, 0,0,0},
    {"pfAuth",           NULL, "mcMD",    2, 0,0,0},
    {"pfAsymAuth",       NULL, "mcDS",    3, 0,0,0},
    {"pfPrivacy",        NULL, "mcCBC"    4, 0,0,0},
    {"pfKeyAgreement", NULL, "mcDH",    5, 0,0,0},
    {"pfTransport",      NULL, "mcATM",   0, 0,0,0}
}
```

Fig. 8. Sample Protocol Data Base Entry

A sample protocol definition for a secure video transmission is shown in Figure 8. All fields from left to right contain the name of the protocol function, the name the instance should get (if needed for communications between otherwise unrelated modules), the name of the preferred mechanism to be used, the order in which the modules should be executed, and swap and module options including side swapping. The processing order must be specified, because it was originally planned to allow for the parallel execution of independent protocol functions. This has not been implemented, since the synchronization overhead between parallel threads turned out to be much higher than the performance improvements achievable.

C. Module Configuration and Operation

Although the modules can be used very flexibly, knowledge of only a few simple interfaces is needed to implement a module (cf. Figure 9). In general, modules are passive and are called, when they need to perform a function, directing the caller using return codes. If a module wants to make use of an interface, it simply provides a function which will be called at appropriate times.

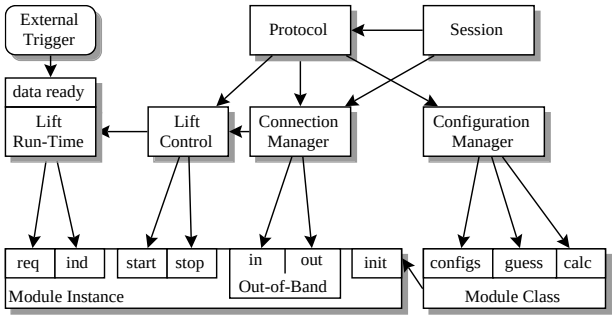


Fig. 9. Module Interfaces

At session setup, the Connection Manager and all other requested protocols are created. The Connection Manager is a regular communication protocol, but with the special duty to help in connection set-up and transmission of control and out-of-band messages. Each protocol fulfills requirements given by the application, which the Configuration Manager resolves in a two-pass process: In the first pass, traversing from the A- to the T-module, it determines all module requirements using the `guess` function. In the second pass, traversing the opposite direction, it resolves these requirements using the `calc` function. If the preferred configuration of the modules is not able to match all requirements, each module is queried for other potential configurations using the `configs` function. After the decision has been made, all modules are instantiated accordingly.

At run-time, modules' `start` and `stop` functions are called, whenever data transport is allowed to start/resume, or is paused/stopped, based on instructions of local and remote applications. After that, modules' `req` and `ind` (request/indication) functions are called as long as at least one module signals that it has more data ready. After that, the Lift turns idle and waits for anyone calling its data ready function to continue, *e.g.*, because of a timer event or the backward path signaling the forward path that it has finally received the acknowledge.

Return values of the `req` and `ind` functions are especially powerful. They signal, *e.g.*, whether the module has data that should be transported, whether the module is busy and cannot accept new data now, whether it or it's sibling in the other path has more packets ready, or whether it wants to send out-of-band data. After signaling that out-of-band data is ready, the Connection Manager picks up the data by calling the `out` function and will deliver it through the network to the matching module's `in` function.

All `req` and `ind` functions are called after each other (if both are defined), where the idea of the request function is to send data into the module and of the indication function is to get data out of the module again. This looks redundant at first, but in fact, this can be used to simplify the design of modules due to a design particularity of the Lift. Whenever the Lift has transferred a packet through all modules, it performs a reverse scan through the indication functions to find out, whether any module has anything more to send, which it would start transporting. This results in a simplification of segmentation or retransmit modules, while still assuring the packet order.

TABLE II
EXAMPLES OF SOME DEVELOPED MODULES AND PROTOCOLS

Protocols	A-modules	C-modules	T-modules
Audio Ethernet	AudioFile	-	McastSocket
Audio ATM	AudioFile	-	McastSocket
Video Ethernet	VideoFile	Measure	ATMMultiSocket
Video ATM	VideoFile	Measure	ATMMultiSocket
Reliable Data	RawAPI	-	MultiTCPSocket
Unreliable Data	RawAPI	-	UDPSocket
CryptoDES	RawAPI	MD5, DES, RSA	UDPSocket
Crypto-IDEA	RawAPI	MD5, IDEA, RSA	UDPSocket
CryptoRC5	RawAPI	MD5, RC5, RSA	UDPSocket

To obtain a detailed view on some different protocols supported, Table II depicts an excerpt and configuration in terms of configured A-, C-, and T-modules, where short module and protocol names are presented.

D. Security Modules and Protocols

To implement and evaluate the basic QoS mapping mechanisms for user and abstract application requirements, the Da CaPo++ middleware offers different security modules and protocols. This allows to show their usability in the context of multimedia protocols and continuous media support. Modules for key agreement, privacy, and authentication are provided (cf. Table III). Note that MD4 is not practically used anymore, since it has been broken in the meantime, and DH for agreement on a shared secret is only usable in conjunction with RC4. It can be used when no peer authenticity is required or perfect forward secrecy has to be provided.

TABLE III
IMPLEMENTED SECURITY MODULES

Function	Algorithm	Parameter
Key Agreement	DH	Size of shared secret
Privacy (block cipher)	DES, 3DES, IDEA, RC5	Key change interval, ECB or CBS modes, number of rounds and key length for RC5
Privacy (stream cipher)	RC4	Key length, Key change interval
Symmetric Authentication	MD4, MD5	MAC on/off, Key change interval
Asymmetric Authentication	RSA	Signing interval

Protocols providing either encryption, authentication, or both can be configured. By specifying appropriate QoS requirements the application can choose which cryptographic algorithm is to be used in appropriate security modules. QoS requirements can be changed on runtime, while users can influence directly the behavior of active protocols, change the employed cryptographic algorithms, or switch off cryptographic mechanisms completely.

