The design and implementation of a flexible middleware for multimedia communications comprising usage experience

Author(s):
Stiller, Burkhard; Class, Christina; Waldvogel, Marcel; Caronni, Germano; Bauer, Daniel; Plattner, Bernhard

Publication Date:
1998

Permanent Link:
https://doi.org/10.3929/ethz-a-004287902 →

The Design and Implementation of a Flexible Middleware for Multimedia Communications Comprising Usage Experience

Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni[1], Daniel Bauer[2], Bernhard Plattner
Computer Engineering and Networks Laboratory TIK, Swiss Federal Institute of Technology ETH Zürich
Gloriastrasse 35, CH – 8092 Zürich, Switzerland
E-Mail: [stiller | class | waldvogel | caronni | bauer | plattner]@tik.ee.ethz.ch
[1] now with Sun Microsystems Inc., Internet Commerce and Security, Palo Alto, California, U.S.A., [2] now with Swiss Life, Management Services, Zürich

Abstract

Distributed multimedia applications require a variety of communication services. These services and different application demands have to be provided and supported within 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. The Da CaPo++ system presented in this paper provides an efficient middleware and application framework for multimedia applications, capable of handling various types of applications in a modular fashion. Applications' needs and communication demands are specified by values in terms of QoS attributes and functional properties, such as multicast groups, encryption or authentication requirements. Da CaPo++ automatically generates suitable communication protocols, provides for an efficient run-time support and offers an easy-to-use, object-oriented application programming interface. Its applicability for real-life scenarios was shown by various prototype implementations. Extensive performance evaluations have been carried out and practical experiences have yielded numerical results and conclusions.

1 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, have to be supported by well-suited communication protocols and efficient middleware hiding away the details of network technologies transparently. 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 and standard run-time environments for protocol processing are not able to cope with high data rates.

Therefore, emerging applications require various communication features that must be integrated and supported efficiently and which have been considered separately in traditional approaches. This is explained by an example. A video conference on public networks for confidential management meetings requires communication protocols providing appropriate encryption and authentication functionality in addition to 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 multicasting and security functionality in addition to traditional multimedia services.

Extending a multimedia middleware far beyond traditionally layered communication architectures has offered manifold opportunities for the provision of efficient multimedia communication services and avoids common design pitfalls for multimedia communications, such as low efficiency or dedicated functionality. Therefore, the following three requirements are made for communication middleware in general:

- 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.
- Usability: A homogeneous *Quality-of-Service-based (QoS) multimedia communication interface* is essential. It should offer the same performance as the underlying communication subsystem, i.e. not incur avoidable overhead. Additionally, the interface should be easy to use for application programmers and independent of specific multimedia applications.
- 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; in addition, the API should enable the application programmer to design re-usable components whenever possible.

These requirements imply major design goals of the Da CaPo++ communication middleware, while the following specific claims for the particular design, implementation, and evaluation of Da CaPo++ are made:

- Efficiency: The Da CaPo++ run-time system and its protocol processing algorithm – called Lift – represent a flexible processing scheme for *efficiently controlling modular protocol tasks* which is based on a standard workstation's operating system.
- Usability: A set of abstract interface functions of the *QoS-based Application Programming Interface* (API) achieves application transparency by assisting an efficient exchange of control and user data between applications and the middleware. In addition, this allows (1) for the new specification of certain functional requirements, such as various degrees of privacy or reliability of authentication, multicast group management, and addressing and, at the same time, (2) for 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.
- Modularity: Based on such QoS specifications, *modular communication functions and protocols* are selected flexibly, *e.g.*, for live audio, stored video, or plain data transfers, where protocols consists of building blocks. A series of various protocols and functions, particularly for security and multicast, has been implemented as prototypes and are integrated into the Da CaPo++ software environment.
- Real-life applicability: The middleware, including above mentioned features, has been utilized within the *application framework* offering by itself a modular structure. It has been implemented for real-life scenarios and applications, such as a tele-seminar and a picture phone, and it acts as a vehicle for performance evaluations under real-life conditions.

Concerning the overall operation of a communication, peer application processes communicate by a Da CaPo++ association which is established by middleware functions of the API. Specifically, the Connection Manager, the Security Manager, a Protocol Database, and the API cooperate to set up associations which are defined by specified QoS attributes and which, in turn, are supported by flexibly selected communication protocols and functions.

This paper is organized as follows. Section 2 briefly compares related work on various aspects related to middleware for multimedia communications. Section 3 discusses central components of Da CaPo++. While Section 4 shows its practical use, Section 5 evaluates obtained results. Finally, Section 6 summarizes the work and draws conclusions

2 Related Work

Due to the wide range of relevant topics that are integrated in the Da CaPo++ approach presented, a variety of different areas of related work is relevant. Four main aspects are dealt within Da CaPo++:

- Flexible communication middleware,
- Efficient run-time system support,
- Detailed QoS specification and exploitation, and
- Provision of advanced communication functionality.

Some selected approaches of related work have been categorized according to these main aspects and additional comparison criteria as depicted in Table 1. Flexible middleware is designed for communication subsystems to support adjustable protocol processing for high-performance applications and high-speed networks, as done within ADAPTIVE [32] or F-CSS [50]. Facilitating 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 can run in an efficient run-time system, the Lift algorithm.

For many distributed applications a strong architecture-oriented and less performance-sensitive approach of communication middleware covers transaction-based applications, directory services, location-independent services, or dynamic object invocation; examples comprise DCE [23], CORBA [26], COM [30], or ANSA [22]. However, the performance of middleware is crucial for multimedia-capable approaches and, therefore, constitutes an integral part of Da CaPo++.

Efficient run-time support for general protocol processing tasks has been investigated in, *e.g.*, the *x*-kernel for modular protocols [20], the Scout operating system for path-based module interconnections [24], and the Crossbow project supporting a high-performance toolkit for experimenting with IP next generation protocols [15]. In particular, accommodating the needs of communications by a suitable run-time system for fine-grained and inter-operating modules is essential for middleware that supports tailored communication protocols. In contrast to the Integrated Layer Processing approach (ILP) [9], Da CaPo++ favors a modular protocol processing approach which is integrated with the Application Level Framing (ALF) approach [9] to achieve good application performance.

Most of the existing approaches make available application knowledge to the middleware environment by providing an interface for the specification of QoS parameters. In OSI'95 [12], a QoS-based transport service including QoS parameter definitions was developed; the Lancaster QoS-Architecture [7] defined a QoS concept for end-systems, and the QoS Broker [25] investigated QoS management issues. A well-documented comparison may be found in [37]. These approaches allow for the detailed characterization of applications and the specification of their communication requirements on different levels, such as application-level, transport-level, or end-system-level. In addition, they are lacking open, extensible, and

Table 1. Comparison of Selected Related Work

Criteria	ADAPTIVE	F-CSS	CORBA	Scout	OSI'95	QoS-A	MCF	Da CaPo++
Multimedia Middleware	Medium	Medium	No	N/A[a]	N/A	Yes	Yes	Yes
Flexible Middleware	High	High	No	N/A	N/A	No	High	High
Portability	UNIX	Transputer	Various	UNIX	N/A	UNIX	UNIX	UNIX
Protocol Configuration	Flexible	Flexible	N/A	Flexible	No	No	Flexible	Flexible
Run-time System	No	No	No	Yes	No	No	Yes	Yes
Application Support	API	QoS-API	IDL	Path	Protocol	Interface	QoS-API	QoS-API
QoS Specification	Yes	Detailed	In progress	No	Yes	Detailed	Detailed	Detailed
QoS Management	No	Limited	No	Yes	No	Yes	Yes	Yes
QoS Exploitation	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Multicast Support	No	No	Yes	N/A	No	No	Yes	Yes
Security Functionality	No	No	No	N/A	No	No	No	Yes
Synchronization	No	No	No	N/A	No	No	No	Yes
Application Framework	No	No	No	No	No	No	No	Yes

a. N/A: This criterion is not applicable.

efficient Application Programming Interfaces, *e.g.*, in support of multimedia applications. Object-oriented interfaces for stand-alone systems have been studied, *e.g.*, in IPC-SAP [31] or Sockets++ [5] in addition to procedural ones, as WinSock2 [47]. Da CaPo++ integrates an open, QoS-based object-oriented interface with an end-system 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.

Furthermore, its communication functionality encompasses security, multicasting, and synchronization capabilities of modern middleware. Security issues are dealt with a number of approaches, *e.g.*, work for high-level network protocols such as the Secure Socket Layer [18] and a number of specific security algorithms and protocols. A good overview of security relevant policies and solutions may be found in [29]. Many algorithms deal with multicast communications, such as for Audiocast [8] and multicast routing [16]. A feature-rich and efficient multicast framework for end-to-end QoS guarantees for multipoint communications (MCF), is presented in [2]. A number of synchronization issues have been dealt with by various approaches which are discussed in [4]. However, it has not been well understood as yet how QoS requirements, security mechanisms, and multicast communication protocols inter-operate within one middleware environment. Thus, Da CaPo++ offers a new approach for handling security requirements as QoS attributes, integrating technology-independent multicasting, and providing synchronization mechanisms for multimedia data streams.

Concerning the variety of distributed applications, a spectrum of projects are concerned with the specific handling of multimedia applications. Among others, examples comprise transmission of continuous media via the World Wide Web [48], [34] or a number of video conference applications and video server work [3], [14], [17], [35], or [43]. Tele-teaching support in terms of workstation-based approaches encompass, *e.g.*, the BETEUS project [51]. As most of these applications require some effort to be integrated into broader scenarios, Da CaPo++ provides a modular application framework being capable of combing basic application building blocks into real-life application scenarios.

Generally spoken, 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 on standard workstations, including a close cooperation between applications, the API, security and multicast, QoS concepts, and the communication middleware itself.

3 Da CaPo++ Middleware

The Da CaPo++ middleware is end-system-based and located between the network access and applications (cf. Figure 1). While the low-level middleware is capable of directly utilizing different underlying network services, the high-level middleware provides modular and re-usable application components for the application framework which in turn involves the flexible definition and implementation of applications and scenarios. The Application Programming Interface (API) interconnects both middleware levels. The middleware as well as the API supports multimedia communications, since multiple time-dependent media flows in addition to native data flows being part of a multiple-flow session can be processed on standard workstations. This is due to the middleware's good performance and its provision of appropriate functions.

