Also publ. as: Computer Engineering and Networks Laboratory (TIK),
Swiss Federal Institute of Technology Zürich (ETH), 1997, TIK-Report No. 28

Project Da CaPo++, Volume I: Architectural and Detailed Design

TIK-Report No. 28

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

General Information

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, *e.g.*, 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.

This report is structured in two separate, but interconnected parts.

Part I, the “Architectural Design”, presents in the beginning the set of ideas and their architectural design for various aspects. It dates from February 1996 and determines this status of the project. Within the following 44 pages many important issues are discussed.

Part II, the “Detailed Design”, presents the deatiled design goals achieved for the Da CaPo++ project. Its content determines the project status in July 1996 in the next 84 pages. The structure of both parts is identical and shown by two separate tables of contents. However, part II discusses many refinements of initially stated design issues, while referring to the architectural design once in a while.

This page has been left blank intentionally.

Also publ. as: Computer Engineering and Networks Laboratory (TIK),
Swiss Federal Institute of Technology Zürich (ETH), 1997, TIK-Report No. 28

Architectural Design: KWF – Da CaPo++ – Project

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

1. Introduction and Goals

The research project KWF–Da CaPo++ is based on the project Da CaPo (Dynamic Configuration of Protocols) at the ETH. The extended system of Da CaPo++ shall provide an application framework for, *e.g.*, 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.

One main goal for Da CaPo++ includes the provision of a real-life application framework. A variety of different applications has to be managed modularly. Therefore, the selection of special services, application components, applications, and application scenarios results in the transparent handling of communication relevant tasks. Specifically, details on the type of network to be used or the functionality of the applicable communication protocol is hidden completely form the user´s perspective.

Another main goal of the extended Da CaPo++ system is to provide privacy and authentication of transferred data. Therefore, in the banking environment a configurable degree of security is supported. The range of parametrizable security functionality includes varying degrees of authentication and privacy. A set of Quality-of-Service (QoS) parameters, attributes in the Da Capo++ terminology, allows for the specification of various security algorithms to be used or time to live boundaries for cryptographic keys to be specified.

Furthermore, the design of Da CaPo++ is 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), will be supported.

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.

Currently, Da CaPo supports one single application, including multiple protocols for different data streams, *e.g.*, a Picture Phone handles an audio and a video data stream separately by two different logical communication protocols which are represented by four different flows (sending and receiving audio and video) and supported by four different instantiated communication protocols in the Da CaPo core. A specific run-time system, which is located above the standard operating system level, supports on a module basis a variety of tasks. Every module used offers a unique interface, including control and user data manipulations.

The Da CaPo++ core is responsible for handling 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.

The core system is internally structured into eight components. The attribute translation accepts the application requirements and translates them to a structure suitable for the Configuration and Resource Allocation (CoRA). CoRA calculates the appropriate module graph. The module graph is locally instantiated by the data transport component and distributed to peer systems by the connection manager. The security manager validates users and applications and assures that the necessary modules are contained within the module graph. The monitor supervises the execution of the protocols and issues notifications if the application requirements are violated.

1.2 Structure of this Architectural Document

This document contains a discussion of necessary changes and extensions to Da CaPo and describes key functionality that has to be added to different elements of Da CaPo. Furthermore, the application framework is presented, including the number of designed elements, such as application components and applications. 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. Section 5 on page 7 includes the architectural 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.

Section 6 on page 18 contains security aspects. In detail the specification and translation of security requirements is discussed in addition to keying and assurance of security at run-time. The Security Manager as the main Da CaPo core component for dealing with security issues is introduced and discussed.

Furthermore, Section 7 on page 25 covers relevant aspects of multicasting. This is on one hand the adaptation of a Da CaPo core component the Connection Manager to multicast requirements. Additionally, the protocol functionality for handling multicast connection in the transport level is presented.

Finally, the Application Framework of Da CaPo++ is extensively provided in Section 8 on page 32. Applications (Picture Phone, Video Conference, Extended WWW Browser) and application components (File Server, File Client, Multicast Support) are presented. Application scenarios have not been included due to project partner responsibilities.

Due to the architectural design phase of the project, all issues are subject to change in more detail. 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.

