Also publ. as: Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich (ETH), 1997, TIK-Report No. 29

Project Da CaPo++, Volume II: Implementation Documentation

TIK-Report No. 29

Burkhard Stiller, Germano Caronni, Christina Class, Christian Conrad, Bernhard Plattner, Marcel Waldvogel
Computer Engineering and Networks Laboratory (TIK)
ETH Zürich, CH – 8092 Zürich, Switzerland
E-Mail: <last-name> @ tik.ee.ethz.ch

1. Introduction and Achievements

The research project KTI–Da CaPo++ is based on the project Da CaPo (Dynamic Configuration of Protocols) at the ETH. The extended system of Da CaPo++ provides a basis for an application framework for banking environments and tele-seminars. It includes the support of prototypical multimedia applications to be used on top of high-speed networks including dynamically configurable security and multicast aspects.

The main goal for this document is the description of the implemented design as stated elsewhere. It covers the application framework parts carried out at ETH and Da CaPo++ core system internal details concerning the security and relevant C-modules. These goals have been achieved and implemented under Solaris 2.5.1 on Sun workstations including multimedia equipment, such as cameras, microphones, and speakers.

Finally, the implemented design of Da CaPo++ has remained independent of any specific transport infrastructure, as long as the considered network offers minimal features, *e.g.*, bandwidth, delay, or bit error rates that are requested by an application. A heterogeneous infrastructure, including Ethernet and ATM (Asynchronous Transfer Mode), is supported, which has been demonstrated in the final project demonstration July 1, 1997.

1.1 Brief Survey of Da CaPo++

The kernel system of Da CaPo++ – called Da CaPo++ core system – provides the possibility to configure end-system communication protocols. This process is based on currently available application requirements, local resources, and network prerequisites. The result is defined as an adapted and best possible communication protocol under well-defined circumstances. Basic building blocks, in particular protocol functions and their mechanisms, form the basis for the process of configuration.

The Da CaPo++ core is responsible for handling multiple data flows and protocol processing completely. Via its application programming interface (API) Da CaPo++ offers unicast- and multicast-services to applications. The API consists of a control access point, which allows to manipulate and configure entire sessions, consisting of several flows. Data access points serve as means to specify the handling of data after protocol processing. This might be a transfer to the application or the specification of the window in which video has to be displayed.

1.2 Structure of this Implementation Documentation

This document contains a discussion of the implementation necessary changes and extensions to Da CaPo and describes key functionalities that have been added to different elements of Da CaPo, resulting in the Da CaPo++ core system and the appropriate application framework.

This document does not include descriptions of tasks and application scenarios that are being handled by the project partners of Schweizerischer Bankverein, Basel (SBV) and XMIT AG, Zürich.

This documents is organized as follows. Initially, Section 2 delivers the table of contents and Section 3 and Section 4 additionally list all figures and tables.

Section 5 includes the implementation and design of the Application Programming Interface (API), which covers an internal structure of an upper and a lower API as well as the model of sessions and flows including a short view into the applied buffer management. Detailed methods and attributes are discussed.

Section 6 contains security aspects that have been implemented not conforming with the corresponding section 3 of security within the detailed design document. In addition, it contains flow charts that deal with the sequence of actions taken for various steps in the Security Manager and the Da CaPo++ core.

Furthermore, Section 7 briefly summarizes the main multicasting features already achieved.

The implementation of the Application Framework of Da CaPo++ is provided in Section 8. In particular, the WWW application scenario is discussed. The implementation of the extended WWW browser and server and the multimedia file client are presented. In addition, the video viewer, its shared version, and the multimedia sender and server are described. Further application scenarios have not been included due to project partner responsibilities.

Finally, a short discussion and initial approach to synchronization issues are handled in Section 9.

1.3 Remarks

This document covers workpackages done at ETH Zürich, in more detail – concerning the internal numbering of workpackages of the project – package B (Application Programming Interface), package C (Multicasting), package D (security), package K1/L1 (Picture phone), package I1 (Audio/Video), package I2 (Da CaPo++ Video Viewer), and package K4/L4/K5 (World Wide Web). Work package A (core system) has been documented in a separate set of documents.

Finally, project partner reports for packages I4 (Application Sharing), package L3 (Tele-Banking), and package L4 (Tele-Seminar, Tele-Referat) will be found elsewhere.

Due to the prototypical implementation of the project, all issues are subject to change in the future. This is not only limited to functions, methods, or tasks, but may include certain conceptual changes due to reasons discovered within the detailed design phase. Implementation restrictions have been added as far as they form a major aspect of interest. The implementation is provided “as is” without express or implied warranty.

2. Table of Contents

3. List of Figures

4. List of Tables

5. Application Programming Interface (API)

5.1 Introduction

This document presents in a detailed way all design and implementation issues concerning the Da CaPo++ application programming interface (API). This report is a synthesis of both fine design and implementation documents that were conceived before and during the implementation, respectively. Thus, it provides a unified view over the programmed API and constitutes a reference as some aspects of both previous documents may no longer be valid.

Some more technical parts of this document are intended for anybody who wishes to extend the functionality of the API. These parts can be most of the time skipped by application programmers, who will just have to understand both the fundamental design principles and the "visible" functionality provided by the API.

5.2 API's Design Overview

This chapter introduces the fundamental design principles that guided the implementation of the API. It should be carefully read by any programmer who wishes to build applications.

In order to make the management of resources easier, it was decided to split applications and Da CaPo++ core system in different "UNIX" processes and to use only one Da CaPo++ process per machine. Thus applications have their own processes and communicate with the Da CaPo++ server via interprocess communication (IPC) mechanisms. The upper API part is therefore linked to the application, whereas the lower API part is the interface to the Da CaPo++ system. A complete illustration of this environment is illustrated in Figure 1 on page 8.

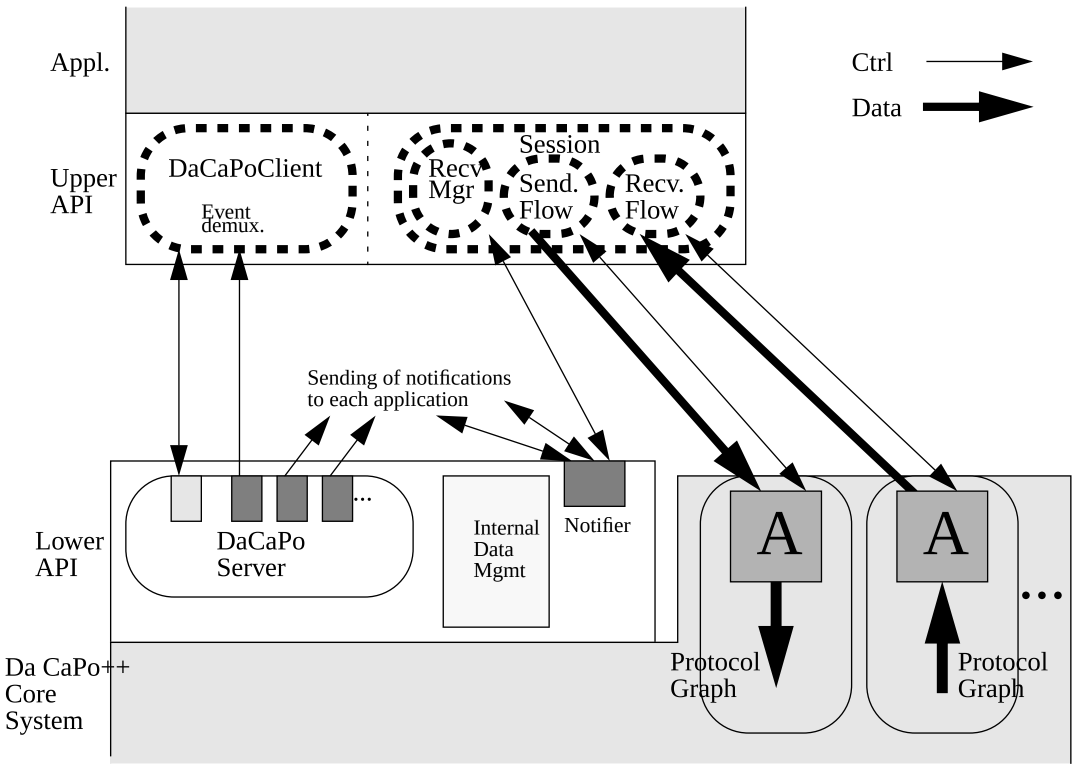


Figure 1. API Components

Thus the design consists in an upper API (described in Section 5.4 on page 16), in a lower API (described in Section 5.5 on page 29) and finally in various IPC mechanisms to deal with the sending and receiving of both data and control information between the two separate processes (see Section 5.3 on page 9). The lower API consists basically in a `DaCaPoServer` (responsible for the main control protocol between each application and the Da CaPo++ process), in an internal data management (storage of all protocol related information, such as application requirements for example) and in a notifier which receives a notification each time data is ready to be received from the application. To make the Da CaPo++ core concepts visible on the application side, extensive use of abstractions was performed. The use of an object-oriented programming language (C++) allowed to define a set of classes that can be instantiated by the application programmer. There are only three such main classes: the `DaCaPoClient` that is responsible for all security aspects when communicating with the Da CaPo++ core, the `Session` and the `Flow` that represent both service and protocol graph respectively. These classes and all their attributes and methods will be further explained in Section 5.4.1 on page 16. The IPC mechanisms are either UNIX domain sockets (for the main control protocol) or shared memory (for data transfers where efficiency is a critical issue).

5.3 Interprocess Protocols

As already mentioned in the design overview, the elected IPC mechanisms are either UNIX domain sockets for the main control protocol or shared memory for critical data transfers such as application data. Both mechanisms are further detailed in the coming subsections.

5.3.1 Main Control Protocol

The main control protocol is responsible for the setup of a connection between the application and Da CaPo++ processes. First an application must register itself towards the core security manager, then a session must be created (with all its flows and application requirements) and configured. After the session has successfully connected, the data transfers can start or stop according to the application. Finally, a session must be destroyed, and all resources reserved in the core system may be deallocated, and eventually re-used by a new session.

All these operations constitute the task of the main control protocol. As timing issues are not critical for this protocol and there is a dacapo process located on each machine, a UNIX domain socket connection was used. This protocol is basically bidirectional, however, it is always the upper API which sends a request, followed by an acknowledge from the lower API. Thus it is not possible for the lower API to asynchronously send a request to the application (the event mechanism is used for this type of communication, cf. Section 5.4.5 on page 29).

The format of the main control protocol PDUs is given in Section 5.3.1.1 on page 9. For the successful sending/receiving of PDUs over this channel, a set of management functions was developed (cf. Section 5.3.1.2 on page 15).

5.3.1.1 Control PDUs Format

Each main control protocol PDU has the format illustrated in Figure 2 on page 9. It consists of a command field followed by a list of parameters.

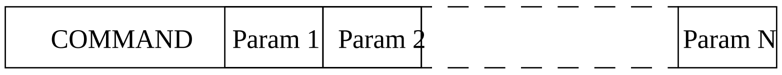


Figure 2. Main Control Protocol PDU Format

For each command there is a fixed number of parameters. Each parameter is delimited via delimiter characters such as "[" and "]". In case where such a delimiter is present in the contents of a parameter, a simple character stuffing is used (with a "*" in front of each "[", "]" and "*"). Knowing that the communication over a UNIX domain socket is reliable, no error detection scheme has to be provided over this channel.

To meet the needs of such a protocol, the set of all necessary PDUs is illustrated in Table 1. The “Src” field indicates who is the sender of the PDU, either the upper or the lower API. For each parameter, the data type is provided in brackets (either `int` or `char*`, C-syntax).

Table 1. PDU Types

Command	**Src**	**Description + Reaction + Parameters (ordered)**
CTRL_REGISTER_APP	Up	An application wishes to register in Da CaPo++ system (lower API). Reactions are either CTRL_ACCEPT_APP or CTRL_REJECT_APP (determined by the security manager). Parameters are: • (int) ProcessId: • (char*) UserName: • (char*) ApplicationName: • (char*) ApplicationCertificate: • (char*) UserPassPhrase: • (char*) PublicKey: • (char*) SecretKey:
CTRL_ACCEPT_APP	Low	The application is authorized to communicate with Da CaPo++. No reactions. Parameters are: • (int) coreApplicationId: global application identifier on the machine • (char*) mmapFileName: to set up the mmap area for event notification • (int) mmapInitialSize: initial size of the shared area
CTRL_REJECT_APP	Low	The application is NOT authorized to communicate with Da CaPo++. No reactions. No parameters.
CTRL_REG_NEW_SES	Up	The configuration file is parsed in the upper API and only the session relevant information is transmitted with this command. CTRL_TMP_REG_SES (failures are not considered...). Parameters are: • (int) coreApplicationId: • (int) SessionRole: • (int) SessionType:
CTRL_TMP_REG_SES	Low	The new session relevant data was stored in a temporary buffer (it will be stored in the lower API internal tables only when the session is fully registered with all flows, ...) No reactions. Parameters are: • (int) CoreSessionId: machine unique id. for the session
CTRL_REG_NEW_FLOW	Up	The configuration file is parsed and only the flow relevant information is transmitted with this command (one call for each session’s flow). CTRL_OK_FLOW (failures are not considered) Parameters are: • (int) CoreApplicationId: • (int) CoreSessionId: • (int) FlowType: audio, video or data • (int) FlowDirection: up-down or down-up • (int) FlowSinkSource: application, file, device

Table 1. PDU Types

Command	Src	Description + Reaction + Parameters (ordered)
CTRL_OK_FLOW	Low	The flow was properly registered (without any ARs!) No reactions Parameters are: • (int) CoreFlowId:
CTRL_REG_NEW_AR	Up	A new application requirement is transmitted to the lower API (either during the session configuration setup, or before initiating a reconfiguration during run-time). No reaction as it is assumed that the parser component in the upper API already checked if the AR and the current flow were “compatible”. Parameters are: • (int) CoreFlowId: • (char*) ARId: identifier of the application requirement • (char*) First value • (char*) Second value • (char*) Third value • (int) Type: Meaning of all three values
CTRL_END_FLOW	Up	All application requirements for the current flow have been parsed and sent. The reaction is TMP_REG_FLOW (failures are not considered) No parameters:
CTRL_TMP_REG_FLOW	Low	The new flow relevant data was stored in a temporary buffer. No reactions. Parameters are: • (char*) mmapAddressCtrlUpDown: to set up the shared area for the sending of control information • (int) mmapInitialSize: initial size of the shared area • (char*) mmapAddressCtrlDownUp: to set up the shared area for the receiving of control information • (int) mmapInitialSize: initial size of the shared area • (char*) mmapFileAddressData: to set up the shared area for the sending of data information (only if necessary) • (int) mmapInitialSize: initial size of the shared area (only if necessary)
CTRL_REG_NEW_SYNC	Up	Two (or more) flows are synchronized together. CTRL_TMP_REG_SYNC (failures are not considered...). Parameters are: • (int) CoreSessionId: • (int) SyncLevel: • (int) NbOfSyncFlows: to know how many parameters there are • (char*) FlowName1: • (char*) FlowName2: • (char*)... (maximum number of flows is currently 7)
CTRL_TMP_REG_SYNC	Low	The synchronization information was properly registered. No reactions. No parameters.
CTRL_REGISTER_SES	Up	The session features were now completely defined, the whole session can be definitely registered. Reactions are either CTRL_ACCEPT_SES or CTRL_REJECT_SES. No parameters:

Table 1. PDU Types

Command	Src	Description + Reaction + Parameters (ordered)
CTRL_ACCEPT_SES	Low	The session is accepted by the lower API No reactions. Parameters are: • (char*) mmapAddressNotifUpDown: to set up the shared area for the sending flow notifica- tions • (int) mmapInitialSize: initial size of the shared area • (char*) mmapAddressNotifDownUp: to set up the shared area for the sending of flow notifi- cations • (int) mmapInitialSize: initial size of the shared area
CTRL_REJECT_SES	Low	The session was rejected by the lower API. No reactions. No parameters.
CTRL_CONNECT_SES	Up	The session requests to be connected. Reactions are either CTRL_ACCEPT_CONNECT or CTRL_REJECT_CONNECT. Parameters are: • (char*) grpAddress: • (char*) grpService: • (char*) localAddress: • (char*) localService: • (char*) remoteAddress: • (char*) remoteService: • (char*) localInterface: • (char*) remoteInterface: • (int) CoreSessionId:
CTRL_ACCEPT_CONNECT	Low	The core system successfully performed the connect on the session. No reactions. No parameters.
CTRL_REJECT_CONNECT	Low	The core system could not perform the connect on the session. No reactions. No parameters.
CTRL_LISTEN_SES	Up	The session requests to be connected. Reactions are either CTRL_ACCEPT_CONNECT or CTRL_REJECT_CONNECT. Parameters are: • (char*) grpAddress: • (char*) grpService: • (char*) localAddress: • (char*) localService: • (char*) remoteAddress: • (char*) remoteService: • (char*) localInterface: • (char*) remoteInterface: • (int) CoreSessionId:
CTRL_ACCEPT_LISTEN	Low	The core system successfully performed the listen on the ses- sion. No reactions. No parameters.
CTRL_REJECT_LISTEN	Low	The core system could not perform the listen on the session. No reactions. No parameters.

Table 1. PDU Types

Command	**Src**	**Description + Reaction + Parameters (ordered)**
CTRL_CONFIGURE_SES	Up	The session requests to be configured. Reactions are either CTRL_ACCEPT_CONFIGURE or CTRL_REJECT_CONFIGURE. Parameters are: • (int) CoreSessionId:
CTRL_ACCEPT_CONFIGURE	Low	The core system successfully performed the configure on the session. No reactions. No parameters.
CTRL_REJECT_CONFIGURE	Low	The core system could not perform the configure on the session. No reactions. No parameters.
CTRL_ACTIVATE_SES	Up	The session requests to be activated. Reactions are either CTRL_ACCEPT_ACTIVATE or CTRL_REJECT_ACTIVATE. Parameters are: • (int) CoreSessionId:
CTRL_ACCEPT_ACTIVATE	Low	The core system successfully performed the activate on the session. No reactions. No parameters.
CTRL_REJECT_ACTIVATE	Low	The core system could not perform the activate on the session. No reactions. No parameters.
CTRL_DEACTIVATE_SES	Up	The session is deactivated Reactions are either ACCEPT_DEACTIVATE or REJECT_DEACTIVATE Parameters are: • (int) coreSessionId • (int) stopWay: either GRACEFUL or GRACELESS
CTRL_ACCEPT_DEACTIVATE	Low	The core system successfully performed the deactivate on the session. No reactions. No parameters.
CTRL_REJECT_DEACTIVATE	Low	The core system could not perform the deactivate on the session. No reactions. No parameters.
CTRL_CLOSE_SES	Up	Closing of the session Reactions are either CTRL_ACCEPT_CLOSE or CTRL_REJECT_CLOSE Parameters are: • (int) coreSessionId
CTRL_ACCEPT_CLOSE_SES	Low	The session could be properly closed in the lower API. No reactions. No parameters.
CTRL_REJECT_CLOSE_SES	Low	The session could not be properly closed in the lower API (because some components may still be active). No reactions. Parameters are: • (char*) Warning:

Table 1. PDU Types

Command	Src	Description + Reaction + Parameters (ordered)
CTRL_CLOSE_APP	Up	The application requests to leave Da CaPo++. Reactions are either CTRL_ACCEPT_CLOSE_APP or CTRL_REJECT_CLOSE_APP. No parameters.
CTRL_ACCEPT_CLOSE_APP	Low	The application could be properly deallocated in the lower API. No reactions. No parameters.
CTRL_REJECT_CLOSE_APP	Low	The application could not be properly deallocated in the lower API (because some components may still be active). No reactions. Parameters are: (char*) Warning:
CTRL_SEC_MGR	Up	The application sends a request transparently to the core Da CaPo++'s security manager. Reactions are either CTRL_SEC_MGR_ACCEPT or CTRL_SEC_MGR_REJECT. Parameters are: • (char*) Request:
CTRL_SEC_MGR_ACCEPT	Low	The request was successfully processed by the security manager. No reactions. Parameters are: • (char*) Reply:
CTRL_SEC_MGR_REJECT	Low	The request could not be processed by the security manager.
CTRL_SET_AR	Up	A new application requirement is transmitted to the lower API No reaction Parameters are: • (int) CoreFlowId: • (char*) ARId: identifier of the application requirement • (char*) First value • (char*) Second value • (char*) Third value • (int) Type: Meaning of all three values
CTRL_SET_AR_ACCEPT	Low	The new AR was properly registered. No reactions. No parameters.
CTRL_SET_AR_REJECT	Low	The new AR could not be properly registered. No reactions. No parameters.
CTRL_GET_AR	Up	The computed (configured) value of an application requirement is required. Reaction is either CTRL_RETURN_AR or CTRL_FAILURE_AR. Parameters are: • (int) CoreFlowId: • (char*) ARId:

Table 1. PDU Types

Command	Src	Description + Reaction + Parameters (ordered)
CTRL_RETURN_AR	Low	The computed (configured) value of the AR is returned. No reactions. Parameters are: • (char*) ARId: identifier of the application requirement • (char*) First value • (char*) Second value • (char*) Third value • (int) Type: Meaning of all three values
CTRL_FAILURE_AR	Low	The computed (configured) value of the AR could not be returned.

5.3.1.2 Management of Control PDUs

A set of functions to create a new PDU of a given type (command), to append a new parameter and to send it on the UNIX domain socket as well as the corresponding set to receive a PDU from the socket and to extract its command identifier and all its parameters are provided in the file `ctrl-util.h`. These functions are used by both upper and lower API. For more information over these functions, the reader is invited to directly consider the files `ctrl-util.[ch]`.

5.3.2 Additional Data Exchanges

For the transmission of application data from the application process to the Da CaPo++ process, it is not possible to use the same IPC mechanism as for the main control protocol. The main reason is efficiency, as sending data over a UNIX domain socket implies data copying from the user-space to the kernel and vice-versa. This scheme leads to major bottlenecks and must not be considered if a huge amount of data is sent/received directly in the application. To counter this unnecessary data-copying, we transmitted data over shared-memory. This solution reduces the load due to data-copying at the expense of several synchronization system calls (semaphores are necessary to “signal” the arrival of new data over the shared-memory). However, simple simulations showed that using a semaphore synchronized shared-memory was up to ten times more efficient than a socket based solution (either UNIX or INTERNET domain).

Figure 1 on page 8 illustrates two types of shared-memory communications. First the sending/receiving of user/control data between the flows and the A-modules, and then the sending of both events and notifications. These data types have different properties and are illustrated in Table 2 on page 15.

Table 2. Shared-memory Channel Properties

Type	Properties	Packet size
User/Ctrl Data	One single writer One single reader	variable
Event[a]/Notification	Several writers One single reader	fixed

a. During implementation, it appeared that it was more efficient to consider variable size events

The most important differences between the two channel types is the packet size (either fixed or variable) and the number of concurrent “processes” (threads in our context) that simultaneously try to write data. These differences and their implications are further described in the following subsections.

5.3.2.1 Ctrl/User Data Exchanges

For control/user data, packet sizes may be of any size (mainly dependent of the application). This is the most general case where the shared-memory can be viewed as a “continuous” area as in part (a) of

Figure 3 on page 16. As there is only one writing thread, only the access to the shared memory must be

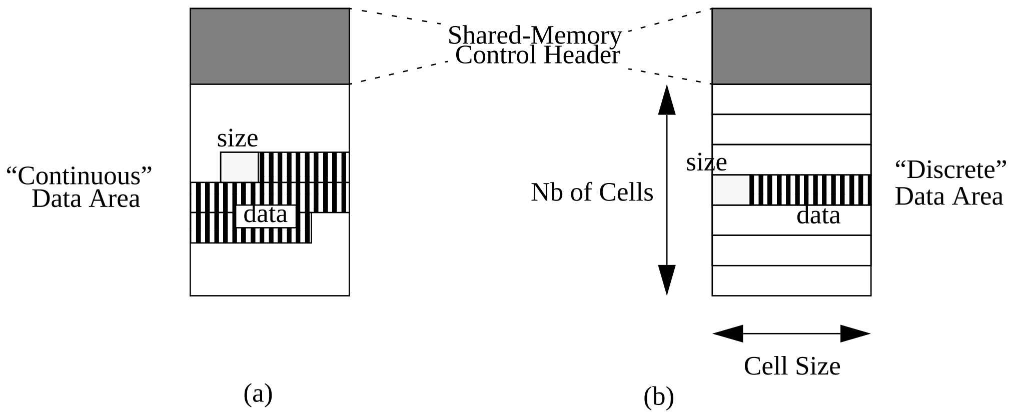


Figure 3. Fixed and Variable Packet Size Shared Areas

granted through semaphores. Actually, two semaphores are required for both writing and reading data. Each written packet consists first in its size (one integer) and then its contents (this enables the reader to read exactly one packet). A code sample on how to use the access semaphores is provided in Figure 4 on page 16.

```
int read(char *buffer,
         int *bufferLen) {
    sema_wait( newData );
    /* reading of new data */
    buffer = ...;
    bufferLen = ...;
    sema_post( freePlace );
    ...
}
```

```
int write(char *buffer,
         int bufferLen) {
   /* test if enough room */
   /* for new data */
   while( bufferLen <
         areaSize-curBytes)
       sema_wait( freePlace );
   ... = buffer;
   ...
   sema_post( newData );
}
```

Figure 4. Read/Write Functions for a Variable Packet Size Shared Area

5.3.2.2 Sending/Reception of Events/Notifications

For the fixed packet size, the elected structure is slightly different (although the same scheme as previous could have been used). We no longer have a "continuous" memory area, but a set of cells that are designed to contain only one message. As before, two semaphores are required to grant access to the shared-area. However, as there are several writing threads, the access to the next free cell must also be protected by a mutex. Again, a code sample is provided in Figure 5 on page 17.

5.4 Upper API

As already mentioned in the overview, there are three main abstractions in the object model of the upper API. In the following subsection, each class specification is given, with all attributes and methods. The application programmer may skip this subsection and see directly a more functional description of each class from Section 5.4.2 on page 22 to Section 5.4.5 on page 29.

5.4.1 API's Object Model

The three main data abstractions are provided in the following subsections. It has to be noted that during the implementation, some classes were split in a base class (hidden to the application programmer) and

```
int read(char *buffer,
        int *bufferLen) {
    sema_wait( newData );
    /* reading of new data */
    buffer = ...;
    bufferLen = ...;
    sema_post( freePlace );
    ...
}
```

```
int write(char *buffer,
        int bufferLen) {
    sema_wait( freePlace );
    mutex_lock( cell );
    /* writing of new data */
    ... = buffer;
    ... = bufferLen;
    mutex_unlock( cell );
    sema_post( newData );
    ...
}
```

Figure 5. Read/Write Functions for a Fixed Packet Size Shared Area

the visible class (e.g., the `DaCaPoClient` class was split in `DaCaPoClientBase` and `DaCaPoClient`, the former being a base class of the latter).

A more complete description of the used classes can be found directly in the implementation files (see Appendix B).

NOTE: for most of the listed methods, the parameters are not given.

5.4.1.1 The DaCaPoClient Object

See Section 5.4.2 on page 22 for a functional description of the `DaCaPoClient` class. All attributes and methods are illustrated in Table 3.

Table 3. DaCaPoClient Methods and Attributes

Name [attr|methode]	**Visibility**	**Description**	**“Friend”**
DaCaPoClient()	Public	Setup of a connection between the application and the Da CaPo++ core system. Two exceptions are raised if the constructor could not perform its job: “ServerNotAccessible” and “ServerAccessDenied”. Parameters are: • Security information.	None
int Close()	Public	The connection to the Da CaPo++ server is closed and proper resource deallocation is performed. No parameters.	None
int SendSecMgrRequest()	Public	Sending of a transparent message to the security manager of the core system. Parameters are: • Buffer and its size	None
~DaCaPoClient()	Private	Destructor	None
int coreApplicationId	Private	Global identifier of the application (on a single machine)	Session
int open()	Private	Creation of the UNIX socket and connecting to DaCaPoServer (blocking call). Return value is either SUCCESS or FAILURE. Parameters are: (no parameters)	Session Flow
int sockFd; int servLen struct sockaddr_un unixPathServerAddress	Private	Attributes for the UNIX domain socket creation	None

Table 3. DaCaPoClient Methods and Attributes

Name [attr|methode]	Visibility	Description	"Friend"
int close()	Private	Closing of the UNIX socket. Return value is either SUCCESS or FAILURE. Parameters are: (no parameters)	Session Flow
int sendCtrl()	Private	Sending of a control packet over the UNIX socket Return value is either CTRL_NO_ERROR or errCode Parameters are: • (CtrlPacket*) cp	Session Flow
int recvCtrl()	Private	Receiving of a control packet over the UNIX socket (blocking call). Return value is either CTRL_NO_ERROR or errCode Parameters are: • (CtrlPacket*) cp:	Session Flow
int status	Private	Status of the DaCaPoClient object, either AUTHORIZED or NOT_AUTHORIZED	None
EventDemux *eventDemux	Private	Responsible for the event processing in the application.	None

5.4.1.2 The Session Object

All attributes and methods are provided in Table 4.