The Da CaPo++ middleware core provides the possibility to flexibly configure end-system communication protocols built out of protocol functions according to application requirements expressed in terms of QoS parameters [39]. The configuration process is directly supported by a number of internal Da CaPo++ components (cf. Figure 3). In addition, it is based on application requirements, availability of local resources, and network prerequisites as well as protocol functions and mechanisms including their properties. These protocol functions, *e.g.*, checksumming or flow-control, are processed by individual communication modules (later referred to as C-modules) being located in the heart of the protocol. Application components access communication services of configured communication protocols via the API through application support modules (A-modules) including the direct multimedia data support. This is achieved by integrating the end-system architecture in terms of data producing or consuming devices into the Da CaPo++ design, such as for multimedia devices cameras, microphones, or speaker boxes.

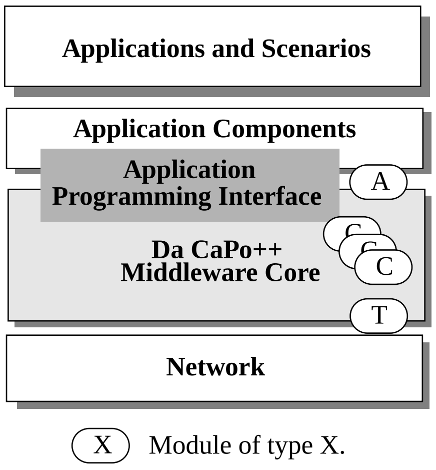

Applications and Scenarios
Application Components
Application Programming Interface
A
Da CaPo++ Middleware Core
C
C
C
T
Network
X
Module of type X.

Figure 1: Middleware Architecture

On the network side of Da CaPo++, available networks in terms of an ATM (Asynchronous Transfer Mode) and an Ethernet-based Internet is utilized, particularly offering certain levels of guarantees for network performance, such as bandwidth guarantees. As an application does not need to care about differences in transport mechanisms used, properties and especially semantics of different transport mechanisms are hidden away. This level of abstraction is provided by transport modules (T-modules) being part of the configured communication protocol.

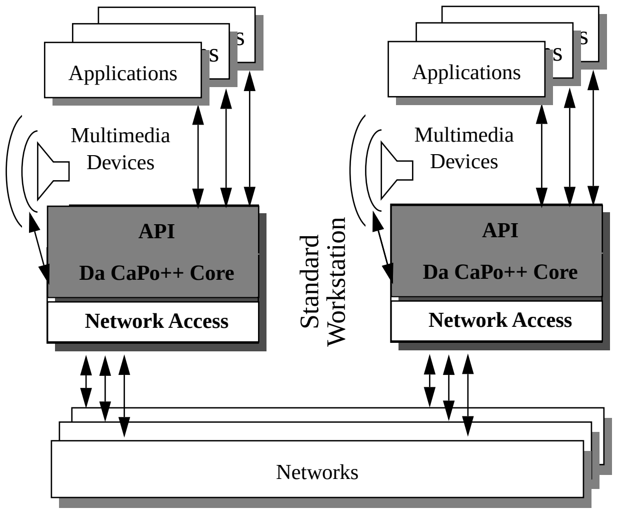


Figure 2: Overall Communication Architecture

In its implementation the Da CaPo++ middleware covers end-system issues on standard workstations, common multimedia devices, and applications on top (cf. Figure 2). The Da CaPo++ core and an API resides once per workstation, while multiple applications may utilize the middleware. Different network scenarios and middleware internals are transparently dealt with [40]. To accommodate diverse network technologies, QoS specifications are used in the API and the core as powerful abstractions, enabling application programmers to ignore specific properties of transport mechanisms within the Da CaPo++ core and the network access. Commitment to these abstractions especially influenced the design of modules. In addition, most applications and protocol modules of the core also do not need to care how network-dependent unicast or multicast or end-system-internal security services are internally implemented. The API determines the appropriate level of abstraction.

3.1 Da CaPo++ Core

The Da CaPo++ core is the central component of the middleware, performs all functions related to session management and data transfer and is an evolution of the original Da CaPo system [27]. Its central goal is to take as much burden off the application and the programmer as possible 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:

- The application names the source or sink for data and specifies communication requirements. It may also choose among predefined protocols offered by the protocol database, instead of specifying parameters itself.
- The application identifies the communication peer, requests the establishment of an association, and starts the data transfer for flows and sessions.
- After that, data transport is performed independently of the application, freeing it from further transmission control, *e.g.*, to perform user interaction.
- Whenever important communication events happen, the application is notified so it can take appropriate measures. The application can also query and modify the state of a session at any time.
- When transmission is finished, the application requests a session tear-down.

All important details of these tasks and additional internals of the middleware core are discussed below. This section on the Da CaPo++ core introduces how sessions are configured and set up. Afterwards, the data transport mechanism and module concepts used in Da CaPo++ are explained. Finally, it gives an overview of the protocol database and resource management facilities used.

3.1.1 Sessions and Parameter Specification

Upon an application's request to the Session Manager, sessions are created in a modular fashion (cf. Figure 3). They form bundles of unidirectional data flows which are 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 an analogous pair of video flows.

Each of these data flows correspond to a specific protocol, created out of a sequence of protocol modules, each performing a specific protocol function (Figure 4 depicts a number of examples). This approach for abstraction allows for a high flexibility in that the application does not need to know any details of how data is transferred from the source to the sink, but still can influence whatever is needed. For example, an application might request compression and does not care about how the video stream is compressed, as long as a specified minimum quality is met, but it might request a specific encryption scheme with specified parameters.

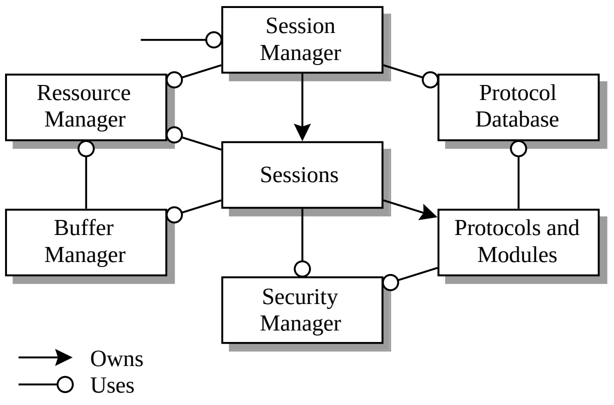


Figure 3: Da CaPo++ Component Relations

Each session includes a reliable flow which is used at session set up 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.

All modules performing the necessary service are selected and configured according to “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 which 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 all requirements specified may conflict with or have influences on each other, or may not even fully specify the system, a 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 whether configuration succeeded and what values have been selected.

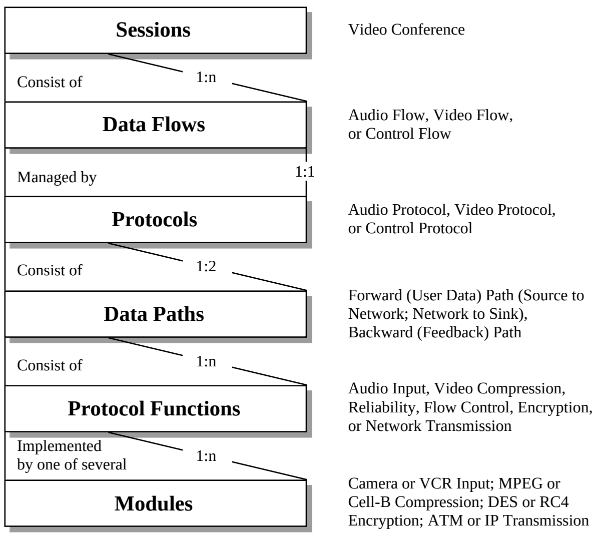

Sessions
Video Conference
Consist of
1:n
Data Flows
Audio Flow, Video Flow, or Control Flow
Managed by
1:1
Protocols
Audio Protocol, Video Protocol, or Control Protocol
Consist of
1:2
Data Paths
Forward (User Data) Path (Source to Network; Network to Sink), Backward (Feedback) Path
Consist of
1:n
Protocol Functions
Audio Input, Video Compression, Reliability, Flow Control, Encryption, or Network Transmission
Implemented by one of several
1:n
Modules
Camera or VCR Input; MPEG or Cell-B Compression; DES or RC4 Encryption; ATM or IP Transmission

Figure 4: Session Hierarchy

During the development and testing of the Da CaPo++ middleware, it turned out that this level of flexibility is seldom needed, but causes a high configuration evaluation overhead at session setup time (each module may specify pre- and postconditions as requirements, *e.g.*, that it needs a reliable transport below) and may cause indeterminism. 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 in 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. More details on the database are discussed in Section 3.1.4.

3.1.2 Lift

Protocols determine the middleware's view of application flows. Flows are split into two Data Paths (cf. Figure 4) for the forward (data and control) and backward (feedback) direction. Therefore, they are always bi-directional. Bulk 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 encompass usually small amounts of data. Although Figure 4 may suggest that there is a lot of hierarchical overhead involved, this overhead is negligible. A limited amount of overhead occurs at session setup and no penalties are to be paid at run-time, where protocols' selected module instances are directly accessed.

In each flow, data is transported by an algorithm called Lift, an active transport mechanism, originally started in [44]. Once started, it works autonomously calling modules' processing functions in turn 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 working per protocol runs in a separate thread – frees other system parts from duties (cf. Section 3.1.5 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 all local variables. When modules return the control back to the Lift, only minimal module state needs to be saved. However, this simplified thread switching is not taken advantage of currently, as the underlying operating system (Solaris 2.5.1) handles intra-process thread switches efficiently. Changing thread-switching behavior would also have tied Da CaPo++ much more closely to a single operating environment than we considered useful.

Concerning memory requirements of modules, communication packets for a complete packet contain a fixed header, where each module owns its own part, plus 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 deducted 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 3.1.6).

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 3.4). The A-module can directly 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).