2. Table of Contents

3. List of Figures

4. List of Tables

5. Application Programming Interface (API)

This Section intends to introduce the architectural design of the API. It was initially planned to bind the API IPC mechanisms with the buffer management strategy in the Da CaPo++ project, however, this has been separated for the first approach, since the data transport is not intensive, i.e., a special access module cares about large data volumes. All function names, arguments, table fields that are proposed just give hints on how to implement desired functionality and are thus subject to change. Final versions of these information will be given within the detailed design phase.

5.1 Design

In order to make the management of resources easier, it was decided to use only one Da CaPo process on a machine. Thus applications have their own processes and communicate with the Da CaPo server via 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.

The design goals are on the one hand to provide a suitable API that can be easily used by an application programmer, and on the other hand to design it in an efficient way, with special care to all classical culprits that are unnecessary data copying and system calls. This latter goal is bound with the definition of a buffer management strategy in the whole Da CaPo system, from the API to the T-module in case of a sending protocol, and conversely from the T-module to the API for a receiving protocol.

5.2 API Components

The aim of this section is to introduce the main components of the Application Programming Interface and their interactions with both applications and Da CaPo kernel system.

The components are illustrated on Figure 1 on page 7 and are further described in the following subsections. First the upper API with the object model to define abstractions for Da CaPo core objects. Then the lower API whose main task is to manage the control of several applications with the Da CaPo system. The IPC mechanism between upper and lower APIs is also considered. The A module is then closer examined and, finally, an example on how to set up a connection in Da CaPo is illustrated.

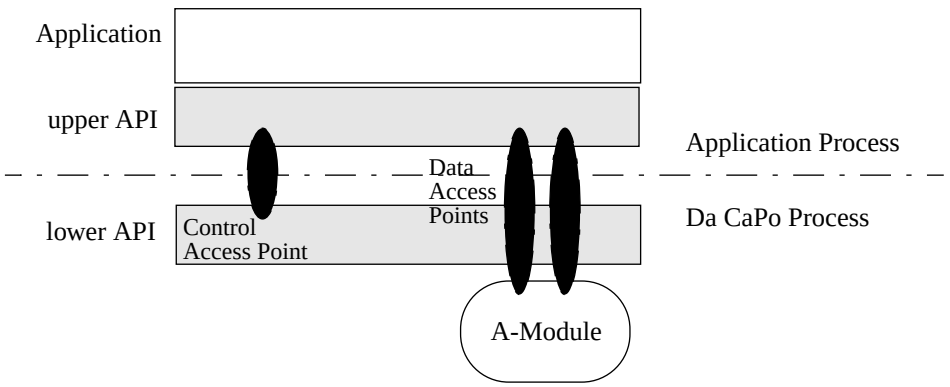
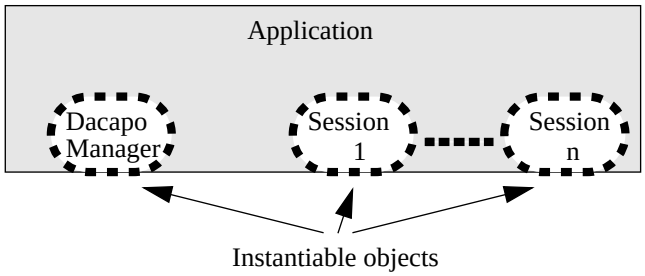


Figure 1API's main components

5.3 Upper API

The upper API part is linked to the application and represents exactly what the application sees from Da CaPo kernel system. To meet first design goal (use of abstractions to facilitate developer's job) and to gain experience in object-oriented technology, applications and thus upper API are written in C++ (the Da CaPo kernel system remains pure C code). Both following sections introduce the most important

objects that build the object-oriented interface to the Da CaPo core system, namely the manager objects on the one hand, and the session objects on the other hand (cf Figure 2 on page 8).



NOTE:

In this figure, only the session objects are directly visible to the programmer (cf variant C in Section 5.3.2 on page 8)

Figure 2Upper API Objects

5.3.1 Manager Objects

Before working with Da CaPo, the user has to be authorized. This is done by creating a DacapoManager object which will first set up a control connection to the Da CaPo core system and then be used for sending/receiving data/events between upper and lower API. This manager object will look as follows:

```
class DacapoManager {
     public:
         DacapoManager(“security relevant parameters”);

     private:
         int RegistrationOk;               // intern flag
         SendCtrlDacapo(char *data);
}
```

The constructor security relevant parameters may be either a password, a passphrase, a public or secrete key, a user id. It is left to the application to provide this data when wishing to communicate with Da CaPo core system.