E. Implementation of the API

While the complete API is discussed in [37], an excerpt of main interface functions offered for the session- and flow-level are listed in Table IV. These functions are applied from the application programmer to utilize the Da CaPo++ middleware as exemplified in (cf. Section V-C).

As a general task, the API has to cross a process boundary between applications and the Da CaPo++ core. The application itself is considered as the "API client process" utilizing the upper part of the API. The "API server process" offers the lower part of the API. Multiple API clients, one for each application, reside in a multi-threaded process on a workstation and applications including the upper part of the API generate a request followed by a response from the lower part of the API. Events can be directed towards the application in an asynchronous fashion. Shared memory and Inter Process Communication paradigms are offered by the API to efficiently support various types of stored data coming from applications. Particularly, bypassing the API for data originating from devices achieves a sufficient throughput for continuous multimedia data streams (cf. Section VI).

TABLE IV
EXCERPT OF PUBLIC SESSION- AND FLOW-LEVEL API METHODS

Function	Description
`Session()`	Constructor of a session object requiring the configuration file and a reference on the previously instantiated API client object as parameters.
`Connect()`	The session either actively connects to a peer or passively waits for a connect() from a peer. Parameters are addresses and ports, necessary for both unicast and multicast connections.
`Configure()`	Every flow of a session is configured by the communication subsystem.
`Activate()`	The transport of data is started or resumed for every flow of a session.
`Deactivate()`	The transport of data is stopped or paused for every flow of a session.
`FlowJoin()`	A new flow is dynamically added in the session (allocation of resources).
`FlowLeave()`	An existing flow dynamically leaves the session (deallocation of resources).
`Close()`	The session is terminated and all resources are deallocated.
`SetReqFlow()`	New QoS requirements can be forwarded to the Da CaPo++ core during run-time.
`GetReqFlow()`	Actually configured values of QoS requirements can be retrieved.
`SendDataFlow()`	User data can be sent within a flow using its flow descriptor.
`ReceiveDataFlow()`	For receiving data the asynchronous approach via a callback function is available.

V. EXAMPLE: IMPLEMENTATION OF A PICTURE PHONE ON TOP OF DA CAPO++

Da CaPo++ has been validated by the implementation of an extensive application framework on top of the middleware. A modular design has been retained also for complex applications resulting in a three-level framework [38], [39]. Since its modularity reflects the modularity of Da CaPo++ on the application level, its basic idea is shortly introduced. Based on the example of a picture phone implementation on top of Da CaPo++ the usability design goal of the Da CaPo++ middleware is discussed.

A. A Three-level Application Framework

As control mechanisms and user interfaces for different data and connection types may be reused in different applications, a three-level *application framework* has been defined and is depicted in Figure 10.

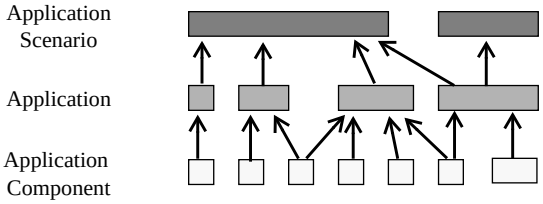


Fig. 10. 3-Level Application Framework

It is per se independent of Da CaPo++ and can be applied to all sorts of applications. The *application component* level comprises atomic units providing a well-defined functionality, *e.g.*,

the display of video data. This functionality is system specific and directly can make use of, *e.g.*, Da CaPo++ A-modules. An *application* consists of one or more application components and offers a single, homogeneous functionality being provided in close cooperation by the application components. *E.g.*, a picture phone determines an application in this sense. An *application scenario* fulfills a completely specified task within a real-world scenario. It consists of one or more applications being logically structured. As the application and application scenario level often cannot be separated clearly, a picture phone can be used as part of, *e.g.*, a telebanking scenario (application) or as a simple picture phone (application scenario).

B. Applying the Da CaPo++ Middleware

In order to make use of the Da CaPo++ middleware, it has to be installed on all involved end-systems of senders and receivers. Within development environment, Da CaPo++ has been implemented on Sun workstations operating Solaris 2.5.1. The Da CaPo++ core including the lower part of the API is implemented in C. A compilation is required on the dedicated end-system. The compiled core is running permanently on these end-systems and applications can connect to the core and utilize the middleware. The upper part of the API is currently implemented as a C++ library and must be linked to applications built on top of Da CaPo++.

Dedicated functionality like video compression using Sun's video card [40] can be used only, if the required hardware is available on the end-systems. Applying, *e.g.*, compression schemes like JPEG (Joint Pictures Expert Group), which are also supported, the interoperability is increased as JPEG can be decoded on other platforms as well. In order to use Da CaPo++ on other platforms, like Windows NT™, the Sun specific part of the code, *e.g.*, the thread management, of Da CaPo++ needs to be ported.

Existing applications can run on top of the middleware after the integration of the Da CaPo++ API. Data generated and consumed by the application is transmitted to and received from the Da CaPo++ core via the API methods `SendDataFlow` and `RecvDataFlow`. These methods are called in a ported application whenever data is written to or received from, *e.g.*, TCP (Transmission Control Protocol) sockets. Data transmission is provided by Da CaPo++ transparently to the application. While this is a valid approach to apply the Da CaPo++ middleware, applications hardly profit from the supported middleware functionality. Especially the handling of multimedia data within the Da CaPo++ middleware eases and supports the efficient implementation of new multimedia applications. This is demonstrated by the example of a picture phone implementation on top of Da CaPo++.