3.1.3 Object-oriented Module Implementation

To achieve the desired modularity of the middleware, the entire Da CaPo++ core is object-based. This is most noticeable with modules: neither the module selection and configuration components, 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.*, name, and a list of function pointers to call and perform well-known functions. Additionally, they have an assigned partner module, the one 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, based on what it will expect next. 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 (cf. Section 4.2).

Instances can not only find out about their class or other instances of the same class, they can also find out about any module within their Session, both on the local and remote site. After having found them, 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.

Besides internal test, protocol trace/debug, measurement, and traffic generation modules, a number of real-world 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.

3.1.4 Protocol Database

A module alone usually is not of much use in a protocol. It needs at least a corresponding peer at the other end of a connection. 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. In Da CaPo++, these four parts are treated independently (cf. Figure 5). A forward path consists of a "Down" part in the sender transporting (usually a lot of) data towards the network and a matching "Up" part in the receiver and 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 also has its separate Lift thread caring for the data transport, giving a total of four Lifts, two in each process.

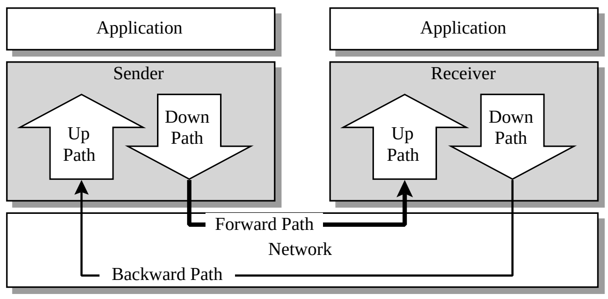


Figure 5: Modules in a Mechanism

Each of these paths is implemented as a series of individual modules in Da CaPo++, although usually the matching Up and Down modules in the sender or receiver 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, up and down, or forward and backward.

A sample protocol definition for a secure video transmission is shown in Figure 6. More information on security modules mentioned can be found in Section 3.3. 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 given, because it was originally planned to allow for the parallel execution of independent protocol functions. This has not been implemented fully, to avoid synchronization overhead between parallel threads.

```
{
 {"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}
}
```

Figure 6: Sample Protocol Data Base Entry

3.1.5 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 7). 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.

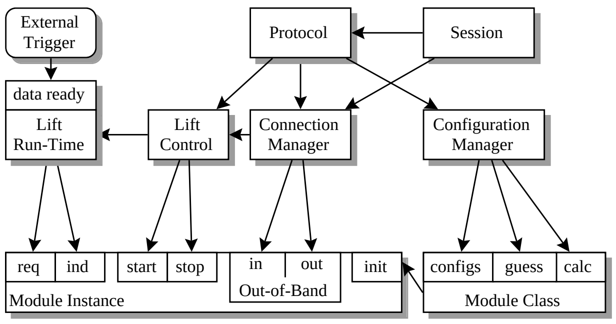


Figure 7: Module Interfaces

At session setup, the Connection Manager (a regular communication protocol, but with special duties) and all other requested protocols are created. 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' `request` and `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 signalling the forward path that it has finally received the acknowledge.

Return values of the `request` and `indication` 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 signalling that out-of-band data is ready, the Connection Manager will pick up the data by calling the `out` function and will deliver it through the network to the matching module's `in` function.

All `request` and `indication` 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 order of packets.

3.1.6 Resource Management

One of the main goals for designing the core was to reduce data copying as much as possible. 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 variable-length data that is to be transmitted between A-modules located at peers. Another Buffer holds the constant-sized header, where all modules can communicate information to its partner module at the receiving peer. A third Buffer is used for communications between modules in a single data path and is not transmitted over the network. Packets and Buffers are managed by the Buffer Manager which also keeps track of reference counts for each Buffer. This allows modules to keep a copy 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 and zero-copying of Application Layer Framing [9] even for segmenting/reassembling.

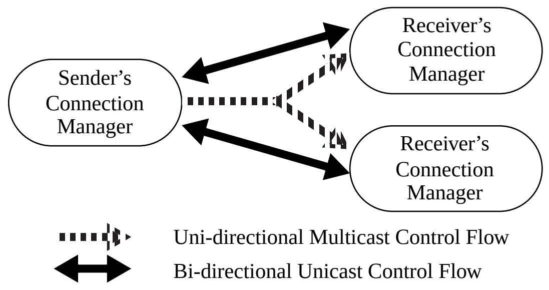
An excellent approach on minimizing copying operations is presented in [6]. Within Da CaPo++ this can not achieved directly, as the Da CaPo++ core runs in the UNIX user space and at least one copy operation between user and kernel space (network access) is required. Integrating Da CaPo++ into the UNIX kernel together with the recent introduction of multi-media mbufs of [6] would result in real zero-copying, but would require a major rewrite of the Da CaPo++ core.

The Buffer Manager – and all other components requiring system resources – request these resources from the Resource Manager (cf. Figure 3). It provides an abstraction layer for memory, CPU resources (in the form of threads), and timers. Memory and threads are directly passed on to the operating system, while timers are multiplexed, since the operating system chosen, Solaris 2.5.1, only supports a limited amount of timers. This reduces memory management overhead and minimizes, if not avoids, memory copying.

3.2 Support for Multipoint Communications

The multipoint communication support implemented in Da CaPo++ allows for point-to-multipoint flows with dynamic join capabilities. A multipoint flow is always set up by the single sender. Receivers may join and leave a multipoint flow at any time. Dynamic joins are mainly useful, when multimedia data is being distributed, where data integrity is not violated by late joining receivers. However, since applications may block dynamic joins, they may implement any form of data integrity.

Multipoint communication is supported in Da CaPo++ by a multipoint-capable Connection Manager and a set of multipoint-capable protocol modules. The Connection Manager is responsible for the compatible protocol usage by all participants of a multipoint session. In order to allow for dynamic joins, a centralized approach has been implemented, where the sender's Connection Manager administers all flows of a multipoint session. Receivers join a flow by requesting flow parameters, such as protocol and QoS parameters, from the sender's Connection Manager. These parameters are used to instantiate the respective protocol stack. If some flow's QoS parameters are changed during operation, the sender's Connection Manager distributes them to all participants. It distributes control information reliably to all participants using a uni-directional multicast and a bi-directional unicast flow (cf. Figure 8). The unicast flow is based on TCP, whereas the multicast flow uses its own error-control protocol based on negative acknowledgments.



Sender's Connection Manager
Receiver's Connection Manager
Receiver's Connection Manager
Uni-directional Multicast Control Flow
Bi-directional Unicast Control Flow

Figure 8: Control Data Flows for Multicasting

Da CaPo++ contains three multicast T-modules. They hide away details of the underlying network infrastructure and provide access either to UDP/IP multicasting, TCP multicasting, or native AAL5/ATM multicasting. This involves initializing communication end-points and forwarding data units. In addition to these functions, the ATM module also implements the dynamic join capability by adding new receivers to existing ATM connections. Additionally, a reliable and an unreliable multicast transport protocol module are available, both implemented as C-modules and being capable to run on top of any of the two T-modules. The unreliable multicast protocol is used for a transport protocol appropriate for audio and video services in addition to segmentation and reassembly functionality of multimedia packets. However, the reliable multicast protocol uses an error-control mechanism based on retransmissions for assuring the correctness of data [1]. In order to avoid a packet implosion at the sender, a negative acknowledgment scheme is being used. Receivers detect loss of packets by two methods. Firstly, they compare sequence numbers in arriving packets. If a gap in these numbers is detected, missing packets are requested from the sender. Secondly, a different method uses time-outs. If the sender has no data to send, it sends a heartbeat packet containing the last sequence number only. Since heartbeats are sent in predictable intervals, receivers detect a packet loss, if no packets arrive during a heartbeat interval.

3.3 Security Functionality

In the Da CaPo++ middleware 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 volume, computational resources required to provide the highest level of security usually exceed available CPU power. For this reason, the amount of security (the strength and, therefore, computational requirements of employed protocols) provided is variable, depending on user demands.

Privacy and authenticity of communications are as much considered a basic service quality parameter of the network as packet loss rate, bandwidth, and delay. Together with C-modules, which provide the actual data security, the Security Manager (cf. Figure 3) 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) [49]. For integration purposes, PGP has been changed into a library and linked into the middleware.

3.3.1 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 3.4). During the delegation of identity, the application indicates which keys in the database are to be used and provides a passphrase to unlock them. This information (cf. Figure 9 for details on data structures shared between the Security Manager and the Da CaPo++ Application Programming Interface) is passed from the application via the API to the Security Manager. In practice, only the provision of a key ID for a public/private key pair and the corresponding passphrase is required to prove ones identity to the middleware. The middleware then involves 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 the capture of the passphrase as it is transferred from the user to the middleware.

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 in the Da CaPo++ middleware.

Admission control of communicating peers is done by the application, because it has been found that criteria for admission control are too varied to be efficiently delegated to the middleware. Upon arrival of a new peer, the Security Managers of the involved middleware exchange (1) certified keys and (2) additional authenticating material forming an authentication hierarchy in the form of "remote user before remote application before remote middleware" which can be forwarded to the application. If the application allows the connection, communication begins.

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

Figure 9: Application Authentication Data Structure

3.3.2 QoS Translation

Security parameters can be controlled as application requirements. Security parameters in Da CaPo++ express requirements on four separate layers: (1) UR: user, (2) AAR: abstract application, (3) LLR: low-level, and (4) IR: 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, memory consumption, or network bandwidth.

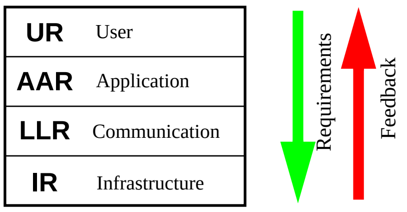


Figure 10: Abstraction of Requirements

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 can not 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 that statically demand it would need to be changed.

To address this problem, low-level requirements are derived from application and user level requirements, as outlined in Figure 10. The application can specify the required strength of 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, and how much the information is presumed to be worth. The goal is to make the system provide security mechanisms which cost the attacker more to break than the information itself is worth. Additionally, a probability is given to specify how likely these promises should be met.

This type of parameters can be determined usually more easily by the user than the low-level requirements. They 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. 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 done once by the developer or administrator of the middleware and not by every application programmer or even 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 the adding of new and improved encryption algorithms without user or application programmer involvement.