If the user or the application is authorized by the security manager core component to start working with Da CaPo, a positive return value is delivered by the DacapoManager object constructor and the manager object is properly initiated. This instance of the DacapoManager will then have to be transmitted as parameter each time a new session object is created (the sending of data to the Da CaPo core system being not directly accessible to the application programmer).

5.3.2 Session and Flow Objects

Sessions and flows are both abstractions for internal Da CaPo “objects” services and protocol graphs respectively.

From an application view, a flow encompasses both sending/receiving of data/ctrl information to a dedicated A-module. Thus flows come in different flavors as they have to reflect internal properties of protocol graphs. Flow properties can also be decomposed according to the direction (either a sending or receiving protocol graph), to the data type (either audio, video, data or any other user-defined type) and finally to the way data is processed in the A-module (either data is read from a file and then sent over Da CaPo, from a dedicated device such as a camera or a microphone or data is directly generated in the application).

In order to facilitate the management of several flows, the concept of session was also introduced. A session encompasses several flows which may be synchronized (*e.g.*, audio with video). The number of flows in a session is static and must thus be known during initialization of the session. Having the possibility to dynamically add/remove new flows in a session would not bring an increase of the functionality,

and is bound with difficulties with the current implementation of the connection manager component. Moreover, unsolved problems would occur when trying to perform synchronization with already active flows in a session.

Although the complete API object model is not yet available, a set of instantiable flow classes is defined. These basic classes are illustrated in Figure 3 on page 9.

Per data type (audio, video, general data) there is a set of 6 instantiable classes, according to the direction and the origin/destination of data. The programmer always has the possibility of defining his own data types (and thus flow classes), provided he also cares for the corresponding protocol graphs (A-modules included).

It is now clear that these 18 flow classes are to be used by an application programmer. When trying to bind them with the session concept, several policies may be applied. These different policies have no influence in the lower API and in the core system, they just modify the way the application programmer "sees" the flow and session objects in the upper API. Three such policies are now explained (called A, B and C). For each policy, an example of object definition is provided (in a C++-like syntax) and positive/negative points are listed (only relevant parameter, attributes and methods are listed in the examples, in no way there are complete and definitive class definitions.).

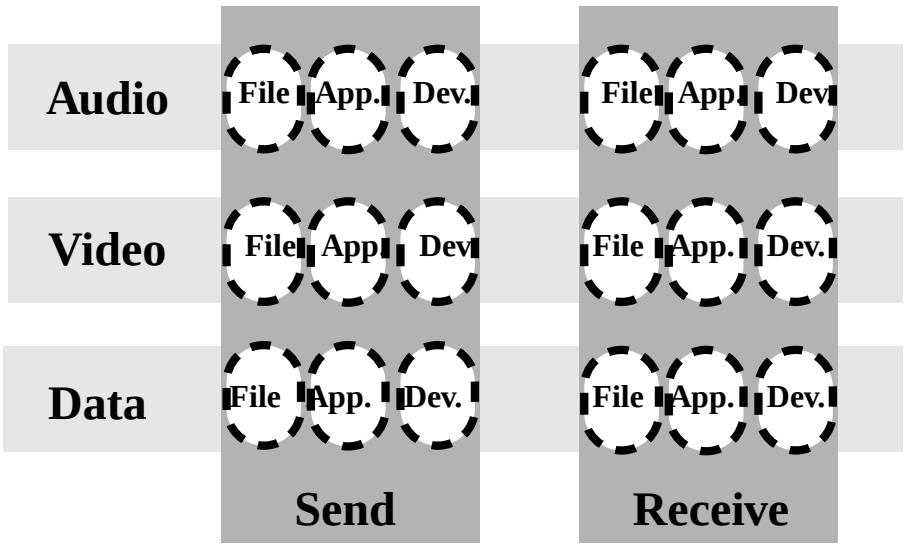


Figure 3Instantiable Flow Classes

5.3.2.1 Variant A:

Static solution.

```
class Session {
     Session();           // Constructor
     VSFlow *VSF;         // Video sending flow
     VRFlow *VRF;         // Video receiving flow
}

class VSFlow {
     VSFlow();            // Constructor for video sending flow
     SendData();
     SendCtrl();
     SetReq();
}
```

TABLE 1. Advantages and Disadvantages for Variant A

Advantages	Disadvantages

* Use of object-oriented paradigm, simple to address flows of the session:
`// ...`
`Session *PP = new Session();`
`// ...`
`PP->VSF->SendData();`
`// ...`
	* Static session definition (not possible to dynamically add new flows in a session)
	* When invoking the constructor of the Session object, the constructors of the flow objects are first executed, involving difficulties for the session setup
	* A session configuration file could only be used for transmitting application requirements, and not to build a whole scenario.