C. Implementation of a Picture Phone

The Da CaPo++ picture phone allows two participants to communicate by exchanging live audio and video. Da CaPo++ A-modules capturing and presenting live audio or live video data, respectively, are combined with unreliable, unicast data transmission T-modules to live audio and live video protocols.

The configuration file for the picture phone session in the creator, *i.e.*, the caller, is depicted in Figure 11. The specified session determines a unicast session, while it consists of four flows, each one for sending and receiving audio and video, respectively. Every flow is assigned a type, *e.g.*, `VIDEO_RECV_DEVICE`. This type specifies the data type, the source/sink of the flow, and the direction of the flow. In this case the session creator is a receiver of a `VideoRecvFlow`. In this example, data is sent directly to the device and not passed via the application (cf. Section III-F). Depending on the data, on end-systems, and on the communication medium available, QoS parameters are specified for every flow. They may encompass, *e.g.*, throughput, frames per second, samples per second, bits per pixel. The communication protocol is configured out of selected modules, where the decision is based on the configuration file, and the protocol is instantiated by the Da CaPo++ core (cf. Section III-A) according to the specified QoS.

```
SESSION CREATOR UNICAST Picturephone
FLOW VIDEO_RECV_DEVICE VideoRecvFlow
     THROUGHPUT  4.5   2.0
     FPS         5     13
     DELAY       0.2   0.45
     JITTER      0.001 0.0035
ENDFLOW
FLOW AUDIO_RECV_DEVICE AudioRecvFlow
     THROUGHPUT  1.41  1.41
     DELAY       0.2   0.45
     JITTER      0.001 0.0035
ENDFLOW
FLOW VIDEO_SEND_DEVICE VideoSendFlow
     THROUGHPUT  4.5   2.0
     FPS         5     13
     DELAY       0.2   0.45
     JITTER      0.001 0.0035
ENDFLOW
FLOW AUDIO_SEND_DEVICE AudioSendFlow
     THROUGHPUT  1.41  1.41
     DELAY       0.1   0.3
     JITTER      0.001 0.003
ENDFLOW
ENDSESSION
```

Fig. 11. Picture Phone Configuration File

The example of the configuration file (cf. Figure 11) specifies two values (maximum and minimum) per parameter for audio and video data. In the sample configuration file, the communication requests different delay and jitter characteristics in both directions, specifying an asymmetric communication.

Multimedia data capture and presentation is performed by the instantiated A-modules, whereas data transmission is performed by different Da CaPo++ protocols.

Implementing a Da CaPo++ picture phone requires the following steps, which are depicted in Figure 12:

1. A Da CaPo++ client must be created exactly once per application. The corresponding method (`DaCaPoClient` constructor) in the upper API creates a Da CaPo++ client and connects it to the Da CaPo++ middleware. Authentication information of the application is passed in the `securityInfoStruct` and evaluated in the Security Manager during creation of the Da CaPo++ client. After this method invocation the functionality of Da CaPo++ can be used.
2. Sessions for data transmission must be instantiated. The picture phone is implemented within one single data session. The corresponding API method requires a configuration file and the

identification number of the Da CaPo++ client. The configuration file passed to this method is depicted in Figure 11.
3. A connection based on the specified QoS parameters must be established between sender and receiver. This is done by the `Connect` method invocation. The connection information passed to this method contains a structure specifying the address of the peer, its port number, as well as the own address and port number. Both port numbers are used to establish the connection between the Connection Managers of both peers. A callback function is specified within this method call. This function is called whenever an event must be passed from the upper API to the specified session.
4. The session is activated (`Activate`) to start data flows sending and receiving data. Due to the specification in the configuration file (cf. Figure 11), data is captured directly by the device (microphone and camera) and displayed on the device (speakers and monitor). Data transmission continues until the method `Deactivate` is called.
5. To stop data transmission the session is deactivated (`Deactivate`). If the session is deactivated gracefully, the Lift algorithm delivers all data pending in the middleware for this session to the participating modules before the session is stopped. Deactivating the session instantaneously would result in a graceless deactivation of all participating flows. After deactivation, the session can be resumed again by the `Activate` method.
6. Before leaving an application, involved sessions must be closed (`Close`). This frees resources reserved by the Da CaPo++ middleware and deletes all data structures related to this session. Afterwards the destructors for session objects and the Da CaPo++ client objects are called.

```
    /* create Da CaPo++ client */
1: client = new DaCaPoClient(securityInfoStruct);
   /* create session */
2: pp = new Session (configFileName, client);
   /* connect session to peer */
3: pp->Connect (connectInfo, callbackFct);
   /* start or resume session */
4: pp->Activate ();
   /* stop/pause session or hold on line */
5: pp->Deactivate (GRACEFUL);
   /* close the session */
6: pp->Close ();
```

Fig. 12. Applying the API to the Picture Phone

These method invocations are sufficient to implement a picture phone on top of Da CaPo++. Additionally, a graphical user interface has been implemented in Tcl/Tk [41]. The connection information is specified in this user interface. The user can enter the communication peer's machine, a default is used for the port number. By activating user buttons, the peer can be connected, data transmission can be started or stopped, and the connection can be closed.

Within the Da CaPo++ project, numerous applications including a teleseminar scenario and a media server have been implemented in order to evaluate the Da CaPo++ middleware further. As the results obtained have shown, the support of multimedia data provided by Da CaPo++ is adequate and even complex multimedia applications can be implemented easily [39].

VI. Evaluation of Da CaPo++

The performance of data communication obtained in a given implementation determines the quality of communication services and protocols. The Da CaPo++ middleware as described above has been implemented on standard workstations [31], such as Sun SPARC20 and Sun UltraSPARC 170E (evaluation machine) running the non real-time operating system Solaris 2.5.1. The Da CaPo++ middleware has been evaluated using the Quantify tool [42] and high-resolution system time measurements directly. Standard Sun multimedia equipment has been utilized, such as cameras, microphones, and the SunVideo board [40] offering real-time image capture and compression for digital video.