3.3.3 Security Assurance

The Da CaPo++ middleware allows for modification of security parameters in an ongoing communication. In this reconfiguration, cryptographic algorithms can be switched off or changed 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 how much data and for how long a traffic key has been in use. Whenever necessary, 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 which provides key management and a variety of encryption and authentication functions to flows, sessions, and protocols. It has been determined as the goal to make security functionality available to users in the same simple as the request of reliable transfer of their messages which has been achieved.

3.3.4 Security Modules and Protocols

While QoS mapping mechanisms for user and abstract application requirements have not been implemented completely, the Da CaPo++ middleware offers different security modules and protocols, allowing to show their usability in the context of multimedia protocols. Modules for key agreement, privacy, and authentication are provided (see Table 2). 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 is to be provided.

Table 2. 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 CBC modes, No. of rounds and key length for RC5
Privacy (stream cipher)	RC	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 those 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 cryptographic mechanisms off completely.

3.4 Application Programming Interface

The Application Programming Interface (API) for communication services is the only interface visible to application programmers in end-systems above the middleware. Since data streams may vary according to their type, location, and origin of data, two basic abstractions for application data streams, so-called flows and sessions have been designed (cf. Section 3.1.1) allowing for hiding all communication protocol specific features [11]. In addition, basic operations for dealing with Quality-of-Service (QoS) have been introduced [27]. *E.g.*, although 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 4.1. However, the application always handles communications in an association-based manner, where the API handles association context information (*e.g.*, including session identifiers or current participating flows) and the underlying protocols may provide a connection-oriented or connectionless service.

An important difference is encountered for live data, originating from multimedia devices, and data from applications. As the BSD socket interface [36] 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 or a multimedia server (cf. Sections 4.2 and 4.3) 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 multimedia application, only the less expensive control of devices – in terms of the amount of data – still remains under the responsibility of applications which 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. Table 3). This process is similar to opening and binding a BSD socket with options. Secondly, a set-up process is involved to establish a connection 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 keeps in mind issues, such as application QoS requirements playing an important role not only during the establishment phase (including configuration and reservation), but also during run-time (QoS re-negotiation).

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 functionality 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.

As the complete API is discussed in [11], main interface functions as required for the application framework's discussion in Section 4 for sessions are listed in Table 3. These functions are applied from the application programmer to utilize the Da CaPo++ middleware.

Table 3. Session-level API Interface Functions

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.
`FlowLeave()`	An existing flow dynamically leaves the session (deallocation of resources).
`Close()`	The session is terminated and all resources are de-allocated.

As a general API 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 API (belonging to the high-level middleware as depicted in Figure 1) and the middleware as one "API server process" offering the lower part of the API (belonging to the low-level middleware, cf. Figure 1). Multiple API clients, one for each application, reside in a multithreaded 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. Direct 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 5).

4 Application Framework

To validate the concept of Da CaPo++ multiple multimedia applications have been implemented in an object-oriented manner using the API (cf. Section 3.4) and the Da CaPo++ core (cf. Section 3.1). These applications exploit the flexibility of the Da CaPo++ middleware by requiring various different protocols.

Protocols utilized for applications encounter (1) live or stored audio, including a unicast or multicast transmission either on top of IP or ATM, (2) live or stored video, including a unicast or multicast transmission either on top of IP or ATM, as well as (3) encrypted or unsecured data. Moreover, data encryption has also been integrated into multimedia applications using a secure audio protocol. All applications have been developed easily on top of Da CaPo++ and cover a broad range of applications from a simple picture phone to a complex teleseminar scenario.

As control mechanisms and user interfaces for different data and connection types may be reused in different application types, the design of applications is highly modular to allow for an easy reuse of components. As Figure 11 shows, the defined *application framework* consists of three levels: (1) 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 belongs to the topmost part of the middleware. (2) An *application* consists of one or more application components and offers a single, homogenous functionality which is provided in close cooperation by the application components. *E.g.*, in this sense a picture phone determines an application. (3) 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).

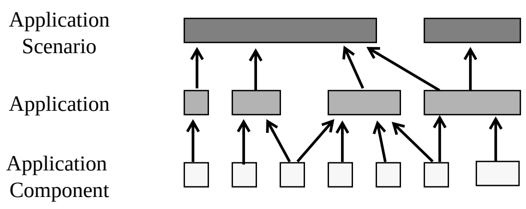


Figure 11: 3-Level Application Framework

For example Figure 12 depicts the structure of the teleseminar scenario which has been implemented utilizing the Da CaPo++ middleware and illustrating the flexibility and performance gained. The teleseminar scenario (cf. Section 4.5) is built out of two applications. The document sharing application implements a shared view on a common document together with a suitable floor control to allow for jointly editing the document. The video conference application (cf. Section 4.2) enables the exchange of audio and video data between participants. This application currently is built of two application components, the audio/video application component controlling the presentation of audio and video data streams and the multicast support component implementing n:m multipoint-to-multipoint communication (cf. Section 4.2). The synchronization control component (cf. Section 4.6) could be integrated into the video conference to offer better data quality to the user. The document sharing application consists of one application component, the application sharing component. This component is also used within the multimedia server (cf. Section 4.3) showing the re-usability of defined components.

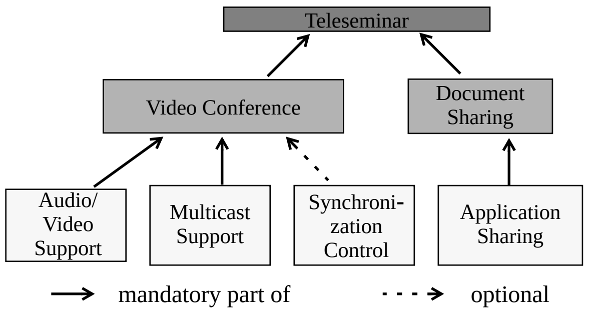


Figure 12: Structure of the Teleseminar Scenario

4.1 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. All four protocols are combined within one Da CaPo++ session. The picture phone has been implemented simply by using API functionality to create and connect sessions (cf. Section 3.4) for data transmission and by providing control functionality to the user in form of an appropriate graphical user interface.

The configuration file for the picture phone session in the creator, *i.e.* the caller, is depicted in Figure 13. The specified session determines a unicast session, while it consists of four flows, each for sending and receiving audio and video respectively. Every flow is assigned a type, *e.g.* video_recv_device. This type specifies the source/sink of the flow as well as the direction of the flow, in this case the session creator is a receiver. In this example, data is directly sent to the device and not passed via the application (cf. Section 3.4). Depending on the flow, on end-systems, and on the communication medium available, QoS parameters are specified for every flow. These may encompass, *e.g.*, throughput, frames per second, samples per second, bits per pixel. The communication protocol is configured based on the configuration file and instantiated by the Da CaPo++ core (cf. Section 3.1) according to the specified QoS.

The example of Figure 13 specifies two values (maximum and minimum) per parameter for audio and video data. In the example configuration file, the communication requests for different delay and jitter characteristics in both directions, specifying an asymmetric communication. The video sent to the peer requires less quality than the video received. This can result from a sender's camera being less powerful as it can capture only frames of a lower quality.

```
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
```

Figure 13: Picture Phone Configuration File

Concerning the implementation of the picture phone, API session-level methods (cf. Section 3.4) are applied. Figure 14 depicts an extract of the picture phone's implementation. A Da CaPo++ client object created once per application (line 1 in Figure 14) receives security information (cf. Section 3.3.1). Afterwards, the client object `client` has to be passed to the constructor of the session object as well as the name of the session configuration file.

```
/* create DaCaPo++ 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 ();
```

Figure 14: Applying the API to the Picture Phone

To connect a session, specified information like the peer (in case of multicast this information is not required), the personal address, and interface are required. This information is handed over in a C-structure `connectFile` to the connect method of the session. The connect method (line 3) also requires a pointer to a callback function `callBackFct` for a session. Callbacks from the middleware to applications are delivered on a session base.

The `Activate` (line 4) and `Close` (line 6) methods of the session object do not require parameters. To call the deactivate method, the deactivation type has to be specified. In this example it is a `GRACEFUL` deactivation (line 5), *i.e.*, the Lift algorithm delivers all data pending in the middleware for this session to participating modules before the session is stopped. Deactivating the session instantaneously would result in a graceless deactivation of all participating flows.

The Da CaPo++ picture phone offers a user interface implemented in Tcl/Tk. Several buttons allow for connecting, starting, stopping, and pausing a picture phone. Related button functions are implemented as in line 3 to 6 as Tcl commands using method calls.

4.2 Video Conference

The video conference application scenario supports audio and video communication between multiple parties. The number of receivers and senders may vary, *e.g.*, people may listen only to the conference without being involved actively speaking. A similar A-module as in the picture phone can be used to capture and present audio and video data. In addition to the picture phone, audio A-modules are required to offer the functionality of audio mixing, as the audio device on the SUN workstation only supports one reader and one writer at a time. As the video conference is a multiparty application, multicasting over supported networks is used for data transmission by adapted live audio and video protocols. Unlike in a Da CaPo++ unicast session, in a multicast session all flows either send or receive data (cf. Section 3.2). Therefore, the conference has to create for each participant (1) as many receiving sessions as other participants send audio and video data, and (2) a sending session, if the participant takes an active role in the video conference.

As these sessions are required in all n:m-applications, the multicast session creation and administration functionality is implemented within the application component multicast support (MCS). A multicast control session is created between all participants. Information on joining and leaving participants is transmitted within this session. Pure data protocols of the Da CaPo++ middleware are used to implement the application protocol for the multicast support application component. The communication between the multicast support components itself is supported by the Group Management System (GMS) [46], which is not fully integrated into Da CaPo++. The multicast component extends the offered multicast service of Da CaPo++ to n:m-multicasting, inclusively allowing for dynamic joins and leaves. Applying the full API functionality to Da CaPo++ multicast audio and video protocols in combination with the multicast support application component, the video conference has been implemented with minimal effort by additionally integrating a user interface for control.