5.3.2.2 Variant B:

In this solution, flows and sessions are separately processed, the information on which flow belongs to which session is done through session-level methods (see appendFlow()).

```
class Session {
     Session(char *SessionName);
     Flow *FlowList;
     appendFlow(char *FlowName);
}

class Flow {
     Flow(char *FlowName);
     SendData();
     SendCtrl();
     SetReq();
}
```

TABLE 2. Advantages and Disadvantages for Variant B

Advantages	Disadvantages
* Managing a dynamic list of flows instead of a static list allows to further add/remove flows to a session.	* Flows can be addressed independently from any session, so it can be tedious for the programmer to remember which flow belongs to which session. * As flows and sessions are treated separately, an object creation process has to be performed for each flow/session object (leading to several function calls in the source code and a huge number of transactions between lower and upper API,). * No configuration file can be provided (except only for the application requirements)

5.3.2.3 Variant C:

In this solution, a configuration file is parsed to create all necessary flows. Each flow can then only be accessed through its identifying string in the configuration file.

```
class Session {
     Session(char *ConfigurationFile);
             // an ASCII file is provided, which contains all flows
             // of a session with their application requirements

     Flow *FlowList;
```

```
            // the following method provides a flow descriptor
            // to access the flow in a faster way (optimization)
        int GetFlowDescriptor(char *FlowName);

            // the following methods were initially in the flow object
        SetReqFlow(int FlowDescriptor, "QoS Value");
        SendDataFlow(int FlowDescriptor, ...);
        SendCtrlFlow(int FlowDescriptor, ...);
    }

    class Flow {
        // no longer accessible for the programmer
        // the flow methods are directly accessed through the session
        //           object
    }
```

TABLE 3. Advantages and Disadvantages for Variant C

Advantages	Disadvantages
* Managing a dynamic list of flows instead of a static list allows to further add/remove flows of a session. * Well adapted for the use of a "complete" configuration file (for both flows and application requirements). A human readable scenario script language can be defined for specifying the application. * Efficiency, only one transaction is needed between upper and lower APIs. * The correspondence between the flow and its session object is made visible at any time.	* Loss of the nice property of OO programming, where the method of a flow can be directly addressed through both session and flow instances (*e.g.*, PP->VSFlow->SendData()). * Need of a parser in both upper and lower APIs

Due to the static aspects of A, this variant will no longer be considered.

The main difference between variants B and C resides in the way how flow objects are actually accessed. In B, each flow object can be accessed through a variable as in the following piece of code:

```
// start of example program for variant B
// it is assumed the programmer got access right to Da CaPo
// dynamic creation of objects
Session *PicturePhone = new Session("PicturePhone");
Flow *VideoOut = new Flow("VideoOut");
Flow *AudioOut = new Flow("AudioOut");
...
// building of the session (it is assumed only flow can be appended
//                                              at a time)
PicturePhone->appendFlow("VideoOut");
PicturePhone->appendFlow("AudioOut");
...
// setting of application requirements (no weight function here)
VideoOut->setReq("FPS", 10);
...
// data transfer
VideoOut->SendData(...);          // direct access to flow objects
AudioIn->RecvData(...)
...
```

For each flow/session object creation and for each appendFlow() method call, a registration process has to be performed between upper and lower APIs. Therefore, the application has a direct access to the flow and session objects (through *PicturePhone and *VideoOut variables).