Concerning the performance numbers below, first the overall overhead due to modularity is discussed. Afterwards security relevant performance figures are outlined and, finally, the API's efficiency is presented. Both, writing and sending of user data requires semaphore operations for accessing the shared memory. For this reason, the sending ($<$460 ms) and the receiving delays ($<$12 ms) for data originating in applications on top of the Da CaPo++ middleware on an end-system were measured. The throughput numbers achievable for various protocols differ, specifically based on the special protocol configuration applied. For example, the results for an unreliable protocol processing determine: The sender requires on average 45 μs vs. 18 μs and the receiver requires 41 μs vs. 31 μs. Concerning the measured upper bounds in the unreliable case, 273 μs for the sender and 323 μs for the receiver have been observed within the Solaris operating system environment. Therefore, Da CaPo++ achieves in this case an average sender throughput of 38.4 Mbit/s for 88 Byte packets and of 44.8 Mbit/s for 1024 Byte packets. The worst but guaranteed case throughput for the unreliable protocol is determined by 2.4 Mbit/s and 3.7 Mbit/s respectively.

A. Lift Performance and Protocol Processing Overhead

The performance of the Lift determines the overhead involved in the concept of achieving modularity within Da CaPo++. The processing overhead is shown in Figure 13. These numbers include the overhead incurred by the Resource Manager to allocate necessary packet memory and is referred to in total as protocol overhead. 1000 Lift runs were performed. All data were measured in wall clock time and modules in the protocol were measurement modules, having a measurement overhead of 0.6μs each. This results in an overhead of 9 μs for the packet allocation and the per-run Lift overhead, plus 0.4 μs for each module in the data path. The high maximum numbers stem from occasional context switches in the non real-time multitasking system. Therefore, the goal of achieving efficiency has been reached while remaining as modular as possible.

The total run-time overhead depending on the amount of memory requested for a flow with 3 modules is shown in Figure 14. As it can be seen, the memory management (using the standard C library) takes a constant 5 μs (difference between memory allocation of 0 Byte and allocation of ≥ 1 Byte), except for the first run, which takes additional 60 μs. The overall maximum value occurs the first time a buffer is requested, all future requests are handled fast. The first run also includes inherent

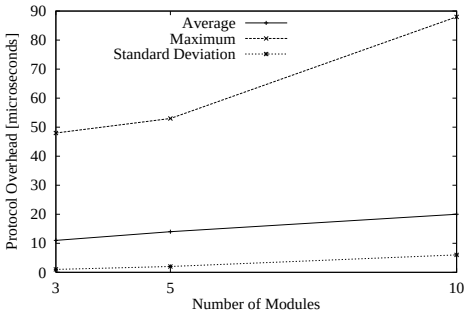


Fig. 13. Protocol Overhead Based on Modules

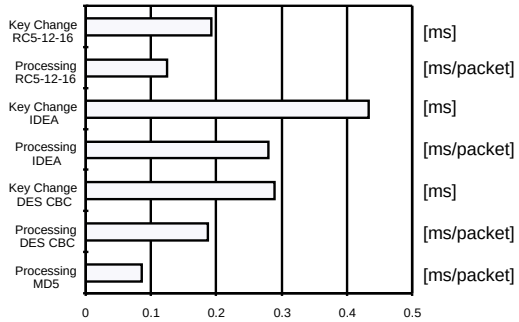


Fig. 15. Comparison of Security Modules

semaphore signaling overhead by the operating system needed to start the Lift thread blocked initially. It was considered important to reduce the high overhead inherent to handling multimedia devices before addressing the much smaller overhead related to module processing. It turned out to be impossible to reduce the multimedia device overhead, since specifications describing their operation could not be obtained in enough detail. The maximum value in the figure is again due to context switches. Note, the required time is independent of the requested memory size in terms of buffers. Only the initial setup time increases slightly with buffer size.

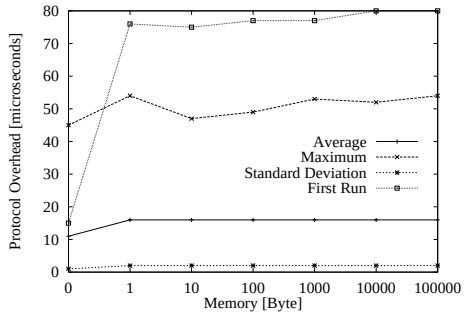


Fig. 14. Protocol Overhead Based on Memory

B. Security Modules

For the evaluation of the security performance, 1000 packets of 1000 Byte length each were sent using the TCP T-module over Ethernet connecting two Sun UltraSPARCs 170E as sender and receiver. For every 100 packets sent, a key change for the symmetric algorithm took place, while an RSA operation including their encryption and decryption was performed every 500 packets. The user CPU consumption of the authentication module Message Digest MD5 and the encryption module DES CBC are studied in detail, while further mechanism numbers are given for additional comparisons.

An overview of all numbers is depicted in Figure 15. Specifically, within the MD5 module the calculation of the MD5 checksum accounts for 97.6% of the CPU usage. 2.1% of the time was used for extracting keys, the rest is accounted by module specific overhead. The per-packet CPU usage for 100 packets without a key change is 0.086 ms. This corresponds to a theoretical throughput of 92 Mbit/s.

To perform the encryption and certification of transmitted session keys and to signal required control data to the Lift, 30.65 ms per key change are required, where the certification takes 93.48% of the time and the encryption of the session key with a peer's public key takes 6.47%. This behavior shows that operations using a public RSA key are much cheaper than operations using a private RSA key which is caused by the difference in time consumed by the modular exponentiation algorithm, depending on the number of 1-bits in the exponent.