Figure 15 shows the example of a video conference with three participants. Each participant sends its audio and video data to the two other participants via a Da CaPo++ multicast audio and video session and it receives data from its participants in such sessions. Data capture and data presentation are controlled by audio and video (A/V) application components. The multicast support component together with the three instances of the A/V component are combined to the video conference application for three participants. This application can be used stand-alone as an application scenario or can be integrated into more complex application scenarios, *e.g.* a teleseminar.

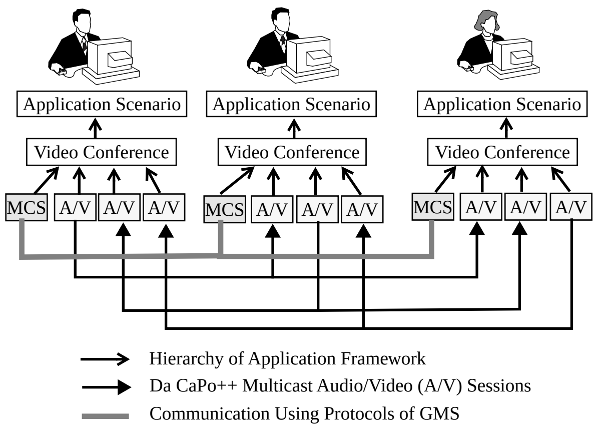


Figure 15: Video Conference with 3 Participants

4.3 Multimedia Server

Multimedia audio and video data may be stored and transmitted to users. This transmission can be unicast or multicast depending on the number of receivers. A multimedia service has been defined enabling the retrieval of multimedia data being stored on a multimedia server.

This service has been defined such as to offer a very flexible, general, and powerful service to the user by hiding away all details on connections, QoS, and multimedia and back-up servers. Figure 16 shows the Da CaPo++ multimedia server scenario that will be explained in this Section. To offer the transparent access for users to different servers and back-up servers, if necessary, an application protocol has been defined. The application protocol uses a pure data A-module to exchange the protocol information. The multimedia client requires information on the requested data that is specified in a specification file which is written in a specific file format. The specification file defines separate alternative service specifications. Each alternative specification comprises (1) the server to connect, (2) the multimedia application type for the local configuration of the multimedia client, as well as the (3) session and flow specification including the QoS definitions for the service. This allows for the definition of back-up servers for the same application in form of different alternative specifications as well as for the specification of different services with decreasing quality. If the high quality service cannot be provided, *e.g.* the specified server cannot accept any more clients, the multimedia client still receives a sufficient service of lower quality if specified in another alternative.

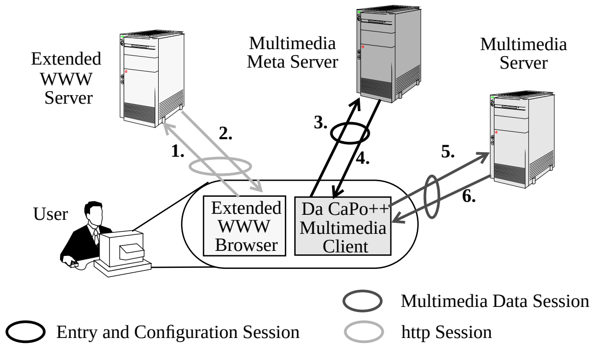


Figure 16: The Multimedia Server Scenario

The specification file is evaluated in the client and relevant information, e.g., the Da CaPo++ protocols and the local configuration of the client, is saved for local use. Afterwards it is sent to the multimedia metaserver (depicted 3. in Figure 16) which in turn provides information on the address of the multimedia server (4.) that is to be used. Information on the communication protocol are retrieved from the QoS specification and the configuration information from the configuration specification in the specification file. This data exchange is defined as entry and configuration session.

Based on the information provided by the server, the multimedia client connects to the multimedia server (5.) and receives data being presented by A-modules for stored audio and stored video (6.). The transmission in the multimedia data session applies audio and video protocols specifically based on the detailed service specification. Applying Da CaPo++ out-of-band communication which allows for sending of control information from receiving to sending A-modules, multimedia control commands as play, stop, and fast forward have been implemented. Table 4 shows control commands that are implemented for the video server. To provide different implementations of the control command, an A-module is to be implemented with the new functionality. The application can access this functionality by simply requesting during communication setup a protocol containing this A-module.

In case of a video service, the multimedia client has been combined with an application component providing application sharing functionality [19] to allow for a user controlled join of additional video clients. The second video client takes part in the video retrieval initiated by the first client and the user interface is shared providing floor control. Due to performance reasons, application sharing is not used to share video data, but the second video client dynamically joins the multicast video flow. An application protocol has been defined using data protocols of

Table 4. Control Commands for the Video Server

Command	Video Server Actions
OPEN	The requested video file is opened to be read.
PLAY	Video data transmission is started or resumed (default is normal play-out, but if FF or FR have been specified transmission is performed for the requested play-out mode).
PAUSE	The data transmission is interrupted, it is resumed with the PLAY command.
STOP	The data transmission is stopped and the session is closed.
FWD	A forward is performed, *i.e.* the current play-out data is set to the last frame of the video data.
FF	Fast forward is provided. It is implemented in the prototype by displaying every10th frame.
RWD	A rewind is performed, *i.e.* the current play-out data is set to be the first frame of the video data.
FR	Fast rewind is provided. In the prototype it is implemented to display every 10th frame in the reverse order.

Da CaPo++ to exchange required information between the first and second video client. The implemented multimedia clients can be started from a WWW Browser using the external viewer concept. The so-called extended WWW browser connects to an extended WWW server (depicted 1. in Figure 16) and retrieves the specification file (2.). This allows for applying well-known service interfaces to start multimedia data retrievals.

This application scenario shows that Da CaPo++ provides the means to implement very flexible and general services by hiding away complexity, but offering high quality in both, services and communications.

4.4 Secure Messaging and Secure Audio

Two additional application scenarios have been developed to demonstrate Da CaPo++ security facilities integrated into the middleware. Secure messaging provides a secured data communication between participants. The secure data protocols consists of a T-module, a data A-module, and possibly different C-modules for data encryption and authentication. Both data encryption as well as authentication can be switched on and turned off during data transfer. Traffic encryption keys are exchanged depending on elapsed time and amount of data transferred since the last key change. Based on the experiences of the secure messaging demonstrator it is concluded that a flexible, secure messaging tool allowing for the dynamic adaptation of the provided security level can be implemented easily with the Da CaPo++ middleware.

To demonstrate the integration of security in multimedia applications, and to outline the attainable performance, audio sender and audio receiver components from the multimedia server scenario have been enhanced with encryption by using secure audio protocols instead of native ones. Thus, high quality audio (*e.g.*, CD quality) is encrypted, transmitted, decrypted, and received in real-time. In addition, it is possible within the given implementation to switch on and turn off the data encryption module during data transmission.

4.5 Teleseminar

An interactive teleseminar scenario has been implemented by applying different applications and application components (cf. Figure 12). A teacher is viewed and listened to by students and he can view all of them. However, the audio of students is received by the teacher only, if a student signals the teacher he wants to asks a question. Documents to be used in the lesson are shared using the application sharing component. Every student can participate in a teleseminar by asking questions or giving answers which are heard to by all other students after the teacher has granted the floor. This is enabled by temporarily adding audio and video flows while the student is active.

Figure 17 shows all data flows between the teacher and two students. One student is active. Its video and audio data is received by the teacher and all other students. Video of all passive students is received by the teacher only. These students do not send audio data as long as they are passive. The video and audio data of the teacher as well as the document data (*e.g.*, slides) are transmitted to all students participating in the teleseminar.

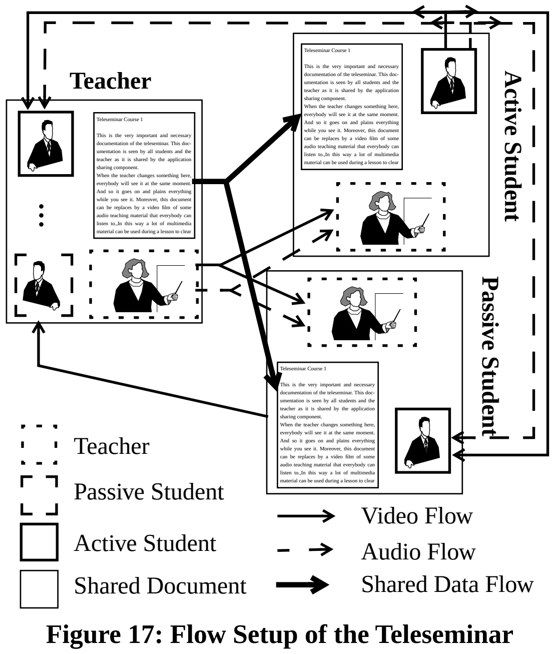


Figure 17: Flow Setup of the Teleseminar

For implementing the audio and video data transmission, similar concepts and protocols have been used as described for the video conference. Furthermore, a floor control has been added to control dynamic joins and leaves of students' audio and video streams during a lesson. It is implemented by Da CaPo++ data protocols. However, instead of using the application sharing component to share documents, multimedia presentations can be included in a teleseminar course by integrating the multimedia server interface. A prototypical implementation has been performed in close cooperation with XMIT which validated main concepts of the flexible Da CaPo++ middleware.

Based on the technology used in the teleseminar, a telebanking scenario can be implemented as well. The presented demonstrator applications (cf. Section 4.4) can be integrated into the telebanking scenario to provide the required privacy to the bank client.

Various application scenarios implemented show that the Da CaPo++ middleware is suited to support efficiently a wide range of diverse multimedia applications

4.6 Synchronization in Da CaPo++

Synchronization is an important issue for the quality of multimedia applications. Since the design of Da CaPo++ is based on protocols and flows providing data transmission of one data type into one direction, synchronization as implemented in Da CaPo++ [40] is designed on a flow based level. Synchronization is required for audio and video flows to ensure that data is played back at the same rate as captured (intra-stream synchronization [4]). In addition, multiple data streams may be synchronized by inter-stream synchronization. To preserve the independence of different protocols in the Da CaPo++ core, synchronization is based on the principle: if flows are intra-stream synchronized and if the first data of all flows are synchronized, synchronization can be guaranteed [10].