In variant C, a configuration file named “PicturePhoneScenario” would look as follows:

```
SESSION PicturePhone;

FLOW VIDEO_SEND_DEVICE VideoOut;
     FPS 10;
     DELAY 0.1;
     COLOR NO;
     ...
FLOW AUDIO_SEND_DEVICE AudioOut;
     SAMPLING 32;
     DELAY 0.1;
     ...
FLOW AUDIO_RECV_DEVICE AudioIn;
     ...
FLOW VIDEO_RECV_DEVICE VideoIn;
     ...
SYNCHRONIZE VideoOut WITH AudioOut;
END PicturePhone;
```

A code fragment using variant C is now presented:

```
// start of example program for variant C
// it is assumed the programmer got access right to Da CaPo
// dynamic creation of objects
Session *PicturePhone = new Session(“PicturePhoneScenario”);
         // all objects are instantiated in this constructor call,
         // flows can then only be accessed through their identifiers
         // strings (“VideoOut”, “AudioOut”, ...)
         // application requirements were also transmitted to the lower
         // API where a parser read them from the configuration file
...
// data transfer (flow access through strings and session object)
int FlowDescVideoOut = PicturePhone->GetFlowDescriptor(“VideoOut”);
PicturePhone->SendDataFlow(FlowDescVideoOut, ...);

int FlowDescAudioIn = PicturePhone->GetFlowDescriptor(“AudioIn”);
PicturePhone->RecvCtrlFlow(FlowDescAudioIn, ...);
...
```

Due to its greater flexibility in terms of leaving C-code unchanged, variant C was preferred to variant B, and thus will now be further considered. *E.g.*, variant C offers a better extensibility for new QoS attributes. The functionality of the API is now illustrated in Table 4 on page 12. As it is not planned to use the dynamic add/removal of a flow in the first project phase, this point is not considered in the following table.

TABLE 4. API Functionality

Function	Remarks
Session(char *configuration-File, DacapoManager *mgr);	When creating a new instance of a session, the only parameters to transmit to the object constructor are the application manager object and the contents of the configuration file to set up all flows with their application requirements
ConnectSession(“peerInfo, connManId, reqConnMan, CREATOR|PARTICIPANT, upcFunc”)	Information on the peer, connection manager identifier, conn. man. requirements and CREATOR or PARTICIPANT are necessary to set up a default configuration with the peer, finally, the upcall function is responsible to process incoming events from Da CaPo system.
ConfigureSession()	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.

TABLE 4. API Functionality

Function	Remarks
PauseSession(int stopWay)	All sending A-modules are stopped, meaning that no new data is accepted from the A-module. According to the stopWay parameter (smooth or brutal), the already present data in the graph is either transmitted or the lift is simply stopped. At the receiving side, the A-module simply discards incoming data (without displaying them). A stopped session can be reactivated with a ContinueSession() command.
ContinueSession()	All A-modules (T-modules for receiving flows) are re-activated and start sending data or "displaying" incoming data.
CloseSession(int closeWay)	The session is deallocated (*e.g.*, the connection with peer is destroyed, the protocol graphs resources are returned to the system). It is possible to perform either a graceful or graceless close on the session
GetFlowDescriptor(char *FlowName)	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.
SetReqFlow(int FlowDescriptor, "QoS value")	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
GetReqFlow(int FlowDescriptor, "QoS value")	The GetReqFlow() function returns the actual configured values of required attributes, they may be identical to those set by SetReqFlow().
SendDataFlow(int FlowDescriptor) RecvDataFlow(int FlowDescriptor, funcPtr *upcFunc)	Sending/Receiving of data to/from the corresponding A-module. The upcall function processes incoming data from A-module (this upcall function must only be provided if the data is received up to the application, *e.g.*, for device and file video/audio receiving flows, it is not necessary)
SendCtrlFlow(int FlowDescriptor) RecvCtrlFlow(int FlowDescriptor, funcPtr *upcFunc)	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 has to be processed only from 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.

As mentioned in the above table, the creation of a new session is performed through the C++ session object constructor. When invoking the constructor, new entries are created in the lower API lists (cf Section 5.4 on page 13). The actual data transfer between upper API and Da CaPo is hidden in the Dacapo Manager object (cf Section 5.3.1 on page 8). Unlike the connectSession() function of Table 4 on page 12, no interaction with network is performed when creating a new Session object. The only purpose is to make this object known in both upper and lower APIs.

5.4 Lower API

The purpose of the lower API part is to manage the communications between several applications and a single Da CaPo kernel system. As illustrated in Figure 1 on page 7, there is a control access point in the lower API. This entry point has a well known address and can therefore be addressed by all application processes (features on this IPC mechanism will be considered in Section 5.5 on page 14).