Table 4. Session Methods and Attributes

Name [attr/method]	Visibility	Description	"Friend"
Session(char *configurationFile, DaCaPoClient *client);	Public	When creating a new instance of a session, the only parameters to transmit to the object constructor are the DaCaPoClient object and the contents of the configuration file to set up all flows with their application requirements	None
Listen(ConnectInfo conInfo, EventAction *upcFunc)	Public	This function is invoked exclusively by the session's CREATOR. It can be used for both unicast and multicast cases. All provided addresses are on the one side the group address and service (only for multicast), the local address and service and the local interface (either `machine_e` or `machine_a` for IP and ATM respectively). finally, the upcall function is responsible to process incoming events from Da CaPo++ system.	None
Connect(ConnectInfo conInfo, EventAction *upcFunc)	Public	This function is invoked exclusively by the session's PARTICIPANT. It can be used for both unicast and multicast cases. All provided addresses are on the one side the group address and service (only for multicast), the remote address and service (those of the CREATOR), the local address and service and finally the local interface (either `machine_e` or `machine_a` for IP and ATM respectively). Finally, the upcall function is responsible to process incoming events from Da CaPo++ system.	None

Table 4. Session Methods and Attributes

Name [attr/method]	**Visibility**	**Description**	**“Friend”**
Configure()	Public	All flows belonging to this session are now configured. In case of multicast, this can be performed only once as no reconfiguration is authorized. In unicast case, only the flows whose requirements were modified are newly reconfigured.	None
Activate()	Public	All A-modules (T-modules for receiving flows) are activated and start sending data or “displaying” incoming data (this corresponds actually to the activation of the lift algorithm). A test is performed to see if all incoming flows were properly initiated with the necessary upcall functions to process incoming data and control information.	None
Deactivate (int stopWay)	Public	All sending A-modules are stopped, meaning that no new data is accepted from the A-module. According to the stopWay parameter (graceful of graceless), the already present data in the graph is either transmitted or the lift is simply stopped. A stopped session can be reactivated with an ActivateSession() call.	None
Close()	Public	The session is deallocated (e.g., the connection with peer is destroyed, the protocol graphs resources are returned to the system). It is only possible to perform a Close() if the session was before deactivated.	None
GetFlowDescriptor (char *FlowName)	Public	The goal of this function is to provide a flow descriptor for a given flow. If not available, each reference to a flow through the char *FlowName would imply a loop on all current flows which each time a string comparison. With this function, this expensive process is performed only once.	None
SetReqFlow(int FlowDescriptor, “QoS value”)	Public	Single requirements are transmitted to the lower API (requirement identifier, min/max values, weight function) through the `SetReqFlow()` command. These requirements are stored in the flow table of the lower API. The application requirements are normally set during the session creation through the configuration file, but for further reconfiguration (if allowed), it is necessary to have the possibility to change at any time	None
GetReqFlow(int FlowDescriptor, “QoS value”)	Public	The GetReqFlow() function returns the actual configured values of required attributes, they may be identical to those set by SetReqFlow().	None

SendDataFlow(int FlowDescriptor, DataPacket *dp)
RecvDataFlow(int FlowDescriptor, funcPtr *upcFunc)
	Public	Sending/Receiving of data to/from the corresponding A-module. Both these functions are only provided by the flows which receive/send data from/to the application.
		The upcall function processes incoming data from A-module in the application.
			None

Table 4. Session Methods and Attributes

Name [attr/method]	Visibility	Description	"Friend"
SendCtrlFlow(int FlowDescriptor, CtrlPacket *cp) RecvCtrlFlow(int FlowDescriptor, funcPtr *upcFunc)	Public	Sending/Receiving of control information to/from the corresponding A-module. The upcall function processes incoming control data from A-module (it has always to be provided as control information is only processed by the application). Control and data are sent to the A-module asynchronously (on two different channels). To avoid losing synchronization between data and control information, a special mechanism to send control over the data channel should be available.	None
char *name	Private	Name of the session (only valid in the application)	None
int coreSessionId	Private	Global identifier of the session (on the machine)	None
int role	Private	Session's role, either SES_ROLE_CREATOR or SES_ROLE_PARTICIPANT	None
int type	Private	Session's type, either SES_TYPE_UNICAST or SES_TYPE_MULTICAST	None
FlowList flowList	Private	Linked list to all flows belonging to the session	None
ShmAreaControl *notifChannel	Private	Reception of notifications from the A-modules each time a new packet is available for this session.	None
EventCallback evHandler()	Private	Session's event handler	None
NotifDemux *notifDemux	Private	Responsible to process notifications for the arrival of ctrl/data for the flow objects.	None

5.4.1.3 The Flow Object

The flow object model is illustrated on Figure 6 on page 20.

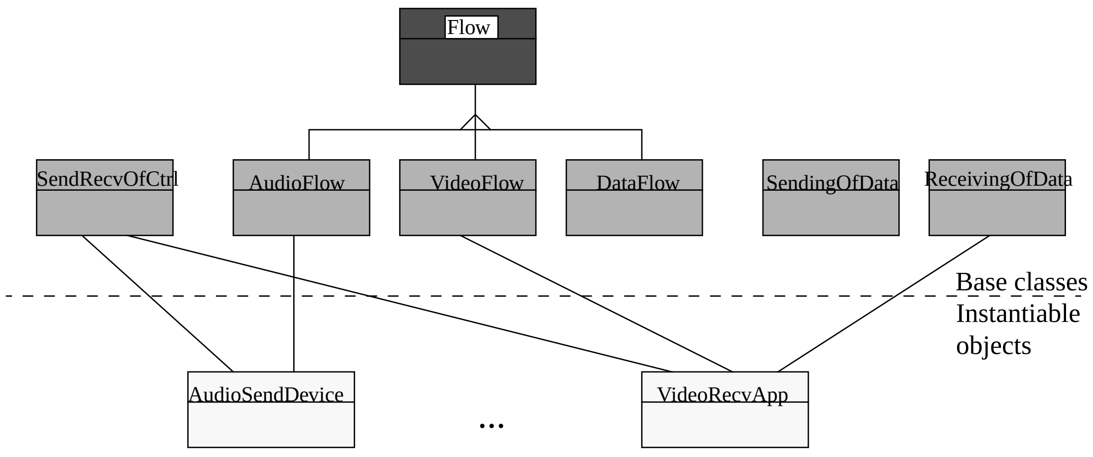


Figure 6. Flow Object Model

Each instantiable flow object may be directly derived from 6 base classes (the base class flow is only visible through the derived classes `AudioFlow`, `VideoFlow` and `DataFlow`), according to the functionality the flow has to provide. Among all base classes (`Flow` excepted), three of them specify the type of data that is transmitted over the protocol graph. The remaining three base classes are related to the sending/ receiving of data/ctrl information (NOTE: the sending and receiving of control information are encapsulated in the `SendRecvOfCtrl` base class, as there must always be a bidirectional control connection between each flow object and the corresponding A-module). In Figure 6 on page 20, the base classes

`AudioFlow`, `VideoFlow` and `DataFlow` do not currently contain any special functionality. Attributes and methods of all other classes can be found in tables starting from Table 3 to Table 7.

Table 5. Flow Class

Name [method|attr]	Visibility	Description	"Friend"
Flow(char *name, int coreSessionId, DaCaPoClient *client)	Protected	Base class for all flow objects. The constructor must know the session the flow belongs to and a reference on the DaCaPoClient object	None
~Flow()	Private	Proper deallocation of the Flow object	None
char *name	Private	Name of the flow (significance only in the session the flow belongs to).	None
int typeDesc	Private	Flow's type (AUDIO_SEND_DEVICE, ...). The data type (audio/video/data), the direction (updown, downup) and the sink/source (app, device, file) can be respectively computed by the mappingType() method.	None
int coreFlowId	Private	Global identifier of the flow (on the machine)	None
int coreSessionId	Private	Reference on the session the flow belongs to	None
DaCaPoClient *dacapoClient	Private	Each flow must know how to reach the Da CaPo++ Server, especially to register application requirements	None
int mappingType()	Private	Decomposition of typeDesc in a finer granularity (cf. typeDesc attribute above)	None
int preRegisterFlow()	Private	The new flow is temporarily registered in the lower API (without any ARs). A coreFlowId is returned which is then used for the registration of all ARs.	None
int registerAR(QoSParameter *ar)	Private	A new AR is registered in the lower API. This function expects NO acknowledge from the core system and thus is non-blocking.	None
int registerAllARs()	Private	This is a "super" function for the registration of all ARs (consists internally of successive calls to registerAR())	None
int registerFlow(ShmAreaControl *cud, ShmAreaControl *cdu, ShmAreaControl *cd)	Private	After all ARs have been properly registered, this is the definitive registration of the flow. Return values are all necessary shared areas (these are attributes of the SendRecvOfCtrl, SendingOfData and ReceivingOfData classes)	None

Table 6. SendingOfData Class

Name [method|attr]	Visibility	Description	"Friend"
SendingOfData()	Protected	This class is responsible for the sending of application data from the application to the core system. Thus, this is a base class only for the specialized flows that actually send data from the application, and not from a device or a file.	None
~SendingOfData()	Protected	Proper deallocation of object resources	None
ShmAreaControl *dataChannel	Protected	Shared area identifier for the sending of data	None
int sendData(DataPacket *dp)	Protected	Actual sending of data from the application to the core system over shared-memory.	None

Table 7. ReceivingOfData Class

Name [method|attr]	Visibility	Description	"Friend"
ReceivingOfData()	Protected	This class is responsible for receiving application data from the core system. Thus, this is a base class only for the specialized flows that actually receive data from the core system.	None
~ReceivingOfData()	Protected	Proper deallocation of object resources	None
ShmAreaControl *dataChannel	Protected	Shared area identifier for the receiving of data	None
int recvData(DataPacket *dp)	Protected	Actual reception of data from the core system in the application over shared memory.	None

Table 8. SendRecvOfCtrl Class

Name [method|attr]	Visibility	Description	"Friend"
SendRecvOfCtrl()	Protected	This class implements both the receiving and the sending of control data directly between the upper API and the corresponding A-modules. It is a base class of all specialized flow objects.	None
~SendRecvOfCtrl()	Protected	Proper deallocation of resources	None
shmAreaControl *ctrlUpDownChannel	Protected	Shared identifier for the control exchange in the direction up-down (application-core).	None
shmAreaControl *ctrlDownUpChannel	Protected	Shared identifier for the control exchange in the direction down-up (core-application).	None
int sendCtrl(CtrlPacket *cp)	Protected	Actual sending of ctrl data to the A-module.	None
int recvCtrl(CtrlFlowCallback *upcFunc)	Protected	The reception of control data from the A-module is currently only allowed through a callback function.	None

5.4.2 Da CaPo++ Access and Security

Before being authorized to do anything with the Da CaPo++ communication subsystem, an application (and therefore a user) has to be properly registered. This authentication task is performed by the core component "security manager". Due to the splitting of the Da CaPo++ core and the application in different processes, the interface of the security manager in the application is done through the `DaCaPoClient` abstraction. On successful completion, a `DaCaPoClient` object is responsible for guaranteeing a safe channel between a given application and the Da CaPo++ core system. Therefore, no application is directly allowed to communicate with Da CaPo++ (for either creating new sessions, sending application data, ...), everything is managed via the `DaCaPoClient` object. The programmer's visible interface of this object consists in its constructor and two additional methods illustrated in the following subsections. A time-sequence diagram is provided in Figure 7 on page 23 to illustrate the application registration process. In this figure, the main protocol primitives may be retrieved from Table 1.

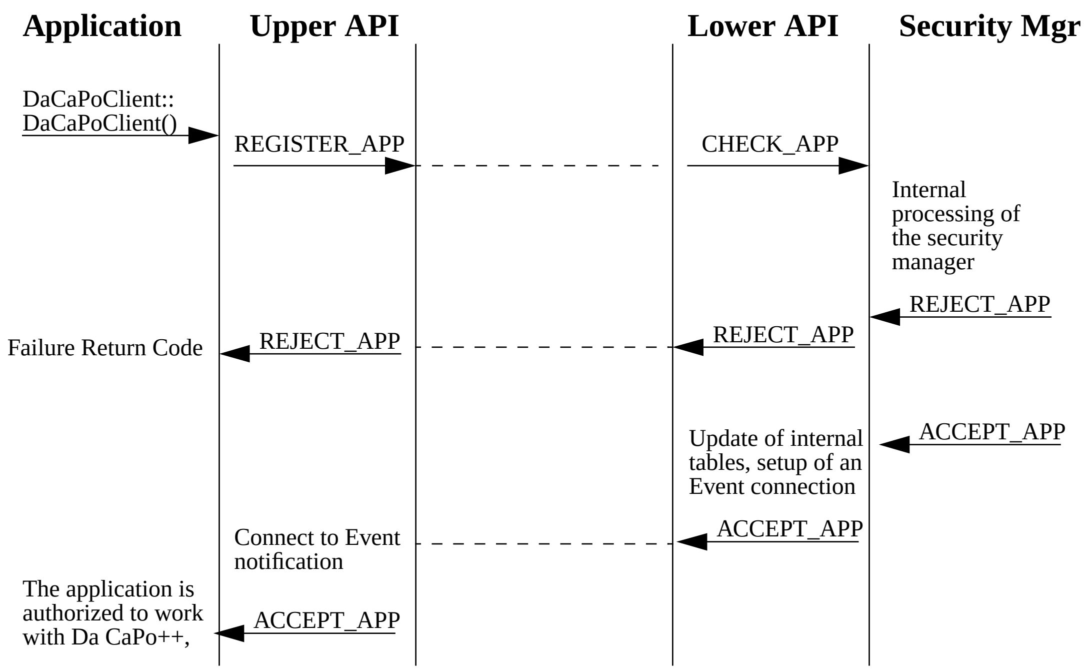

Application
Upper API
Lower API
Security Mgr
DaCaPoClient::
DaCaPoClient()
REGISTER_APP
CHECK_APP
Internal processing of the security manager
REJECT_APP
Failure Return Code
REJECT_APP
REJECT_APP
ACCEPT_APP
Update of internal tables, setup of an Event connection
Connect to Event notification
ACCEPT_APP
The application is authorized to work with Da CaPo++,
ACCEPT_APP

Figure 7. Registration of a New Application

5.4.2.1 Security Data Structure

The constructor `DaCaPoClient::DaCaPoClient(SecurityInfo &secInfo)` requires as single parameter the security information. This is provided via the `SecurityInfo` data structure:

```
typedef struct SecurityInfo_t {
     int pid;
     String64 userName;
     String64 applicationName;
     String1024 applicationCertificate;
     String64 userPassPhrase;
     String1024 publicKey;
     String1024 secreteKey;
} SecurityInfo;
```

The structures `StringXX` are used to contain variable length character strings. They are defined as follows (example with a maximal size of 64 bytes):

```
typedef struct String64_t {
     char data[64];
     int size;
} String64;
```

5.4.2.2 Instantiation of a DaCaPoClient Object

Once the constructor of the `DaCaPoClient` object is invoked with the security information (not all parameters are mandatory; for more explanations, please refer to the security design documents), a request is sent to the Da CaPo++ core system (resp. to the security manager) which then decides if the new application can be authorized or not. In case of a positive reply, the `DaCaPoClient` object is properly created and the application can continue. On the other hand, if the application cannot be authorized, an exception[1] is raised in the constructor of the `DaCaPoClient` object. This is due to pure C++ implementation reasons as no return value can be delivered during the creation of an object. There are two types of exceptions that can be raised by a `DaCaPoClient` object, namely `ServerAccessDenied` (the security manager refuses to register the application or the user) and `ServerNotAccessible` (the service access point of Da CaPo++ cannot be reached). An example on how to use C++ exceptions in the case of a constructor is illustrated below:

```
DaCaPoClient *client;
try {
     client = new DaCaPoClient( &secInfo);
}
catch ( ServerAccessDenied f) {
     // corresponding error code;
}
catch ( ServerNotAccessible f) {
     // corresponding error code;
}
// ...
```

5.4.2.3 Transparent Requests to Core Security Manager

When an application properly registered, it has the opportunity to communicate directly with the security manager. This is done in a transparent way by sending a buffer (of type `String128`) directly through both upper and lower APIs to the security manager. This latter then may generate a reply that is made available to the application in the second buffer of the `SendSecMgrRequest()` request. The format of this method is as follows:

```
int SendSecMgrRequest(String128 *requestBuf, String128 *replyBuf);
```

The maximal buffer size of 128 was arbitrarily chosen, and is thus subject to change if necessary. The return value is smaller than 0 if an error occurred either in the security manager or during the interprocess communication. A return value of 1 indicates that no failure occurred.

This method is synchronous as it only returns when an acknowledge has been properly generated by the security manager. An asynchronous way for the security manager to send messages to the application is possible via events.

5.4.2.4 Closing Connection

When an application wishes to resume communication with Da CaPo++, it has to invoke the `Close()` method of the primarily created `DaCaPoClient` object. Currently no parameters are transmitted, such as the way how to stop existing active sessions (graceful or graceless). If active components are still in use when destroying a `DaCaPoClient` object, some warning messages will be displayed to inform the user, but a graceless stop will be performed. Internally, the close request is sent to the Da CaPo++ core system which then sends an acknowledge to the application process (actually the upper API). This acknowledge is then transformed in a return value and made available to the application.

5.4.3 Session and Flow Specification

In order to specify a session, a script approach was chosen. The advantage of such a solution is to provide a human readable session definition (in comparison with a huge set of function calls). Moreover, the use of a script language allows to modify session parameters without having to recompile each time the application. The syntax of the configuration file can be found in /*Sprache zur Definition von Kommunikationsparametern*, Dario Straulino, Studienarbeit am TIK, 1996/. In this document, only an example of syntax is provided. Then, more practical topics on how to deal with a session are provided.

5.4.3.1 Configuration File Example

This example demonstrates the use of a configuration file for a picturephone application with two bidirectional flows for audio and video.

1. Due to compiler compatibility problems, it was not possible to use the C++ exception mechanisms. The elected solution consists in calling a method after each object instantiation to check the successful instantiation process (*“object”*`::CheckValidity()` delivers a value smaller than 0 in case of failure)

```
SESSION CREATOR UNICAST PicturePhone

FLOW VIDEO_SEND_DEVICE VideoSendFlow
THROUGHPUT 20000000.0 40000000.0 % WF_CONST
ENDFLOW

FLOW AUDIO_SEND_DEVICE AudioSendFlow
THROUGHPUT 20000000.0 40000000.0 % WF_CONST
ENDFLOW

FLOW VIDEO_RECV_DEVICE VideoRecvFlow
THROUGHPUT 20000000.0 40000000.0 % WF_CONST
ENDFLOW

FLOW AUDIO_RECV_DEVICE AudioRecvFlow
THROUGHPUT 20000000.0 40000000.0 % WF_CONST
ENDFLOW

ENDSESSION
```

5.4.3.2 Creation of a Session

The session creation is a pure local function, this means that no interaction with the network is performed at this level. The necessary data structures are reserved and initialized in both upper and lower APIs.

For a successful session creation, the constructor of the `Session` object needs an error-free session configuration file and a completed `DaCaPoClient` object. The format of this constructor is as follows:

```
Session::Session(char *configurationFile, DaCaPoClient *client);
```

The parameter `configurationFile` is the name of the file (with full path). The `client` parameter is a reference on the primarily created `DaCaPoClient` object. As for the `DaCaPoClient` constructor, exceptions[1] are raised if any problem occurs in the constructor's code. These exceptions are `WrongFileSyntax` (if the configuration file does not follow EBNF defined syntax), `AccessServerDenied` (if the reference on the DaCaPoClient object is not valid) and finally `SessionNotCreated` (if any internal error occur). The session constructor must therefore be invoked in the following way:

```
Session *ses;
try {
     ses = new Session( configurationFile, clientRef);
}
catch ( WrongFileSyntax f) {
     // corresponding error code;
}
catch ( ServerAccessDenied f) {
     // corresponding error code;
}
catch ( SessionNotCreated f) {
     // corresponding error code;
}
// normal execution
```

An illustration of the session creation process is provided in Figure 8 on page 26.

1. Due to compiler compatibility problems, it was not possible to use the C++ exception mechanisms. The elected solution consists in calling a method after each object instantiation to check the successful instantiation process (*“object”*`::CheckValidity()` delivers a value smaller than 0 in case of failure)

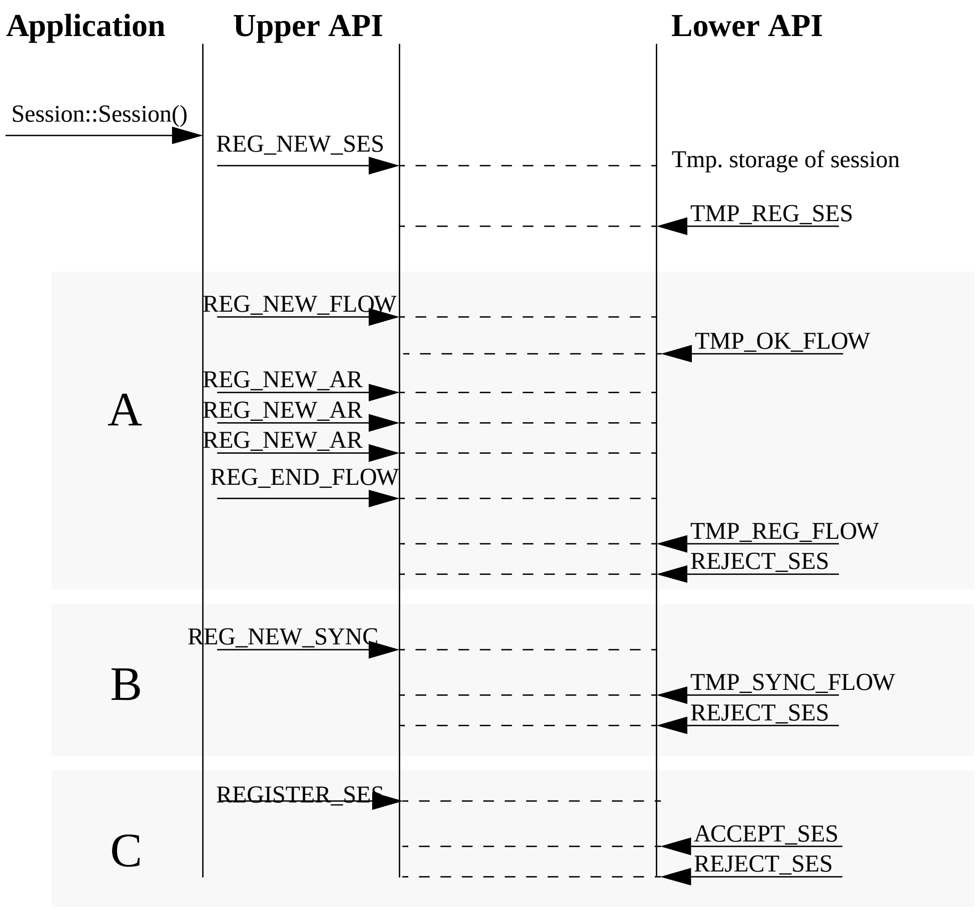


Figure 8. Registration of a New Session

5.4.4 Connecting to Da CaPo++ Core System

Once a session is properly instantiated, interaction with the network can be performed via the `Listen()` and `Connect()` methods. The format of these methods is as follows:

```
int Listen(ConnectInfo *conInfo, EventCallback upcFunc);
int Connect(ConnectInfo *conInfo, EventCallback upcFunc);
```

As events may be generated asynchronously by the Da CaPo++ core system, an event handler must be provided for each session. The type of this event handler is `EventCallback` and will be introduced in Section 5.4.5 on page 29.

5.4.4.1 Connect Data Structure

All necessary information to retrieve communication partners is contained in the `ConnectInfo` structure:

```
typedef struct ConnectInfo_t {
     String64 grpAddress;
     String64 grpService;
     String64 localAddress;
     String64 localService;
     String64 remoteAddress;
     String64 remoteService;
     String64 localInterface;
     String64 remoteInterface;
} ConnectInfo;
```

Not all fields are mandatory according to the communication type (MULTICAST or UNICAST, ...)

5.4.4.2 Connect/Listening Process

This process is strongly related with the one in the core system. All relevant parameters of the connect()/ listen() functions are properly translated in internal Da CaPo++ attributes, which are then used in the corresponding transport infrastructures (T-modules). For more information over this translation process, see Section 5.5.5 on page 33.

5.4.4.3 Session's Runtime Environment

Once a session is properly connected, a set of functions (actually all public methods of the `Session` class, see Table 4) are provided to "control" the runtime environment of the corresponding session.

5.4.4.4 Session Configuration

No effect, as the core system currently does not allow [re]configuration.

5.4.4.5 Session Activation/Deactivation

Activation means starting of data transfer between application, A-modules and T-modules. Sessions are activated and deactivated by respectively:

```
int Activate();
int Deactivate(int way);
```

The `way` parameter of `Deactivate()` tells if a graceful or graceless deactivation of the session has to be performed. Thus, `way` can have either the value `GRACEFUL` or `GRACELESS`.

5.4.4.6 Sending/Receiving of Control/Application Data

Flows belonging to a session can only be addressed via their names (ASCII string in the session configuration file) through the session interface. To avoid having to perform many costly string operations (string comparisons) each time a flow is invoked, a bypass via a flow descriptor has been set up. Therefore one has to first retrieve the flow descriptor with the `GetFlowDescriptor(char *flowName)` before sending or receiving anything on or from the flow. This method is defined as follows:

```
int GetFlowDescriptor(char *flowName);
```

The parameter `flowName` represents the file name as in the session configuration file, the integer value returned by this function has then to be used for all subsequent calls to the flow `flowName`. A return value of zero means the requested flow could not be found.

5.4.4.7 Sending of Data (Up-Down Direction)

The sending of either application data or control data directly to the Da CaPo++ corresponding A-modules is performed through the following session methods:

```
int SendData(int flowDescriptor, DataPacket *dp);
int SendCtrl(int flowDescriptor, CtrlPacket *cp);
```

Again, both DataPacket and CtrlPacket are data structures which allow for a maximum packet size as illustrated below:

```
typedef struct DataPacket_t {
     char data[MAX_DATA_PACKET_SIZE];
     int size;
} DataPacket;
```

and

```
typedef struct CtrlPacket_t {
        char data[MAX_CTRL_PACKET_SIZE];
        int size;
    } CtrlPacket;
```

Both constants `MAX_DATA_PACKET_SIZE` and `MAX_CTRL_PACKET_SIZE` currently have values of 512, which is subject to be changed. It must be reminded that not all flow types allow for the sending of application data.

If failure occurs, both above functions return a value smaller that zero.

5.4.4.8 Receiving of Data (Down-Up Direction)

Receiving of either control data or application data is currently only possible via upcall functions (asynchronous mode). This means that it is mandatory for each receiving flow to provide such a callback function to process incoming data. Therefore, the interface to both receiving functions is as follows:

```
    int RecvData(int flowDescriptor, DataFlowCallback upcFunc);
    int RecvCtrl(int flowDescriptor, CtrlFlowCallback upcFunc);
```

The format of the upcall functions is illustrated below:

```
    typedef int (*DataFlowCallback)(DataPacket *dp);
    typedef int (*CtrlFlowCallback)(CtrlPacket *cp);
```

An C++ example on how to set up such a callback function is provided in the following example:

```
    // declaration of the callback function
    int DataPacketHandler(DataPacket *dp) {
        // handler’s code
    }

    // Setting of the callback function
    DataFlowCallback dfCbPtr = &DataPacketHandler;
    if ((errCode = ses->RecvData(flowDesc, dfCbPtr)) < 0) {
        // error when setting callback function
    }
```

It must be reminded that not all flow types allow for the receiving of application data direct in the application.

Both functions return values smaller than zero if any error occurred during set up. If no callback function is provided for a flow, an error is raised during run-time if data happens to be received by this flow.

5.4.4.9 Setting/Getting Single Application Requirements

Two functions are provided for setting and getting single application requirements for a flow. These may be invoked before a reconfiguration to adapt some QoS parameters. However they are currently not implemented.

```
    int SetReqFlow(int flowDescriptor, char *reqToSet);
    int GetReqFlow(int flowDescriptor, char *reqToGet, char *reqValue);
```

Setting new application requirement means simply erasing the older value, if there was any. Getting a value means getting the configured one, not the one that figures in the session configuration file (it is left to the responsibility of the application to keep such values, if necessary).

5.4.4.10 Closing of a Session

After a session is created, it is possible to remove it and to free the allocated resources. This is done by the `Close()` method of the session object. There are currently no parameters available (such as graceful or graceless) as it is assumed that the programmer already de-activated the session. As for all previous functions, a close request is sent to the core Da CaPo++ system and an acknowledge is returned to confirm or not the close of the session.

5.4.5 Event Handling

As already mentioned in this document, the Da CaPo++ core system has the possibility to send asynchronous data to applications encapsulated in so-called events.

5.4.5.1 Event Data Structure

The event class structure has the following definition:

```
class Event {
     public:
          Event();
          ~Event();
          int type;
          String128 parameter;
}
```

A simple C structure would have been sufficient to characterize this data type, however a C++ class may eventually allow for defining operations on the events.

An event consists mainly of a type and a parameter field. There is a set of predefined event types that are generated by the Da CaPo++ core system (this list is not yet available). Again, the maximum 128 bytes available for the parameter is an arbitrary limit that may be changed if necessary.

As for the receiving of data from the A-modules, a callback function is provided to process incoming events. These upcall functions have the following format:

```
typedef int (*EventCallback)(Event *ev);
```

As the use of such callback functions is identical to the receiving of data case, no example is provided here.

5.4.5.2 Session's Event Handler Registration

A session's event handler always has to be registered during the `connect()/listen()` process. The upper API maintains the event handlers for each session in a static table. The static table has a fixed maximal size and the event handler can be retrieved at the index corresponding to the core session identifier of the session (provided by the core system). This scheme is not the most efficient in terms of memory utilization (only few entries of the static table are actually relevant), however the retrieval of the right event handler can be performed almost instantaneously via pointer arithmetic operations.

5.5 Lower API

The lower API part is the front end of the Da CaPo++ core system. It deals mainly with the registration of applications, the setup of sessions and the control during runtime. The lower API consists mainly in a service access point (the DaCaPoServer component, see Section 5.5.1 on page 29), an internal data management part (storage of sessions/flows properties, see Section 5.5.3 on page 32) and a notifier component (see Section 5.5.2.1 on page 31).

As represented in Figure 1 on page 8, the A-modules are not part of the lower API. However, as they are the data access points for the application and as they have to closely collaborate with the lower API, especially during the configuration process, their specification will be considered in Section 5.5.2 on page 31.

5.5.1 Service Access Point

The Da CaPo++ service access point is a well-known address on each machine. It is the end-point for the main control protocol between each application process and the Da CaPo++. As this service access point has to deal concurrently with several applications, two different approaches were considered:

- There is only one finite state machine (FSM) which deals with all applications. This means that special care must be taken to guarantee a fair access to the Da CaPo++ server for each application. This can be done by using timers for each transaction.
- There is one control process for each application. This means that a private instance of the FSM is allocated for each application. This guarantees the fair access to the server without the burden of having to use timers. On the other hand, the concurrent execution of several instances of the FSM requires a greater attention from the programmer (no global variables may be used, only dynamic and private instances).

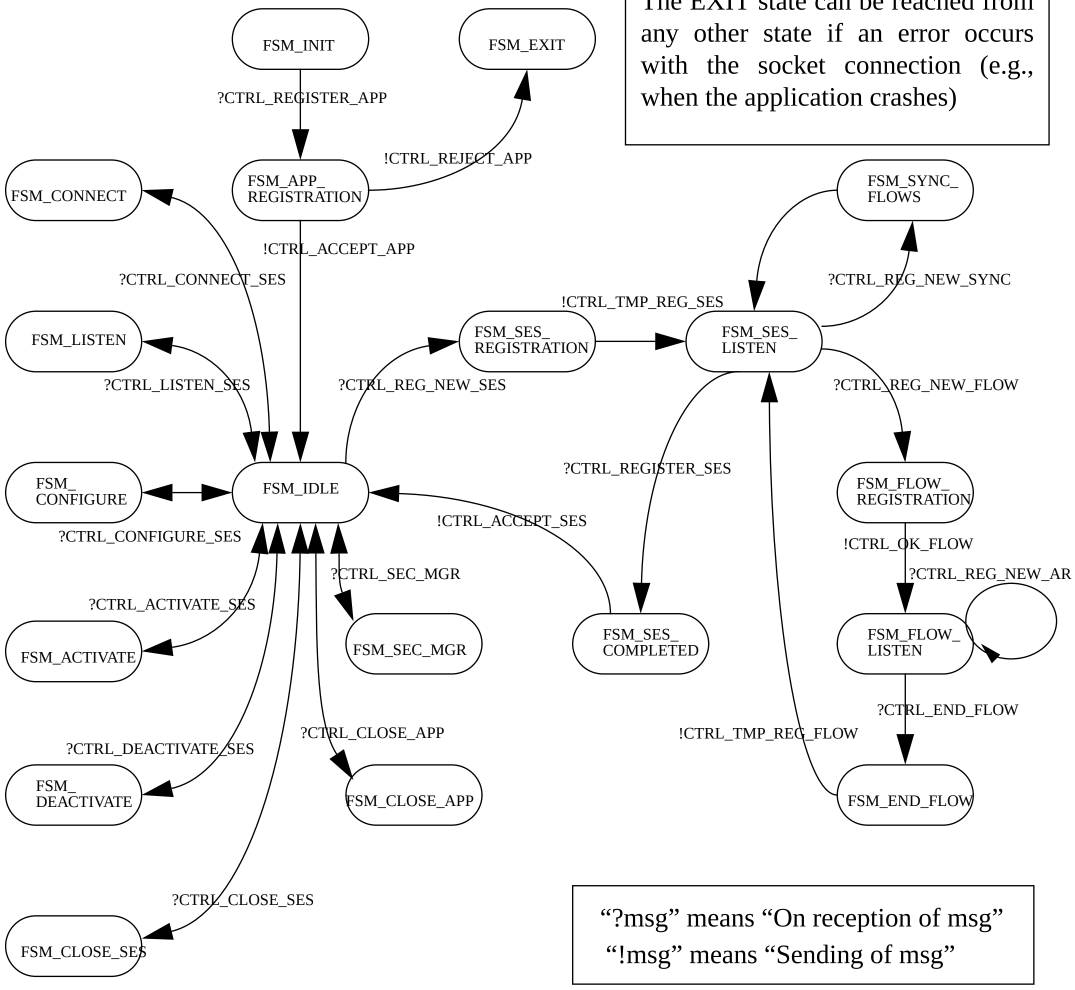
Because of its flexibility, we chose the second variant with a DaCaPoServer thread for each application.