The synchronization in Da CaPo++ is based on the assumption of synchronized clocks. Intra-stream synchronization based on timers and periodic data has been included into A-modules of audio and video protocols. A synchronization control application component implements a directory service to administrate all flows in an application that have to be synchronized. The synchronization is based on the fact that a maximum delay can be specified for all data and control transmissions by Da CaPo++ protocols. The task of the synchronization control is to ensure that the first data units of all streams are presented simultaneously, *i.e.* synchronized.

The synchronization control is performed by defining an application protocol to exchange information between the synchronization control component and A-modules on sender and receiver side. Sending A-modules can be located on different sender's, *i.e.* synchronization can be achieved for data flows originating from different senders. The synchronization control receives delay specifications for flows from applications and determines the maximum of all delays specifying the total delay in addition to processing times. A start command for the data transmission is issued by the synchronization control and sent to all A-modules (cf. Figure 18). It specifies the time the first data is to be presented in a receiver. A-modules forward the start command to sending A-modules based on local knowledge of this connection's delay. Start commands are issued early to allow for stopping applications, if problems occur within senders. This mechanism guarantees that the first data of all data flows is synchronized.

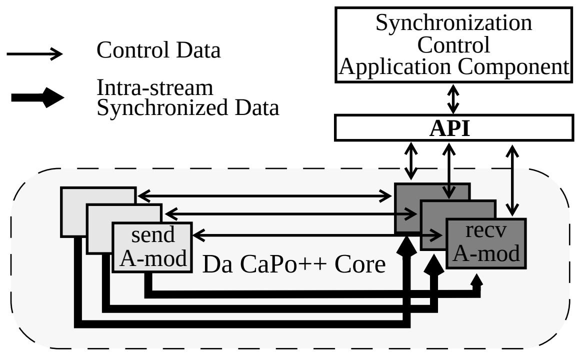

Control Data
Intra-stream Synchronized Data
Synchronization Control Application Component
API
send A-mod
Da CaPo++ Core
recv A-mod

Figure 18: Synchronization in Da CaPo++

To profit from the synchronization functionality in Da CaPo++, applications have to include the synchronization control application component and provide information on (1) data flows that need to be synchronized and (2) delay specifications of these flows. All control commands of applications and the start command are passed through the synchronization control which specifies the maximum delay of all data streams. Every command, in spite of the stop command, that is always executed, is performed following a two phase protocol. In the first phase, the synchronization control announces the command and the computed start time to participating A-modules which in turn forward the command to peer A-modules. They provide an answer indicating whether the requested command can be performed in time or not. In the second phase, the synchronization control either stops the execution of the command, if it cannot be provided for all flows, or the command is executed. This mechanism guarantees for inter-stream synchronization, as the transmission of all data flows is started synchronously at the specified start time. Moreover, data flows remain synchronized also during the execution of control commands. As synchronization is designed on a per flow basis, where each flow may suffer a different delay, data flows originating from different servers can be synchronized also.

5 Evaluation of Da CaPo++

The performance of data communications obtained in a given implementation determines the quality of communication services and protocols. The Da CaPo++ middleware and applications as described above have been implemented on standard workstations [40], such as Sun SPARC 20 and Sun UltraSPARC 170E running the non real-time operating system Solaris 2.5.1, while they were as idle as possible.The Da CaPo++ middleware has been performance evaluated using the Quantify tool [28] and high-resolution system time measurements directly. Standard Sun multimedia equipment has been utilized, such as cameras, microphones, and the SunVideo board [42] offering real-time image capture and compression for digital video. All performance evaluations of Da CaPo++ below have been carried out on Sun UltraSPARC 170E machines and are performed on four different levels [41]: (1) middleware internal modules and components, (2) multimedia and security protocols, (3) the API, and (4) application control.

To obtain a detailed view on different protocols being evaluated, Table 5 depicts a number of them and their configuration in terms of configured A-, C-, and T-modules, where short module and protocol names are given instead of following full implementation naming conventions.

Table 5. Evaluated 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

5.1 Lift Performance

The performance of the Lift is shown in Figure 19. These numbers include the time 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 0.6 µs each. This results in an overhead of 9 µs for the packet allocation and 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.

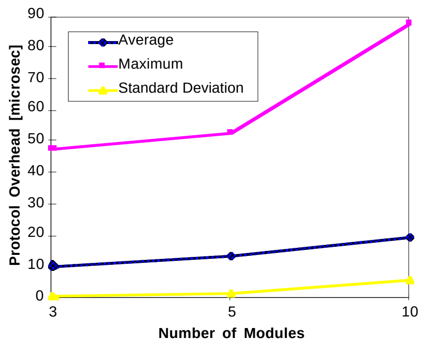


Figure 19: Protocol Overhead Based on Modules

The total run-time overhead depending on the amount of memory requested for a flow with 3 modules is shown in Figure 20. 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 in a quick manner. The first run also includes inherent semaphore signalling overhead by the operating system needed to start the Lift thread initially blocked. Buffers are not reserved in advance, because there is currently no chance to identify how many messages will be stored in each module. This has not been addressed yet, because it was considered more important at first to reduce the high overhead inherent to handling the multi-media devices. The maximum value is again due to context switches. Important, the required time is independent of the requested memory (buffer). Only the initial setup time increases slightly with buffer size.

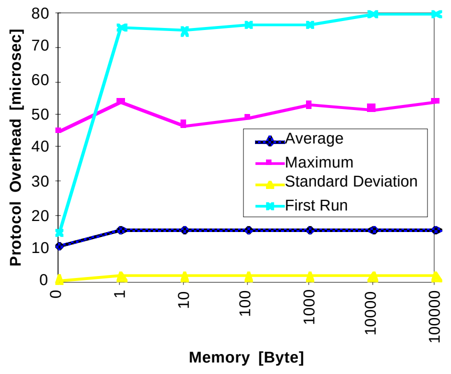


Figure 20: Protocol Overhead Based on Memory

5.2 Modules for Multimedia Data

Besides Da CaPo++ components, modules determine protocol functionality which are especially important for run-time performance evaluations. To achieve traceable results A-modules have been evaluated which handle stored audio and stored video for reading data and passing them to the Lift or for receiving data and displaying them on the screen or playing them on speaker boxes.

The Sun audio device interface uses a similar structure and commands as the file system interface, *i.e.* the system calls `read` and `write` are used to access the device. In the sending A-module for stored audio the mean value of this Lift function was less than 1 ms, whereas once in a while a maximum value came closely to 25 ms. The high variance is caused by the `read` call in this function, *i.e.* by the device. The Lift function in the receiving A-module perceives a remarkable variance, the mean cost are less than 0.1 ms, whereas the maximum reaches 2.5 ms. Again, this variance is caused by the device, in this case it is inherent to the `write` function call to the audio device.

Costs of Lift functions for stored video A-modules are highly dependent on the device, in the case implemented the SunVideo board and the Xil-library used to compress, decompress, capture, and display images of a video sequence. The Xil-library handles within the first image data relevant to the entire image sequence which occupies a larger amount of storage. Reading and decompressing this data takes up to 1.2 s or 0.12 s respectively. If the Xil-library has created the image sequence based on this data, the sending Lift function takes a mean processing time of 0.5 ms (worst case: 9 ms) which is sufficient for excellent video quality. The measured overhead of A-modules to access a multimedia device compared to directly using this devices is negligible.

Plain data transmissions in the Da CaPo++ middleware use a particular A-module for data. This A-module creates Da CaPo++ data packets from the application data received by the lower API or sends data received by the Lift via the API to the application. Both, writing and sending this data requires semaphore operations for accessing the shared memory. For this reason, the sending (≤460 ms) and the receiving function (≤12 ms) were measured. Therefore, the pure data transmission A-module (Raw-Data) should be used only for time-dependent multimedia data, if a direct access to multimedia devices via appropriate A-modules is not possible.

5.3 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. 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 figures are given. An ideal system is assumed, eventual network congestions or multitasking overhead would reduce the observed performance.

An overview of all numbers is depicted in Figure 21. 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.

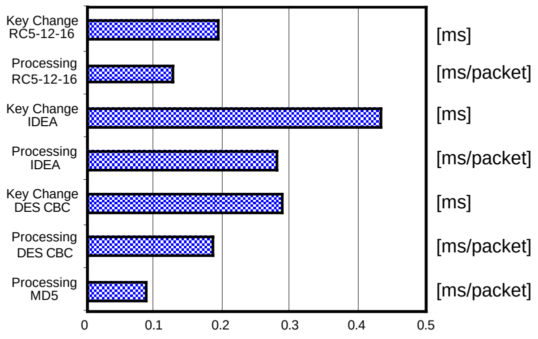


Figure 21: Comparison of Security Modules

5.4 Security Protocols

Different security protocols encrypting plain data have been evaluated by sending 10 MByte of data in 10,000 packets. Keys have been changed every 100th packet and an asymmetric encryption operation (RSA) has been performed every 500 packets. Table 6 shows the real overall (application-to-application) throughput values that can be achieved using security in Da CaPo++. 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.

Table 6. 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

To provide for 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).

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

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:

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

To combine required costs for authentication and encryption, both CPU seconds per Mbit values must be summarized. Table 7 represents cost values and achievable middleware throughputs as derived for the measurement platform of Da CaPo++.

Table 7. 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

5.5 Multimedia Protocols

To be independent from variances induced by devices and to measure the time required by Da CaPo++ protocols only, measured times encompasses (a) the time elapsed between “data is available in an A-module”, but not yet copied to the Da CaPo++ data packet and “data is ready” to be sent into the network for transmission or (b) the time “data is received” from the network and the time “data has been copied” already from the Da CaPo++ data packet to the application. Due to the shared memory concept developed for the API it is possible to use the same buffers for application data in the application process (except for data originating from the SunVideo board) as for the Da CaPo++ core.