The information coming from the applications (creation of new sessions and flows) is stored in two internal tables (for efficiency goals, these tables are likely to be implemented as lists), namely the session table and the flow table. Da CaPo kernel components can then access these tables (*e.g.*, to retrieve appli-

cation requirements during configuration process or for security purposes). Both tables contain information according to Table 5 on page 14 and Table 6 on page 14.

TABLE 5. Session Table Structure

Field	Description
char *sessionName	Name of the session, only valid in the corresponding application
char *commChannel	Information on how to communicate with the application (done through a communication socket to transmit Da CaPo intern events to the application)
int sessionId	Session identifier (scope on the local Da CaPo system), not visible to the programmer.
int appId	Application identifier (scope on the local Da CaPo system), not visible to the programmer.
int status	Status of the session (connected, ...)
int *flowList	List of all flows belonging to the session.

TABLE 6. Flow Table Structure

Field	Description
char *flowName	Name of the flow, only valid in the corresponding application
char *commChannel	Information on how to communicate with the application (shared memory area and access semaphores used by the A-module) for both data and control exchange.
int flowId	Flow identifier (scope on the whole Da CaPo system), not visible to the programmer.
int sessionId	Session identifier the flow belongs to (only if it belongs to a session, *e.g.* if the flow has already been appended to a session).
int appId	Application identifier (scope on the local Da CaPo system), not visible to the programmer.
int type	Type of the flow (audio, video, data)
char *sync	List of all other flow identifiers it is synchronized to, and information on the kind of synchronization
int status	Status of the flow (configured, modified, ...)
“Protocol graph access”	The modules being part of the corresponding protocol graph are made accessible through this information.
“QOS parameters”	All application requirements (or default values). These are made available for each Da CaPo core component.

As already mentioned the lower API has a component to read and parse the application requirements (QoS parameters) coming from the application. It is intended to provide a script file to Da CaPo, containing the session properties and all flow QoS parameters (thus the programmer does not have to call a huge number of methods to set all attributes, on the other hand, transmitting a single file name is much more efficient than to perform a huge number of IPC calls to set up each requirement separately). This data will then be parsed, and in case of an error, a message is sent to the application.

5.5 IPC between Application and Da CaPo

As illustrated in Figure 4 on page 15, there is a single control access point and several data access points between each application and the Da CaPo kernel system. The control access point is used to set up new sessions, to transmit application requirements, to configure protocols, to close a session, whereas data access points are related to both data and control information to the A-modules. Over data access points, each application can either send/receive data or send/receive control information, this control information being only relevant for the corresponding A-module (or its peer). Thus there are different channels between applications and the Da CaPo kernel system:

- a bidirectional communication for control information between application and Da CaPo, such as when the application creates a new session with the corresponding flows or when a Da CaPo internal event has to be transmitted to the application
- a unidirectional data communication for each flow of the session (to send/receive data directly to the A-module)
- a bidirectional control communication for each flow of the session, such as to resize a video window or to get information from the A-module

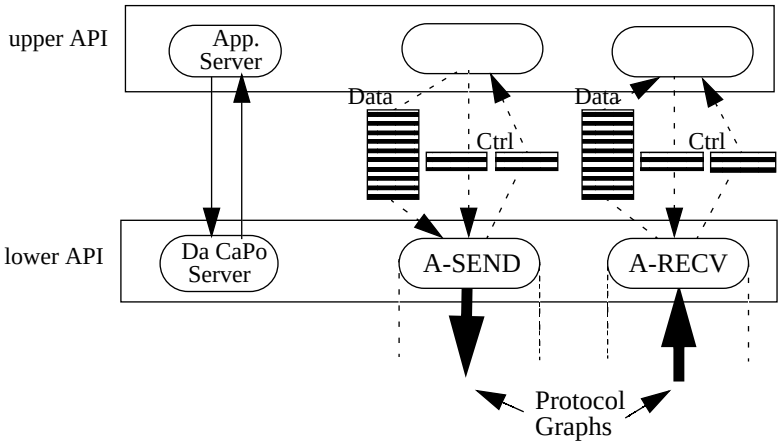


Figure 4Data and Control Interfaces

5.5.1 Data and Control Interfaces

Basically, shared memory with semaphore synchronization (or socket synchronization for the control access point channel) was chosen to implement an efficient IPC mechanism between both parts of the API. It can be seen on Figure 4 on page 15 that the only static interface is the bidirectional communication between the application (application server component) and the Da CaPo system (through the Da CaPo server). The address of the Da CaPo server is well-known on a machine, therefore each application has to go through it to register (for security), to set up the flows and sessions it requires, ... The data access points are dynamically created for each receiving or sending A-module during protocol configuration.