Concerning the encryption module, the encryption of 999 packets of 1000 Byte length each with DES in CBC mode takes 199.71 ms. This includes 10 DES key changes, 0.73 ms each, and two refills of the pool of session keys holding 5 keys at a time. This takes 1.08 ms per refill. The per packet CPU usage without key changes amounts to 186.94 ms. This results in 0.187 ms per packet or in a theoretical throughput of the purely software-based DES implementation of 42 MBit/s.

C. Security Protocols

Different security protocols encrypting plain data have been evaluated by sending 10 MByte of data in 10,000 packets, determining the full end-to-end performance achieved. Keys have been changed every 100th packet and an asymmetric encryption operation (RSA) has been performed every 500 packets. Table V shows the real overall (application-to-application) throughput values that can be achieved using security in Da CaPo++.

TABLE V
ACHIEVED THROUGHPUT OF SECURITY PROTOCOLS

Security Protocol	Throughput
DES/MD5	5.87 Mbit/s
IDEA/MD5	4.19 Mbit/s
RC5-12-16/MD5	8.30 Mbit/s

These values include runtime, operating system, application, API and A-module overhead, as they are calculated from elapsed times. Normally multimedia data communication in Da CaPo++ would run even more efficiently, because data is transferred from the middleware directly to output devices and vice versa. Even when coming from the application, throughput is sufficient for multimedia data applications, *e.g.*, five encrypted CD-quality audio streams may be transmitted from a SUN workstation utilizing an RC5 with 12 rounds and 128 bit keys in conjunction with keyed MD5 authentication.

To perform the translation of abstract application requirements to low level requirements in the Da CaPo++ middleware, a way to predict resource consumption as a function of employed security algorithms needs to be found. The solution is a formula that can be fed by implementation and platform-dependent figures, resulting in the number of CPU seconds required for the encryption or authentication of a certain amount of data - including key change and internal processing overhead. For simplification purposes, the calculated resource consumption represents the maximum of the cost on the sending and the receiving side.

The formula below determines required CPU seconds per Mbit of processed data. π indicates the number of packets that are contained in a Mbit, P_{cost} represents the system inherent per-packet protocol processing cost (*e.g.*, 0.015 ms for the measurement environment), A_{cost} indicates the per-megabit module inherent overhead, ρ stands for the number of RSA key encryptions done per Mbit (not equal to the number of key changes, as several keys can be grouped together for one RSA operation), κ determines the number of keys that are grouped together, and κ_b defines for the number of bytes in one single key. RSA_{cost} stands for the cost of a single RSA operation (approximately 3.6 ms per Byte). K_{cost} and R_{cost} represent the cost for changing the key of an algorithm and the cost for gathering the random material used to form the key (about 1.3 ms per key).

$$\text{CPU}\left[\frac{s}{\text{Mbit}}\right] = \kappa P_{\text{cost}} + A_{\text{cost}} + \rho\kappa\kappa_b RSA_{\text{cost}} + \kappa K_{\text{cost}} + \kappa R_{\text{cost}}$$

Applying an example to this formula shows that the result depends on the number of packets per megabit, the number of key changes, and the key encryption/exchanges per megabit. Assuming 1000 Byte packets, key changes to be performed every 100 kByte, and RSA operations performed every 5 key changes, the following algorithm dependent cost result:

$$C = 125 \cdot 0.000015 + A_{\text{cost}} + 0.25 \cdot 1.25 \cdot \kappa_b \cdot 0.0036 + 1.25 \cdot K_{\text{cost}} + 1.25 \cdot 0.0013$$

TABLE VI
ALGORITHM COSTS AND THROUGHPUT

Algorithm	MD5	DES	3DES	IDEA	RC5-12-16
A_{cost}	0.0108	0.0234	0.0701	0.0380	0.0129
κ_b	16	8	16	16	16
K_{cost}	0	0.0007	0.0022	0.0002	0.0001
CPU [s/Mbit]	0.029	0.035	0.091	0.056	0.031
Mbit/s	35	29	11	18	32

To combine required costs for authentication and encryption, CPU seconds per Mbit values for authentication and encryption must be summarized. Table VI represents cost values and achievable middleware throughput as derived for the measurement platform of Da CaPo++.

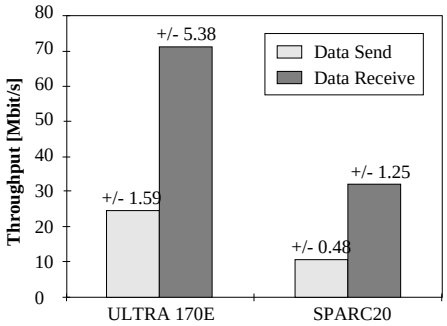


Fig. 16. Saturated Raw API Throughput

D. API Performance

The API plays an important role during connection establishment and data transfer. Control data are exchanged between the application and the Da CaPo++ core over a Unix domain socket by an IPC mechanism. User data exchange is supported by a shared memory concept [37] and data is either injected or received by an application in the upper part of API. Within the lower part of the API, the A-module either generates new data or consumes incoming data. Figure 16 depicts the maximum performance that can be expected for sending or receiving data over the API. These results were obtained by sending and receiving, respectively, 1000 packets of 1000 Byte size each. The difference between the maximal sending throughput and the maximal receiving throughput is due to the overhead in A-modules, since data coming from the application are available to the Lift after an additional thread-switch for accessing the call-back function from the Lift.