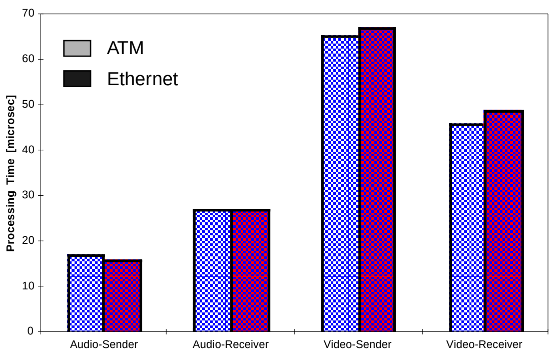


Figure 22: Audio/Video Protocol Processing Times

The time required to process continuous media protocol data has been measured using UDP and ATM T-modules including context switches. In case of video, the respective T-module, one C-module with empty functionality (to study module overhead), and a single file video A-module compose the protocol. In case of audio, a T-module together with a file audio A-module compose the protocol. Figure 22 depicts that all processing times for the ATM and UDP T-modules are almost similar for the same data type (audio and video at 17 µs and 65 µs for the sender). The overhead introduced by the measurement C-module for communication is below 40 µs. Therefore, if C-modules are required in protocols, *e.g.*, for the encryption of data, Da CaPo++ is suitable for real-time multimedia applications on standard workstations, even though more Lift actions (cf. Section 3.1.5) are necessary.

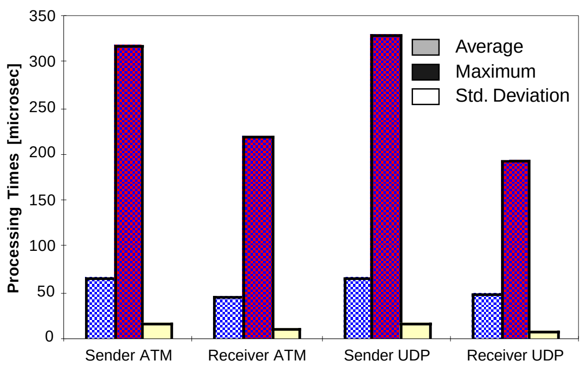


Figure 23: Video Protocol Processing Times

Based on these per protocol measurements, the elapsed processing times for the video protocol are presented in Figure 23, assuming that 999 video frames have been captured and compressed by the M-JPEG algorithm which is performed on the SunVideo board. Video frame sizes varied between 3,076 Byte and 14,282 Byte determining a standard deviation of 3,035 Byte. In this case, elapsed processing times encompass for the sending side the elapsed time, when data is accessed from the file until it is sent, and for the receiving side the elapsed time, when data arrives at the T-module and is handed over to the multimedia device. Standard deviations of these times are caused mainly by standard Xil-library functions. Nevertheless, due to the advantage of applying standard hardware and software, the drawback of these fluctuations experienced by single media streams or complete multimedia applications remains properly within tolerable limits.

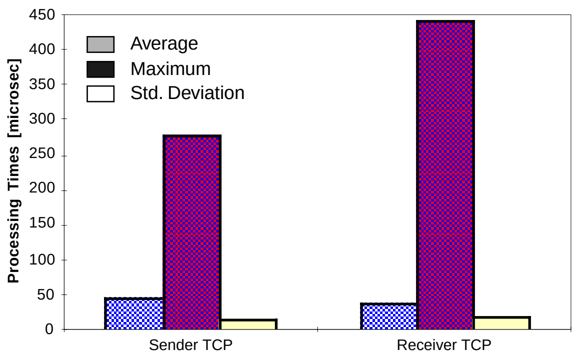


Figure 24: Reliable Protocol Processing Times

Two scenarios for measuring native user data protocol times have been evaluated. For a reliable data transmission protocol the protocol stack consists of a RawData A-module (allowing for accessing data from the application) and a multicast-capable TCP T-module for Ethernet access. The unreliable protocol utilizes the same RawData A-module and a unicast-capable UDP T-module instead. Note that these T-modules run in user space and are not part of the Solaris protocol stack.

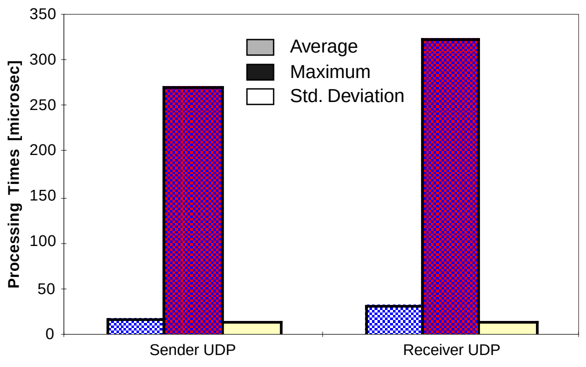


Figure 25: Unreliable Protocol Processing Times

The definition of measurement points is identical to the one described for Figure 23, however, results for the reliable protocol (cf. Figure 24) and the unreliable protocol (cf. Figure 25) include processing times of the access to the upper and lower part of the API data structures and the writing of data to the receiving data structures for small packet sizes of 88 Byte (cf. Section 3.4 and [11]). Therefore, these results determine that the reliable protocol processing is slightly more expensive: on average 45 µs vs.

18 µs for the sender and 41 µs vs. 31 µs for the receiver. Concerning the upper bound on harder processing time guarantees in the unreliable case, 273 µs for the sender and 323 µs for the receiver have been observed. For example, by re-calculating these numbers into throughput values for the unreliable data transmission, Da CaPo++ achieves an average a sender throughput of 38.4 Mbit/s for 88 Byte sized packets and of 44.8 Mbit/s for 1024 Byte sized packets which is lower for the worst, but guaranteed case (maximum values) determined at 2.4 Mbit/s and 3.7 Mbit/s respectively

5.6 API Performance

The API plays an important role during the connection establishment and the data transfer. Control data are exchanged between the application and Da CaPo++ over a Unix domain socket by Inter Process Communication. User data exchange is supported by a shared memory concept [11] 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 26 depicts the maximum performance that can be expected for sending or receiving data over the API. These measurements were obtained by sending and receiving, respectively, 1000 Byte sized 1000 packets. 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 not available immediately to the Lift due to an additional thread-switch for accessing the call-back function from the Lift.

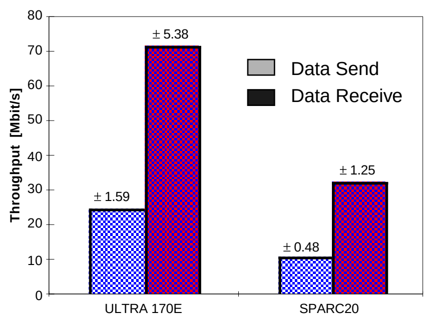


Figure 26: Saturated Raw API Throughput

API measurements with varying packet sizes in the sending direction are presented in Figure 27. An almost linear relation between the packet size and the throughput is achieved, reaching the maximum for 8 kByte packets at app. 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.

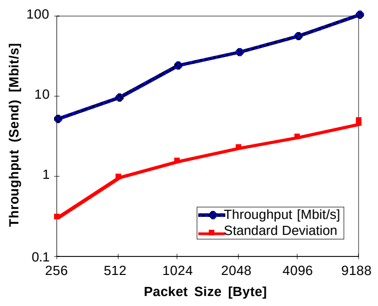


Figure 27: API Throughput

5.7 Application Control

In addition to pure data transmission as evaluated in the context of protocols and API measurements, the user of applications also sees the time influence of application control. As an example the multimedia server scenario has been measured in real-time, as the control handled in this scenario by the user is quite complex. Two specific points have been evaluated, the entry and configuration session as well as the response time to control commands.

The entry and configuration session time has been measured for one successful connection in the first alternative specification. If several alternative specifications have to be tried out, the effective time for the entry and configuration session will result in a multiple of the measured time. The entry and configuration session is based on a Da CaPo++ data protocol on top of IP. The time elapsed between the connect to the entry session of the multimedia client and the retrieval of the address of the multimedia server was measured between 181 ms and 219 ms with a very high variance resulting from the variance in network transmission over the local Ethernet. The flexibility gained by the entry and configuration session in the multimedia server scenario is, thus, not paid by long waiting times of the user.

The response time to commands is an important issue for the quality perceived by the user. If he/she has to wait too long, the total quality of the application will probably be perceived to be low. The response time has been measured for both, video and audio data transmission. In response to the command, regular packets have been created with a specified flag in the header that indicated the packet to be a response to the command. The sending of packets has been triggered by a time-out as for all multimedia packets. For this reason the response time depends on the periodicity of the packets which is reasonably higher in audio data transmission. Also the response time depends on the functionality needed to generate the packet and the transmission times of command and packet.

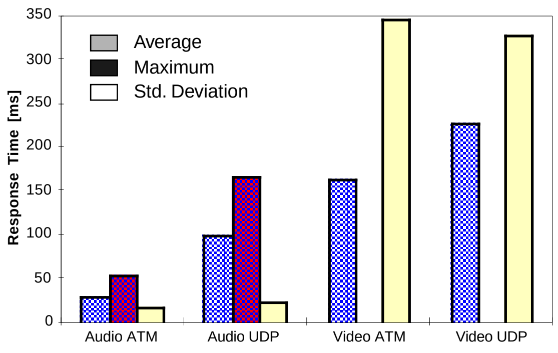


Figure 28: Response Time of Control Commands

Figure 28 shows that the application-to-application response time of commands is lower in case of an ATM connection than in case of IP over the shared medium Ethernet. The differences in the response times for audio and video applications result mainly from (1) the higher periodicity of audio packets as well as from (2) the different processing times in A-modules. A maximum of largely more than 1 s in case of video commands is excluded from Figure 28, since it results from the processing time of about 1 s for only the first video frame of an image sequence in the Sun Xil-library. Therefore, mean values in case of audio of 100 ms on top of UDP and 30 ms on top of ATM are excellent values. In addition, mean values of video on top of UDP of 227 ms and on top of ATM of 150 ms are well suited to provide a high quality service of application control to the user.

6 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 are summarized as follows:

- 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 real-time requirements, when secure and multicasting protocols are used.
- Security and multicast as well as synchronization requirements are seamlessly integrated into the QoS specification offered by Da CaPo++.
- The Application Programming Interface hides away all complex protocol issues from the programmer, providing an abstract and QoS-based interface.
- The Da CaPo++ approach provides valuable, efficient, and well-adapted services and protocols to application programmers. By applying it in the implementation of sample real-life application scenarios, a significant proof of concept has been presented.

Da CaPo++ is capable of accommodating a variety of multimedia applications 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 protocol efficient processing efficiency for a highly modular approach at the same time.