5.5.1.1 DaCaPoServer's Finite State Machine

The exact specification of the FSM is illustrated in Figure 9 on page 30. The initial state is FSM_INIT. When the a new connection is accepted at the UNIX domain socket, a CTRL_REGISTER_APP control packet is sent by the application with all the security information.

Note:

The EXIT state can be reached from any other state if an error occurs with the socket connection (e.g., when the application crashes)



"?msg" means "On reception of msg"
"!msg" means "Sending of msg"

Figure 9. DaCaPoServer's Finite State Machine (FSM)

5.5.2 Data Access Points

Although the data access points (A-modules) are not part of the lower API, here are some considerations. During the configuration process, each flow, according to its type, is allocated 2 or 3 different shared-memory areas (if the A-module expects user data directly from the application, 3 areas are required, in other cases where only a bidirectional control channel is required, 2 areas are sufficient).

An important detail to mention here is the way an A-module notices either user or control data is available from the application. One solution would be to have a receiving thread for each A-module that would simply wait for the access semaphore. We rejected this solution because of the overhead created by this amount of additional threads (actually, some A-modules would have required 2 threads, for both user and control data). To circumvent this problem, the notification approach was elected.

5.5.2.1 Notification Process

As it can be seen on Figure 1 on page 8, there is in the lower API a notifier component. There is one instance of this component for each session (implemented as a thread). The notifier listens permanently on a shared-memory channel (over the access semaphore) and processes notifications from the upper API. Such a notification has the format illustrated in Figure 10 on page 31.

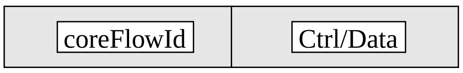


Figure 10. Notification Format

On reception of a notification, the notifier component has to inform the A-module corresponding to the coreFlowId flow that either control or data information was sent by the upper API (Ctrl/Data field of the notification).

The complete process of the notifier is now depicted in Figure 11 on page 31.

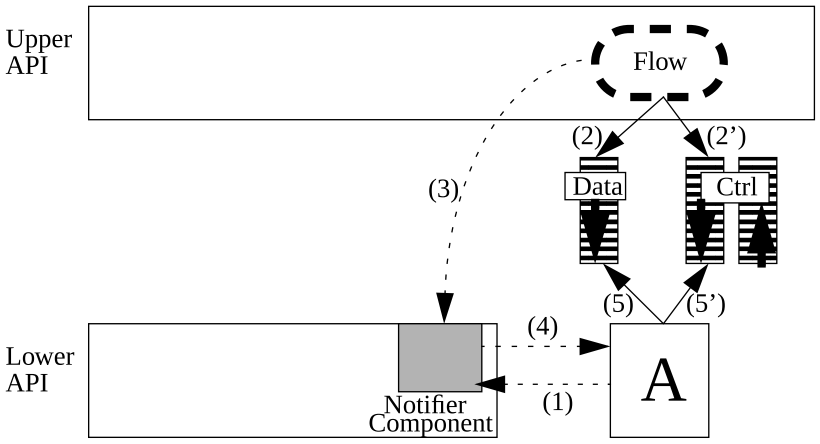


Figure 11. Notification Process

First, each A-module must register itself towards the notifier component by providing two different callback functions for control or data information (1). When a flow writes data in a shared-memory, either (2) or (2'), it then sends a notification to the lower API, specifying its coreFlowId and if it wrote control or user data (3). On receiving a notification, the notifier invokes the callback function of the corresponding A-module according to the type of data (4). It is then the task of the callback functions to go and read data in the shared-memory (5-5'), and to either process it locally (e.g., the control information may adjust a parameter of the A-module, or it may be sent out of band to the peer A-module) or to push it in the Da CaPo++ lift (typically for user data).

5.5.3 Internal Data Management

After a session was registered in the Da CaPo++ core, the lower API has to keep track of all information related to the session. This is either static (list of flows, application requirements, ...) or dynamic information (status of the session, either connected or not, ...). This data is stored internally in three dynamic lists according to the model illustrated in Figure 12 on page 32.

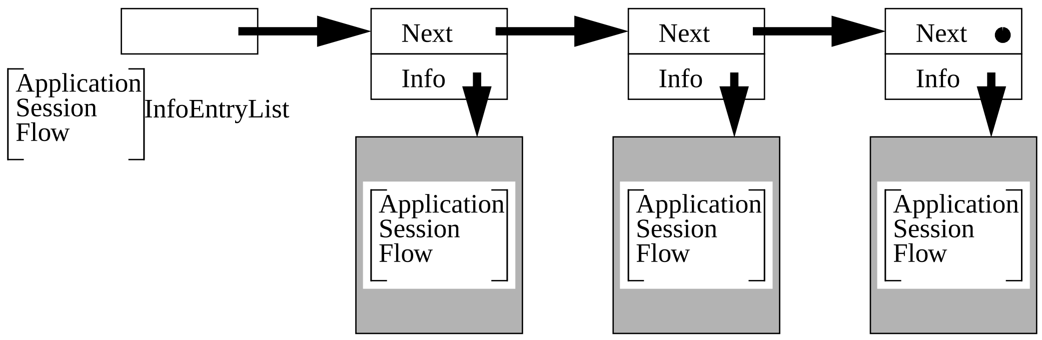


Figure 12. Internal Data Management Structures

There are three different lists: for the application, session and flow data. The contents of the application data is given in Table 9. Similarly, the contents of the session data in Table 9, and finally, the contents of the flow data in Table 10 and Table 11.

Table 9. Application Data

Field	Description
int coreApplicationId	Global identifier for the application. This global identifier is only valid on the machine where the Da CaPo++ process is active.
intList *sessionList	A list of all sessions that belong to the application. IntList is a type that represents a simple list of integers. Each session is characterised by its unique coreSessionId.
int applicationStatus	Current status of the application. Possible values are: STAT_APPLICATION_IDLE, STAT_APPLICATION_ACTIVE, STAT_APPLICATION_CLOSED
ShmAreaControl *eventChannel	Identifier of the shared-memory channel where events have to be sent to the application.
SecurityInfo *secInfo	Reference on the security data that were provided when the application registered itself.

Table 10. Flow Data (1)

Field	Description
int coreSessionId	Global identifier for the session. This global identifier is only valid on the machine where the Da CaPo++ process is active.
int coreApplicationId	Reference on the application the session belongs to.
IntList *flowList	A list of all flows that belong to the application. IntList is a type that represents a simple list of integers. Each flow is characterised by its unique coreFlowId.
int sessionStatus	Current status of the session. Possible values are: STAT_SESSION_NOT_CREATED, STAT_SESSION_CREATED, STAT_SESSION_CONFIGURED, STAT_SESSION_CONNECTED, STAT_SESSION_LISTENING, STAT_SESSION_ACTIVATED, STAT_SESSION_DEACTIVATED, STAT_SESSION_CLOSED,
int sessionType	Type of the session, either SES_TYPE_MULTICAST or SES_TYPE_UNICAST

Table 10. Flow Data (1)

Field	Description
int sessionRole	Type of the session, either SES_TYPE_CREATOR or SES_TYPE_PARTICIPANT
ShmAreaControl *notifChannel	Shared-memory channel where notifications have to be sent
Dc_ServicePtr servicePtr	Reference on the Da CaPo++ core structure service (equivalent to the session abstraction)

Table 11. Flow Data (2)

Field	Description
int coreFlowId	Global identifier for the flow. This global identifier is only valid on the machine where the Da CaPo++ process is active.
int coreSessionId	Reference on the session the flow belongs to
int flowStatus	Current status of the flow. Possible values are: STAT_FLOW_CREATED
int flowType	Type of the flow: either FLOW_AUDIO_TYPE, FLOW_VIDEO_TYPE or FLOW_DATA_TYPE
int flowDirection	Direction of the flow: either SHM_UPDOWN or SHM_DOWNUP
int flowSinkSource	Sink/Source of the flow: either FLOW_FROMTO_DEVICE, FLOW_FROMTO_FILE, FLOW_FROMTO_APP
ShmAreaControl *dataChannel	Shared-memory to send/receive user data (direction depending of the flowDirection field). This field is equal to NULL if the flowSinkSource field is different of FLOW_FROMTO_APP
ShmAreaControl * ctrlUpDownChannel	Shared-memory for the control information transfer from the application to the A-module.
ShmAreaControl *ctrlDownUpChannel	Shared-memory for the control information transfer from the A-module to the application.
QoSList *qosList	List of all application requirements that were provided by the application.
Dc_ProtocolPtr protocolPtr	Reference on the Da CaPo core protocol graph (equivalent to the flow abstraction)

5.5.4 Event Handling

It was already mentioned several times in this document that the Da CaPo++ core system has the possibility to send asynchronous events to the application. The format of an event is illustrated in Figure 13 on page 33. Thus the corresponding C-type `EventInfo` consists of the coreSessionId (events are always

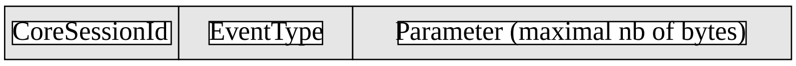


Figure 13. Format of an Event

related to sessions), an event type and a single parameter (which is a string of fixed maximal size). The interface in the lower API to send events is as follows:

```
int lapi_SendEvent(EventInfo *ev);
```

5.5.5 Integration in the Da CaPo++ Core System

Most of the design process of the API was done in close relation to the Da CaPo++ core system. However, special care was taken during the implementation to clearly separate between a general purpose application programming interface (allowing service definition and efficient interprocess communications) and the Da CaPo++ communication subsystem. As a consequence, most Da CaPo++ specific issues are grouped in a single C-file (`dcapi.c`). The other Da CaPo++ parts can be found in the notification process.

The integration consists mainly in “translating” the upper API’s objects (sessions, flows) into their counterparts in the Da CaPo++ core (services, protocol graphs). Then, the application requirements also have to be “translated” into attributes. Once this process is completed, the lower API just has to invoke the internal core API. Hence, the core system cannot distinguish that it is managed by an application located in another process.

6. Security Aspects of Da CaPo++

It is assumed that the reader is familiar with section 3 of the Da CaPo++ Detailed Design. This section describes especially the functionality that (1) was not implemented or (2) that was implemented not conforming to the specifications in the Detailed Design document.

6.1 Authenticity and Keys

Contrariwise to the design, authenticity and integrity of applications has not been covered in the implementation. The mechanisms to allow the security manager a verification of the application exist, but are not used. Likewise the separate user-identification and authentication interface only exists in a very rudimentary form, applications are expected to provide user authentication data in form of public key ID's and matching passphrases to decrypt the private key.

While Da CaPo++ is in principle able to communicate in a secure manner, even if the applications are not security-aware, this feature has not been exploited by the realized system. The class of peer authenticity required for a communication to succeed is always 'user authenticity', as described in section 3.1.1 of the design. The Da CaPo++ user interface to control associations (user vs application vs protocol) is realized in Da CaPo++, and can be used to force dissociation, or changes in the behavior of secure protocols. It interfaces directly with the communication thread of the security manager.

Key management works as specified. The GMS (see section 3.2.5 of the design) is not integrated.

6.2 Requirements, Configuration, and Runtime Issues

QoS translation and dynamic instantiation of protocols (presumably done by CoRA) do not exist in Da CaPo++. For this reason, the application configuration file (see section 6.3 for security relevant parameters) already specifies LLRs, (see section 3.2.2.2 in the design). It is assumed that a separate application provides the translation from AARs to LLRs, when the configuration of the other communication-relevant parameters takes place. If an application currently specifies AARs instead of LLRs, a default set of (strong) LLRs is assumed and enforced by the instantiated relevant communication modules.

The 'small reconfiguration' mentioned in section 3.1.3 of the design exists, albeit the initiating mechanism is not integrated with the communicating application, but with the user-interface of the security manager instead.

The 'security association block' provides for key changes and notifications as specified, this logical functionality is split over the actual C modules, the security manager itself and the Da CaPo++ runtime system respectively.

6.3 API Configuration and Behavior

As mentioned above, the configuration files are the mechanisms of choice to transfer application requirements to the core system. Besides the abstract parameters 'privacy' and 'authenticity', which are only used to enable default parameter sets, the following value types exist:

- CRYPT_ALG
  Specification of the algorithm to use, may be extended by '-keysize-rounds' for algorithms such as RC5.
- AUTH_ALG
  Defines the authentication (hashing) algorithm to use, when '-asymm' is attached, asymmetric authentication is performed in the core.
- OWN_USERID

- PEER_USERID
  These two IDs are used by the protocol to ask the right keys from the security manager, and are also transferred to the communicating peer to check for matching identities on both sides.
- The value of these parameters is passed via API to the instantiated module graph, where it is evaluated by the C modules themselves. It is required that both sides use exactly the same requirements. Mismatches are detected and signalled to the application.
- While asymmetric authentication is realized in the core, the forwarding of return-receipts is not done. It would have to be added as an event to the API. Other events such as data received that is incorrectly authenticated, participant join and leave are operational.

6.4 Protocols

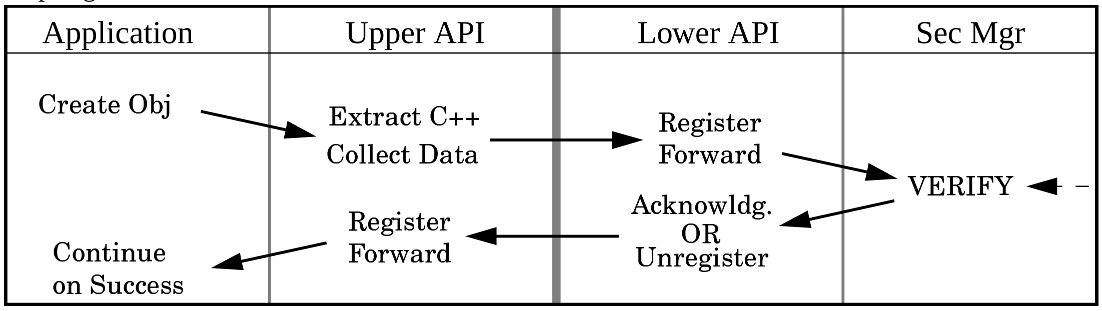
The following protocols have been used for the demonstrations:

Private Audio:	Datagram transfer with ECB or CBC encryption included.
Authenticated Text:	Reliable data transfer with asymmetric authentication included.
Secure text:	Reliable data transfer with asymmetric authentication and strong encryption.

An arbitrary protocol graph (*e.g.*, video) may additionally be secured simply by adding the appropriate modules, as described in section 3.2.3 of the design. The available modules are ‘ECB’ and ‘CBC’, offering DES, RC5 and IDEA, and ‘MD’ offering MD4 and MD5 as hashing algorithms and optionally authentication via RSA. Key exchanges are sender-initiated and done via RSA, the session keys being sent by out-of-band transmission. A DH key-agreement module for the use in the connection manager protocol exists but is not used. A stream cipher implementation RC4 is also available as module. MD4 has been broken in February 1996, which has been foreseen during the implementation of the module before that time. For testing purposes, MD4 is still included in the final implementation.

6.5 Implementation Details

This section describes security relevant functionality of the implementation, giving graphical support. First the setup of the application–core link in terms of security relevant data flow during connection setup is given:



This clarifies section 3.3.1 and 3.3.2 of the design document. The verification step in the security manager currently only encompasses the registration of the application, the user IDs it carries, and corresponding keying material. To allow unsecured protocols to work, verification never fails. If, however, an application that did not pass on appropriate data during connection setup tries to use a secured protocol graph, the initialization of the protocol graph will fail, because of missing keying material. The application receives a corresponding fatal notice.

6.6 Da CaPo++ Start

In the following flow charts, only security-relevant operations are depicted. When the Da CaPo++ process is started (cf. Figure 14.), the action described in the flow chart are taken. The security manager (SM) is initialized, and prepares its internal data structures. This includes the start of the communication thread (CT) which offers the user interface service. Next, the generator for random data (‘RNG collector’) is started. This generator is used later on to provide session keys to the different protocols. The generator collects system data via various asynchronously executed statistics commands, and maintains a pool of ‘fresh’ random data. Finally, the Da CaPo++ core unlocks its own machine key, and performs a ‘login’ under the name of the machine it runs on. If the login fails, the security initialization is aborted, and no security-relevant function will work. In the case of success, the unlocked key can then be used to check keyring and module integrity, or check the integrity of the entire Da CaPo++ process text segment. The functions marked by ‘*’ in the flow charts have not been implemented.

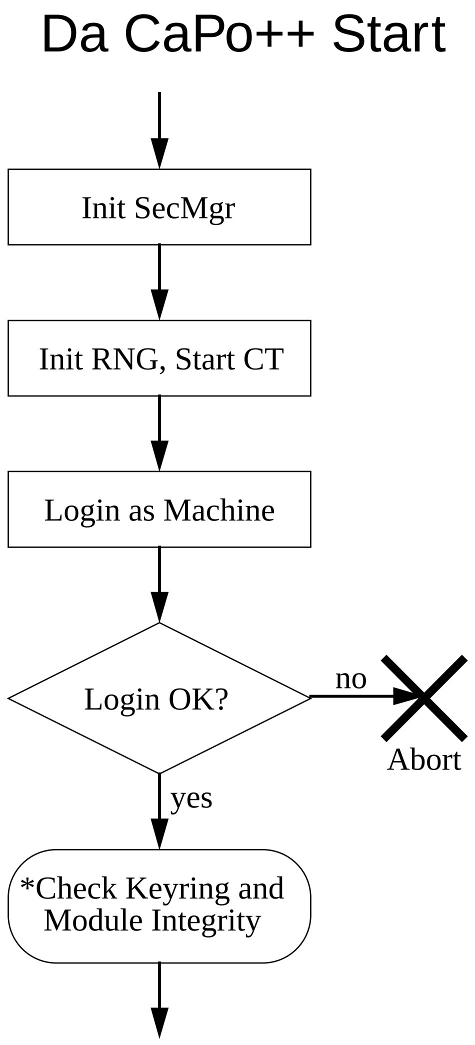


Figure 14. Da CaPo++ Start

6.7 Application Run

‘Application Run’ describes relevant actions of an application that communicates via upper and lower API with the Da CaPo++ process (cf. Figure 15.). To connect to the Da CaPo++ process, the application has to provide an authentication structure, which will be described under ‘API Connect’. If the connection succeeds, and no error message is returned from the lower API, or the SM, the application will create one or more sessions. The upper API transfers the configuration data to the lower API and creates corresponding session

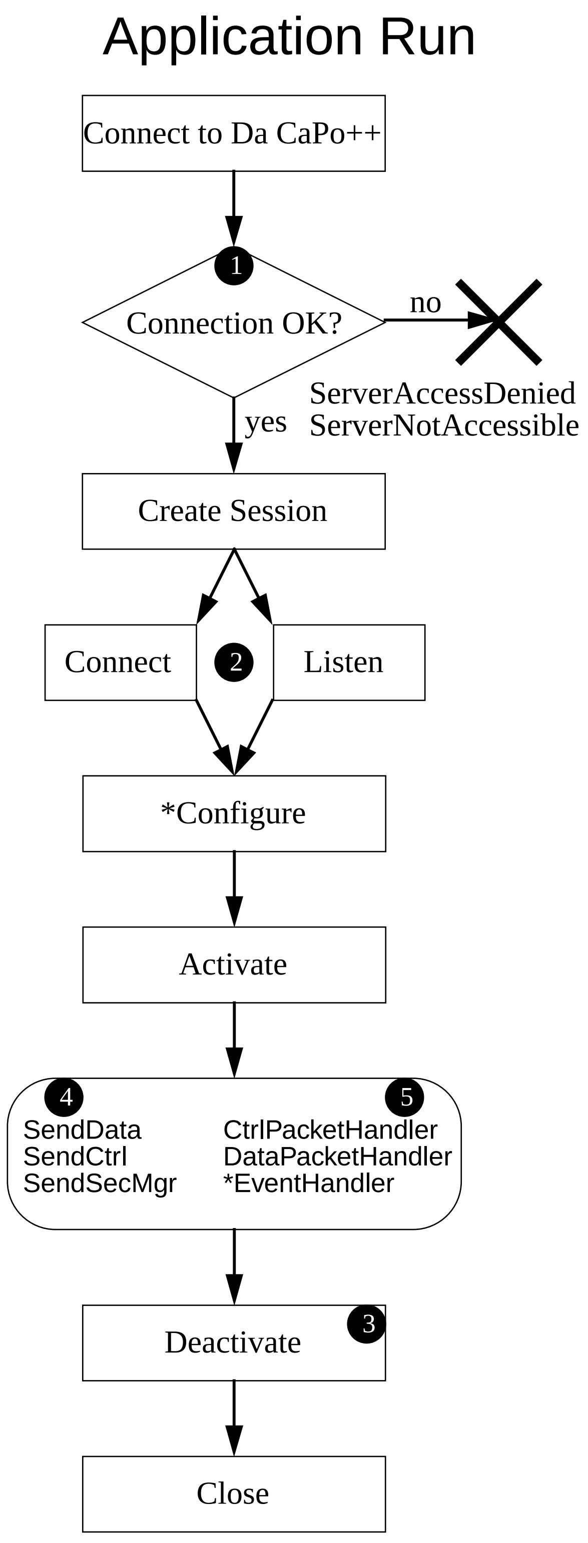


Figure 15. Application Run

and flow objects for use by the application. The following security-relevant parameters are currently understood by the lower API and translated into the aiSecurity requirement: PRIVACY, AUTHENTICATION, CRYPTALG, AUTH_TYPE, OWN_USERID, PEER_USERID.

As a next step, an application issues a ‘connect’ or ‘listen’ request, usually depending on command line parameters. During this blocking operation the two core systems contact each other, and all protocol graphs are instantiated and connected to each other. In this moment, the connection manager (CM) exchanges information of the actually employed protocols. This is also the time when a ‘Join’ event is generated. During this time, ‘protocol initialization’ is processed. After the (for now empty) configuration phase, the protocols are activated (this translates to a lift_start) and the actual data transfer takes place. It is described under ‘Send Data’ and ‘Receive Data’.

Finally, sessions can be activated and deactivated multiple times, and a close deallocates protocols (see ‘Protocol Termination’). Afterwards, a destruction of the session object leads to the deallocation of flows and session in the lower API, and destroying the connection to Da CaPo++ unregisters the application in the SM.

The numbers ❶ to ❺ in above flow charts indicate which of the numbered diagrams in the following subsections further explain the depicted functionality.

6.8 API Connect

When an application connects to the Da CaPo++ core system, the lower API itself register the application for communication purposes and the security manager (SM) registers it under a different view (cf. Figure 16.). The registration in the SM allows the user interface to kill an arbitrary application or modify its protocol properties later on. During the registration with the SM, the authorization of the application can be checked. This is not yet implemented, and would just signify that the security manager holds a certificate for this application, and can check it’s integrity at runtime. It then could consult an access control list to decide whether the application has the rights to use Da CaPo++ protocols. As a next step, the user keying information as provided by the application is processed in the SM. Either a simple ‘login’ is performed (if the keys are already held in the global database), and the private key is unlocked with the passphrase the user provided -- or the keys provided by the user are used for further communication purposes. If the login fails, no keying material is available, and, depending on the configuration of the SM, the application will be allowed to connect to the core, and use it in insecure mode only, or connection setup itself will be abandoned. If the login is successful, the application is registered within the SM and control is returned to the lower API.

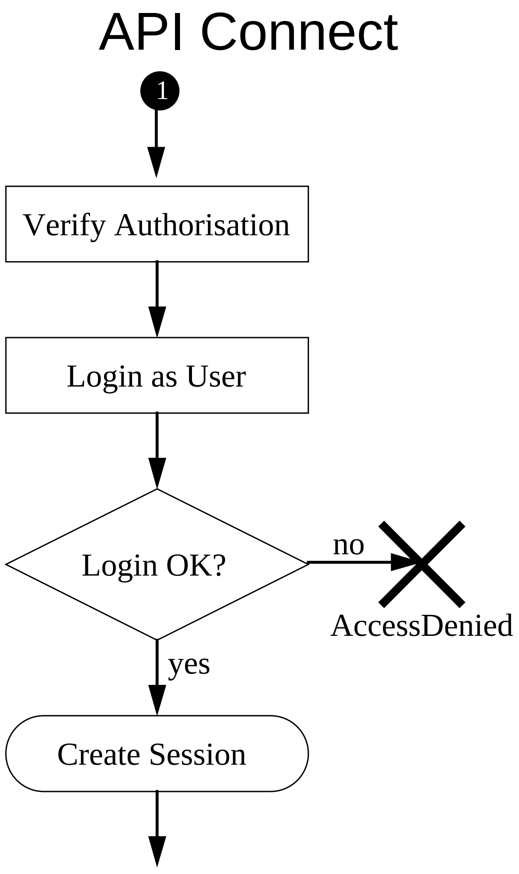


Figure 16. API Connect

6.9 Protocol Initialization

During protocol initialization (cf. Figure 17.), the encryption and authentication algorithms allocate their internal data structures. Then, protocol requirements are scanned for the user name of the person using the protocol, and this user name is then sent to the security manager, to fetch the associated password for private keying material. If the password is not available, this signifies, that no corresponding login has been performed upon connection setup between application and core, and that thus the keying material for this user is not accessible to that protocol. A dcError is created which will result in a failure to build this particular protocol. If, on the other hand, the password was available, the user name was assumed to be valid, and the asymmetric (private and public) keying material of the user are fetched from the SM. In the future a test must be included whether the protocol requesting keying material is associated with the application that did the initial login – if not, the SM may not give out the keying material. As a last step of the cryptographic specific initialization, the module registers itself with the SM, to allow for the ‘security ON/OFF function’ of the user interface to work. Note that session keys have not yet been set up, this happens when the first data packet is sent. The selection of a specific encryption and/or authentication module happens by examining the aiSecurity requirements for the corresponding switch. Assuming that CORA will not be provided by the core system, only preconceived protocols/module graphs may be used at runtime. Nevertheless a certain dynamicity is possible, as the module itself offers multiple algorithms, and a switch from one to the other may be performed at runtime in a sender-initiated small reconfiguration. But for now it is assumed that the actual module properties are specified in the configuration file of the upper API. The generation of this configuration file may be automated, and coupled with the Security QoS engine.

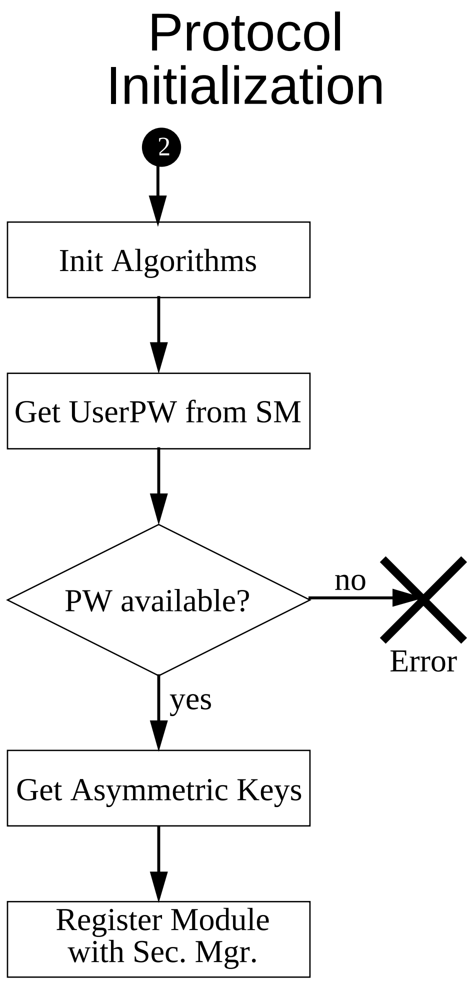


Figure 17. Protocol Initialization

6.10 Protocol Termination

During the _exit functions of the cryptographic C-modules, they are unregistered from the SM, and all keying material that is currently held by the modules is destroyed (cf. Figure 18.).

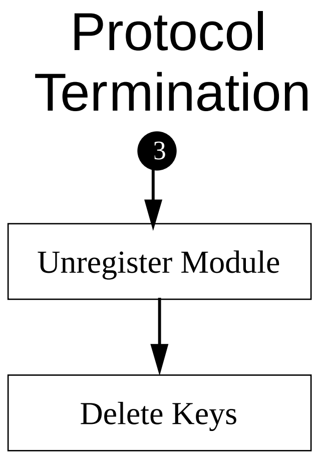


Figure 18. Protocol Termination

6.11 User Interface

The user interface to the security manager offers two functions at the moment (cf. Figure 19.). First, the key management library of the PGP subcomponent can be invoked, allowing for the handling of the certificate and key database. Second, a command can be issued to the security manager such that the security function in all modules can be toggled on or off. Toggling this for a specific protocol only is a trivial extension, as the security manager knows what protocols are registered with it, and will hold the information on what application they appertain. Another addition will be the possibility to dissociate an application from Da CaPo++ via the security manager, this is a management function that can be extend to control the actual parameters of an application and its instantiated protocols.

The user interface communicates with the Da CaPo++ via an internet socket, and currently no access control is performed. A special communication thread runs in the core system, accepts requests from the socket, and invokes the corresponding functions in the SM.

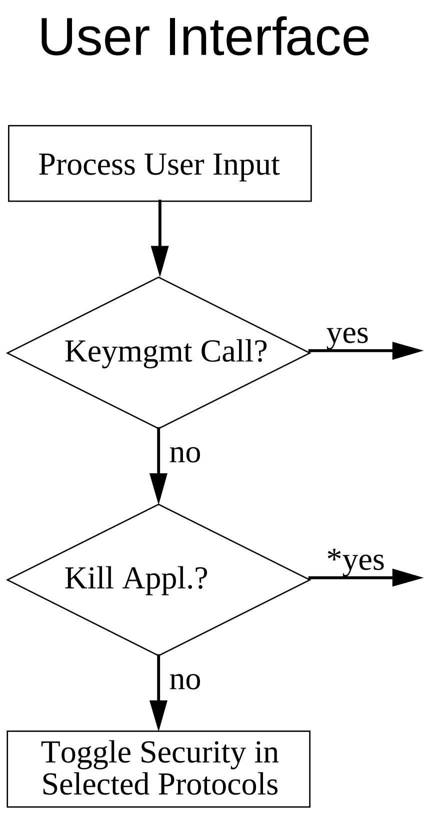


Figure 19. User Interface

6.12 Send Data

The two flow charts ‘send data’ (cf. Figure 20.) and ‘receive data’ (cf. Figure 21.)describe the operation of the encryption module. Keep in mind that the authentication module performs a very similar operation, the only difference is that the out-of-band communication flows from the receiver to the sender if receiver-nonrepudiation has been requested. Receiver-non-repudiation has not been implemented.