The three different interfaces are implemented in the following way:

- The bidirectional communication for control information between application and Da CaPo.

  To make communication easier, a simple bidirectional socket connection is provided. On the lower API side, a “Da CaPo server” component is provided to deal with incoming data from the applications. On the application side, a similar “Application Server” component is instantiated with an upcall function to process incoming events from Da CaPo (cf connectSession() function of Table 4 on page 12).

- The unidirectional data communication for each flow of the session.

  It consists of a shared memory area, synchronized with 2 semaphores. This memory area is also used for the buffer management in the whole Da CaPo system (further details in Section 5.7 on page 16). Thus the shared memory is organized as a circular buffer, where the packet size is fixed by the current flow (depending on the protocol graph), and the maximum number of packets in the circular buffer has to be determined during configuration (if this size is too small, it is likely that the buffer will always be full, involving failures or waiting times when an application tries to send data in a non-blocking or blocking way respectively, on the other hand, important delays can be encountered if this size is too large).

- The bidirectional control communication for each flow of the session is also implemented with a shared memory and 2 semaphores for each direction. The size of this shared area still has to be determined.

The implementation details on how exactly the IPC mechanisms are set up between upper and lower APIs will be explained in the fine design document (*e.g.*, how the receive of Da CaPo events is actually multiplexed between all active sessions on the control channel, how many threads are necessary in the upper API to wait for either incoming data or control information from all flows, ...).

5.6 A-Module

The A-module is the implementation of the data access points. Thus it has an interface to the application (to get/send control/data). On the Da CaPo side, it has similar interfaces as any other module.

5.6.1 Concurrency for control and data information

At the A-module, data and control information are being received simultaneously from the application. Thus a policy has to be set up to deal with this concurrency. One solution would consist in giving higher priority to control versus data information.

5.6.2 QoS Mapping

Da CaPo offers a set of predefined attributes as throughput, delay, delay-jitter, error-rate, ... These basic attributes are however insufficient to characterize the behavior of some applications. In a video application context, the most significant Da CaPo attribute would be the throughput, though it is likely a programmer (or a user) would rather speak in terms of frames per second, color depth and image size. Actually the effective throughput can be computed by a combination of the three latter values (compression is ignored).

This operation is called QoS mapping and can be performed in each A-module. Each A-module has the necessary mapping functions to translate specific application requirements to its type (*e.g.*, video, audio or data). All original application requirements are stored in the flow table in lower API (cf. Section 5.4 on page 13). It is then the task of the A-module to process this information.

5.7 Buffer Management

As the buffer management will not be part of the first implementation, this section delivers a vague idea initially.

As already mentioned, the circular buffer which is used to transfer data from the application process to the Da CaPo A-modules (or vice versa) is also the core of the memory management in Da CaPo. Storage is thus allocated for each data packet in the upper API (or directly in the application if it is reliable enough) and then released when the packet leaves the T-module (this is valid for the sending direction, in the receiving direction, allocation is performed at the T-module and release in the upper API).

5.7.1 Structure

As illustrated in Figure 5 on page 17, the buffer management consists of a shared memory area. This buffer is structured in cells, which have the maximal size a packet of the corresponding flow can reach (this should be a property of the flow). The initial number of cells in the buffer is set up during protocol configuration, but it can always be adjusted if necessary.