Security and multicasting functionality is made available to users in the same way as a request of reliable transfer of messages. The Da CaPo++ approach integrates multicasting and security functionality into middleware by synthesizing security and multicasting 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.

The Application Programming Interface 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 very useful exploitation is possible with QoS attribute specifications. Although breaking the transparency (applications are requested to specify their communication requirements for underlying “layers”) by handling application QoS within the Da CaPo++ core in the first place, 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 showed to be very suitable and easy-to-use for application programmers providing QoS specifications and supporting efficient data transfers 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. An application programmer discerns advantages and drawbacks of an interface, while constructing complex application scenarios with a given set of API functions. 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.

The independent user interface provided by the API allows for the three-level, modular application framework to enable an easy design and implementation of re-usable application building blocks to be combined to complex real-world application scenarios. In addition, the flexible application framework offers many features of an experimental toolkit. The real-world scenarios that were implemented show that Da CaPo++ is well suited to support a wide range of diverse multimedia applications. This diversity is supported best by a configurable middleware.

Concluding, the approach of integrating applications and supporting them by modern middleware is a promising one; it's viability has been demonstrated by our design and implementation of the Da CaPo++ middleware. The advantage of Da CaPo++ is its independence of the underlying operating system, making the prototypical Da CaPo++ implementation easily portable. Even though the performance of Da CaPo++ is not optimal completely at a few places in its current implementation due to undesirable operating system interactions, Da CaPo++ is fully portable, the proof of concept for flexibly configured communication protocols has been furnished, and an efficient multimedia support on standard workstation's hardware has been successfully accomplished.

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. In addition, George Fankhauser and Bettina Bauer commented on earlier versions of this paper.

References

[1] D. Bauer, B. Stiller, B. Plattner: *An Error-control Scheme for a Multicast Protocol Based on Round-trip Time Calculations;* 21st IEEE Conference on Local Computer Networks, Minneapolis, Minnesota, U.S.A., October 1997, pp 212 – 221.

[2] D. Bauer, B. Stiller, B. Plattner: *Guaranteed Multipoint Communication Support for Multimedia Applications;* SYBEN'98 Broadband European Networks Conference, Zürich, Switzerland, May 18–21, 1998.

[3] E. W. Biersack, C. Bernhardt: *A Fault Tolerant Video Server Using Combined RAID 5 and Mirroring;* SPIE Vol. 3020 for Multimedia Computing and Networking, San Jose, California, U.S.A, February 1997, pp 106 – 117.

[4] G. Blakowski, R. Steinmetz: *A Media Synchronization Survey: Reference Model, Specification, and Case Studies*; IEEE Journal on Selected Areas in Communications, Vol. 14, No. 1, January 1996, pp 5 – 35.

[5] S. Böcking: *Sockets++: A Uniform Application Programming Interface for Basic-level Communication Services;* IEEE Communications Magazine, Vol. 34, No. 2, December 1996, pp 114 – 123.

[6] M. Buddhikot, J. Chen, D. Wu, G. Parulkar: *Extensions to 4.4 BSD UNIX for Networked Multimedia in Project MARS;* IEEE Conference on Multimedia Computer Systems, Austin, Texas, U.S.A., June 1998.

[7] A. Campbell, G. Coulson, D. Hutchinson: *A Quality-of-Service Architecture*; ACM Computer Communication Review, Vol. 24, No. 2, April 1994, pp 6 – 27.

[8] S. Casner, S. E. Deering: *First IETF Internet Audiocast*; ACM Computer Communication Review, Vol. 22, No. 3, July 1992, pp 92 – 97.

[9] D. D. Clark, D. L. Tennenhouse: *Architectural Considerations for a New Generation of Protocols;* ACM Computer Communication Review, Vol. 20, No. 4, September 1990, pp 200-208.

[10] C. Class, *Synchronization Issues in Distributed Applications: Definitions, Problems, and Quality of Synchronization;* Technical Report No. 31, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, December 1997.

[11] C. Conrad, B. Stiller: *A QoS-based Application Programming Interface for Communication Middleware;* SPIE Vol. 3233 for the Voice, Video, and Data Communications Symposium, Dallas, Texas, U.S.A., November 1997, pp 248 – 259.

[12] A. Danthine: *The OSI´95 Transport Service with Multimedia Support – Research Reports ESPRIT, Project 5341; Volume No. 1*, Springer, Berlin, Germany, 1994.

[13] M. Dasen, G. Fankhauser, B. Plattner: *An Error-tolerant, Scalable Video Stream Encoding and Compression for Mobile Computing;* ACTS Mobile Summit 1997, Granada, Spain, Vol. II, November 1997, pp 762 – 771.

[14] A. De Jong-A, K. Hsing, D. Su: *A VoD Application Implemented in Java;* Journal on Multimedia Tools and Applications, Vol. 5, No. 2, September 1997, pp 161 – 170.

[15] D. Decasper, M. Waldvogel, Z. Dittia, H. Adiseshu, G. Parulkar, B. Plattner: *Crossbow – A Toolkit for Integrated Services over Cell-switched IPv6;* IEEE ATM Workshop, Lisboa, Portugal, June 1997.

[16] S. E. Deering: *Multicast Routing in a Datagram Internetwork*; Ph. D. Thesis, Stanford University, California, U.S.A., December 1991.

[17] C. Ed, A. Zakhor: *Disk-based Storage for Scalable Video;* IEEE Transactions on Circuits and Systems for Video Technology, Vol. 7, No. 5, October 1997, pp 758 – 770.

[18] A. Frier, P. Karlton, P. Kocher: *The SSL 3.0 Protocol;* Netscape Communications Corporation, http://home.netscape.com/eng/ssl3/, November 1996.

[19] T. Gutekunst: *Shared Window Systems*; Ph. D. Thesis No. 11120, Swiss Federal Institute of Technology, Zürich, 1995.

[20] N. C. Hutchinson, L. L. Peterson: *The x-Kernel: An Architecture for Implementing Network Protocols;* IEEE Transactions on Software Engineering, Vol. 17, No. 1, January 1991, pp 64 – 76.

[21] E. Isaacs, J. Tang: *What Video Can and Can't do for Collaboration: A Case Study*; ACM Multimedia, June 1993, pp 199 – 206.

[22] R. J. van den Linden: *An Overview on the Advanced Network Systems Architecture (ANSA);* Architectural Report AR.000.00, APM Ltd., Cambridge, England, 1993.

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

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

[25] 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.

[26] Object Management Group: *CORBA: The Common Object Request Broker Architecture*; Revision 2.0, July 1995.

[27] T. Plagemann, B. Plattner, M. Vogt, T. Walter: *A Model for Dynamic Configuration of Light-Weight Protocols*; 3rd IEEE Workshop on Future Trends of Distributed Systems, Taipeh, Taiwan, April 1992, pp 100 – 106.

[28] Pure Software: *Quantify User's Guide*, 1995.

[29] M. Purser: *Secure Data Networking*; Artech House, London, England, 1993.

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

[31] D. Schmidt: *IPC_SAP: An Object-oriented Interface to Operating System Interprocess Communication Services*; C++ Report, Vol. 4, No. 8, SIGS, November/December 1992, pp 1 – 10.

[32] D. Schmidt, T. Suda: *Transport System Architecture Services for High-Performance Communication Subsystems*; IEEE Journal on Selected Areas in Communications, Vol. 11, No. 4, May 1993, pp 489 – 506.

[33] B. Schneier: *Applied Cryptography*; 2nd Edition, Wiley & Sons, New York, U.S.A., 1996.

[34] J. Soo: *Live Multimedia over HTTP*; 2nd International World Wide Web Conference, Mosaic and the Web, Chicago, Illinois, U.S.A., October 1994.

[35] A. Srivastava, A. Kumar, A. Singru: *Design and Analysis of a Video-on-Demand Server;* Journal on Multimedia-Systems, Vol. 5, No. 4, July 1997, pp 238 – 254.

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

[37] B. Stiller: *Quality-of-Service – Dienstgüte in Hochleistungsnetzen*; International Thomson Publishing, Bonn, Germany, 1996.

[38] B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel: *Da CaPo++ – Communication Support for Distributed Applications;* Technical Report No. 25, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, January 1997.

[39] B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: *Project Da CaPo++, Volume I: Architectural and Detailed Design;* Technical Report No. 28, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, August 1997.

[40] B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: *Project Da CaPo++, Volume II: Implementation Documentation;* Technical Report No. 29, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, August 1997.

[41] B. Stiller, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: *Project Da CaPo++, Volume III: Performance Evaluations;* Technical Report No. 42, TIK, Swiss Federal Institute of Technology, Zürich, Switzerland, February 1998.

[42] Sun Microsystems: *SunVideo User's Guide;* Mountain View, California, U.S.A., 1994.

[43] T. Turletti, C. Huitema: *Video Conferencing on the Internet;* IEEE/ACM Transactions on Networking, Vol. 4, No. 3, June 1996, pp 340 – 351.

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

[45] V. Voydock, S. Kent: *Security Mechanisms in High-Level Network Protocols*; ACM Computing Surveys, Vol. 15, No. 2, June 1983, pp 135 – 171.

[46] E. Wilde, B. Plattner: *Transport-independent Group and Session Management for Group Communication Platforms;* European Transactions on Telecommunications, Vol. 8, No. 4, July/August 1997, pp 409 – 421.

[47] *WinSock2: Information, Architecture, and Specification;* http://www.sockets.com/, 1997.

[48] K. Wolf, K. Froitzheim, M. Weber: *Interactive Video and Remote Control via the World Wide Web*; European Workshop on Interactive Distributed Multimedia Systems and Services, B. Butscher, E. Moeller, H. Pusch (Eds.), Berlin, Germany, March 1996, pp 91 – 104.

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

[50] M. Zitterbart, B. Stiller, A. Tantawy: *A Model for Flexible High-Performance Communication Subsystems*; IEEE Journal on Selected Areas in Communications, Vol. 11, No. 4, May 1993, pp 507 – 518.

[51] S. Znaty, T. Walter, M. Brunner, J.-P. Hubaux, B. Plattner: *Multimedia Multipoint Teleteaching over the European ATM Pilot;* International Zürich Seminar 1996, Lecture Notes in Computer Science No. 1044, Springer, Berlin, Germany, February 1996, pp 225 – 238.