Processing of data to be sent happens on the ‘request’ side of the module, the indication side only forwards information to the next module. When data comes in, the security on/off flag is checked first and, if security, is disabled the data is forwarded without further processing. In the header, and indication is set that security was turned off on the sending side. If security is enabled, the module checks for the existence of a fresh session key. If none exists or the current session key is too old, a new one is generated. Part of the generation happens synchronously, namely, random material is requested from the SM, the expanded keys for the algorithms are generated using this random material as the new session key and the session key is forwarded to the control interface of the module. The control interface is marked ‘ready’ such that it will be activated when the Da CaPo++ lift checks its state the next time. As a last step, the header in the packet to be sent is marked such that the receiver will detect that a new session key is required.

When a session key is present, the data is padded and encrypted and forwarded to the next module.

Asynchronously, when the control interface is marked ready and is thus ‘executed’ by the controlling thread, the session key is signed using the private key of the module and encrypted using the public key of the peer of the module. Those keys have been gathered during protocol initialization. This encrypted key packet is then send as an out-of-band message to the peer module. If the keys to be transmitted are the first session keys, the actual IDs of the employed asymmetric keys are also transmitted, to allow for a cross-check on protocol level and minimize the risk of undetected inconsistencies of provided keying material.

6.13 Receive Data

The first performed check is whether the protocol is switched to ‘security off’ (cf. Figure 21.). Currently this is done by examining the status field of the incoming packet. If security is off, the packet is forwarded to the next module directly.

If the packet has been encrypted, the header field for the employed session key generation number is checked. If the currently available session key matches with the one employed to encrypt the packet, normal processing continues. If the session key has changed, the indication procedure of the module is replaced by one which fetches data form a session key stack, where asynchronously incoming session keys are stored by control-In, the out-of-band receiver of this module. If the correct key is not in this stack, the indication procedure will be called again a short time later. Obviously, this works only if the out-of-band data transmission is reliable, which is usually the case as the connection manager protocol is employed. The indication procedure finally initializes the cryptographic algorithms with the new session key that was fetched from the stack and normal processing continues by replacing the indication procedure.

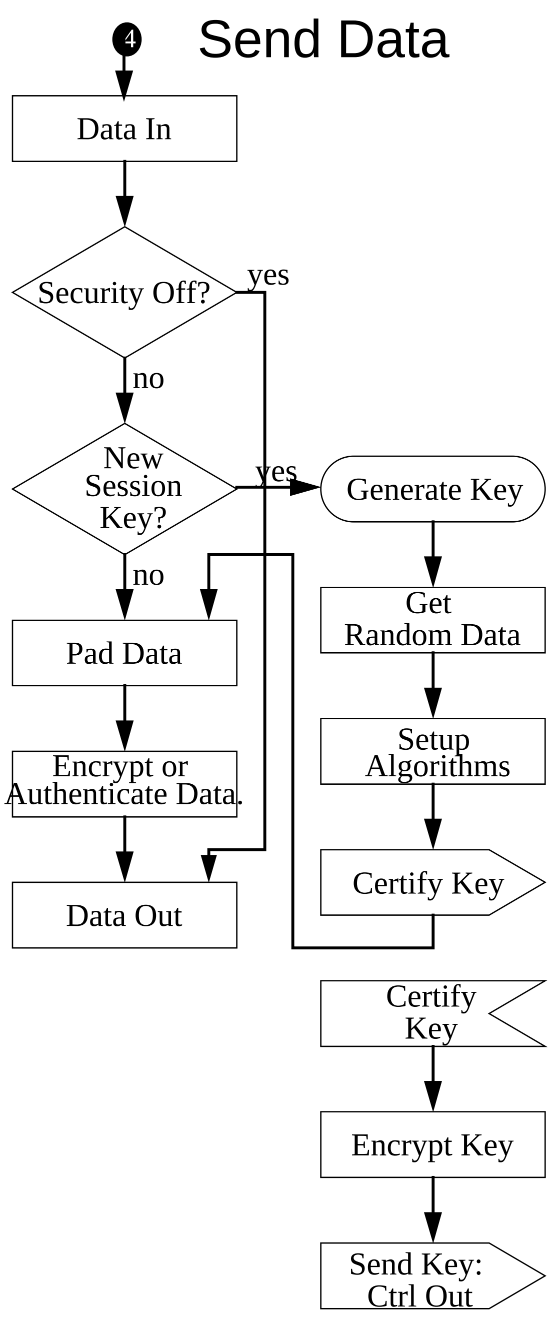


Figure 20. Send Data

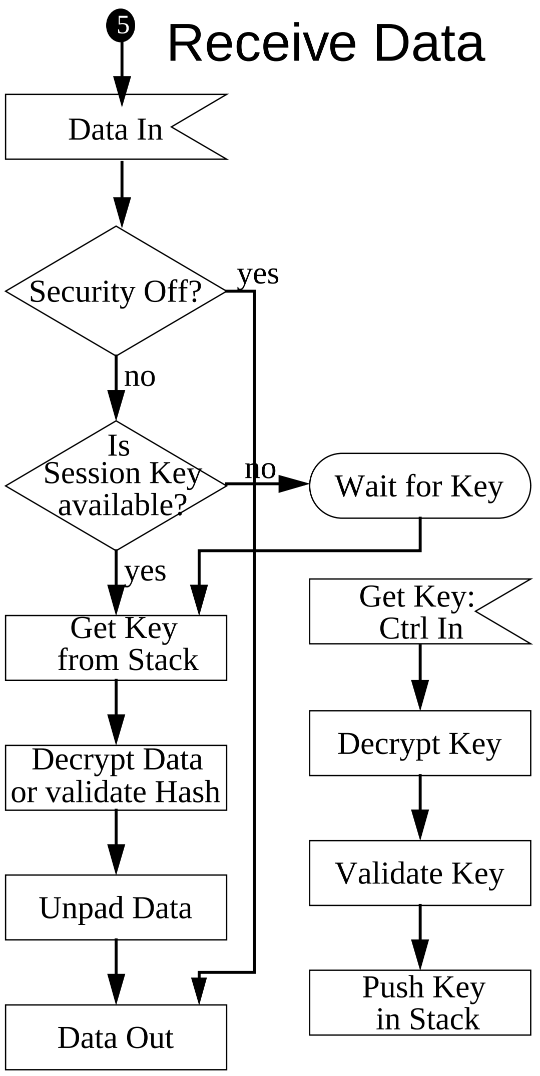


Figure 21. Receive Data

Normal processing simply decrypts the data, removes any padding, and forwards the data to the next module.

When key packets arrive on the ctrl-In interface, they are decrypted using the installed asymmetric keys, and the integrity and authenticity of the session key is check by validating the certificate appended to the session key on the sending side. If the key generation number or the transmitted initial key IDs do not match, or the validation fails, an error is returned.

7. Multicast Aspects of Da CaPo++

Multicasting within Da CaPo++ is supported by a multicast-capable Connection Manager as part of the Da CaPo++ core system and protocol support for multicasting connections as part of configurable modules to provide necessary protocol processing support. The detailed design document remains valid, even though the error-control layer has not been integrated by implementation into the Connection Manager. A stand alone implementation is available within MCF (Multicast Communication Framework) that proves the feasibility of the detailed design.

Summarizing, Da CaPo++ implements a basic multicast model where the creator of a multicast flow is the only sender. Participants are, therefore, always receivers. Multicast flows must be part of a session, as every Da CaPo++ flow. Multicast flows inside a session originate all from the same machine. Multicast flows must not be mixed together with unicast flows in the same session, since the connection manager expects homogeneous sessions.

Dynamic join and leave is allowed, *e.g.,* participants can attach to a running multicast session. However, the session is controlled by the creator only. The multicast dynamics prevent a global reconfiguration. This means that the protocol configuration is based on the creator's application requirements only.

8. Application Framework of Da CaPo++

A huge number and variety of traditional and modern applications offer a broad range of user-oriented services. Therefore, a hierarchical structure of control and, subsequently, graphical user interfaces of these applications has been identified to structure relevant elements. They include, *e.g.*, audio and video transmissions, picture phones, video conferencing, tele-banking, tele-seminar, tele-shopping, tele-teaching. Due to a set of well-defined differences between these applications, this spectrum of applications looks quite unstructured. For instance, a tele-seminar includes features and functionality of a video conferencing; a picture phone includes inevitably the transmission and presentation of audio and video data. Additionally, the type of control applied and used within these applications is different. As data transfer requires a simple interface only, a picture phone has to offer a separate graphical user interface for sufficiently controlling the handling and manipulation of audio and video data. Finally, a tele-seminar involves meta-control for integrating floor-control issues, managing and synchronizing video, audio, and data flows, or joining new participants.

The basics for defining the application framework for the KWF–Da CaPo++–Project comprise a layered hierarchy. Especially a defined three-layer hierarchy allows for a very flexible and modular design and implementation of a variety of application scenarios. The lowest layer comprise application components that are placed directly via a specified application programming interface on top of the Da CaPo++ core system. In the middle layer, applications are constructed out of application components in addition to special application functionality and a separately usable graphical user interface. In the upper layer application scenarios are used to consolidate multiple applications. They provide extensive functionality and features for complex user requirements, including a specifically designed graphical user interface for control and meta-control purposes. All these elements (application components, applications, and application scenarios) are placed in one of the layers based on their specific objectives and features.

The application component – just component in short – forms the basic building block for the application framework. It defines in the lowest level of the hierarchy differentiated and separately usable parts of traditional applications. They provide a separated functionality only, a set of tightly bound features including an application programming interface, but no graphical user interface. Examples include but are not limited to, audio/video presentation, messaging service, or application sharing. Traditional applications, such as picture (video) or standard (voice) phone or video conferencing, have been placed in the middle of the hierarchy. However, within the framework they are functionally structured out of single or multiple application components. Additionally, application provide a separate graphical user interface for controlling exactly this one only. Specific user control features to run this application sufficiently is provided. Nevertheless, an application in this sense is able to run stand alone. Finally, a huge variety of applications may be combined for designing complex application scenarios – scenario in short – that provide functionality, graphical user interfaces, and meta control interfaces to fulfill emerging user requirements in tele-operating environments. In the defined terminology, modern applications such as tele-seminar or tele-teaching belong to the layer of application scenarios.

Compared to an object-oriented design or reusable code and elements respectively, within the application framework the layered structure of elements describes a novel approach. Application building blocks allow for the flexible construction of applications, their control parts, and their graphical user interfaces.

Within the following the extended WWW server and Browser are discussed. In addition, the video viewer, an audio player, and the shared video viewer including the video daemon are presented. Finally, the file client and server, and the multimedia server and sender are shown.

8.1 Introduction into the Extended WWW Server and Browser

A traditional World Wide Web (WWW) application offers access to numerous information stored on servers in the Internet. These servers are called WWW Servers. The information is presented using so-called WWW pages that are written in a hypertext markup language like HTML. Clicking on a hyperlink, the requested data is transmitted via an HTTP (Hypertext Transport Protocol) link to the user site. in general all data is transmitted first before it is processed resp. presented on the user’s machine. This may result in time gaps. Also the HTTP protocol does not allow for bandwidth reservation or for the specification of QoS requirements for the transmission. This is very inconvenient, especially if continuous media is to be transmitted. To overcome these restrictions it was decided to facilitate the establishment of a link following a communication protocol allowing for the guarantee of QoS requirements and the continuous transmission and processing of data via an HTML hyperlink on a usual WWW page. The requested multimedia data will then be transmitted via the newly established communication link. Da CaPo++ was chosen as protocol for the data transmission.

This document describes the designs as well as the implementation of the WWW application scenario in the Da CaPo++ project. It also describes the used A-Modules (SunVideoFile and AudioFile) as well as the relevant out-of-band communication in these modules. As the Da CaPo++ project included a *Shared WWW Browser* as well, this document includes the description of the implementation of the *dynamic multicast* in case of video data, as this allows for the realization of the Shared WWW Browser. The Shared Video Viewer scenario is described in detail in this document. Also synchronization issues are mentioned, as they may be relevant in future File Client applications.

8.2 Extended WWW Browser and Server

The Extended WWW Browser and Server scenario[1] enables the transmission of multimedia data over connections established with the Da CaPo++ protocol. All in this scenario relevant connections as well as the included servers and protocols are shown in Figure 22. for multimedia data, that is transmitted in one session. This figure serves as reference for all future discussions in this document. The whole scenario will be introduced first before details will be explained.

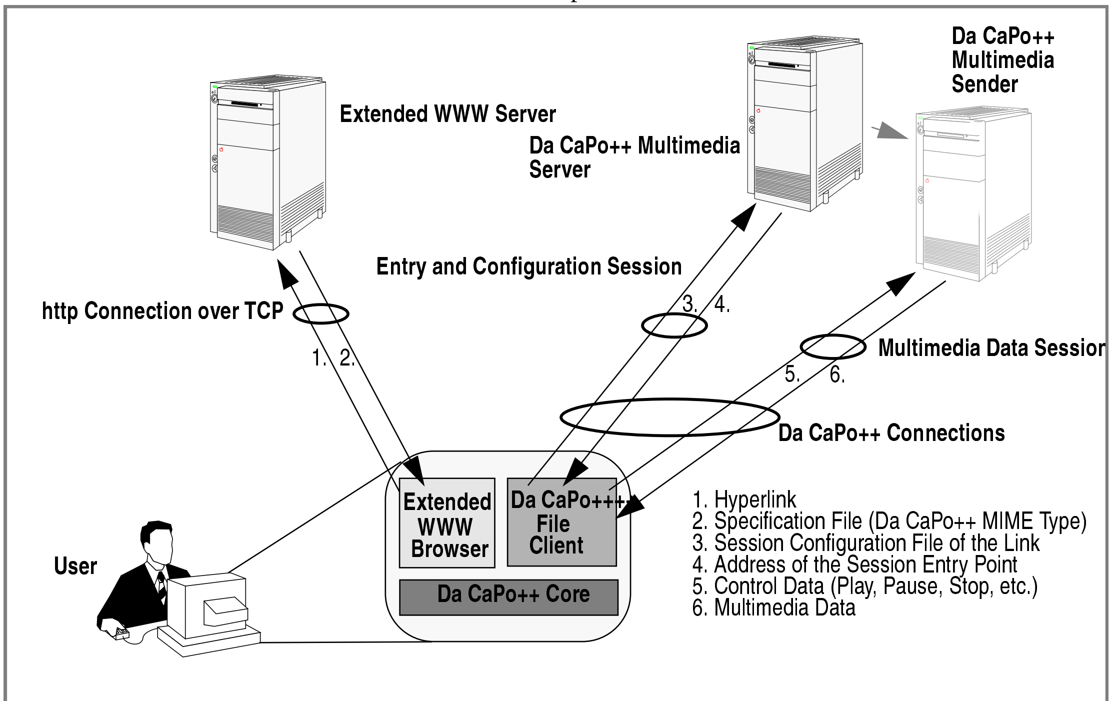


Figure 22. Extended WWW Browser in Case of Multimedia Data Transmission

Comparing this document with former design documents, you will notice that the *Da CaPo++ File Server* has been split up into the *Da CaPo++ Multimedia Server* and the *Da CaPo++ Multimedia Sender.* This was done during the implementation. The design of the former Da CaPo++ File Server and its port administration is described in Paragraph 8.15.2, whereas the implementation of the Da CaPo++ Multimedia Server and Multimedia Sender is described in Subsection 8.16.

A user clicks on a hyperlink demanding data from an Extended WWW Server. The Extended WWW Server recognizes a specific MIME (Multipurpose Internet Mail Extension) type for the demanded data,

1. in this document it if referred to as an application scenario. It also is an application that may be included into other application scenarios.

which in fact is a specification file[1], and transmits it to the client's machine. The Extended WWW Browser on the client's machine will recognize the specific MIME type and starts a script itself starting the application that establishes a Da CaPo++ connection to the multimedia server and demands the requested data. This establishment is based on information in the specification file.

The whole scenario follows the concept of *External Viewers.* This concept is based on the recognition of MIME types. Every data referenced by an HTML hyperlink has such a MIME type assigned to. A WWW Browser recognizes the MIME type and performs an action for these data. These actions are specified in the file*.mailcap*. As part of such an action, a program can be started, referred to as *External Viewer.*

To enable this the following changes have to be made.

8.2.1 Extended WWW Server

The Extended WWW Server has to recognize the new MIME type *application/x-dacapo.* In our implementation we used the httpd Server from CERN. In its configuration file the following line is to be added:

```
AddType .dacapo application/x-dacapo   7bit    1.0
```

This line assigns to each file with the extension *.dacapo* the MIME type *application/x-dacapo*. This is all to be done to extend the WWW Server.

8.2.2 Extended WWW Browser

To extend a Unix WWW Browser for our purpose, two files have to be changed. In the file *.mime.types* the following line is to be added, so that the browser may recognize the MIME type sent by the server:

```
application/x-dacapo           dacapo
```

In the file *.mailcap* the following line was added:

```
application/x-dacapo;        xterm -T "Da CaPo++ File Client" -n "File
Client" -e /proj/dacapo/class/src/appl/www/video_client/video_client %s
```

That is, whenever a file of the MIME type *application/x-dacapo* is read, an *xterm* is automatically started to start the *video_client* program on the client's site. The *xterm* is not necessary but useful to see the program's output like, e.g. warnings.

8.3 Multimedia File Client and Multimedia File Server

The program started by the Extended WWW Browser when receiving data with the MIME type *application/x-dacapo* may of course also be started stand-alone under the premise that a specification file is passed to the application. Therefore the rest of the document examines the design of a stand-alone Multimedia File Server and Multimedia File Client. The later will then be started as *External Viewers* in the Extended WWW Server and WWW Browser scenario. The implementation of the Da CaPo++ File Client as well as of the Da CaPo++ Multimedia Server and the Da CaPo++ Multimedia Sender is described in this document, too.

8.4 Presenting Multimedia Data: General Concepts

The scenario of the Multimedia File Client and Multimedia File Server (in short: File Client and File Server) serves for the transmission of continuous multimedia data via the Da CaPo++ communication protocol. The data will be stored on a multimedia file server and be transmitted to the user's site. In the

1. The grammar of the language used to write the specification file is defined in the extra document "The Specification File". The meaning of this file is explained in the following sections of this document.

Da CaPo++ protocol stack, data is passed to/from the application/Upper API via A-Modules. In case of multimedia data these A-Modules also present and process data.

In case of the File Server and File Client the A-Modules:

- read the data from the file
- present the data (video, audio) on the output device
- provide functionality as play, pause, fast forward, etc.

The A-Module itself is embedded in the Da CaPo++ core. The A-Module is part of the protocol graph and "communicates" directly with the kernel, i.e. the functionality of the kernel is directly used. Therefore the A-Modules are written in C. In this chapter, presenting the general functionality and the design of the involved A-Modules, some technical details can not be avoided.

8.4.1 Communication in A-Modules

The A-Modules for File Server and File Client are designed for the following scenario:

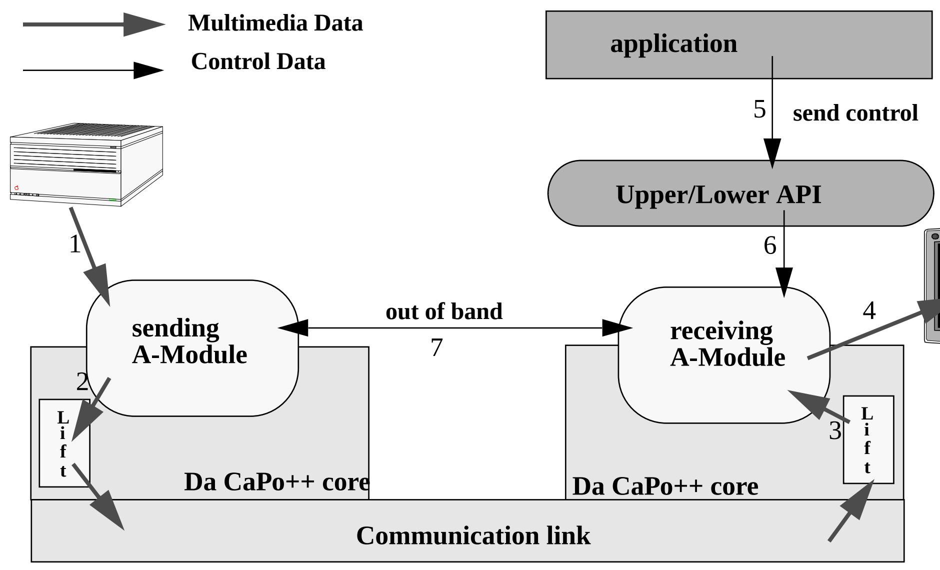


Figure 23. Scheme of the Communication Paths Between A-Modules

- **Sender:** The sending A-Module reads the data from a file and transmits it. Furthermore, all VCR[1] control the user initiates, is at last performed here. To do so, the A-Module receives the control commands from the receiving A-Module. Therefore, a communication between the A-Modules on both sides is necessary. This kind of communication is called *out-of-band communication* in the Da CaPo++ context. The data packets for the out-of-band communication are transmitted by the Connection Manager.
- **Receiver:** The receiving A-Module receives the data and presents it to the device. Furthermore it receives the control information via the Lower API and sends it to the peer's A-Module to be processed. During the initialization process it may also receive out-of-band communication from the peer. This occurs in the SunVideoFile A-Module, when the sender sends the required size of the X-Window displaying the video data to the receiver.

Figure 23 on page 47 gives an overview of these different forms of communication involved.

The sending A-Module reads the data from the file (1), generates a data packet and hands the packet to the Lift (2), which in its turn sends the packets to the peer's site. In general, this is provided within the

1. VCR control: VCR is the video record player. Play, pause, stop, fast forward, etc. are understood as VCR control functions in this document.

function `asIndiA-Module-Name().` The return value for the Lift will be `sbDataOk`. For the continuous media audio and video the transmission of one data packet is triggered by a timer event.

The Lift on the receiving site, after receiving the data, will call the function (3) `arIndiA-Module-Name` or `arRequA-Module-Name` in the receiving A-Module. These functions process the received data and present it on the output device (4).

This functionality is sufficient to continuously transmit data from a multimedia file as long as there is still data left and to display it on the user's site. In case of providing additional control to the user as are the VCR functions, the *out-of-band communication* provided by Da CaPo++ (5 to 7) is required.

8.4.2 Concept of the Out-of-band Communication in Da CaPo++

Transmitting control data from the user to the A-Module on the File Server side involves the application, the Upper and the Lower API, both A-Modules and the Connection Manager on both sides. The Connection Manager is responsible to correctly transmit the out-of-band data to the other site. This is explained in the Da CaPo++ core documentation and is not scope of this document.

Figure 24 on page 49 illustrates all functions involved in the out-of-band communication as implemented within the File Server A-Modules. These functions are further explained in the following paragraphs.

8.4.2.1 Registration of Call-back Functions

Several call-back functions are involved in the whole out-of-band communication process. These functions have to be registered in the entity of the Da CaPo++ core that calls these functions during the communication process.

An application has the possibility to send control packets to each flow. As explained in the document on the design and implementation of the API, the Upper API communicates control packets received by the application to the Lower API via an IPC mechanism. These control packets are neither interpreted nor are the corresponding functions provided by the Lower API. The Lower API communicates the control packet to the A-Module belonging to the flow's protocol graph instead. To do so, the Lower API calls a function in the A-Module, a so-called call-back function. The A-Module has to register this function at the Lower API. This registration is performed by calling the function `lapi_RegisterAModule`. This function gets two function pointers as parameters, one pointer to the call-back function processing data, that is transmitted from the application via the IPC mechanism, the other to the function processing the control data.

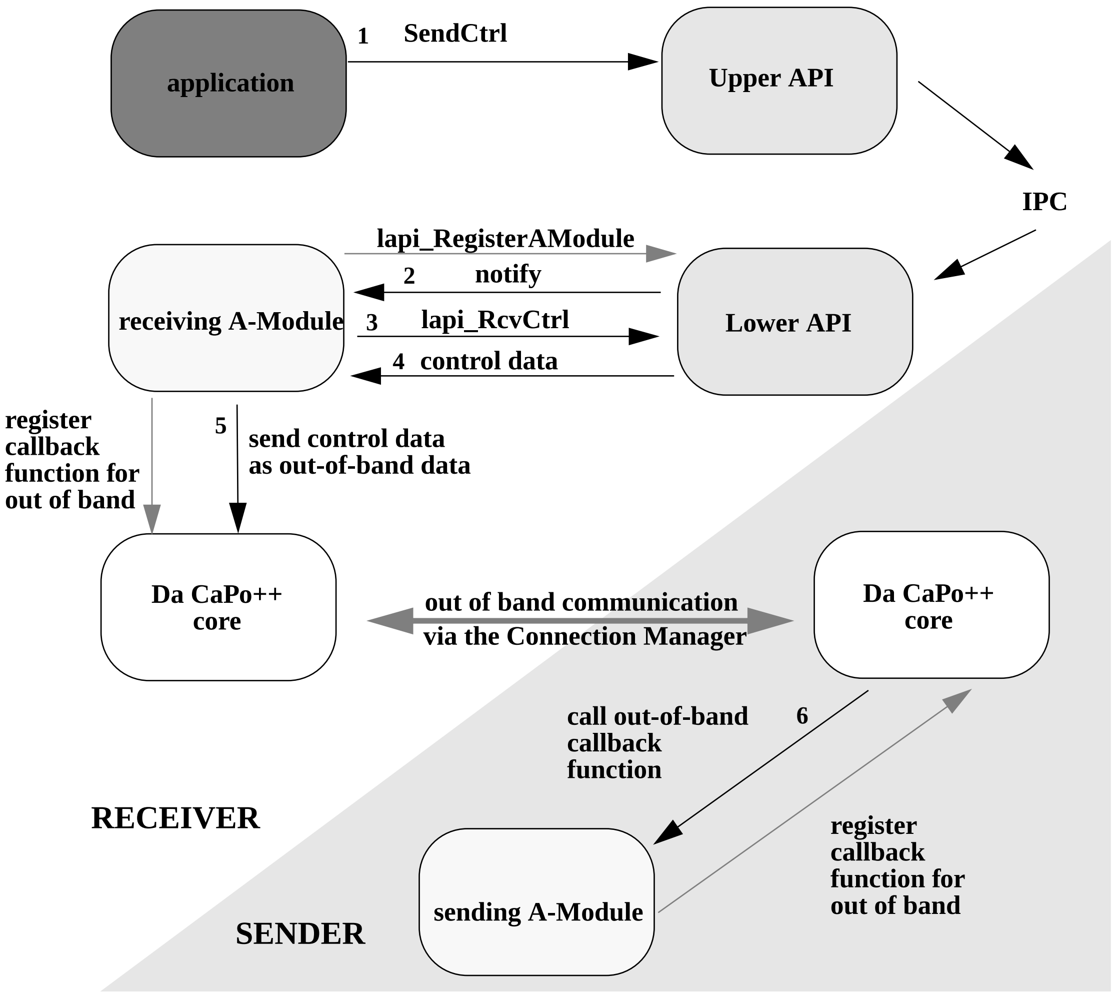
The A-Module also has to register a function allowing the out-of-band communication to its peer's A-Module to the Da CaPo++ core. This is done by defining the values for the `DC_Module` structure. This structure contains one field to specify the function for sending out-of-band data, as well as a field for the function to receive out-of-band data.

These two function are called by the Lift algorithm. The receiving function is invoked whenever the Connection Manager receives out-of-band data from the peer's Connection Manager. The sending function for out-of-band data is invoked after the function `LiftFullStatusTreatement` was called within the A-Module. Giving `sbControl` as parameter to this function, the out-of-band sending function will be invoked by the Connection Manager.

8.4.2.2 Transmitting Control Data Via the Out-of-band Communication

Figure 24 on page 49 outlines how control data is transmitted from the application to the sending A-Module. All VCR control is transmitted in this way, as these functions have a direct impact on the reading of the data from the file, i.e. the processing of the A-Module in the File Server.

The application can send *control packets* to each flow. A control packet contains a string and is send to the Upper API indicating the identification number of the flow the packet is designated for (1). The content of the control packet is written to a shared memory and the Lower API is informed that there is con-



application
1 SendCtrl
Upper API
IPC
lapi_RegisterAModule
2 notify
receiving A-Module
3 lapi_RcvCtrl
Lower API
4 control data
register callback function for out of band
5
send control data as out-of-band data
Da CaPo++ core
out of band communication via the Connection Manager
Da CaPo++ core
call out-of-band callback function
6
RECEIVER
register callback function for out of band
sending A-Module
SENDER

Figure 24. Sending Control Information Via the Out-of-band Communication

trol data for the specific flow/A-Module[1]. The Lower API then invokes the call-back function of the A-Module, this corresponds to a notification (2). The A-Module can get the control packet, i.e. the content of the shared memory, by calling the function `lapi_RcvCtrl` (3+4). The receiving A-Module will interpret the control packet's content and generate control information for the A-Module on the peer's side. After generating this information[2], the function `LiftFullStatusTreatement` is called to invoke the out-of-band communication.(5) On the sender's site, the A-Module receives the out-of band information (6) and performs the requested functionality. Depending of this functionality, the sending of data is resumed or stopped, or modified (e.g., fast forward instead of normal speed playback).

8.4.3 Timer Concept

In the Da CaPo++ core there exist different kinds of timers that can be used. Timers can be used to trigger alarms regularly. They can be initialized with their period. These timers are used on the sending side to send the continuous media with a specific rate in time in the SunVideoFile and the AudioFile A-Module. The timer is created with the command `rsrc_TimerCreate (&timer, callback, lift_pointer)`. The Da CaPo++ core create a periodic timer and each time, the timer triggers an alarm, the *callback* function is called by the Lift. This function, then, calls the function `lift_DataAvailable` which in turn calls the indication function of the A-Module, so that data is

1. The flow on application and Upper API level corresponds to an A-Module on the Lower API level.
2. The control information is intermediatly stored in the module instance pointer `ndInstancePtr` of the `DCnode` data structure.

sent. Thus, data packets are sent periodically which leads in the local environment, to an acceptable video and audio quality. How synchronization may be performed and why it was not integrated in the Da CaPo++ system is described in Section 9 on page 95.

8.5 Presenting Multimedia Data: The SunVideoFile A-Module

This paragraph gives an overview on the current implementation of the SunVideoFile A-Module. The SunVideoFile A-Module is designed following the above mentioned principles for File Server and File Client respectively A-Modules. The code for the A-Module is in the file `src/amod/sunvideo-file/a_sunvideofile.c`.

The movie data for the SunVideoFile was encoded in JPEG with the Sun video card. The X-Imaging Library Xil from Sun which was delivered with Solaris 2.5 was used to process the compressed image sequence (cis).