API measurements with varying packet sizes in the sending direction are presented in Figure 17. An almost linear relation between the packet size and the throughput is achieved, reaching the maximum for 8 kByte packets at approximately 108 Mbit/s. These figures are caused by the relatively large overhead due to semaphore operations of the shared memory which are an inherent problem of the applied operating system. The required time to copy larger packets via the C-library call `memcpy()` is not significant compared to these operations. Since multimedia data originating in devices bypass the API, these obtained numbers specifically determine the upper limit to the application-to-application throughput. The high degree of modularity applied to all threads (Lift, API) and processes (applications) could be reduced further to achieve an ever higher API throughput, however, implemented applications experienced a sufficient performance as these figures show.

The Da CaPo++ API throughput achieved has been compared to a number of different alternatives as depicted in Figure 18. The triangles show that the Da CaPo++ API performs very well ranging from 13.7 Mbit/s for 256 Byte packets to 274.2 Mbit/s for 8 kByte packets [37]. These numbers are in-line with Unix Sockets as well as Internet Sockets. Of course, a shared memory solution would give better performance, however, note that only user data not originating from a multimedia device must cross the API. Therefore, the API does not act as bottleneck for multimedia data transmissions.

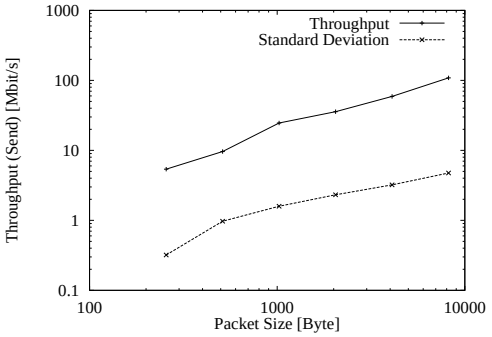


Fig. 17. API Throughput

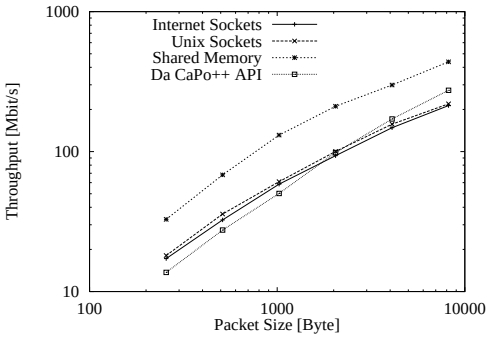


Fig. 18. Throughput Comparison of Different API Mechanisms

VII. Summary and Conclusions

The Da CaPo++ middleware is a comprehensive systems approach providing QoS-based multimedia services to applications. Its key characteristics and major results comprise

- The Application Programming Interface hides away complex protocol issues from the programmer, providing an abstract and QoS-based interface.
- Security and as not discussed in detail multicast are seamlessly integrated into the QoS specification offered by Da CaPo++.
- The Da CaPo++ approach provides valuable and well-adapted services and protocols to application programmers. By applying it in the implementation of a sample real-life picture phone application, a significant proof of concept has been presented.
- A prototype has been implemented and submitted to performance measurements. The results show that the implementation approach taken provides for a highly efficient protocol processing (Lift algorithm) which has been shown to fulfill soft real-time requirements, even when secure protocols are used.

The Application Programming Interface of Da CaPo++ offers the required degree of transparency between applications and the middleware. While the complexity of the Da CaPo++ core and of communication-relevant tasks is completely hidden from the application programmer, a useful exploitation is possible with QoS attribute specifications. Although breaking the transparency by handling application QoS within the Da CaPo++ core in the first place (applications are requested to specify their communication requirements for underlying "layers"), this offers an order of magnitude better alternatives in providing a best-suited communication protocol and service from the middleware's point of view. Even in case of QoS-ignorant applications, communication facilities are provided by the middleware relying on pre-defined standard communication protocols. The API abstractions developed show to be suitable and easy-to-use for application programmers providing QoS specifications. The support of efficient data transfers is achieved at the same time. Since multimedia devices and the middleware are tightly interconnected, applications do not require much effort for controlling these devices. Therefore, many difficult aspects of multimedia support are no longer part of the application, but are completely handled within the middleware. The general-purpose and QoS-based communication API offers a set of functions for communication purposes, where the flow types defined in the API are extensible and may be used naturally to generate objects and protocols required for communications.

Security and multicasting functionality is made available to users in the same way as they request for a reliable transfer of messages. The Da CaPo++ approach integrates security functionality into middleware by synthesizing security into additional QoS attributes. Thus, properties of secure protocols are as changeable as those of insecure protocols, provided that both parties agree on such changes.

Da CaPo++ is capable of accommodating in a tailored fashion a variety of multimedia application requirements due to its internal configuration facility for communication protocols and services. This flexibility achieved is fruitful for applications, however only required at the level of different protocols and for a group of protocol functions concerning security, multicasting, and error control. Experiences gained from the prototypical implementation reveal that protocol processing for the transmission of continuous data, *e.g.*, audio or video, can be performed with the Lift efficiently on standard workstations. Specifically, the exact number of concurrently supported data streams depends on their particular requirements, *e.g.*, in terms of security protocol functions. The Lift as a run-time system for modular protocols shows, in the given implementation environment, a minimum overhead of 9 μs for the packet allocation and a per-run Lift overhead of 0.4 μs for each module in the Lift's data path. This determines adequate protocol processing efficiency for a highly modular approach at the same time.

Concluding, the approach of supporting applications by advanced middleware, in terms of flexible protocol selection as well as QoS support, is a promising one. Its viability has been demonstrated by the design and implementation of the Da CaPo++ middleware. Further advantages of Da CaPo++ are concerned with its independence of the underlying operating system and the possibility to port the prototypical Da CaPo++ implementation easily. Even though the performance of Da CaPo++ is not optimal completely at a few places in its current implementation, specifically due to undesirable operating system interactions, the proof of concept for flexibly configured communication protocols has been furnished, and an efficient multimedia support on standard workstation's hardware has been accomplished successfully.