8.5.1 Restrictions from the Xil Library

The use of the Xil library in the SunVideoFile A-Module led to some un-elegant implementations to overcome the restrictions of this library:

- To obtain one single video frame and get its length we could not use one single Xil function. Therefore, we used a function returning a pointer to the video data from the current image up to the end of the cis. This functions also returns the number of Bytes in the image sequence as well as the number of frames. We called this function for the current cis as well as for the cis when regarding the next image of the movie. Taking the difference of the length information for both cis and shrinking the memory space pointed to by the first pointer to exactly this length, we obtained the image data for one video frame. This data than can be send as a Da CaPo++ packet to the peer's site.
- At the beginning of the cis is some relevant information for the receiver to initialize its own cis. As our performance measurements showed, the sender may need up to 3 seconds (!) to read this information. This leads to a high initial delay. In case of *dynamic multicast*, the information of the first frame has to be transmitted to the new receiving A-Module before it can display any data, as the information in the first frame is relevant to correctly initialize the cis. For this reason the first frame has to be retransmitted to all new-coming receivers, which may block the sender for the time to read the frame and leads to a presentation pause, gap or whatever in the other receivers.[1]

8.5.2 Video File Instance Pointer

Each A-Module has one instance pointer pointing to all data relevant for this instance of the A-Module. The instance pointer already existing in the A-Module for the Picture Phone was not sufficient for the SunVideoFile A-Module and, thus, has been enlarged. The new instance pointer consists of the elements as shown in Table 12.

Table 12. The Video File Instance Data Structure

name	**type**	**structure**	**meaning**	**valid context**
viInstPtr	VideoInstancePtr		the video instance pointer as used in the SunVideo A-Module	in both A-Modules
videoFileName	char *		the name of the video file read by the A-Module	in the sending A-Module

1. For this reason, the Xil is not suited to transmit video data that require a specified quality.

Table 12. The Video File Instance Data Structure

name	type	structure	meaning	valid context
playing	int		1 - perform playing 0 - do not play this parameter serves for the implementation of play, pause and stop	in the sending A-Module[a]
fast_play	int		1 - fast forward 2 - fast rewind 0 - normal playback speed this parameter controls the playback speed[b]	in the sending A-Module[c]
second_client[d]	1		sender: 1 - if set, the next frame has to be sent for a newly joined receiver receiver: 1 - if set, the A-Module belongs to a “not yet totally initialized” late receiver in dynamic multicast	in both A-Modules
compr	int		this parameter indicates the compression type of the film	in both A-Modules
outofband	CtrlPacket	char data[] int size	this parameter contains the out of band data received by the A-Module of the peer’s site or the data that shall be sent to the peer	in both A-Modules
max_frames	int		this parameter indicates the maximum number of frames in the `cis`	in the sending A-Module
current_frame	int		this parameter indicates the current position in the `cis`	in the sending A-Module
frames_per_second	long		this parameter indicates the number of frames that have to be sent in one second	in the sending A-Module
window_sizes	VideoWindowInfoPtr	unsigned int xsize unsigned int ysize unsigned int nbands XilDataType datatype	this parameter contains the video frame information necessary to create the X Window to display the video data	in both A-Modules
timer	Rsrc_Timer-Ptr		timer for the sending of the video data with a specified number of frames per second	in the sending A-Module

a. It would be more reasonable to use this value in the receiving A-Module, too, to avoid sending of un-necessary control packets (like “play” during the already performed playback of the movie).

b. It may also be used to control the playback direction. E.g. the value 3 could indicate to reverse the film data in normal playback speed.

c. see a

d. The name *second_client* comes from the shared WWW application, when the dynamic multicast of video data sent from a file was first used.

8.5.3 VCR Control Commands

For JPEG video files the following VCR commands have been realized up to today in the SunVideoFile A-Module. Two fields in the instance pointer structure have been defined to control the video playback:

- *playing*: if this variable has the value 1, playing is performed. If the value is 0 the retrieval of the video data is stopped.
- *fast_play*: in case of value 0 the playback is performed with the normal speed. In case of value 1 fast forward is performed whereas fast rewind is performed when the value is set to 2.

The control commands are performed as follows.

Play. The *play* command invokes the sending A-Module to generate video frames and send them to the peer's side. After a *stop* or *pause* command the sending of video frames is resumed, when this command is called. The frames are generated depending on the value of the *fast_play* variable. If this variable indicates *fast forward*, *play* provides sending of data in the *fast forward* mode. If the last frame is sent, the reading and transmission of video data is stopped.

Stop. The *stop* command stops the sending of video frames in the sending A-Module. The Lift, however, is not stopped in the current implementation. The transmission of the data is resumed by invoking the *play* command.

Pause. The *pause* command in the current version is performed in the same way as the *stop* command. As *pause* in general is performed only with the purpose to stop the data transmission for a short time only, it is not planned to stop the Lift in the future while performing the *pause* command.

Fast Forward. The *fast forward* command provides a fast play of the video data. This is realized by skipping 9 frames during the playback, i.e. only every 10th frame is sent to the peer's A-Module. The variable *fast_play* is set to 1. Calling of the *fast forward* command for a second time resumes the normal play back speed.

Fast Rewind. The command *fast rewind* is performed similarly to the command *fast forward*. The variable *fast_play* is set to 2 and in the rewind mode every 11th frame is played[1]. If the command *fast rewind* is called for a second time, the playback speed is reset to normal.

Forward. The command *forward* sets the pointer of the cis to the last frame, i.e. the video data can only be displayed in the *fast rewind* mode,

Rewind. The command *rewind* sets the pointer of the cis to the first frame, i.e. the video data can be displayed from the beginning.

Commands that may be provided in the future. A *reverse* command could be provided in the future. This could be easily realized by playing back every 2nd frame in the reverse mode.

The following table presents the control commands that may be sent to the SunVideoFile A-Module.

Table 13. The Control Commands for SunVideoFile

command	control packet structure	meaning	status
play	PLAY	the playing of video data is started or reset	implemented
pause	PAUSE	the sending of video data is paused	implemented
stop	STOP	the sending of video data is stopped	implemented but Lift does not stop

1. This results in the same as skipping 9 frames in the normal reverse mode.

Table 13. The Control Commands for SunVideoFile

command	control packet structure	meaning	status
forward	FWD	skip to last video frame	implemented
fast forward	FF	display each 10th frame while displaying	implemented
rewind	RWD	go back to first video frame	implemented
fast rewind	FR	display each -11th frame	implemented

8.5.4 Initialization of A-Modules

Before displaying video data, the corresponding `cis` has to be created. This `cis` contains information needed to create a corresponding X-Window. This information cannot be obtained before reading the `cis`. The sending A-Module may create a X-Window to display the data as soon as it created the `cis`. The receiving A-Module may not create the X-Window before receiving the first frame. As the creation of the X-Window takes a certain time in which several video images may be sent, it was decided to create the display window on the receiving site before transmitting the video data. To do so, the sending A-Module sends the necessary information as out-of-band data to the receiving A-Module. As soon as the receiving A-Module has created the X-Window, the transmission of the video data is initiated.

For performance reasons on the sender side, in case of the SunVideoFile A-Module, the sending A-Module does not display the video data.

In which concerns the SunVideoFile A-Module we have to distinguish two different cases for the initialization phase of both A-Modules:

1. the *stand-alone A-Module application* for testing purpose: this application is useful as long as there are tests running testing the data transmission as well as the data presentation. This A-Module can be tested by transmitting data written to a file with a fixed, dedicated name. In this case, the name of the file to read needs not to be transmitted from the receiving to the sending A-Module.
2. the *File Server application using the A-Module*: in this application the name of the file to read can be chosen by the client and must be transmitted via the out-of-band communication from the client's site (= receiving A-Module) to the server's site (= sending A-Module) after the A-Module received the file name as control data from the lower API.

In all cases the sending A-Module has to open the file to be read, create the corresponding `cis` and send the information needed to create a corresponding X-Window and the `cis` to the receiving A-Module. There is only one A-Module used in case of the two application contexts. To distinguish these two cases, one global variable was introduced: `a_VideoFileName` of the type `char *`. This variable contains the (fixed) name of the read file defined by the stand-alone A-Module application and is a `NULL` pointer in case of the File Client application. The initialization process in both cases is very similar. For this reason Figure 25 on page 55 illustrates the process for both cases. The involved control packets and functions are explained in the following paragraphs.

Table 14 on page 54 gives an overview on the control packets involved in the initialization process of the A-Modules.

8.5.4.1 Initialization of a New Receiver in Case of Dynamic Multicast

As already stated in Section 8.5.1, in case of dynamic multicast the sender must transmit the first frame of the `cis` to the new receiver for him to correctly initialize its `cis`.[1] This is controlled and performed

1. If this is not done, the Xil simply crashes.

using several out-of-band control packets, one header Byte in the data packets and the *second_client* parameter in the Video File Instance Pointer.

Table 14. The Control Packets for the Initialization Process in A-Modules

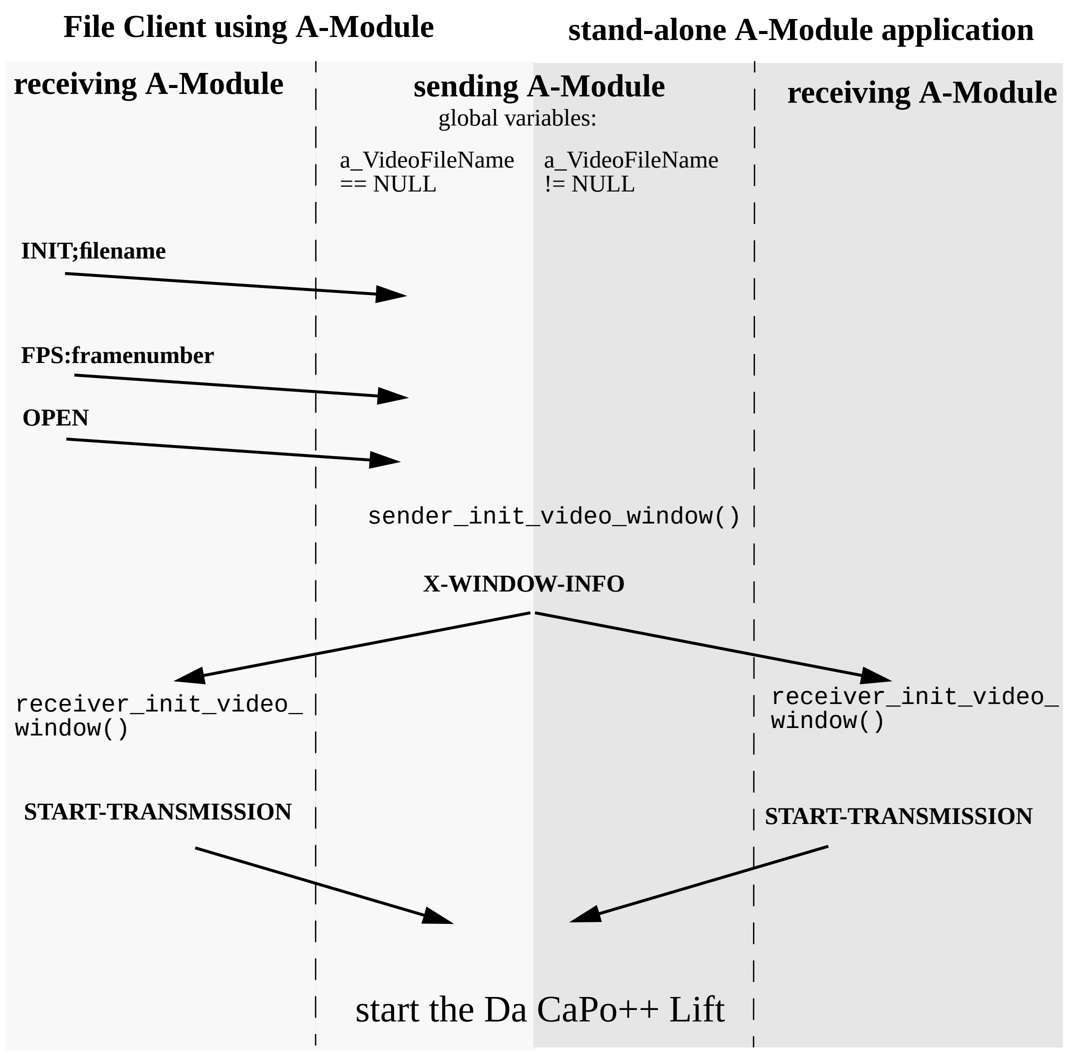
control packet	parameters	meaning
INIT	filename	the file name of the file to read is sent from the receiving to the sending A-Module
FPS	number of video frames per second	the number of video frames that has to be sent per second is specified by this value
OPEN		the receiving A-Module demands the sending A-Module to open the file
X-WINDOW-INFO	VideoWindowInfo structure	the necessary data to create the X-Window is sent from the sending to the receiving A-Module
START-TRANSMISSION		the receiving A-Module indicates that it created its display window and that the video data transmission may start

A new receiving A-Module in a multicast group asks the sender for the retransmission of the X-Window information as well as of the first frame of the `cis`. To do so, it sends an out-of-band packet *GET-WINDOW-INFO* to the sender. The receiver has realized to be a new receiver, by receiving data packets while its own `cis` is not yet created. When sending its *GET-WINDOW-INFO* packet to the sender, the receiver sets the parameter *second_client* in the Video File Instance Structure to 1 so that the sender does not receive one *GET-WINDOW-INFO* for every normal data packet the client receives. The sending A-Module sends a packet *X-WINDOW-INFO* so that the receiver can initialize and create its X-Window. When the receiver has initialized its X Window, it sends the control packet *SECOND-CLIENT-READY* to the sender. The sender now sets its *second_client* parameter to 1 and retransmitts the first frame in its `cis` to the multicast group when the timer triggers its periodic alarm.[1] To prevent the other receivers from displaying this frame that is not in its normal place within the image sequence, the sender sets the data header to 1. A normal data packet has a header of 0 and every receiver that is not in the process of initializing as new receiver in case of dynamic multicast ignores the packet with a header of 1. After sending the first frame with the header 1, the sender resets the *second_client* to 0. This is also done by the receiver after it has received the first frame in a data packet with header 1.

Table 15. The Control Packets for the Initialization Process of New Receivers in Case of Dynamic Multicast

control packet	parameters	meaning
GET-WINDOW-INFO		a new receiver asks for the X-WINDOW-INFO control packet from the sender
X-WINDOW-INFO	VideoWindow-Info structure	the necessary data to create the X-Window is sent from the sending to the receiving A-Module
SECOND-CLIENT-READY		the new receiver indicates that it has initialized its X-Window and asks for the (re-)transmission of the first frame

1. To avoid problems if this packet gets lost, the new receiver should start a timer and retransmit its SECOND-CLIENT-READY packet in case of not having received a packet with the header 1 in the meantime. This is not implemented.



control packets:

INIT;filename - FPS:framenumber - OPEN - X-WINDOW-INFO - START-TRANSMISSION

functions:
`sender_init_video_window() - receiver_init_video_window()`

Figure 25. The Initialization Process in A-Modules

8.5.5 Implementation of A-Modules

8.5.5.1 Hard coded elements

The current implementation of the SunVideoFile A-Module has the following hard-coding:

- The compression standard is hard coded to be JPEG.

8.5.5.2 File a_sunvideofile.c

The following functions are defined in the file `a_sunvideofile.c`:

- void data_ctrl_back

- This function is the call-back function for the sending of data from the application to the A-Module (flow). As this is not provided in the SunVideoFile A-Module, an error message is generated.
- `void sender_ctrl_back`

  This function is the call-back function for the sending of control data in the sending A-Module. As this is not provided by the SunVideoFile A-Module, an error message is generated.
- `void receiver_ctrl_back`

  This function is the call-back function for the control data in the receiving A-Module. The content of the control packet is copied to the *outofband* field in the instance pointer and the out-of-band communication is invoked calling the function `liftFullStatusTreatement`.
- `void sender_OoBCommand`

  This function is called in the sending A-Module. The content of the *outofband* field in the instance pointer is interpreted and the corresponding actions are performed. These actions consist mostly in setting the appropriate values for the variables *playing* and *fast_play*. The initialization of the sending A-Module is performed within this function in case of the File Client Context. When the sending A-Module receives the *START-TRANSMISSION* control command, the Lift is started. In case of dynamic multicasting the control packets *GET-WINDOW-INFO* and *SECOND-CLIENT-READY* are recognized and the appropriate action is performed.
- `void receiver_OoBCommand`

  This function is called in the receiving A-Module and performs the out-of-band commands in the receiving A-Module (especially during the initialization process of the A-Modules). The implementation is similar to the function `sender_OoBCommand`. The only valid control packet in the receiver is the *X-WINDOW-INFO* packet.
- `Lift_Status videoFile_rcvOutIn`

  This function is called by the Da CaPo++ core in case the Connection Manager receives out-of-band data for the receiving A-Module. The data is copied into the *outofband* field of the video file instance data structure and the function `receiver_OoBCommand` is called.
- `Lift_Status videoFile_rcvOutOut`

  This function is called in the receiving A-Module and provides the sending of the out-of-band data. The content of the *outofband* field of the instance pointer is written into a Da CaPo++ packet. The return value `sbDataOk` informs the Lift that there is a packet to send.
- `Lift_Status videoFile_sndOutIn`

  This function is called in the sending A-Module and receives the out-of-band data. This data simply is copied to the *outofband* field in the instance pointer structure and the above mentioned function `sender_OoBCommand` is called.
- `Lift_Status videoFile_sndOutOut`

  This function is called in the sending A-Module and provides the sending of the out-of-band data. The content of the *outofband* field of the instance pointer is written into a Da CaPo++ packet. The return value `sbDataOk` informs the Lift that there is a packet to send.
- `void receiver_init_video_window`

  This function initializes the `cis` and the X-Window in the receiving A-Module. It is called after the control packet *X-WINDOW-INFO* was received. When the X-Window is created, the control packet *START-TRANSMISSION* is sent to the sending A-Module using the out-of-band communication channel. The compression type is hard-coded to JPEG. The sending of *START-TRANSMISSION* after initializing the receiver’s window in case of dynamic multicasting makes that the sender sends one frame out of synchronization with the time, and thus leads to making up the 1 frame delay in the sending of the video sequence after having repeated the first frame.

- `void sender_init_video_window`[1]

  This function initializes the `cis` in the sending A-Module. After initializing the `cis` the video format information is sent to the receiving A-Module. The compression type is hard-coded to JPEG. The periodic timer is created based on the value in *frames_per_second.*

- `Lift_Status asInitVideoFile`

  This function is the initialization function in the sending A-Module. The call-back functions are notified to the Lower API using the function `lapi_RegisterAModule`. In case of a stand-alone A-Module, the instance variables *frames_per_second* and *videoFileName* are initialized and the function `sender_init_video_window` is called.

- `Lift_Status asExitVideoFile`

  This function is called in the sending A-Module and destroys the `cis`.

- `Lift_Status asIndiVideoFile`

  This function provides the sending of the video data to the client's site and is part of the sending A-Module. The values of *playing* and of *fast_play* are taken into account within this function. As already stated, the Lift is not stopped for instance when *playing* is set to 0. Instead the function is returned at once with the value `sbNoData.`

  If *playing* is set to 1, the `cis` is set to the next frame depending in the value of *fast_play* (to the next frame in case of 0, to the 10th frame in case of 1, to the -11th frame in case of 2). Then the video frame to send is computed by using two pointers as stated in the beginning of Section 8.5.1.

  In case of *second_client* being set to 1, the sender sends the data of the first frame of the `cis` with a 1 in the data packet header, otherwise the header Byte is set to 0 and the next packet in the frame sequence (depending on the value of *fast_play)* is transmitted.

- `static void asTimerVideo`

  This function is the callback function for the periodic timer in the SunVideoFile A-Module.

- `Lift_Status asInfiVideoFileEmpty, asEmpty, asStartVideoFile` and `asStopVideoFile`

  These functions are called when the Lift is started and stopped respectively. In the `asStartVideoFile` the Lift is not yet started.

- `Lift_Status arInitVideoFile`

  This function is called for the sending A-Module. The call-back functions are notified to the Lower API and the `cis` to take the received data as well as the X window to display the data are initialized.

  The X window size as well as the compression type are hard coded for the moment.

- `Lift_Status arExitVideoFile`

  This function is called for the receiving A-Module and destroys the `cis` as well as the X window.

- `Lift_Status arStopVideoFile`

  This function is called by the Lift in case the protocol is stopped.

- `Lift_Status arRequVideoFile`

  This function is called by the Lift when data is received. The received video frame is written to the `cis` and displayed within the X window. If this function is called and the `cis` is not yet initialized, the receiving A-Module belongs to a dynamically joined receiver in a multicast group. The *second_client* parameter is set to 1 and a *GET-WINDOW-INFO* control packet is sent to the sending A-Module.

- The rest of the file `a_sunvideofile.c` provides some functions for CoRA in case it will be integrated as well as the definitions of the A-Module data structures for the database.

1. The function name is quite misleading. In the first versions of the A-Module, the sender also displayed the video window. When this was stopped, the name was kept nevertheless.

8.5.5.3 File xilcis_color.h

The file `xilcis_color.h` contains among other definitions the definition of the `VideoFileInstance` data structure, which is the relevant instance pointer for the SunVideoFile A-Module.

8.5.5.4 File xilcis_color.c

The file `xilcis_color.c` is located in the `src/amod/library` directory as it contains some function used by the SunVideo as well as by the SunVideoFile A-Module. Its functions are the following:

- `XilLookup suvCreateCmap`

  This function creates a lookup colormap for the Xil Library. This colormap is used in case the system does not provide *truecolor*.
- `Xil_boolean suvErrorRHandler` and `suvErrorSHandler`

  These two functions are the error handlers needed for Xil for the receiving and the sending A-Module. They do abort the process by generating an appropriate error message.
- `void suvStartXil`

  This function starts and initializes the Xil.
- `void suvMakeCis`

  The cis is created within this function.
- `void suvMakeWindow`

  The X window is created in this procedure.
- `void suvPrepareDecompressedOutput`

  The decompressed output is prepared (creation of a Xil image, installation of the colormap if necessary).
- `void suvCellInstallCmap`

  This function installs the colormap for Cell and CellB compression/decompression.

8.5.5.5 The File memmap.c

The file `memmap.c` which is also suited in the directory `src/amod/library` provides four functions to map one input file in a `MFile` data structure. These functions are:

- `void openfile`

  This functions opens a `MFILE` structure for reading.
- `void init_memfile`

  The `MFILE` is initialized.
- `void detach_file`

  The `MFILE` structure is empty and the storage freed after this function was called.
- `void attach_file`

  The `MFILE` structure points to the file to read.

8.6 Presenting Multimedia Data: The AudioFile A-Module

This paragraph gives an overview on the current implementation of the AudioFile A-Module. The AudioFile A-Module is designed following the in Section 8.4 described principles for File Server and File Client respectively A-Modules. The code for the A-Module is in the file `src/amod/aufiofile/a_audiofile.c.`

The data in the audio file was captured from a music CD and is sent and played in CD quality. The sending of the data is controlled by a timer.

8.6.1 Audio Instance Pointer

Each A-Module has one instance pointer pointing to all data relevant for this instance of the A-Module. The AudioFile instance pointer has the name *AudioInstance* and the following structure:

Table 16. The Audio Instance Pointer

name	type	structure	meaning	valid context
audioFd	int		file descriptor for the audio device	both A-Modules
aufd	int		file descriptor for the sample file	in the sending A-Module
offset	int		offset into sample data, e.g. the header	in the sending A-Module
period	long long		period of samples in microseconds	in both A-Modules
lastsent	long long		last time a frame was sent (to control if a frame was lost)	in the receiving A-Module
count	int		counter for the received frames (to compare with the frame number in the header and control if a frame was lost)	in the receiving A-Module
timer	Rsrc_Timer Ptr		timer for the sending of the audio data	in the sending A-Module
audioFileName	char *		the name of the audio file to read	in the sending A-Module
playing	int		1 - perform playing 0 - do not play (= send) this parameter serves for the implementation of play, pause and stop	in the sending A-Module
fast_play	int		1 - fast forward 2 - fast rewind this parameter could be used to implement fast forward and fast rewind. This is not done in the current implementation of the AudioFile A-Module	in the sending A-Module
outofband	CtrlPacket	char data[] int size	this parameter contains the out of band data received by the A-Module of the peer's site of the data that shall be sent to the peer	in both A-Modules

8.6.2 VCR Control Commands

In the AudioFile A-Module three control commands are provided using the *playing* parameter in the AudioInstance Structure.

Play. The *play* command invokes the sending A-Module to read audio data from the file (triggered by its timer) and send it to the peer's site. This command starts the sending at the beginning and resumes sending after a *pause* or a *stop* command. With this command, the parameter *playing* is set to 1.

Stop. The *stop* command stops the sending of audio data in the sending A-Module. The lift, however, is not stopped in the current implementation. The transmission of the data just is stopped by setting *playing* to 0.

Pause. The *pause* command in the current version is performed in the same way as the stop command. *playing* is set to 0, whereas the timer is not stopped.

Commands that may be provided in the future. There are several commands that may be provided in the future, especially *forward, rewind, fast forward* and *fast rewind*. The implementation of these commands is prepared in the out-of-band functions of both A-Module but was not realized.

Table 17. The Control Commands for Audio File

command	control packet structure	meaning	status
play	PLAY	the playing of audio data is started or reset	implemented
pause	PAUSE	the sending of audio data is paused	implemented
stop	STOP	the sending of audio data is stopped	implemented, but Lift does not stop
forward	FWD	skip to end of audio file	prepared, not implemented
fast forward	FF		prepared, not implemented
rewind	RWD	go back to beginning of audio file	prepared, not implemented
fast rewind	FR		prepared, not implemented

8.6.3 Initialization of A-Modules

The initialization of the AudioFile A-Modules on both sides is not as complicated as process as of the both SunVideoFile A-Modules. The receiver just sends an *INIT* packet to the sender indicating the audio file name. The rest of the initialization process as already done automatically in the Da CaPo++ init functions of the both A-Modules. Before a *PLAY* command can be performed, the *OPEN* command has to be called.

Table 18. The Control Packets for the Initialization Process in A-Modules

control packet	parameters	meaning
INIT	file name	the file name of the file to read is sent from the receiving to the sending A-Module
OPEN		the receiving A-Module demands the sending A-Module to open the file

8.6.4 Implementation of A-Modules

8.6.4.1 File a_audiofile.c

The file first contains the necessary definitions of the audio configuration and sound structure as well as of the `AudioInstance` pointer.

The following functions are defined in the file `a_audiofile.c:`

- `void a_data_ctrl_back`

  This function is the call-back function for the sending of data from the application to the A-Module (flow). As this is not provided in the AudioFile A-Module, an error message is generated.

- `void a_sender_ctrl_back`

  This function is the call-back function for the sending of control data in the sending A-Module. As this is not provided by the AudioFile A-Module, an error message is generated.

- `void a_receiver_ctrl_back`

  This function is the call-back function for the control data in the receiving A-Module. The content of the control packet is copied to the *outofband* field in the instance pointer and the out-of-band communication is invoked calling the functions `liftFullStatusTreatement`.

- `void a_sender_OoBCommand`

  This function is called in the sending A-Module. The content of the *outofband* field in the instance pointer is interpreted and the corresponding actions are performed. These actions consist mostly in the setting of the parameter *playing*. The control commands *forward*, *fast forward*, *rewind* and *fast rewind* are prepared here but not implemented. When the *INIT* control packet is received, the function `sender_start_audio` is called.

- `void a_receiver_OoBCommand`

  This function is called in the receiving A-Module and performs the out-of-band commands. As there are no valid out-of-band commands in the receiving A-Module, a warning message is generated.

- `Lift_Status audioFile_rcvOutIn`

  This function is called by the Da CaPo++ core in case the Connection Manager receives out-of-band data for the receiving A-Module. The data is copied into the *outofband* field of the audio instance data structure and the function `a_receiver_OoBCommand` is called.

- `Lift_Status audioFile_rcvOutOut`

  This function is in the receiving A-Module and provides the sending of the out-of-band data. The content of the *outofband* field in the audio instance data structure is written to a Da CaPo++ packet. The return value `sbDataOk` informs the Lift that there is a packet to send.

- .`Lift_Status audioFile_sndOutIn`

  This function is called by the Da CaPo++ core in case the Connection Manager receives out-of-band data for the sending A-Module. The data is copied into the *outofband* field of the audio instance data structure and the function `a_sender_OoBCommand` is called.

- `Lift_Status audioFile_sndOutOut`

  This function is in the sending A-Module and provides the sending of the out-of-band data. The content of the *outofband* field in the audio instance data structure is written to a Da CaPo++ packet. The return value `sbDataOk` informs the Lift that there is a packet to send.

- `void setAudioInfo`

  This function is called by the receiving A-Module and sets the audio information (channels, sample rate, precision, etc.) of the audio device.

- `int sender_start_audio`

  This function opens the audio file, initializes the relevant values in the audio instance pointer and creates the timer to send the audio data.

- `Lift_Status asInitAudio`

  This function initializes the sending A-Module. The A-Module registers itself in the lower API. In case of a *stand-alone A-Module* the user has to set the environment variable *AUDIODATA*. If this variable is set, it indicates the audio file that is to be read. In this case the function `sender_start_audio` is called in the init procedure, otherwise the function is called after the *INIT* control command has been received by the sending A-Module.

- `Lift_Status asExitAudio`

  This function is called in the sending A-Module and closes the audio file.

- `Lift_Status asIndiAudio`

  This function is called whenever the timer triggers an alarm, i.e. whenever an audio data packet is to be sent. If *playing* is set to 0, this function simply returns. Otherwise, audio data is read from the audio file and written into a Da CaPo++ packet. The functions returns `sbDataOk`, so that the Lift sends the data to the peer.
- `void asTimerAudio`

  This function is the call-back function for the periodic timer created in the sending AudioFile A-Module. It calls the Lift function `lift_DataAvailable` so that the Lift then calls the function `asIndiAudio`.
- `Lift_Status asIndiAudioEmpty`

  This function is empty.
- `Lift_Status asStartAudio`

  This function is empty and called when the Lift is started.
- `Lift_Status asStopAudio`

  In this function, the periodic timer is destroyed.
- `int start_receiver_audio`

  This function opens the audio device for the receiving A-Module. The relevant values of the audio instance data structure are initialized with the values of the audio device.
- `Lift_Status asInitAudio`

  This function registers the A-Module in the lower API and calls the function `start_receiver_audio`.
- `Lift_Status arExitAudio`

  This function closes the audio device.
- `Lift_Status amStopAudio`

  This function is empty and called when the Lift is stopped.
- `Lift_Status arRequAudio`

  This function is called by the Lift whenever a data packet arrives for the receiving AudioFile A-Module. It is controlled, if the packet arrives too late and the data is written to the audio device.
- The rest of the file `a_audiofile.c` provides some functions for the CoRA in case it will be integrated as well as the definitions of the A-Module data structures for the data base.

8.7 (Multimedia) File Client

The Multimedia File Client (File Client in short) is the application on the user's site, that enables the receiving and displaying of multimedia data sent from the Multimedia File Server (File Server in short). The fact that the data is read from a file enables also the use of VCR control commands like play, pause and stop. The scope of this chapter is the design of the File Client as well a its current implementation status. Moreover, we will see how the "File" Client may also be extended to receive any other multimedia data (even live data).

The File Client is, as well as the File Server, an application or an application scenario within the Da CaPo++ application framework. Therefore, the design of the File Client follows the object-oriented paradigm. The implementation is done in the object-oriented language C++.

8.7.1 File Client and the Specification File

The File Client gets a specification file[1] as input. This file is parsed and the information is stored in the global variable `Specification`. This variable is defined as:

1. The language used in the specification file is defined in the document "The Specification File".

```
AlternativeSpec *Specification;
```

The class `AlternativeSpec` is a data structure taking all relevant information that is obtained from the specification file by the parser. Its definitions are in the file `appl_types.H` and its implementations in the file `appl_types.cc`. In the specification file different *alternative specifications* (which are enclosed by the `BEGIN_ALTER_SPEC` and the `END_ALTER_SPEC` tokens) may be specified. Each such specification contains:

- the type of the File Client needed to process the data.
- for each involved File Server[1]: the address of the *Multimedia Server*, the Da CaPo++ port and the multicast address of the *entry and configuration session* and a service flag for the data transmission.[2]
- for each session the session type (consisting in *unicast* or *multicast*) the name and all flows with the related synchronization specification.
- for each flow the name of the file to be read, the flow type in the server's and in the client's site, the name of the flow and the application requirements.

This information is used to instantiate the appropriate File Client type, to create the configuration files needed for the Upper API, as well as to connect to the server and to demand the transmission of the required file data.

While parsing the specification file the parser automatically generates the configuration files needed for the Upper API. These files are automatically named and stored on the temporal disk `/tmp` of the current machine. The naming conventions for the files are:

```
/tmp/configpid.Is.tmp for the sender and
/tmp/configpid.Ic.tmp
```

for the client, whereas *`pid`* is the current process id of the File Client and `I` indicates the number of the actually parsed session within the alternative specification, e.g. `/tmp/config.3s.tmp` or `/tmp/config.1c.tmp`. The language in the files follows the grammar rules for the configuration files.

The File Client application has to specify the global variable `finished` as follows:

```
int finished = 0;
```

before instantiating the class `ClientControlComponent`. This global variable indicates if the parser has finished parsing[3] of the specification file, i.e. the parser has reached the end of the specification file. In case an error occurred during the parsing process, the application will be stopped at once, as this is considered as a fatal error to abort.

8.7.2 Starting of the File Client Application

When all this information is appropriately created the File Client starts the required File Client type. The flow chart in Figure 26 on page 65 illustrates this process, which may be highly iterative in case of multiple failures in connecting sessions and multiple alternative specifications.

The in the flow chart shown steps are presented in detail in the following sections.

8.7.2.1 Parsing of an Alternative Specification

As already mentioned the block enclosed by `BEGIN_ALTER_SPEC` and `END_ALTER_SPEC` corresponds to an alternative specification. The parser parses one such specification, creates the required configuration files from the session blocks (included within `BEGIN_SESSION` and `END_SESSION`) and

1. The most general Multimedia File Server scenario consists in n File Servers which send data to m File Clients.
2. for instance this flag is used to determine whether the connection shall run on Ethernet (*www*) or on ATM (*wwwatm)*
3. This variable also is set 1, if a File Client/File Server application has successfully been established. In this case the other alternative specifications are no longer relevant.

stores all relevant information in the variable `Specification`. Then the parsing process is stopped and the File Client process continues.

8.7.2.2 Instantiating an Appropriate Class for File Client Type

Several different File Client types are possible and shall be introduced here. For each File Client type one specific class is to be implemented.

The Video Viewer. The Video Viewer File Client type enables the receiving and displaying of one video stream on the client's site within a multicast session. The application instantiates one session containing one flow of the type `VIDEO_RECV_DEVICE`. The video data on the server's site is read from a file. Therefore, the type of the flow in the File Server application is `VIDEO_SEND_FILE`. Alternative specifications for the Video Viewer follow the following scheme:

```
BEGIN_ALTER_SPEC
TYPE VIDEO_VIEWER;
PEER (address, group_address, service, interface);
BEGIN_SESSION
SESSION_TYPE MULTICAST;
SESSION video;
FILE name_of_file_to_read;
FLOW VIDEO_SEND_FILE VIDEO_RECV_DEVICE video_flow
{
     /* all necessary application requirements */
}
ENDFLOW
END_SESSION;
END_ALTER_SPEC;
```

The Video Viewer is already implemented. Its implementation is presented in Section 8.10.

The Audio Player. The Audio Player File Client type enables the client to receive and display one audio stream within a multicast session. The audio data is stored in a file. The flow type on the client's site is `AUDIO_RECV_DEVICE` and on the server's site `AUDIO_SEND_FILE`. The Audio Player's alternative specifications follow the following scheme:

```
BEGIN_ALTER_SPEC
TYPE AUDIO_PLAYER;
PEER (address, group_address, service, interface);
BEGIN_SESSION
SESSION_TYPE MULTICAST;
SESSION audio;
FILE name_of_file_to_read;
FLOW AUDIO_SEND_FILE AUDIO_RECV_DEVICE audio_flow
{
     /* all necessary application requirements */
}
ENDFLOW
END_SESSION;
END_ALTER_SPEC;
```

The Audio Player also is implemented. Its implementation will be described in Section 8.12.

8.7.2.3 Connect to File Server (Entry and Configuration Session)

Each File Server provides entry points that specify an *entry and configuration session* which is listening to clients to connect. The characteristics of these sessions are simple. They contain one flow to transmit ASCII data. No special QoS requirements are defined for this session. The configuration file of this session is known to all possible File Servers and Clients and delivered directly with the File Server and File Client program.

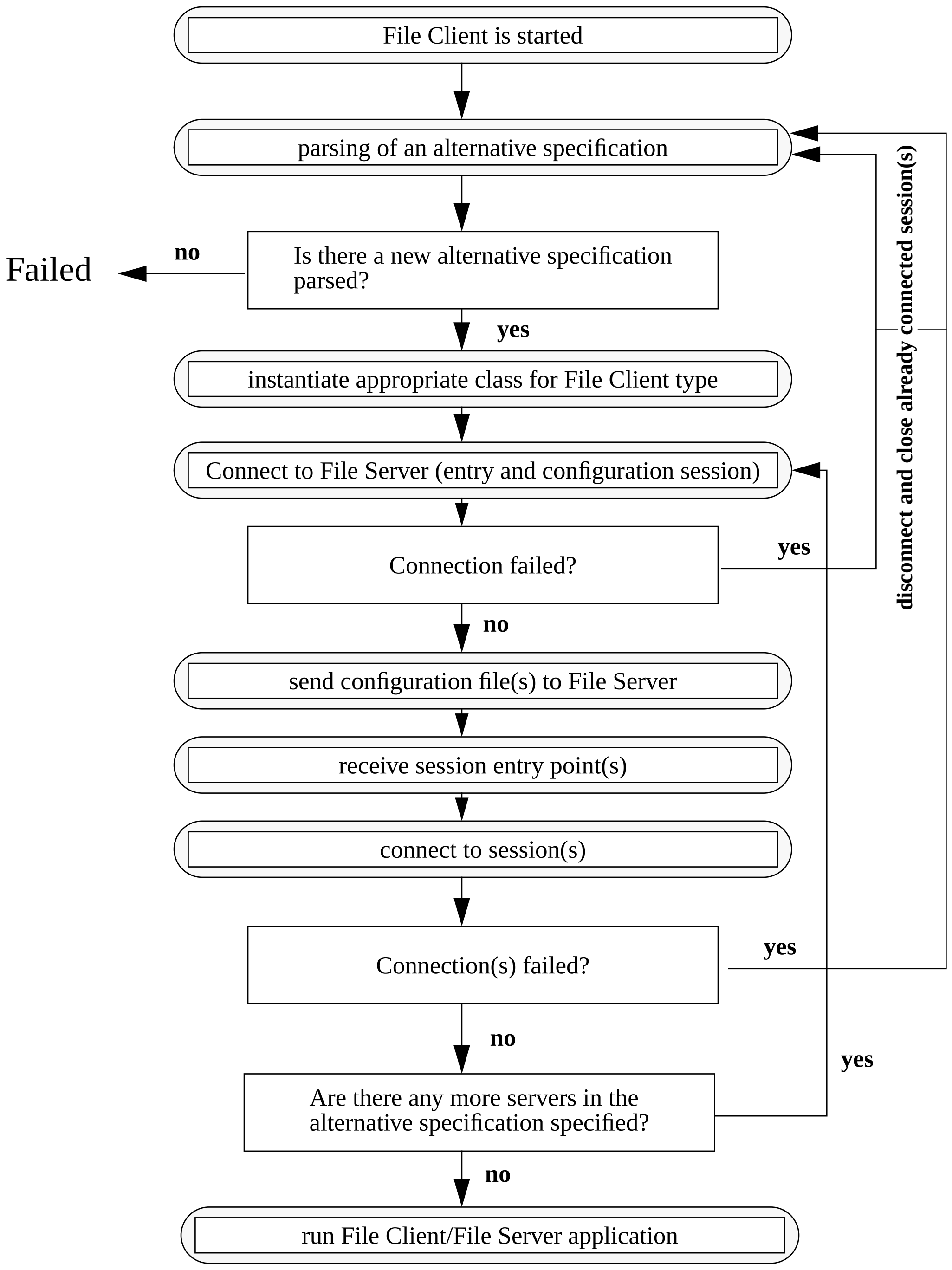

File Client is started
parsing of an alternative specification
Is there a new alternative specification parsed?
no
Failed
yes
instantiate appropriate class for File Client type
Connect to File Server (entry and configuration session)
Connection failed?
yes
no
disconnect and close already connected session(s)
send configuration file(s) to File Server
receive session entry point(s)
connect to session(s)
Connection(s) failed?
yes
no
yes
Are there any more servers in the alternative specification specified?
no
run File Client/File Server application

Figure 26. Flow Chart of the Initialization Process in the File Client

8.7.2.4 Sending Configuration File(s) to File Server

After the connection to the File Server is established, the session configuration file(s) the parser created for the server (`config`*`pid`*`.Is.tmp`) is (are) sent to the File Server. [1]

1. In the current implementation the Multimedia Server (= File Server) does only support client applications with one session. This could be changed easily when necessary.

8.7.2.5 Receiving Session Entry Point(s)

When the session(s) for the application is (are) created and listening, the server sends its (their) entry point(s) to the client to connect to the session. This was the design idea. As the *Listen* method of the session class is a blocking procedure, the address has to be sent *before* the session is really listening. To avoid race conditions, the File Client waits for 3 seconds after receiving the address(es) before connecting to the data session (s).

8.7.2.6 Connecting to Session(s)

After receiving the session entry point(s) for the session(s) in the File Server, the File Client itself will create session(s) using the configuration file(s) (`config`*`pid`*`.Ic.tmp`) the parser created. When its session(s) is (are) properly connected to the File Server session(s), the File Client/File Server application can be run and the data can be transmitted from server to client.

8.7.2.7 Iterations in the Initialization Process

As long as no successful connection to the File Server(s) is established and as there exist at least one more alternative specification, the File Client tries to connect to the File Server according to the specification information parsed from the specification file. The File Client has to be designed carefully so that in case of iterations all already created and possibly connected sessions are properly disconnected and closed.

8.7.3 Implementation of the File Client

8.7.3.1 Class ClientControlComponent

The File Client is implemented within one single class `ClientControlComponent`. The structure of the class is the following:

Table 19. The Class ClientControlComponent

	name	type	description + parameters
public	ClientControlComponent	constructor	instantiation of the Da CaPo++ Client
	RunClientControl	method	return value: void iterative parsing and instantiation of the appropriate class of File Client type parameters: int argc, char* argv[]: these parameters are passed to the File Client type class as the Tcl/Tk user interface depends on them
	~ClientControlComponent	destructor	
protected	DCmgr	DaCapoClient*	reference to the Da CaPo++ Client object

The Da CaPo++ Client has to be initialized once for a whole application, even if multiple sessions are to be connected. As can be seen in Figure 26 on page 65 it may be possible that multiple servers may be connected before the File Client/File Server application is running. For performance reasons, the Da CaPo++ Client is initialized within the constructor of the `ClientControlComponent`.

Possible Security Problem. As the Da CaPo++ Client is instantiated during the class' constructor this may imply some security problems. It is possible, that different alternative specifications in the specification file specify different File Client types. This makes sense, e.g., in case that a connection to a server providing video and audio data is tried first and, if no connection is possible, connections to server(s) only providing the video or the audio information or a textual representation of the movie's content are specified. This may lead to problems in case that the applications provide different security levels or in case that some applications are certified and others are not. For instance, it was decided that the Da CaPo++ Client is instantiated in the constructor, as it seems to be most probable, that different alternative entry points of one or more alternative File Servers providing the same application for an identical File Client type are specified within a specification file. If, later on, the File Client specification file is used to describe different possible application scenarios as alternatives (if the first cannot be established then the second is tried) and if these alternatives have different security characteristics, the class `ClientControlComponent` will be subject to change. The instantiation of the Da CaPo++ Client will have to be moved into the `RunClientControl` function.[1]

8.7.3.2 Function RunClientControl

The function `RunClientControl` contains the core functionality of the File Client. The frame for this function is the following:

```
wwwin = fopen (“file_to_parse”, “read”);
while (finished == 0) {
         wwwparse ();
         if (! finished) {
             switch (Specification->get_type()) {
             case VIDEO_VIEWER:
                  /*
                   * instantiate class for the Video Viewer
                   */
                  break;
             case AUDIO_PLAYER:
                  /*
                   * instantiate class for the Audio Player
                   */
                  break;
             /* other cases.... */
         };
    };
```

The variable `finished` serves to control the iterations in the parsing process. It is set 1if and only if:

1. The parser reached the end of the specification file to parse. In this case it’s the parser to set the variable to 1.
2. A successful File Client/File Server application has been established. In this case it is the implementation of the instance of the File Client type class which sets the variable to 1.

In general the function `RunClientControl` provides a loop to control the iterations of the parsing process and to instantiate the appropriate File Client type classes.

Instantiation of the Da CaPo++ Client. As stated in the Section , “Possible Security Problem,” on page 66, it may be desired in the future to instantiate the Da CaPo++ Client for each possible alternative specification. To do so, the code instantiating the Da CaPo++ Client has to be eliminated from the `ClientControlComponent` constructor. The new code for the `RunClientControl` function should follow the following scheme:

```
wwwin = fopen (“file_to_parse”, “read”);
while (finished == 0) {
         wwwparse ();
         if (! finished) {
             switch (Specification->get_type()) {
             case VIDEO_VIEWER:
                  // provide the relevant security information in the
                  // structure SecurityInfo secInfo
                  ....
                  // instantiate Da CaPo++ Client for this alternative
```

1. How this may be done is explained in `Section 8.7.2.3`.

```
                        // specification
                        DCmgr = new (DaCaPoClient &secInfo);

                        /*
                         * instantiate class for the Video Viewer
                         */
                        // destroy the Da CaPo++ Client
                        DCmgr->~DaCapoClient ();
                        break;
                   case AUDIO_PLAYER:
                        // analogous handling
              };
         };
```

In this case the Da CaPo++ Client is created before a File Client type class is instantiated and destroyed directly after the destructor of the class has been called. This has to be provided like that as the Da CaPo++ Client is started with the security information.

8.7.4 Restrictions in the Implementation

As a failure in the connection process of a session leads to errors in the Da CaPo++ system, the application can hardly recover from the initialization process as indicated in Figure 26. has not been implemented. The File Client just tries the first alternative specification. If this fails, the process is stopped. It should be possible to implement the robust initialization process after stabilizing the Da CaPo++ core.

8.7.5 Vision of Future File Clients

The File Client as well as the language for the specification file was designed as general as possible. For this reason many other File Client types as planned for instance are possible. First of all, we may think of File Clients that enable the synchronous receiving of audio **and** video data. File Clients can be specified that support the receiving of audio and video data stored on different servers. Clients are possible, that allow for different sessions with audio and video data, i.e. the parallel view of different movies. File Clients may be designed to receive text data via a ftp like session and in parallel some animating audio data to shorten the waiting time for the file transfer to finish. With the File Client concept as presented here applications like the Picture Phone, the Video Conference, etc. may be established. As every application component, application and application scenario is supposed to be designed according the object oriented paradigm and to be implemented in C++, every possible Da CaPo++ application and application scenario may be invoked by the File Client, under the precondition that the corresponding File Client type is defined and appropriate classes exist. The only thing to do to start these application will be the creation of appropriate specification files.

8.8 File Client Type Classes

As mentioned above, for each File Client type one designated class is provided that contains the functionality of this class. Each class that implements the functionality of one dedicated File Client can be named a *File Client type class*. The base class for all these classes is called `ClCtrlComp` and has the interface according to Table 20, whereas all methods are virtual and abstract.

All File Client type classes are inherited from this class and have to implement above mentioned three virtual functions.

Table 20. The Class ClCtrlComp

	name	type	description + parameters
public	ClCtrlComp	Construc-tor	Constructor DaCaPoClient *DCmgr: the private attribute pointing to the Da CaPo++ client is set to point to DCmgr
	event_handler	method	return value: int this method is supposed to provide the handling of events if any relevant events occur for the File Client parameter: int EVENT: denotes the number of the event occurred
	RunCtrlInterface	method	return value: void this method is supposed to provide an interface to the functionality of the A-Modules belonging to the File Clients flows to the user parameter: char *astring: this parameters denotes the programs name and is needed for Tcl/Tk GUIs
	error_handler	method	return value: int this method is supposed to provide handling of errors parameter: int error_id: denotes the error id
	~ClCtrlComp	destructor	
protected	DaCaPoClient *DCmgr	attribute	points to the Da CaPo++ client
	Session *entry_session	attribute	the pointer to the entry session of the File Client
	int entry_rcv_flow	attribute	the id of the flow in the entry session
	Str pid	attribute	the process id of the File Client process

8.9 Interface of File Client Type Classes

8.9.1 Motivation

The current implemented Video Viewer does not provide but a very rudimentary GUI. The GUI can be seen (as a screen shot) in Figure 27 on page 69. The current GUI is implemented in Tcl/Tk. As the name does already express, the GUI does not provide but an user friendly graphical based access to the functionality of the Video Viewer. Each File Client is supposed to get a GUI. These GUIs may be subject to change according to the users the program is provided for or according to new knowledge on GUI design. For this reason the GUI is not part of the File Client type classes. A new class has designed providing a general interface to the File Client type class for the GUI.

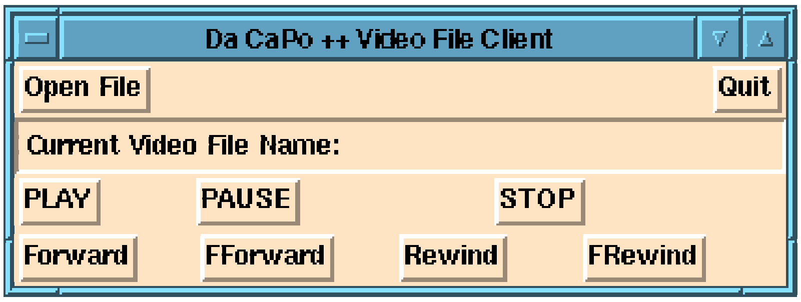


Figure 27. The Rudimentary GUI for the Video Viewer File Client

8.9.2 Interface Class GenericFct

The class `GenericFct` was designed on account of the needs of Tcl/Tk. Pressing buttons like the “PLAY” button in Figure 27 on page 69 initiates the call of a Tcl/Tk command. These commands are created within the application by command lines like:

```
Tcl_CreateCommand (interp, “play”, IPlayVideo,
     (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
```

whereas `interp` is the Tk interpreter and `IPlayVideo` is a C function providing the functionality of the command `play`. The function `IPlayVideo` calls the *playing* functionality of the Video Viewer class.

It is possible that multiple different File Client type classes exist using the same GUI to discard their differences in the implementation to the user. To profit from the challenges of object-oriented programming as much as possible by enabling different File Client classes using the same GUI and related functions and re-using as much code as possible, the class `GenericFct` was provided.

8.9.3 Class GenericFct

The class definition of `GenericFct` is presented in the following table.[1]

Table 21. The Class GenericFct

	name	**type**	**description + parameters**
public	GenericFct	constructor	
	CallFct	method	return value: int this methods provides the interface to the functionality of the class parameter: int flag: this parameters denotes the desired functionality to be performed
	~GenericFct	destructor	
	char *get_filename()	method	this function returns the name of the file to be read
protected	char *read_file_name	attribute	name of the file to be read

The classes inheriting from `GenericFct` have to implement the function `CallFct`. This function gets an integer value passed and based on this integer value the demanded functionality is provided.

8.9.4 Interface to the File Client’s Functionality

To provide an interface for GUIs to the functionality of File Client’s regardless of the class names a global variable

```
GenericFct *FileClient;
```

is defined within the application.

In the File Client’s function `RunClientControl` this pointer is initialized in the following way:

```
case VIDEO_VIEWER:
```

1. In the momentary implementation there is also the read_file_name as attribute of the class and an additional method to obtain the file name called get_filename. This should be subject to change, as this is only sensible in case of one GUI for one single flow. If there are multiple flows being read from files, multiple names of files to be read would be needed. For this reason it is much more sensible to locate this information in the File Client type class.

```
/*
 * instantiate class for the Video Viewer
 * let VideoViewer point to that instance
 */
if (successful connection) {
    FileClient = VideoViewer;
    VideoViewer->RunControlInterface (astring);
    };
/* clean up */
...
```

That is, the global pointer `FileClient` points to the File Client class type. Therefore, all other functionality being in this class cannot be accessed by the `FileClient` pointer. Instead this pointer is a global variable pointing to the interface of File Client type classes and valid in all cases of different possible File Client types and implementations.

The valid values of the parameter `flag` depend on the type of the File Client. In general, they may be defined as an enumeration type and so implicitly be casted to integer values.

The GUI of a File Client needs only to call the `CallFct` of the File Client type class and to pass the `flag` value corresponding to the desired functionality.

Annotation. In spite of the “most general” design, the GUI has of course to be chosen with respect to the functionality of the specific File Client.

8.9.5 Integration of New GUIs for Already Existing File Client Type Classes

On account of the above mentioned design it is quite simple to create a new GUI for an already existing File Client type class. There are two things to be done:

1. The new GUI has to provide access to all desired functionality of the File Client. This is to be done with respect to the File Client type and the defined `flag` values belonging to this class. The File Client’s functionality simply is called by calling the `CallFct` of the File Client. This is done by using the global variable `FileClient`. The function call is
1. FileClient->CallFct(flag);
1. whereas `flag` denotes the desired functionality.
1. A new class has to be derived from the already existing File Client type class which provides the following changes:

Table 22. The derived class for the File Client with new GUI

	name	**type**	**overloaded functionality**
public	newClass	constructor	call the constructor of the base class
	RunCtrlInterface	method	copy the function of the base class and exchange the call of the function to run the GUI
	~newClass	destructor	call the destructor of base class

Nothing more needs to be changed to provide a new GUI for an already existing File Client type class. If the programmer wishes to enhance the event handler and/or the error handler function, he/she is free to do so if these functions were specified to be virtual in the base class.

8.10 Video Viewer

The Video Viewer has already been implemented enabling clients to watch video movies via the Da CaPo++ protocol. The Video Viewer is an application consisting in one session containing one single video flow. The corresponding A-Module for the Video Viewer video flow is the *SunVideoFile* A-Module already described in Section 8.5. The File Client type class implementing the Video Viewer is the class `VideoFileClCtrlComp` presented in the next section.

8.10.1 Class VideoFileClCtrlComp

The class `VideoFileClCtrlComp` has two base classes:

1. the class `ClCtrlComp` which defines the functionality a File Client type class has to provide and
1. the class `GenericFct` defining the interface for the File Client.
2. The class definition is given in Table 23.

Table 23. The Class VideoFileClCtrlComp

	name	**type**	**inherited from**	**description + parameters**
public	VideoFileClCtrl-Comp	constructor		description: see below DaCaPoClient *mgr: this parameter points to the Da CaPo++ Client for the Video Viewer
	RunCtrlInterface	method	ClCtrlComp	return value: void the GUI function is called within this method parameter: char *argstring: this string has to be passed to the GUI function (at least for Tcl/Tk)
	~VideoFileClCtrl-Comp	destructor		closes the session; calls the destructors of `GenericFct` and `ClCtrl-Comp`
	event_handler	method	ClCtrlComp	return value: int the return value denotes the error code, if an error occurred this function provides the handling of events; in this class not imple-mented parameter: int EVENT: this parameter indicates the event that occurred
	error_handler	method	ClCtrlComp	return value: int the return value denotes the error code, if the error could not be recovered this function provides the handling of errors; it is not implemented parameter: int error_id; this parameters denotes the error occurred
	CallFct	method	GenericFct	return value: int the return value is 0 if the obtained control was valid and 1 of it was not valid this function sends the control to the video flow (detailed description see below) parameter: int flag: indicates the functionality being called
	get_filename	method	GenericFct	not overloaded, therefore not visible in the Class' definition.

Table 23. The Class VideoFileClCtrlComp

	name	type	inherited from	description + parameters
public	GM_Attach	method		return value: void this function is called by the group management system for the application sharing and attaches a new Video Client by contacting the Video Demon on the machine and sending the relevant information for dynamic multicast parameters: int ref this is the group management reference id for this request char *host_name indicates the name of the host the application shall be shared with
	GM_Detach	method		return value: void this function is called by the group management system to detach a host in the application sharing scenario. A *kill* packet is sent to the Video Demon on the machine to be detached parameters: int ref this is the group management internal reference id char *host_name indicates the name of the host to detach
protected	Session *VideoFileSession	attribute		the pointer to the video session in the Video Viewer
	int VideoFlow	attribute		this integer is the FlowDescriptor value for the video flow which is needed to send control or data to the video flow via the Upper API
	char *FlowName	attribute		contains the flow name being obtained from the specification file
	char *FileName	attribute		contains the name of the video file to be read
	Session *demon_session	attribute		the pointer to the demon session for the application sharing scenario
	int demon_send_flow	attribute		the flow descriptor for the sending flow in the video demon session
	int demon_recv_flow	attribute		the flow descriptor for the receiving flow in the video demon session
	char *read_file_name	attribute	GenericFct	name of the video data file
	DaCaPoClient *DCmgr	attribute	ClCtrlComp	pointer to the Da CaPo++ Client
	Session *entry_session	attribute	ClCtrlComp	pointer to the entry session

Table 23. The Class VideoFileClCtrlComp

	name	type	inherited from	description + parameters
pro-tected	int entry_rcv_flow	attribute	ClCtrlComp	the flow descriptor for the data flow in the entry session
	Str pid	attribute	ClCtrlComp	the process id of the File Client

In the following sections the constructor and the functions `RunCtrlInterface` and `CallFct` are presented in more detail.

8.10.1.1 Constructor: Design

The constructor of the Video Viewer establishes the connection to the File Server depending on the information in the `Specification` data structure. The flow chart in Figure 28. illustrates this process. In general the flow chart follows the process already shown partially in Figure 26 on page 65. Only the specific, numbered details are explained here:

- ad "successful creation of session" (1): It is possible that for some reason a session cannot be created properly. This case has to be treated by an appropriate action in the application to avoid a brute and improper aborting of the application.
- ad "time-out or disconnect[1]" (2): it is of course possible that in spite of a successful connection to the server, for some reason[2] no session entry point is sent to the File Client. To avoid *endless* waiting, the File Client starts a timer when sending the configuration file to the server.
- ad "connection established" and "failure" (3+4): the successful or not successful connection to the Video Server is equivalent to the `VideoFileSession` pointer pointing to a session or being a NULL pointer respectively.
- ad "A": the configuration file used to create the session is delivered with the Video Viewer as it is identical for all entry and configuration sessions
- ad "B": the address to connect is the peer's address specified in the specification file
- ad "C": the configuration file used to create this session is created by the parser and was retrieved from the information in the specification file
- ad "D": the address to connect to the session is the session entry point sent by the Video Server during the entry and configuration session

If the connection of the session was successfully established, the File Name of the file to be read is sent to the Video Server with an INIT control packet.[3]

8.10.1.2 Constructor: Implementation

The design of the constructor is more general and more robust than its current implementation. This is due to the fact that the Da CaPo++ core generates a *fatal* error in case that connections cannot be established. For this reason the constructor implements the entry and configuration session as described in Section 8.10.3.

8.10.1.3 Function RunCtrlInterface

The function `RunCtrlInterface` provides the following steps:

- set the File Client interface to the currently instantiated object.
- if the connection to the Video Viewer could not be established abort the function

1. The disconnect event is performed by a *disconnect packet* that is sent from File Server to File Client to indicate, that the File Server cannot perform the client's request.
2. This may also include a faulty implementation of a File Server.
3. see also Table 13 on page 52.

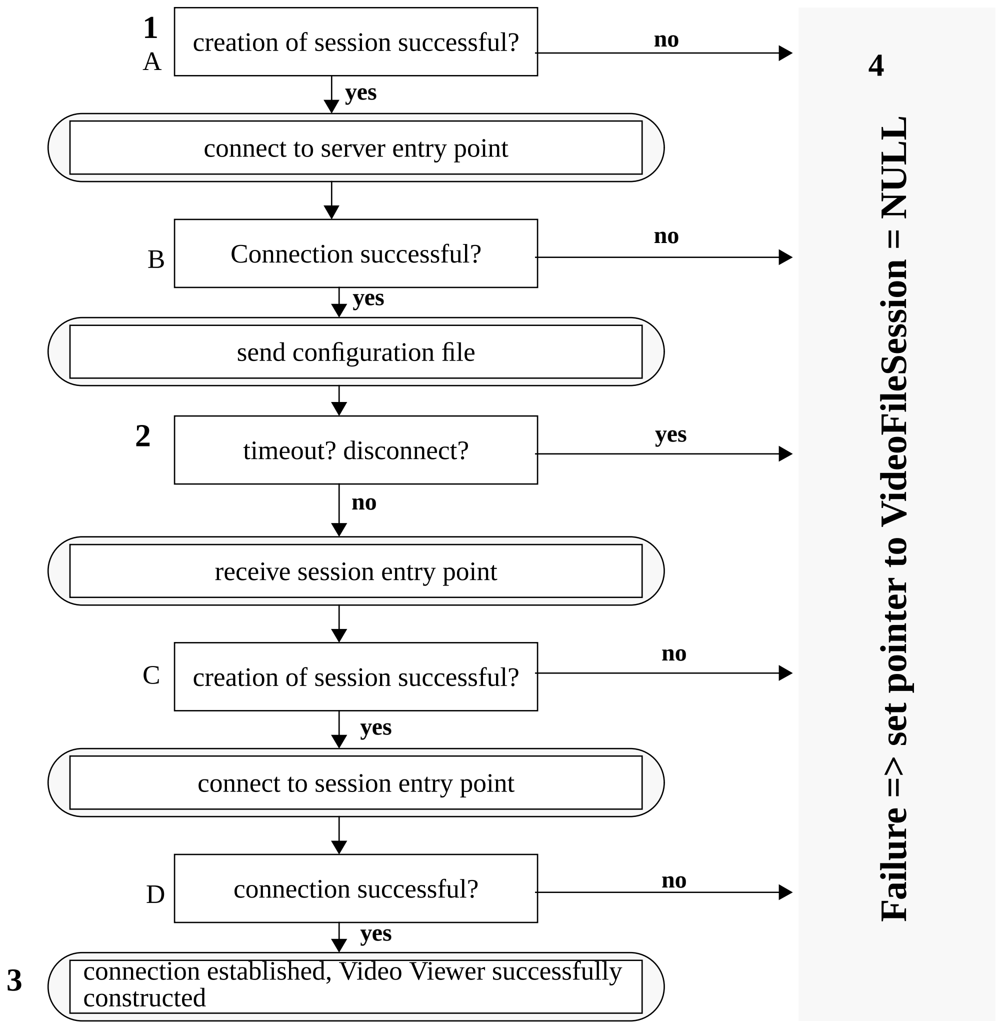

1
A
creation of session successful?
no
yes
connect to server entry point
B
Connection successful?
no
yes
send configuration file
2
timeout? disconnect?
yes
no
receive session entry point
C
creation of session successful?
no
yes
connect to session entry point
D
connection successful?
no
yes
3
connection established, Video Viewer successfully constructed
4
Failure => set pointer to VideoFileSession = NULL

Figure 28. Connection Establishment in the Video Viewer

- set the global variable `finished` to 1 to indicate that a successful connection has been established
- start the graphical user interface by invoking the function providing the GUI

8.10.1.4 Function CallFct

The `CallFct` function's functionality is based on the definition of the possible flags to be passed to the function. The flags defined for the Video Viewer are:

Table 24. Flags Defined for the Video Viewer

flag	**meaning**	**control packet structure**[a]
V_OPEN	send open command to A-Module	OPEN
V_PLAY	send play command to A-Module	PLAY
V_PAUSE	send pause command to A-Module	PAUSE
V_STOP	send stop command to A-Module	STOP
V_FWD	send forward command to A-Module	FWD
V_FF	send fast forward command to A-Module	FF

Table 24. Flags Defined for the Video Viewer

flag	meaning	control packet structure[a]
V_RWD	send rewind command to A-Module	RWD
V_FR	send fast rewind command to A-Module	FR
V_DISPLAY_FRAME	do nothing; this command is necessary only when the GUI is in the same process thread as the video displaying unit and in case of Tcl/Tk to be provided as default command (and thus give control to the `CallFct` regularly, even if no command button is pressed). This function is not necessary in the Da CaPo++ environment if using Tcl/Tk.	no control packet
V_ATTACH	call the GM_Attach method	no control packet
V_DETACH	call the GM_Detach method	no control packet

a. These packets are the same as defined for the SunVideoFile A-Module in Table 13 on page 52

The control packets corresponding to the flags are generated in the function `CallFct` and sent to the A-Module using the IPC and the out-of-band communication as explained in Section 8.4.2 on page 48.