ACKNOWLEDGMENTS

The authors would like to express many thanks to their former ETH Da CaPo++ team members Christian Conrad, Toomas Tommingas, and Martin Vogt, to a group of diploma students, and to two project partners from former XMIT AG, Dietikon, now with Swisscom, and Swiss Bank Corporation, Basel, now UBS. George Fankhauser and Bettina Bauer commented on earlier versions of this paper.

REFERENCES

[1] H. W. Lockhart, *OSF DCE—Guide to Developing Distributed Applications*, McGraw-Hill, New York, U.S.A., 1994.

[2] Object Management Group, "CORBA: The common object request broker architecture," July 1995, Revision 2.0.

[3] "The TINA-C homepage," http://www.tinac.com, 1998.

[4] D. Rogerson, *Inside COM*, Microsoft Press, Redmond, Washington, U.S.A., 1997.

[5] R. van den Linden, "An overview on the advanced network systems architecture (ANSA)," Architectural Report AR.000.00, APM Ltd., Cambridge, England, 1993.

[6] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, W. Kachro, and A. Gokhale, "Applying optimization patterns to design real-time ORBs," in *5th USENIX Conference on OO Technologies and Systems COOTS'99*, May 1999, San Diego, California, U.S.A.

[7] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W.H. Sanders, D. E. Bakken, M. E. Berman, D.A. Karr, and R.E. Schantz, "AQuA: An adaptive architecture that provides dependable distributed objects," in *17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, West Lafayette, Indiana, U.S.A., Oct. 1998, IEEE.

[8] I. Foster and C. Kesselman, "The Globus project: A status report," in *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, Mar. 1998.

[9] D. Schmidt and T. Suda, "Transport system architecture services for high-performance communication subsystems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 489–506, May 1993.

[10] M. Zitterbart, B. Stiller, and A. Tantawy, "A model for flexible high-performance communication subsystems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 507–518, May 1993.

[11] L. Peterson N. Hutchinson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, Jan. 1991.

[12] D. Mosberger, *SCOUT: A Path-based Operating System*, Ph.D. thesis, University of Arizona, Tucson, U.S.A., 1997.

[13] D. Decasper, M. Waldvogel, Z. Dittia, H. Adiseshu, G. Parulkar, and B. Plattner, "Crossbow—a toolkit for integrated services over cell-switched IPv6," in *IEEE ATM Workshop*, Lisboa, Portugal, June 1997.

[14] D. Clark and D. Tennenhouse, "Architectural considerations for a new generation of protocols," *ACM Computer Communication Review*, vol. 20, no. 4, pp. 200–208, Sept. 1990.

[15] A. Danthine, "The OSI'95 transport service with multimedia support," in *Research Reports ESPRIT, Project 5341*. 1994, vol. 1, Springer, Berlin, Germany.

[16] A. Campbell, G. Coulson, and D. Hutchinson, "A Quality of Service architecture," *Computer Communications Review*, vol. 1, no. 2, pp. 6–27, Apr. 1994.

[17] K. Nahrstedt, *An Architecture for End-to-end Quality-of-Service Provision and its Experimental Verification*, Ph.D. thesis, University of Pennsylvania, U.S.A., 1995.

[18] S. Narayan K. Nahrstedt, H. Chu, "QoS-aware resource management for distributed multimedia applications," *Journal on High Speed Networking*, 1998, (to appear).

[19] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of QoS architectures," *Multimedia Systems*, vol. 2, no. 6, pp. 138–151, 1998.

[20] B. Stiller, *Quality-of-Service—Dienstgüte in Hochleistungsnetzen*, International Thomson Publishing, Bonn, Germany, 1996.

[21] D. Schmidt, "IPC_SAP: An object-oriented interface to operating system interprocess communication services," *C++ Report*, vol. 4, no. 8, pp. 1–10, November/December 1992, SIGS.

[22] S. Böcking, "Sockets++: A uniform application programming interface for basic-level communication services," *IEEE Communications Magazine*, vol. 34, no. 2, pp. 114–123, Dec. 1996.

[23] "WinSock2: Information, architecture, and specification," http://www.sockets.com/, 1997.

[24] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 protocol," Netscape Communications Corporation, http://home.netscape.com/eng/ssl3/, Nov. 1996.

[25] M. Purser, *Secure Data Networking*, Artech House, London, England, 1993.

[26] S. Casner and S. Deering, "First IETF internet audiocast," *ACM Computer Communication Review*, vol. 22, no. 3, pp. 92–97, July 1992.

[27] S. Deering, *Multicast Routing in a Datagram Internetwork*, Ph.D. thesis, Stanford University, California, U.S.A., Dec. 1991.

[28] D. Bauer, B. Stiller, and B. Plattner:, "Guaranteed multipoint communication support for multimedia applications," in *SYBEN'98 Broadband European Networks Conference*, Zürich, Switzerland, May 18-21 1998, pp. 395–404.

[29] B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, and M. Waldvogel, "Project Da CaPo++—Volume I: Architectural and detailed design," Tech. Rep. 28, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, July 1997.

[30] T. Plagemann, B. Plattner, M. Vogt, and T. Walter, "Model for dynamic configuration of light-weight protocols," in *3rd IEEE Workshop on Future Trends of Distributed Systems*, Taipeh, Taiwan, Apr. 1992, pp. 100–106.

[31] B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, and M. Waldvogel, "Project Da CaPo++—Volume II: Implementation documentation," Tech. Rep. 29, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, Aug. 1997.

[32] M. Vogt, T. Plagemann, B. Plattner, and T. Walter, "Eine Laufzeitumgebung für Da CaPo," in *GI/ITG Arbeitstreffen "Verteilte Multimediale Systeme"*, Stuttgart, Germany, Feb. 1993, pp. 3–17, K. G. Sauer, München, Germany.