8.10.2 GUI of the Video Viewer

The graphical user interface of the Video Viewer is implemented in Tcl/Tk as can be seen in Figure 27 on page 69. Its implementation is in the file `appl_VVtcltk.cc`. The file provides the following functions:

- `void IxxVideo:` where `xx` is: `Play, Pause, Stop, Fwd, FF, Rwd, FR` and the functions `playing` and `IOpenVideoFile`:

  These functions provide the Tcl/Tk commands created to call the File Client's functionality. The functions call the `CallFct` of the Video Viewer with the flags (the sequence corresponds to the sequence of functions above): `V_PLAY, V_PAUSE, V_STOP, V_FWD, V_FF, V_RWD, V_FR, V_DISPLAY_FRAME, V_OPEN`
- `void RunVideoViewerGUI`

  This function implements the Tcl/Tk graphical user interface. The Tcl and the Tk command interpreters are created as well as are the Tcl commands[1]. The Tcl/Tk window is created and the control is given to the GUI until the user presses the "quit" button in the GUI. The Tcl/Tk input file that defines the GUI is `appl_VVgui.tcl` In case of a shared Video Viewer, the GUI has to be put onto the display `:10`. Also the *GM_Check* function has to be called regularly.[2]

8.10.3 Implementation of the Entry and Configuration Session

In the *entry and configuration session*, the Video Viewer sends information on the service to get (especially the configuration file of the server session) to the File Server and retrieves an address of the entry point for the Video Session. The entry and configuration session is implemented as a multicast session. As the Da CaPo++ sessions for data transfer must be closed on both sides to properly leave the application and stop data transfer *and* display, the entry and configuration session is retained during the whole application. It is used, too, for a *leave* message in either case, the File Server being shut down or a File Client being closed on user request.

An application on the client side must be identified on the sever. As the entry and configuration session is a multicast session, all possible clients receive packets like address packets or the request that one spe-

1. The creation of the Tcl commands is already shown in the Paragraph 8.9.2.
2. For more details on the shared Video Viewer see Section 8.11

cific client shuts down. For this reason the packets have to identify unambiguously the client they are meant for. For this identification the pair (*host_id, process_id)* has been used.

As described in Section 8.16 the File Server has been implemented as a Multimedia Server and a Multimedia Sender. The *leave* message from the client is received in the server which stops the process of the Multimedia Sender. When the application is stopped from the server side, this is controlled by the Multimedia Server which stops the sender process and sends the *leaving* message to the client.

8.10.3.1 The Data Packets for the Entry and Configuration Session

Following data packets may be sent from the Video Viewer to the Multimedia Server:

Table 25. Packets from Client to Server during the Entry and Configuration Session

packet structure	meaning
SESSIONFILE;[hostname][processid]VIDEO[remote_interface]{sessionfile}	this packet contains the configuration file for the video file session. The server gets the flag *VIDEO* to identify the application. The pair [hostname][processid] is used to clearly identify the client. The remote interface is used so that the Multimedia Sender can correctly set the atmFlag.
LEAVE[hostname][processid]	this packet is sent if the user has left the application

The entry and configuration session is a multicast session and the sending direction goes from File Server to Video Viewer. For this reason, the packets from the Video Viewer to the File Server are sent using the out-of-band communication.

These packets are sent from the Multimedia Server to the Video Viewer:

Table 26. Packets from Server to Client during the Entry and Configuration Session

packet structure	meaning
ADDRESS[client_host]{client_pid][sender_name][dacapo_port][dacapo_multicast_address]	this packet contains the specification of the client it is meant for, i.e. the clients pair of host name and process id. Then the entry point for the data session is described by the host name, the Da CaPo++ port name and the multicast address to use to connect to the Multimedia Sender.
LEAVING[client_host][client_pid]	this packet sends the leaving message to tell the client with *client_host* and *client_pid* that the sender is being shut down.

8.10.3.2 Initializing a New Video Session

Whenever a client connects to the entry and configuration session of a Multimedia Server, the server obtains a *join* event of the Da CaPo++ core system. The join event includes the client host id. The establishment of the connection of the entry and configuration session is indicated by a *start* event on both sides. The Video Viewer sends its *SESSIONFILE* packet to the Multimedia Server. The Multimedia Server starts a Multimedia Sender and provides to the sender the addresses it shall use for the video file session as well as the configuration file for the session. The addresses are also sent to the client which, in turn, can now connect to the appropriate session in the Multimedia Sender.

8.10.3.3 Leaving a Video Session

Both, the Video Viewer and the Multimedia Sender, may stop the video file session. As Da CaPo++ requires a disconnect and close of the session on both sides, the other side has to be properly informed that one side is closing down the session. This information exchange is done between the Multimedia Server and the Video Viewer in the entry and configuration session[1]. In case the client application is shut

1. with this concept of leaving of sessions, the *entry and configuration session* gets the role of a *control session.* The name was retained due to “historical reasons”.

down, the Video Viewer sends to the Multimedia Server a *LEAVE* packet indicating its host name and process id. The Multimedia Server looks up the process id of the correspondent Multimedia Sender and send a *SIGTERM* signal to the Multimedia Sender. The Video Viewer application sleeps for 2 seconds so that the Multimedia Sender as creator of the session may disconnect and close the session first.[1]

The Multimedia Sender has no control interface. The sending application can be shut down on sender side by invoking the command *kill* in the menu of the Multimedia Server for the specified sender. In this case a *LEAVING* packet is sent to the Video Viewer and a *SIGTERM* signal is sent to the Multimedia Sender. So both, the Video Viewer and the Multimedia Sender do properly leave the application, i.e. disconnect and close the video file session.

8.11 Shared Video Viewer

In the project plan a *Shared Extended WWW* scenario has been described. In this scenario, the Extended WWW scenario as described in Section 8.2 is to be shared. To do this, the *xwedge*, as developed in a former RACE project, was to be used to perform the application sharing. As sharing of continuous multimedia data cannot be performed easily by the xwedge, the distribution of the video data information was to be done by the Da CaPo++ system. The xwedge itself could not be fully integrated in Da CaPo++, thus, it still is running over TCP/IP. For this reason a *shared Video Viewer* concept had been developed and implemented as a proof of concept how the Shared Extended WWW scenario could be provided using Da CaPo++.

8.11.1 Process Model

To realize the Shared Video Viewer concept, several processes are involved. A complete outline is given in Figure 29.

The following processes are involved in the scenario:

- **Da CaPo++ Core** on each machine
- **Da CaPo++ Multimedia Server** and **Da CaPo++ Multimedia Sender** in the sender side. As described in Section 8.16, the File Server consists of two processes, the server and the sender process.
- **Da CaPo++ Video Viewer** is the in Section 8.10 presented Video Viewer. It is sharing aware and communicates with the GM control process and with the Video Demon.
- **GM control** is the interface to the application sharing. It allows the Video Viewer to get information if the application is shared.
- **Application Sharing, PC and PS** is the xwedge implementation. PC is the pseudo client and PS is the pseudo server. In fact, pseudo client and pseudo server are always present if the xwedge is started, but the PS is used on the machine that wants to share an application and PC is used on the machine that receives the data of the shared application.
- **Da CaPo++ Video Demon** is the process that controls the receiving of the shared video data on the receiving machine. It communicates with the Da CaPo++ Video Viewer.
- **Da CaPo++ Video Viewer2** is the video receiving application on the receiver of the sharing. It is started and stopped by the Da CaPo++ Video Demon.

8.11.2 Group Management Interface of the Application Sharing

The *group management* as implemented in this scenario provides an interface between the Da CaPo++ Video Viewer and the application sharing, i.e. the xwedge. It was decided not to change the xwedge for this scenario. As the application sharing itself is performed by the xwedge, that shares the graphical user interface in our scenario, and as only the Video Viewer knows the relevant data for a second Video

1. otherwise the Da CaPo++ core system may get some problems.

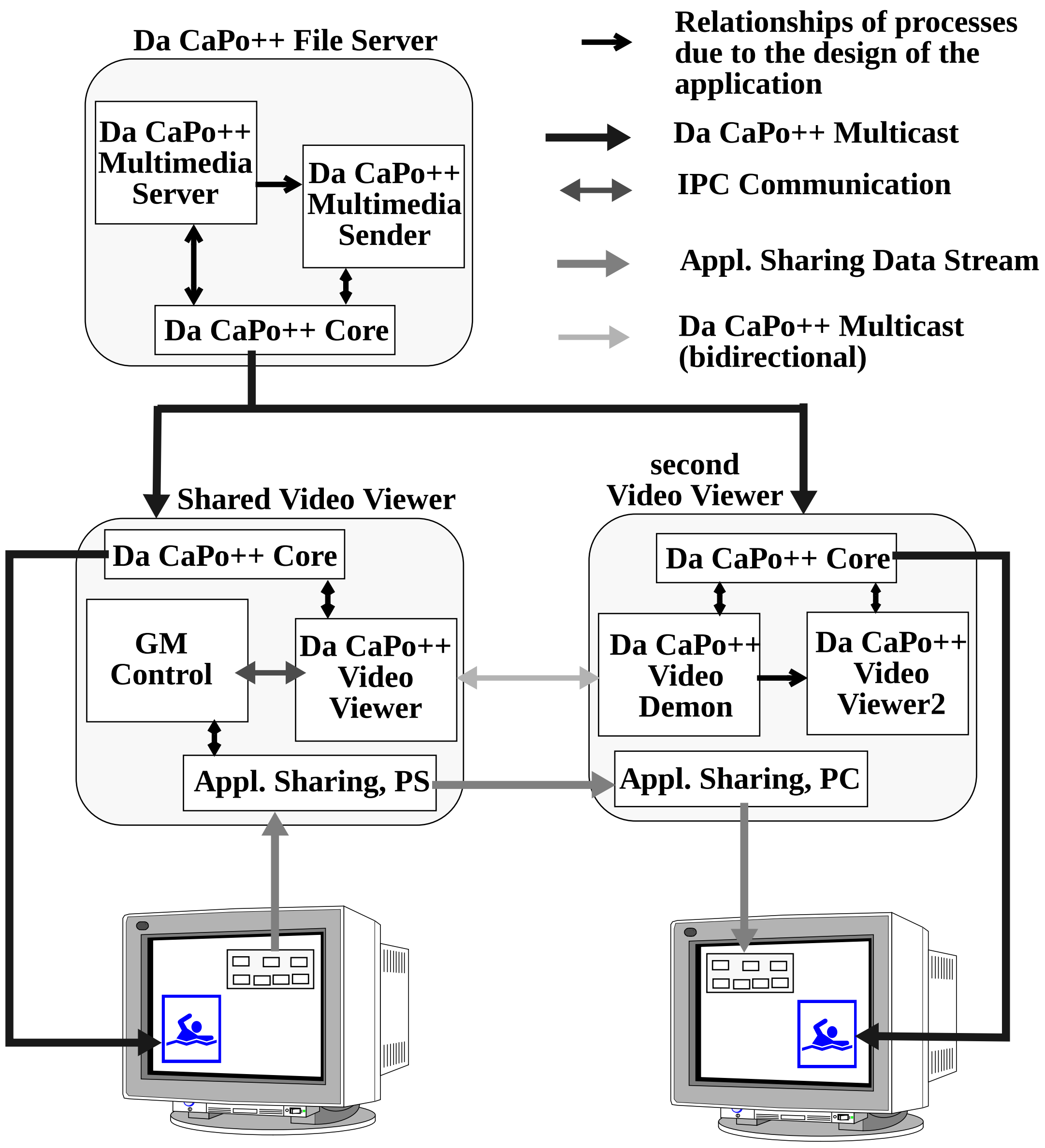


Figure 29. The Process Model in the Shared Video Viewer Scenario

Viewer to join the multicast session in the Da CaPo++ Multicast Sender, both processes have to be involved in the sharing process and have to communicate to each other. This task is provided by the group management process. A Da CaPo++ Video Viewer that can be shared has to register itself to the group management. The group management informs the Video Viewer whenever a sharing client is added to or removed from the sharing process. The Video Viewer then is responsible that the sharing client starts or stops the representation of the shared video data. The xwedge, in turn, is responsible that the sharing client retrieves or not the GUI of the Video Viewer application.

The group management process provides a simple, ASCII based user interface that allows for the specification of commands as the change of the floor control or the retrieval of status information. Only two commands are relevant in the Video Viewer, too. The command *adduser* means that the application shall be shared with another machine. It takes the machine name as command. The command *deluser* stops the sharing of the application with a specific machine that is specified as parameter. These two commands are explained in more detail later.

8.11.3 Involved Packets

The following packets are send from the Da CaPo++ Video Viewer to the Da CaPo++ Video Demon. While Table 27 defines the direction Viewer to Daemon, Table 28 defines packets heading from Video Demon to Video Viewer..

Table 27. Packets from the Video Viewer to the Video Demon

packet structure	**meaning**
CREATE_CLIENT[request_id][sender_host_id][sender_port_id][sender_remote_interface]	this packet is the request to the Video Demon to create a new Video Viewer2. It already contains all information needed by the second Video Viewer to connect to the Multimedia Sender
SESSION{sessionfile}	this packet contains the configuration file data for the video data session between Video Viewer2 and Multimedia Sender
KILL[request_id=0][process_id]	this packet is sent from Video Viewer to Video Demon to indicate that the Video Viewer2 with the Process Id *process_id* has to be terminated. The request_id variable is not yet used.

Table 28. Packets from the Video Demon to the Video Viewer

packet structure	**meaning**
INVALID_PACKET[]	this packet is sent to the Video Viewer in case the Video Demon had not been in the state to accept the packet received
PID[latest_request_id][process_id]	the Video Demon sends the process id of the newly created second Video Viewer to the Video Viewer

8.11.4 Process of Sharing the Video Viewer

Figure 30. shows the setup of the sharing of the Video Viewer.

The Video Viewer has to register itself at the beginning using the function `GM_Register`. This function gets as parameters the window id of the X-window of the GUI as well as pointers to the two functions of the Video Viewer that shall be called by the group management in case the sharing shall be setup or stopped. Those functions have to be pure “C”. As the Video Viewer was designed and implemented following an object-oriented approach to allow for the overloading of functions, it was decided to implement the *attach* and *detach* function for the sharing as methods (see also Table 23). Pure “C” functions were implemented and their pointers are used as parameters in the `GM_Register` function. These functions call C-like C++ functions which in turn call the `CallFct` in the Video Viewer with the flag V_ATTACH or V_DETACH. This function then calls the attach and detach methods of the class.

When the user of the Video Viewer application decides to share the application with a user on another machine, it types in the command *adduser*. The group management process then calls the attach function in the Video Viewer, which, at last, calls the `GM_Attach` method of the class. This method connects to the session of the Video Demon. This session always has the same session configuration file on all machines as well as the same Da CaPo++ port (`dacapo6`) and group address (`224.1.1.99`). For this reason the by the group management process provided host name is sufficient to properly connect the Video Demon Session. The Video Viewer then sends a *CREATE_CLIENT* packet to the Video Demon. This packet specifies the whole address to connect to the Multimedia Sender. Afterwards, it sends also the session configuration file as a *SESSION* packet to the Video Demon. The Video Demon, after receiving both packets, forks itself. The child process starts the second Video Viewer with the sender host name, the sender port, the sender group address, the sender remote address and the name of the locally saved session configuration file as command line parameters. The Video Demon parent process returns a *PID* packet to the shared Video Viewer. The pid packet contains the process id of the forked process, the second Video Viewer. The shared Video Viewer calls the function `GM_UserAttached` of the group management with the reference id and the process id of the second Video Viewer as parameters. The group management process thus has now the information, which process at the peer’s machine acts as second Video Viewer.

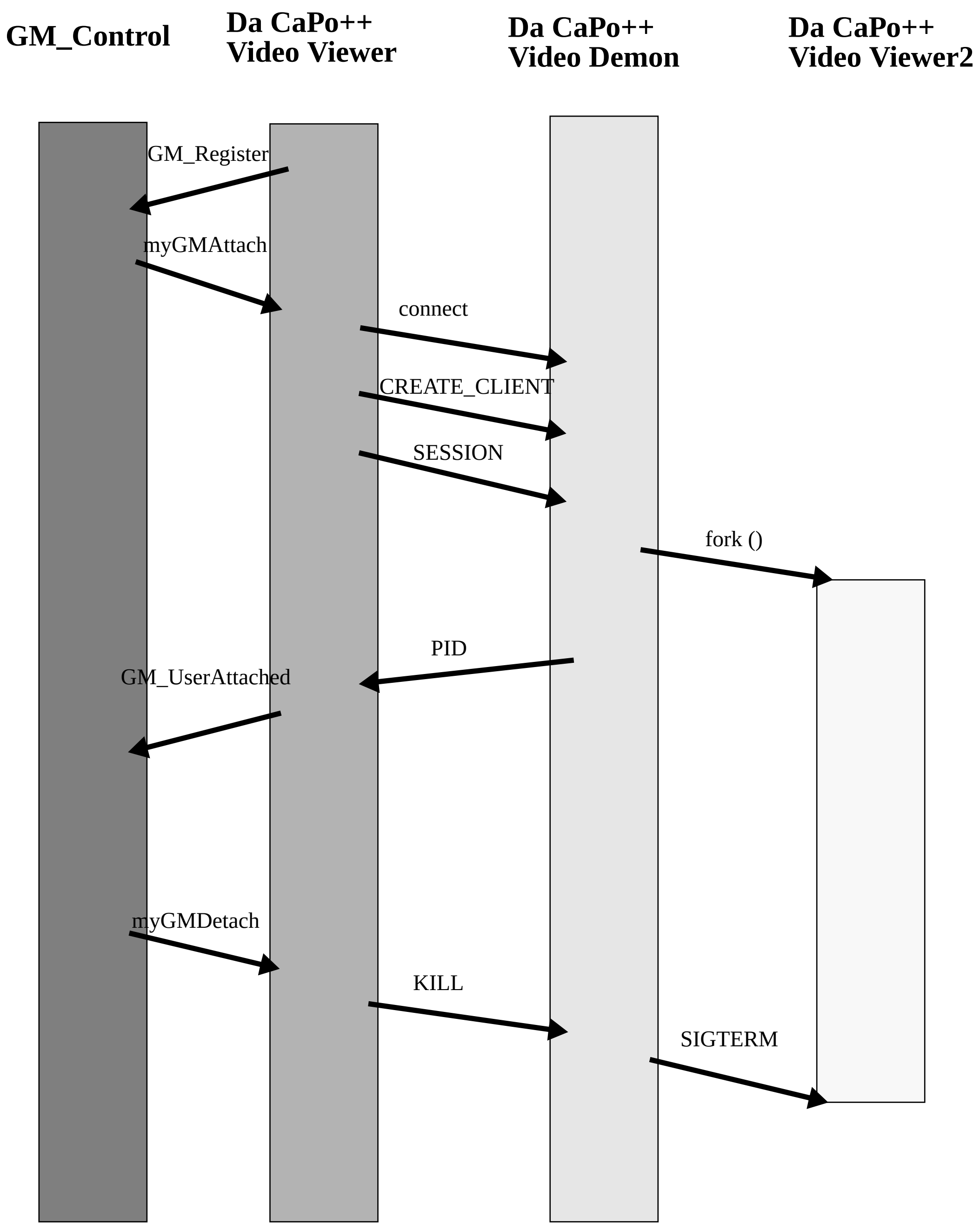


Figure 30. Setup of Sharing of the Video Viewer

The user can stop the sharing of the Video Viewer by typing the command *deluser machine_name* in the group management menu. The group management, then, calls the detach function which, at last, calls the `GM_Detach` method of the Video Viewer. This method gets the peer's host name as well as the process id of the second Video Viewer as parameters. The Video Viewer connects the session of the Video Demon if it has no longer a connection and sends a *kill* packet to the Video Demon. The `request_id` parameter of the kill packet is not used and, thus, for instance, just set to 0. The Video Demon sends a *SIGTERM* signal to the process specified by the process id received in the kill packet. So, the second Video Viewer is stopped.

The group management controls that the application sharing is started for the specified window id, when the *adduser* command is received and that the application sharing is stopped, when the *deluser* command is received.

8.11.5 Shared Video Viewer

The Video Viewer had to be modified to enable the sharing. The modifications are not restricted to the additional methods: `GM_Attach` and `GM_Detach`. The Video Viewer must register itself in the group management process. Before this can be done, the group management process has to be started. Also the xwedge must be started on the machine, as the group management process cannot be run without a present xwedge. For this reason, the Video Viewer can be started in two different modes, a normal mode and a shared mode. In this paragraph the additional points of the shared mode a explained in short. To start the Video Viewer in the shared mode, the flag `shared` has to be the second command line parameter, thus is, the third argument. If this flag exists, the main routine of the File Client initializes a variable `shared`, which normally is 0, with 1. In case of this variable set to 1, the X window for the GUI is sent to display`:10` as demanded by the application sharing in case of the Shared Video Viewer scenario. The Video Viewer then registers itself to the group management process. In case of a normal Video Viewer, the GUI is sent to display`:0` and the registration procedure is not called. In this case, the process is controlled by Tcl/Tk until the GUI is destroyed. The command `Tk_MainLoop ();` is called. In case of the shared Video Viewer, the function `GM_Check` must be called regularly to give the process control to the interface between Video Viewer and group management. So, the group management can call the detach or attach function if necessary. The corresponding code is:

```
while (tk_NumMainWindows >0) {
         Tk_DoOneEvent (0);
        Tk_DoWhenIdle(myGM_Check, (ClientData) NULL);
}
```

Besides these changes, the Video Viewer’s code is exactly the same for the normal and the shared case.

If the Multimedia Sender closes the connection, the second Video Viewer(s) has (have) to be informed to properly close their video session(s). For the prototypical implementation the Video Viewer does not keep a list of all current second Video Viewers related with its application, but knows if it is shared and knows the pid as well as the host name of the last sharing request. For this reason, the Video Viewer can call the detach function itself if it is shared and if the Multicast Sender closes the connection. To extend this feature to properly close down all second Video Viewers that may exist on different machines, a simple list administering process ids and host names of these viewers must be added to the Video Viewer.

8.11.6 Second Video Viewer

The second Video Viewer is called by the Video Daemon and enables the retrieval of video data via the multicast session at the Multimedia Sender. The second Video Viewer has command line parameters.

Table 29. Command Line Arguments of the Second Video Viewer

argument number	name	meaning
1	video_client2	programme name
2	server_name	name of the host the Multimedia Sender is running on
3	server_port	name of the Da CaPo++ port, the video session is connected to in the Multimedia Sender
4	group_adr	the multicast address for the video session
5	remote_interface	the interface for the video session (to distinguish the ATM from the Ethernet case)
6	file_name	name of the locally stored session configuration file for the video session. This file was stored by the Video Demon on the local machine.

With the information of these parameters, the second Video Client creates a Da CaPo Client, connects to the video session in the Multimedia Sender and, thus, retrieves the video data. Necessary changes to enable this dynamic multicast are described in Section 8.5. A signal handler also is implemented. In case of the signal *SIGTERM* received by the Video Demon, the video session is closed and the second Video Viewer exits from running.

8.11.7 Video Demon

The Video Demon is the program that controls the sharing of the video data stream on the receiving machine. The GUI data is received and shared by the xwedge, that has to run on the machine, too. The Video Demon creates a multicast session and listens. Whenever the machine shall be the receiver of a shared Video Viewer application, the shared Video Viewer connects to the Video Demon and transmits a *CREATE_CLIENT* as well as a *SESSION* packet. In case these packets arrive in a wrong order, the Video Demon ignores them and sends an *INVALID_PACKET* packet back to the shared Video Viewer. When the *SESSION* packet is received, the Video Demon stores its content into a local file and forks itself. The child process starts the second Video Viewer. The parent process sends the process id of the recently forked process to the Shared Video Viewer. If the Video Demon shall be used to control different sharing processes in parallel, the packets exchanged between Shared Video Viewer and Video Demon should contain the host id of the Video Viewer as well as its process id, too, to enable both processes perform correctly and to allow the shared Video Viewer to ignore packets not meant for it.

In case the sharing of video data shall be stopped, the Video Demon receives a *KILL* packet with the process id of the second Video Viewer process to stop. It sends a *SIGTERM* signal to the process specified by the process id.

8.12 Audio Player

The Audio Player has been implemented enabling clients to listen to CD quality audio data via the Da CaPo++ protocol. The Audio Player is an application consisting in one session with one single audio flow. The corresponding A-Module for the audio flow is the *AudioFile* A-Module already described in Section 8.6. The File Client type class implementing the Audio Player is the class `AudioPlayer-ClCtrlComp` presented in the next section.

8.12.1 Class AudioPlayerClCtrlComp

Like the class `VideoFileClCtrlComp`, the class `AudioPlayerClCtrlComp` has two base classes:

1. the class `ClCtrlComp` which defines the functionality a File Client type class has to provide and
2. the class `GenericFct` defining the interface for the File Client.

The class definition is given in Table 31.

The constructor as well as the function `RunCtrlInterface` of the Audio Player class is analog to the constructor and the function of the Video Viewer class. The implementation has been realized in an analog way, too. The Video Viewer sends the *SESSIONFILE* packet in the entry and configuration session with the flag *VIDEO*. The Audio Player sends it with the flag *AUDIO*. Please refer to Section 8.10.1.1, Section 8.10.1.2, Section 8.10.1.3 and Section 8.10.3 for details of the design and of the implementation.

8.12.2 Function CallFct

The `CallFct` function's functionality is based on the definition of the possible flags to be passed to the function. The flags defined in the Audio Player are:

Table 30. Flags Defined for the Audio Player

flag	meaning	control packet structure
A_OPEN	send open command to A-Module	OPEN
A_PLAY	send play command to A-Module	PLAY
A_PAUSE	send pause command to A-Module	PAUSE
A_STOP	send stop command to A-Module	STOP

8.13 GUI of the Audio Player

The graphical user interface of the Audio Player as implemented in Tcl/Tk can be seen in Figure 31.. Its implementation is in the file `appl_APtcltk.cc`.

Table 31. The Class AudioPlayerClCtrlComp

	name	**type**	**inherited from**	**description + parameters**
public	AudioPlayer-ClCtrlComp	constructor		description: see below DaCaPoClient *mgr: this parameter points to the Da CaPo++ Client for the Audio Player
	RunCtrlInterface	method	ClCtrlComp	return value: void the GUI function is called within this method parameter: char *argstring: this string has to be passed to the GUI function (at least for Tcl/Tk)
	~AudioPlayer-ClCtrlComp	destructor		closes the session; calls the destructors of `GenericFct` and `ClCtrl-Comp`
	event_handler	method	ClCtrlComp	return value: int the return value denotes the error code, if an error occurred this function provides the handling of events; in this class not imple-mented parameter: int EVENT: this parameter indicates the event that occurred
	error_handler	method	ClCtrlComp	return value: int the return value denotes the error code, if the error could not be recovered this function provides the handling of errors; it is not implemented parameter: int error_id; this parameters denotes the error occurred
	CallFct	method	GenericFct	return value: int the return value is 0 if the obtained control was valid and 1 of it was not valid this function sends the control to the audio flow (detailed description see below) parameter: int flag: indicates the functionality being called
	get_filename	method	GenericFct	not overloaded, therefore not visible in the Class' definition.
pro-tected	Session *Audi-oSession	attribute		the pointer to the audio session in the Audio Player
	int AudioFlow	attribute		this integer is the FlowDescriptor value for the audio flow which is needed to send control or data to the audio flow via the Upper API

Table 31. The Class AudioPlayerClCtrlComp

	name	type	inherited from	description + parameters
pro-tected	char *FlowName	attribute		contains the flow name being obtained from the specification file
	char *FileName	attribute		contains the name of the audio file to be read
	char *read_file_name	attribute	GenericFct	name of the audio data file
	DaCaPoClient *DCmgr	attribute	ClCtrlComp	pointer to the Da CaPo++ Client
	Session *entry_session	attribute	ClCtrlComp	pointer to the entry session
	int entry_rcv_flow	attribute	ClCtrlComp	the flow descriptor for the data flow in the entry session
	Str pid	attribute	ClCtrlComp	the process id of the File Client

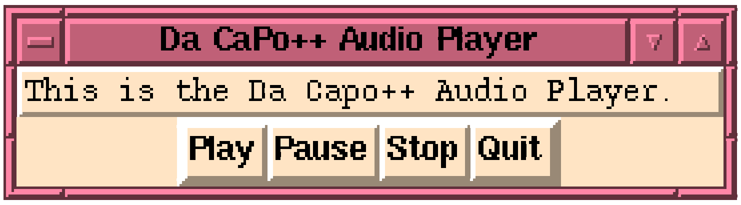


Figure 31. The GUI for the Audio Player File Client

The file `appl_APtcltk.cc` provides the following functions:

- `void IxxAudio`: where `xx` is: `Play, Pause` and `Stop` and the function `IOpenAudioFile`.

  These functions provide the Tcl/Tk commands created to call the File Clients functionality. The functions call the `CallFct` of the Audio Player with the flags: `A_PLAY, A_STOP, A_PAUSE, A_OPEN`

- `void RunAudioPlayerGui`

  This function implements the Tcl/Tk graphical user interface. The Tcl and Tk command interpreters are created as well as are the Tcl commands. The Tcl/Tk window is created and the control is given to the GUI until the user presses the “quit” button in the GUI. The Tcl/Tk input file that defines the GUI is `appl_APgui.tcl`.

8.14 Implementation of the File Client

The File Client is easily implemented with the above mentioned classes (Section 8.7 to Section 8.12.) Some global variables are to be defined:

- `serverf, clientf, errorf` of the type `ofstream` have to be defined. These stream pointers are needed by the parser to write the server and client configuration files and the error file.
- `BASE_DIR` of the type `Str` has to be defined and must be initialized with the name of the directory the configuration files and the error files shall be written into.
- `finished, parsing_started, Specification, FileClient` of the types `int, int, AlternativeSpec *, GenericFct *` respectively have to be defined.[1]

1. See also the document “The Specification File”.

After defining these global variables the relevant code implementing the File Client (which may be as general as possible if the required classes are provided) consists in the following statements:

```
ClientControlComponent *ClientControl = new ClientControlComponent ();
ClientControl->RunClientControl (argc, argv);
ClientControl->~ClientControlComponent ();
```

The File Client is implemented in the file `video_client.cc`.

8.15 Design of the File Server

The design of the File Server is shortly presented here by further explaining the three main points concerning its functionality. The File Server's design is quite easy to understands as it reflects directly the required counter part to the File Client design. A File Server can serve multiple File Client sessions at one time. For this reason the File Server has to somehow *administrate* the File Clients it is communicating to and the *sessions* related to these communications. For this purpose we define the term *user session.* A user session is the set of all sessions belonging to one designated File Server/File Client application between the server and one File Client. The File Server requires the possibility to shut down one user session in its entity (e.g. to free resources). Of course this should not be done without warning the client and without looking for other solutions, but the facility really is necessary to provide a File Server being able to run over a long period of time without the need of being restarted.

Each session has one specific entry point belonging to a *port* in the server's machine. As the number of ports is limited the server has to somehow *administrate free and used ports*.

Similarly to the process of connection establishment in the File Client as presented in Section 8.7.2 on page 63 the process of connection establishment in the File Server is not trivial, even more complex. As the data transmission is directly performed by the A-Modules, the *File Server routine* consists mainly in *connection establishment and closing* between File Server and File Client. These three issues concerning the design of the File Server will be presented in detail in this chapter.

8.15.1 Administration of User Sessions

One user session may contain multiple Da CaPo++ sessions. For this reason the class of user sessions contains a list of all related Da CaPo++ sessions. One list element does not only contain a pointer to the sessions and flow type information but also the port number of the session. The class for the user session needs the following methods:

Table 32. The Class for the User Session

	name	**type**	**description**
public	UserSession	constructor	
	get_id	method	returns id of user session
	get_first_session	method	returns first session of session list
	get_next_session	method	returns current session pointer, current points to the next session in the list after the function
	set_current_to_first	method	sets the current pointer to the first session
	add_session	method	adds one session to the list
	delete_session	method	removes one session from the list
	~UserSession	destructor	

Table 32. The Class for the User Session

	name	type	description
protected	id	integer	number to identify the user session
	session_list	SessionList[a]	points to the first element of the session list
	current	SessionList	points to the session to be retrieved from the list at next
	some user information		information required from the user on the File Client's site

a. SessionList is a class that implements a list of sessions, whereas each element has the flow type information, the port of the session and the session's name as well as a pointer to the next session in the list and the connection information.

The class of the user session itself is an element of a list of user sessions. The list is provided in the class `UserSessionList` and administrated by the class `UserSessionAdm.`

Table 33. The Functionality of the User Sessions Administration Class

	name	type	description
public	UserSessionAdm	constructor	
	add_session	method	adds one UserSession to list
	delete_session	method	discards one UserSession from list parameter: user session id: to identify the session to be deleted
	get_first_session	method	gets pointer to first UserSession
	get_next_session	method	gets pointer to current UserSession, current points to the next session in the list after the function
	set_current_to_first	method	sets current pointer to first UserSession
	~UserSessionAdm	destructor	
protected	session_list	UserSessionList	pointer to a list of user sessions
	current	UserSessionList	pointer to the session to be retrieved from the list at next

A graphical user interface shall be provided to give the File Server service provider easily access to the user session administration functionality of the File Server.

8.15.2 Port Administration

The port administration administrates a list of free and used ports. The implementation of this class is based on the assumption that the File Server obtains an input file specifying free ports it can use for its connections. These ports then are assigned exclusively to the File Server during its whole run time. The port administration class `PortAdmin` provides the following functionality

Table 34. The Class for the Port Administration in the File Server

	name	**type**	**description**
public	PortAdmin	constructor	parameter: char *file_name: this parameter specifies the name of the input file containing all ports that are assigned to the File Server. All these ports will be written to the list of free ports
	get_free_port	method	return value: name the function returns the name of a free port
	free_port	method	return value: void the functions adds a port to the list of free ports parameter: Str port_name: this parameter denotes the name of the port to be freed
	~PortAdmin	destructor	
protected	free_ports	PortList	list of free ports
	used_ports	PortList	list of used ports

8.15.3 Connection Establishment and Closing

The processing of the connection establishment for a user session is presented by the state table depicted in Table 36, whereas the legend is included in Table 35.

Table 35. Legend

name	**meaning**
ECS	entry and configuration session
U	the actual user session in an entry and configuration session
U_i	one user session in the user session administration
StC	session to connect; the File Server has sent the session entry point to the client, the session is listening and waiting for the File Client to connect
EC	entry and configuration state
CS	connected session: one already connected session within a user session
SEP	session entry point

Table 36. State Table for the File Server (Without Error Handling)

main state	**incoming event**	**condition**	**performed action**	**next state**
START			create ECS connect ECS to listen	WAITING

Table 36. State Table for the File Server (Without Error Handling)

main state	incoming event	condition	performed action	next state
WAITING	ECS connected		instantiate class: U = new (UserSession) start ECS timer	EC
	time-out of StC_j of U_i		disconnect sessions in U_i stop all timers for sessions in U_i call `free_port` for the ports of the sessions in U_i call destructor of all sessions in U_i remove U_i from user session administration call destructor of U_i	WAITING
	connect of StC_j in U_i		mark StC_j as connected in U_i stop StC timer	WAITING
	disconnect CS in U_i		disconnect all sessions in U_i stop all timers for sessions in U_i call `free_port` for the ports of the sessions in U_i call destructor for all sessions in U_i remove U_i from user session administration call destructor of U_i	WAITING
	shut down File Server		for all sessions U_i do: disconnect all sessions in U_i stop all timers for sessions in U_i call `free_port` for the ports of the sessions in U_i call destructor for all sessions in U_i remove U_i from user session administration call destructor of U_i	END

Table 36. State Table for the File Server (Without Error Handling)

main state	incoming event	condition	performed action	next state
EC	? configuration file of session	`get_free_port` returns port name (free port available)	create session StC_j with configuration file add StC_j to U start timer for StC_j connect session StC_j to Listen send SEP to client	EC
		`get_free_port` returns NULL (no free port available)	disconnect all sessions in U stop all timers for the sessions in U call destructor for all sessions in U disconnect ECS call destructor of ECS create ECS connect ECS to listen	WAITING
	disconnect of ECS		add U to user session administration call destructor of ECS create ECS connect ECS to listen	WAITING
	time-out of StC_j of U_i		unconnected sessions in U_i stop all timers for sessions in U_i call `free_port` for the ports of the sessions in U_i call destructor of all sessions in U_i call delete_session for U_i	EC
	connect of StC_j in U_i		mark StC_j as connected in U_i stop timer for StC_j	EC
	disconnect CS in U_i		disconnect all sessions in U_i stop all timers for sessions in U_i call free_port for the ports of the sessions in U_i call destructor for all sessions in U_i remove U_i from user session adminis tration call destructor of U_i	EC
	shut down File Server		for all sessions U_i do: disconnect all sessions in U_i stop all timers for sessions in U_i call `free_port` for the ports of the sessions in U_i call destructor for all sessions in U_i remove U_i from user session adminis tration call destructor of U_i	END

8.15.4 Error Cases in the File Server

Table 36 does not contain the state transitions if any error occurred. There may occur errors in each constructor, destructor or connection routine which will have the following effects:

- error in the *constructor of session S*:

  if it is the constructor of ECS the File Server has to be shut down (after sending a warning message to the clients), in case of any other session constructor[1], the user session S belongs to, has to be closed (disconnecting of all sessions, call of all destructors, call of destructor of user session)

- error in the *constructor of a user session*:

  fatal, the server has to be shut down
- error in a *destructor*:

  fatal, the server has to be shut down
- error in the *connection of a session S*:

  if it is the ECS => fatal, server has to shut down

  in case of any other session, close the user session the session belongs to
- error in the *disconnection of a session*:

  fatal, server has to be shut down
- error in the *sending of the SEP*:

  the ECS has to be closed and all sessions in U have to be disconnected and destructed

8.15.5 Validation of the Design

The Da CaPo++ File Server has not been implemented as described above. The port administration as designed was not suitable for Da CaPo++. In Da CaPo++, the Connection Manager of every session occupies two Da CaPo++ ports. For each flow two more ports are allocated, one for the communication in each direction. The port administration must be extended to support different numbers of ports allocated per session and, thus, also per user session.

The complex and very general state machine for the entry and configuration session of the File Server has not been implemented as described her.

The prototypical implementation of the Da CaPo++ File Server, that has been realized with the processes Multimedia Server and Multimedia Sender is described in Section 8.16.

8.16 Implementation of the Da CaPo++ Multimedia Server and the Da CaPo++ Multimedia Sender

As mentioned above, the Da CaPo++ File Server has been prototypical implemented in two processes, the Multimedia Server and the Multimedia Sender. The Multimedia Server provides the entry session for Da CaPo++ File Clients and controls the different File Client applications. For each application, one Multimedia Sender is started. The Multimedia Sever and Sender concept is well-suited for multiple applications served by the same sever. Its implementation is based on the fact, that for the Video Viewer application as well as for the Audio Player application one session with only on flow is instantiated. In the following paragraphs, the two processes will be described.

8.16.1 Da CaPo++ Multimedia Server

The Da CaPo++ Multimedia Sever serves as entry point for multimedia applications and as control interface. It creates the entry and configuration session as a multicast session and listens to clients. The session configuration file for the entry and configuration session is standardized and the counter part to the configuration file of all File Clients.

The Da CaPo++ File Client connects to the entry session of the Da CaPo++ Multimedia Server. As already described in Section 8.10.3, the Da CaPo++ File Client sends a *SESSIONFILE* packet as out-of-band packet to the Multimedia Server. The Video Viewer sends the flag *VIDEO* and the Audio Player sends the flag *AUDIO* with the file. The Multimedia Server is currently only implemented to support

1. In this case the server is in State EC

these two applications. As both application just consist of one session with one flow, the port “administration” in the Multimedia Server has been simplified a lot.

8.16.1.1 The Port Administration

Every application that is supported by the current Multimedia Server consists of one single flow in one session. For this reason, every application needs 4 ports. 2 ports for the Connection Manager and 2 more ports for the flow. For this reason, every application gets one port as first port it can use from the Multimedia Server. The Multimedia Server just has a counter for the number *n* of applications ports have been assigned to so far. It then calculates the first port to assign to a new application as $y = 4*n + 20$. The port name for the application is thus `dacapoy` where y is the previously counted number. The first port that is used for multimedia application is thus `dacapo20`. `dacapo0` to `dacapo3` are used for the entry and configuration session and `dacapo6` to `dacapo12` are used for the demon session and, thus, already reserved.

In the prototypical implementation of the Da CaPo++ Multimedia Server, the ports assigned to for applications are not re-used.

Besides unique port numbers, applications also need unique group addresses.

8.16.1.2 The Group Address Administration

Like the port administration, the group address administration also has been very much simplified. Each multimedia application gets its own group address assigned to. This is done by simply adding 1 to the last of the 4 parts of the group address. As the group address `224.1.1.1` is the group address for the entry and configuration session of the Multimedia Server and as `224.1.1.99` is the group address for the demon session in the Video Demon, the Multimedia Server in its implementation can thus assign 97 different group addresses to the applications without any risk.

8.16.1.3 Setup of a New Multimedia Application

When the Multimedia Server has received a *SESSIONFILE* packet, it saves the session file on the `/tmp` disk. The sessionfile gets a unique name that is specified with the process id of the server together with the number of clients connected so far.[1] The server then determines the port number for the application as well as the group address. An *ADDRESS* packet with its own host name, the determined port name as well as the group address is sent to the File Client. The packet gets the process id as well as the host name of the client as identification. These two values were taken from the *SESSIONFILE* packet sent by the client. The server forks itself. The child process starts the Multimedia Sender with the own host name, the port name, the group address, the remote interface extracted from the *SESSIONFIEL* packet and the name of the session configuration file as arguments. The parent process adds the process id of the currently started Multimedia Sender together with the application flag (*AUDIO* and *VIDEO*) and the client’s host name and process id in a simple list. The list definition is in file `appl_PidAdm.H` and the list implementation in `appl_PidAdm.cc`. The File Client, after having received the *ADDRESS* packet, connects to the Multimedia Sender.

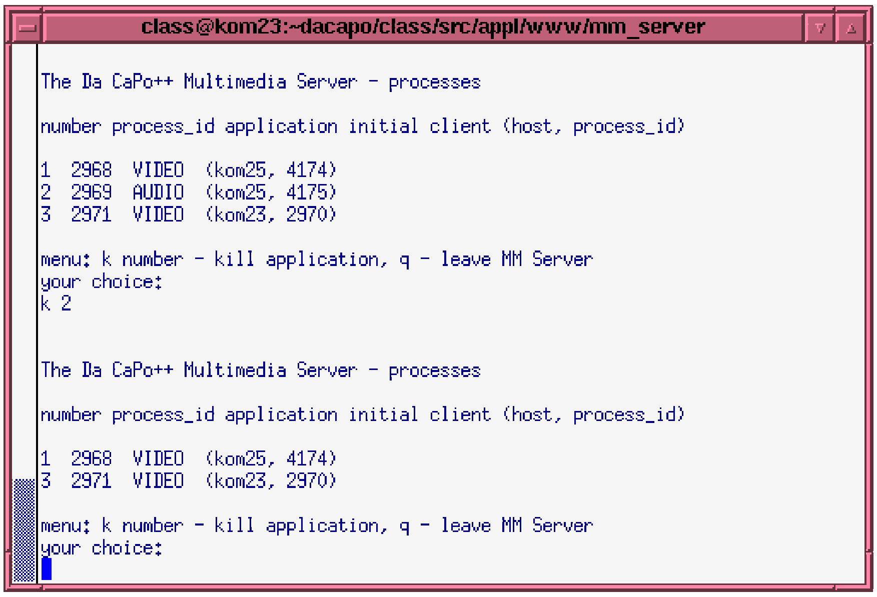
8.16.1.4 Multimedia Server Control Interface

The Multimedia Server control interface is ASCII based, simple and can be seen in Figure 32.

It shows a list of the current active Multimedia Sender processes on the machine. Those processes are identified by a sequence number, the process id of the sender, the application flag as well as the host and the process id of the File Client of this application. The “user” of the Multimedia Server[2] has the possibility to shut down the server which closes all applications as well as to stop specific applications. The interface fulfills monitoring functions in the way, that the service provider can observe which applications are currently running. The user interface and the stored information on the applications is very

1. This must be subject to change if multiple clients are to be supported.

2. In general this is the service provider of the multimedia data applications.



```
class@kom23:~dacapo/class/src/appl/www/mm_server

The Da CaPo++ Multimedia Server - processes

number process_id application initial client (host, process_id)

1  2968  VIDEO  (kom25, 4174)
2  2969  AUDIO  (kom25, 4175)
3  2971  VIDEO  (kom23, 2970)

menu: k number - kill application, q - leave MM Server
your choice:
k 2


The Da CaPo++ Multimedia Server - processes

number process_id application initial client (host, process_id)

1  2968  VIDEO  (kom25, 4174)
3  2971  VIDEO  (kom23, 2970)

menu: k number - kill application, q - leave MM Server
your choice:
```

Figure 32. The User Interface of the Multimedia Server

rudimentary at the moment. The information could easily be extended by information from the Security Manager like the user id of the user of the File Client. The use of multimedia applications could be kept track on a user level in the Multimedia Sever for charging purposes and applications could be stopped if the user has reached its credit limit, e.g. The user interface as provided in the prototype just shows the monitoring and control facilities that are possible in the Multimedia Server application.

As mentioned above, the service provider has the possibility to quit the whole Multimedia Server application as well as to stop certain multimedia applications, i.e. to stop the corresponding Multimedia Sender processes. To do so properly, the affected File Client must get information that the sender shuts down to properly close its session and the Da CaPo++ client.

8.16.1.5 Shutdown of Multimedia Senders

To inform the File Client of a Multimedia Sender process that will be terminated by the server, there exist two possibilities. The Multimedia Sender could send a shutdown message to the client in form of a control message that is transmitted by the data flow. This would result in an application specific, data type independent control message that is transmitted by and can be observed in the A-Modules. The code to forward this control information to the application would to be duplicated in all relevant A-Modules. This would run contrary to design principles that were followed in Da CaPo++. The second possibility is to use the entry and configuration session to also transmit shutdown information. The sender has to inform the receiver so that it properly closes down its session.

The sender gets *leave* events for every receiver that leaves the dynamic multicast session. It could keep a list and automatically shut down if the last receiver has made a leave. The server should be informed, too. To do so, the sender would have to send a signal to the server process. This was not implemented. The shutdown is automatically performed on the sending side, if the initial receiver, the first File Client, stops the application. It sends a *LEAVE* packet to the server. Parameters of this packet are the host name of the client's machine and the process id of the File Client. The server compares the name and the process id with its list and if there is an application process that was initiated by this client process, a *SIGTERM* signal is sent to the corresponding Multimedia Sender.

In case the service provider decides to stop one current Multimedia Sender application, the server sends the *SIGTERM* signal to the sender process and sends a *LEAVING* packet indicating the initial client's host name and process id via the entry and configuration session. The clients compare the information in the packet with their own parameters and only the affected client closes its session.

If the Multimedia Server is completely shut down, the *SIGTERM* signal is sent to all Multimedia Sender processes in the list of the Multimedia Server and *LEAVING* packets are sent to the clients for all multimedia sending processes the server has controlled.

8.16.1.6 Future Work

The prototypical design of the Multimedia Server leads to problems in case multiple clients connect via multicasting to the same multimedia data session. If this is not initiated by the first Video Viewer with the application sharing scenario, neither the first Video Viewer nor the Multimedia Server know of the second client. The second client itself has no knowledge[1] of the host name and the process id of the initiating client. For this reason the second client would not be shut down properly in case of a shut down of the sender. Also the first client would completely stop the sending process by sending a *LEAVE* packet in the entry and configuration session of the Multimedia Server which, then, stops the sending process even if there is another client receiving the data. This problem can only be solved by exchanging control information between sender and server or by controlling the shutdown by the sender process not by the server process.

The Multimedia Server could also be on a different machine as the Multimedia Sender. To do so a concept of a *Sender Demon* similar to the concept of a Video Demon used in the application sharing scenario could be used to start and stop processes on a remote machine. If this was implemented, the Multimedia Server would serve to do load-balancing between different possible senders. It would have the role of a *Metaserver.*

8.16.2 Da CaPo++ Multimedia Sender

The Da CaPo++ Multimedia Sender is the process that creates a multimedia multicast session with one flow. Its command line parameters are included in Table 37.

Table 37. Command Line Arguments in the Multimedia Sender

argument number	name	meaning
1	mm_sender	programme name
2	peer_name	name of the initial client
3	port_id	name of the Da CaPo++ port to use as first port
4	group_address	name of the multicast group address for the session
5	remote_interface	remote interface flag to use (to determine between Ethernet and ATM)
6	file_name	name of the session configuration file locally stored by the Multimedia Server

The Da CaPo++ Multimedia Sender is a very simple program that just instantiates the session with the given configuration file and the specified address information. It also implements a signal handler to properly close the session in case of receiving a *SIGTERM* signal.

1. and probably should not have any knowledge due to privacy issues.

9. Synchronization of Data Flows

The term *session* was once introduced in the Da CaPo++ project to group flows, especially for synchronization purpose. It was decided to implement a synchronization feature in Da CaPo++. For a detailed description refer to the diploma thesis of M. Bamert.

The synchronization of data flows has not been restricted to flows belonging to the same session. Instead modifications in the A-Modules as well as a Synchronization Control Component above the Upper API have been proposed to enable synchronization of flows belonging to different sessions and possibly being sent from different senders. Data flows that have to be synchronized are grouped into a *synchronization group*.

The synchronization is realized by guaranteeing intra-stream synchronization within the data flows per respect to the global time and guaranteeing that all synchronization relevant control commands as PLAY, FF, STOP etc. are always performed at the same time point within all data flows. With this, inter-stream synchronization is guaranteed, too.

9.1 Assumptions

The synchronization approach is *receiver controlled*. The diploma thesis is based on several assumptions:

- all data packets have time-stamps. The time stamps may be the sending time in the sender. In this case the receiver can compute the presentation time by adding the maximum delay between sender and receiver. The time stamp can also indicate the optimal presentation time in the receiver.
- all machines share a global time. This is a very strict assumption. In a local environment we can assume this with a certain deviation. If this deviation is less than 40 ms, the synchronization will be acceptable.
- a maximum end-to-end delay for control packets (out-of-band communication) can be guaranteed.
- a maximum end-to-end delay for data packets is guaranteed. The lase two assumptions are based on the facilities offered by Da CaPo++.

9.2 Architecture and Design

The architectural scheme of synchronization components in Da CaPo++ is given in Figure 33.

The relevant components are shortly explained in the following paragraphs.

9.2.1 Applications

A Da CaPo++ application component, application or application scenario that wants to use the implemented synchronization facility, must instantiate the class implementing the *Synchronization Control Component*. It must pass the information of the flows to be synchronized and of the sessions, the flows are part of, to the Synchronization Control Component. This component also needs information on the application requirements in what concerns the maximum delay. Applications without a guaranteed maximum delay cannot use the synchronization facility.

9.2.2 Synchronization Control Component

The Synchronization Control Component controls the synchronization of the data flows. The synchronization itself as the presentation of data flows, has to happen in the A-Modules to be processed in real time. To avoid IPC communication and its delay, the communication between the Synchronization Control Component and the A-Modules must be minimized. For this reason, there are two phases for the synchronization. The *initialization phase* and the *synchronization phase*. The Synchronization Control Component needs to compute its synchronization delay for all commands. This delay is

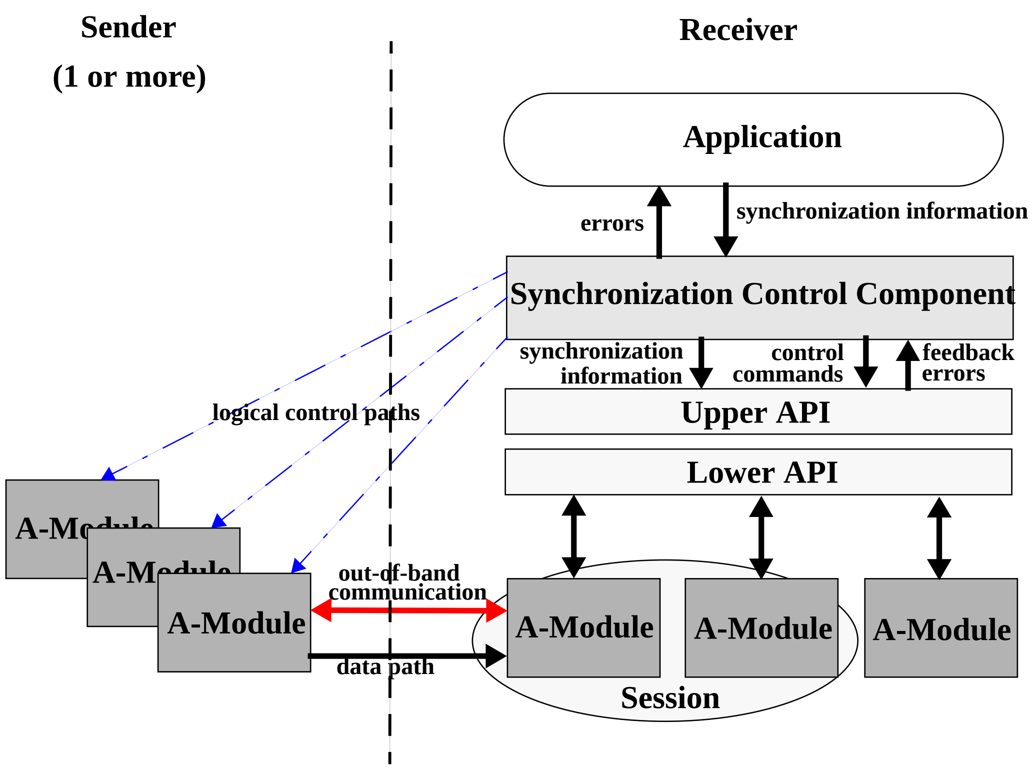


Figure 33. The Relevant Components for the Synchronization of Data Flows

$3 \cdot \Delta_{max} + 3 \cdot t_{IPC} + t_A$

whereas Δ_{max} denotes the maximum delay of out-of-band communication of all data flows in the synchronization group, t_{IPC} denotes the time for the inter process communication in the sender and t_A denotes the processing time in the A-Modules.

All control commands must be sent from the application to the Synchronization Control Component. The Synchronization Control Component determines whether the control command is a synchronization relevant command, i.e. a command that has to be performed for all data flows simultaneously like PLAY or FF, or if the command just refers to one single stream, like “widen the video frame”. The case of non-synchronization relevant control commands is simply treated by forwarding the command to the affected A-Module. This case is not further mentioned here. If the command is synchronization relevant, the Synchronization Control Component forwards it to all A-Module in the synchronization group indicating the target time, that is the current time plus the synchronization delay. The synchronization delay as computed above is large enough to allow for sending the control command to all sending A-Modules, receiving error messages in the receiver, in case a sending A-Modules cannot perform the command, and stopping the whole command in all A-Modules, if one cannot perform it. If no error occurs, the control command will be performed in the senders’ side and all the data flows remain synchronized.

Thus, for all synchronization relevant control commands, the Synchronization Control Component computes the target time for the command and sends the command together with the target time to all A-Modules belonging to flows in the synchronization group. If an error message is sent to the Synchronization Control Component, the control command cannot be performed while remaining in the synchronized state. For this reason, the Synchronization Control Component generates an error message for the application and stops the performance of the control command in all date flows.

9.2.3 A-Modules

The receiving A-Modules receive the synchronization relevant control command together with the target time. The command has to be sent to the sending A-Module together with the target time. The sending A-Module has to control if it can properly perform the demanded control command. If not, an error message must be generated and sent to the receiving A-Module which directly forwards this error message to the Synchronization Control Component. If at least one sending A-Module has generated an error message, the Synchronization Control Component sends a *stop* message to all A-Modules. The A-Modules have to send the stop message to the sending A-Module. Thus, the former received control command is canceled. If the sending A-Modules do not receive a stop message, they have to start the control command at the target time minus the maximum delay specified for the data connection of this specific flow.

All data received data packets are delayed in the receiving A-Module until the optimum playout time (target time for this packet) is reached or until the maximum delay has been passed after the sending time of the packet. If a packet is lost or is received late, the A-Module sends an error message to the Synchronization Control Component which forwards this message to the application.

9.3 Integration into Da CaPo++

The synchronization as described above has been prototypically implemented by M. Bamert. It is integrated into the Da CaPo++ system, but commented out. Tests have shown, that the overhead produced by the synchronization feature leads to a quality decrease compared with a non synchronized application. It is not yet fully investigated if this is due to thread changes, buffer management questions, or due to the fact that Da CaPo++ is not supported by a real-time environment, especially a real time scheduler. If timer routines called by periodic timers cannot be performed at once, this may lead to “late arrived” packets that are thrown away.

10. Da CaPo++ References

/Bame97/ M. Bamert: "Synchronisation multimedialer Datenströme"; Diploma Thesis, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, February 1997, 43 pages.

/BCCC96a/ D. Bauer, G. Caronni, C. Class, C. Conrad, B. Stiller, M. Vogt: "Architectural Design: KWF–Da CaPo++ – Project"; Internal Project Report, ETH-AD-97, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, February 20, 1996, 40 pages.

/BCCC96b/ D. Bauer, G. Caronni, C. Class, C. Conrad, B. Stiller, M. Waldvogel: "Detailed Design: KWF–Da CaPo++ – Project"; Internal Project Report, ETH-DD-21, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, July 20, 1996, 80 pages.

/Brei96/ O. Breig: "Firmenberatung in der Virtuellen Bank – Prototypische Implementierung einer kundenspezifischen Produktgestaltung"; Diploma Thesis, Fachhochschule Furtwangen, Fachbereich Wirtschaftsinformatik, Germany, August 1996, 141 pages.

/BSPl96/ 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 1996, pp 212–221.

/Büch97/ M. Büchi: "Entwurf und Implementierung einer Multicastunterstützung für Multimedia-Anwendungen"; Diploma Thesis, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, March 1997, 60 pages.

/CCCS97/ G. Caronni, C. Class, C. Conrad, B. Stiller, M. Waldvogel: "Implementation Documentation: KTI–Da CaPo++ – Project"; Internal Project Report, ETH-DO-25, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, July 25, 1997, 96 pages.

/CoSt97a/ C. Conrad, B. Stiller: "A QoS-based Application Programming Interface for Communication Middleware"; Broadband Networking Technologies – SPIE Symposium Vol. 3233 on Voice, Video, and Data Communications, Dallas, Texas, U.S.A., November 1997.

/CoSt97b/ C. Conrad, B. Stiller: "The Design of an Application Programming Interface for QoS-based Multimedia Middleware"; 22nd IEEE Conference on Local Computer Networks, Minneapolis, Minnesota, U.S.A., November 1997.

/KGGu96/ A. Karduck, A. Geiser, T. Gutekunst: "Multimedia Technology in Banking"; IEEE Multimedia, Vol. 4, No. 3, Winter 1996, pp. 82 - 86.

/GuRü96/ T. Gutekunst, E. Rütsche: "KWF-Projekt Da CaPo++: Anwendungsszenarien"; Internal Project Report, SBV Basel/XMIT Zürich, Switzerland, January 1996, 9 pages.

/Heng96/ U. Hengartner: "Verwaltung von kryptographischen Schlüsseln und Trust Relations"; Student Thesis, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, February 1996, 51 pages.

/Nigg96/ G. Nigg: "Multicastmodule für Da CaPo"; Diploma Thesis, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, March 1996, 54 pages.

/SBCC97a/ B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel: "Da CaPo++ – Communication Support for Distributed Applications –"; TIK-Report No. 25, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, January 1997, 18 pages.

/SBCC97b/ B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel: "Project Da CaPo++, Volume I: Architectural and Detailed Design"; TIK-Report No. 28, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, July 1997, 144 pages.

/SBCC97c/ B. Stiller, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: "Project Da CaPo++, Volume II: Implementation Documentation"; TIK-Report No. 29, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, August 1997, 99 pages.

/SBCC97c/ B. Stiller, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: "Project Da CaPo++, Volume III: Evaluation"; expected TIK-Report, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, expected October 1997.

/Stil96/ B. Stiller: "An Application Framework for the Da CaPo++ Project"; 5th Open Workshop for High-Speed Networks, ENST, Paris, France, March 20-21, 1996, pp 4-17 – 4-27.

/Stra96/ D. Straulino: "Sprache zur Definition von Kommunikationsparametern"; Student Thesis, Computer Engineering and Networks Laboratory, TIK, ETH Zürich, Switzerland, July 1996, 55 pages.