[33] P. Zimmermann, *The Official PGP Users Guide*, MIT Press, Boston, Massachusetts, U.S.A., 1995.

[34] C. Conrad and B. Stiller, "A QoS-based application programming interface for communication middleware," in *22nd IEEE Conference on Local Computer Networks*, Minneapolis, Minnesota, U.S.A., Nov. 1998, pp. 274–283.

[35] W. R. Stevens, *UNIX Network Programming*, Addison Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1992.

[36] T. Gutekunst, *Shared Window Systems*, Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1995.

[37] C. Conrad and B. Stiller, "A QoS-based application programming interface for communication middleware," in *SPIE for the Voice, Video, and Data Communication Symposium*, Dallas, Texas, U.S.A., Nov. 1997, vol. 3233, pp. 248–259.

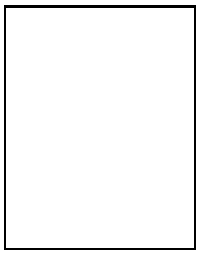
[38] B. Stiller, "An application framework for the Da CaPo++ project," in *5th Open Workshop on High Speed Networks*, ENST, Paris, France, Mar. 1996, pp. 4–17 – 4–24.

[39] B. Stiller, C. Class, M. Waldvogel, G. Caronni, D. Bauer, and B. Plattner, "The design and implementation of a flexible middleware for multimedia communications comprising usage experiences," Tech. Rep. 54, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, July 1998.

[40] Sun Microsystems, "SunVideo user's guide," 1994.

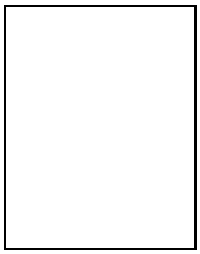
[41] B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall PTR, Upper Saddle River, New Jersey, U.S.A., 2nd edition, 1997.

[42] Pure Software, "Quantify user's guide," 1995.

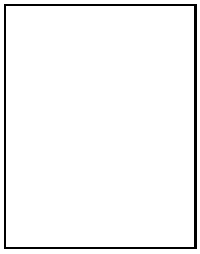


Burkhard Stiller received his diploma degree in computer science and his doctoral degree from the University of Karlsruhe, Germany in October 1990 and February 1994, respectively. From January 1991 until September 1995 he has been a Research Assistant at the Institute of Telematics, University of Karlsruhe, being on leave in 1994/95 for a one-year EC Research Fellowship at the University of Cambridge, Computer Laboratory, England. Since November 1995 he is with the Computer Engineering and Networks Laboratory TIK, ETH Zürich, Switzerland as a Lecturer for multimedia communications and a Research Associate.

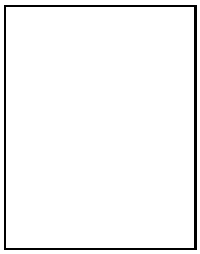
Dr. Stiller currently acts as technical program co-chair for the DSOM'99 and has served as a reviewer for journals, conferences, and workshops. Besides managing research projects in Germany, Switzerland, and the UK, his primary research interests include architectures for multimedia communication systems, middleware, Quality-of-Service models, charging and accounting systems, and ATM networking. He is a member of the IEEE, ACM, and the Gesellschaft für Informatik GI in Germany.



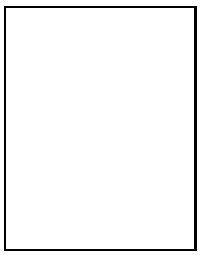
Christina Class received her diploma degree in computer science applied to business administration from the University of Mannheim, Germany in 1995. Since 1995 she has been with the Computer Engineering and Networks Laboratory TIK at ETH Zürich, Switzerland as a Research Assistent. She is currently completing the Ph.D. degree in the Department of Electrical Engineering at ETH Zürich.
Her research interests include communication protocols, multimedia, middleware, Quality-of-Service and synchronization of multimedia data. She is a student member of the IEEE and ACM.



Marcel Waldvogel received his diploma degree in computer science from ETH (Swiss Federal Institute of Technology) Zürich in 1994. He is currently working towards the Ph.D. degree at the Computer Engineering and Networks Laboratory TIK at ETH Zürich. His research interests include security and privacy issues in communication networks and distributed systems, algorithms for high-speed packet classification, and distributed storage systems. Marcel Waldvogel is maintaining the Swiss PGP key server. He is a member of the ACM.



Germano Caronni received his diploma degree in computer science from ETH Zürich in 1993. In the same year, he joined the Computer Engineering and Networks Laboratory TIK at ETH as a research assistant. Currently, he is per-suing his Ph.D. on QoS-based Dynamic Security. Since 1997, he is with Sun Microsystems, now with the Network and Security group of Sun Labs.
Mr. Caronni was one of the first to invent a process to watermark images, participated in the IETF (IPSEC), led the independent implementation effort for SKIP (secure TCP/IP) and its integration into an adaptive firewall. In 1997, he won the RC5/48 challenge of RSA Data Security Inc. His work and publications' focus are in the area of distributed systems and communication security. He is a member of the IEEE and ACM.



Daniel Bauer is a researcher at the IBM Research Laboratory in Zürich, Switzerland. He received his diploma degree in computer science from the Swiss Federal Institute of Technology, ETH Zürich in 1993. From 1993 until 1997 he worked as a Research Assistant with the Computer Engineering and Networks Laboratory, TIK at the ETH Zürich. In 1997 he received his Ph.D. in electrical engineering also from the ETH Zürich.
Dr. Bauer's research interests include routing, Quality-of-Service, distributed computing, multimedia, multicasting, and resource management.