Two additional fields may be useful:

- mark:

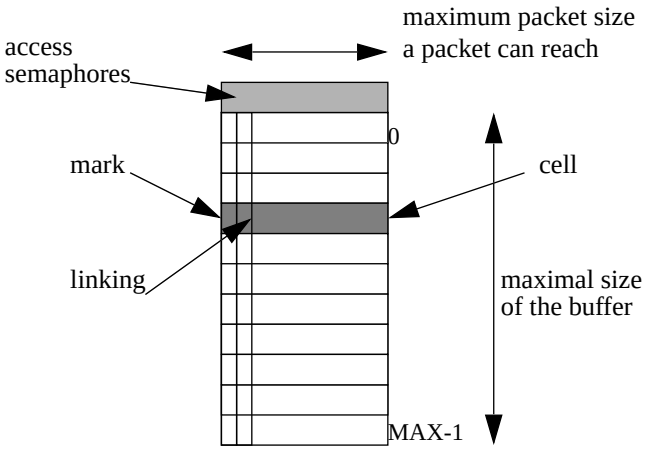


Figure 5Buffer Management Structure

For an IRQ module, it is not possible to deallocate the data before receiving the corresponding acknowledge. In this case, the buffer is marked and cannot be removed when the corresponding data reaches the T-module

- link:

  To perform effective data compression, it may be necessary to gather several packets and to compress then a larger amount of data (compression ratios are likely to be better with more data). In this case, some cells can be linked to form a single large amount of data.

5.7.2 Functionality

The functionality of this buffer management is illustrated in Table 7 on page 17.

TABLE 7. Buffer Management Interface

Function	Description
char *allocateCell(int bufferId)	One single cell is allocated, returns a pointer on this cell
int freeCell(char *cellPtr)	The cell is released and can be allocated again for a new packet
int markCell(char *cellPtr)	The cell is marked, and may thus not be deallocated
int unMarkCell(char *cellPtr)	The cell is unmarked and may thus be deallocated
int linkCells(char **cellPtr)	All specified cells are linked to build a single large packet
int adjustBufSize(int bufferId, int delta)	The size of the buffer is modified

Knowing that buffer may be allocated in the upper API (or application) and in Da CaPo kernel system, it is necessary to provide 2 interfaces for both programming languages C and C++ for some buffer management functions.

6. Security Aspects of Da CaPo++

Securing Da CaPo communications is achieved by defining protocols that include encrypting and authenticating modules. Depending on the security requirements that the application specifies, the configuration process will employ these modules, taking into account security which might be provided by lower level transport infrastructure. A static key and certificate database allows for the application-independent storage and recovery of public keys and related information. The actual control of security in Da CaPo is done by the Security Manager, which consists of several building blocks. This Section does not mention the assumptions related to trust that have been taken to allow for a realistic approach, as they have been already discussed earlier.

6.1 Security Design Overview

'Securing Da CaPo' covers four different areas. First, users have to identify themselves to the Da CaPo core, and have to prove their identity. Second, applications that want to use Da CaPo in a secure fashion have to be identified and authenticated by Da CaPo. Another important area is the machine-machine authentication that allows two Da CaPo endsystems to communicate in an authenticated and secure manner even if no 'security aware' application or end-user is available. Finally, the fourth area covers the actual encryption and authentication of data that is transmitted over an unprotected network infrastructure.

The second and third area may actually be coalesced into one if user authentication is done through the application. Such behavior is not encouraged, as it leads to the necessity of a multitude of 'logins' for the user. The four areas show different behavior depending on whether a delegation of the respective identity to the Da CaPo system takes place. For the sake of simplicity, this is assumed to be the case throughout the following.

6.1.1 Assuring Authenticity: Associations and Identities

Before employing a secure communication system, the participants have to be securely identified and their 'output' must be attributable to them in an easy fashion. This assumption ignores issues like frequently changing identities and desires for anonymity, but is only relevant if authentication is required. In an extreme scenario, all trusted parties that are involved have to be mutually authenticated. These parties consist of the end-systems on which Da CaPo++ is running, the users involved in the communication, and/or the applications actually producing and consuming the data.

In the model employed by the Da CaPo++ communication system these three identities are ordered hierarchically. If no user authenticity can be provided, application authenticity, and failing that, machine authenticity will be provided. The instances participating in the communication can express their minimal requirements, and are notified upon connection establishment with whom they are actually communicating.

Before a communication can actually progress, users and/or applications involved are required to delegate their identities to the Da CaPo core system so that the core can authenticate data on their behalf, and prove their identity to the peer. This mainly consists in giving a secret to Da CaPo++ with which identities can be authenticated. The given secret need not be the secret that was originally employed to prove identities, and may be usable only by Da CaPo for a limited time span and/or for a limited amount of authentications. The typical case (and the one provided) will, nevertheless, be a full delegation.

Public key values and user/application identities are stored in a global key and certificate database, where application identities consist of arbitrary (but structured) strings identifying them, and user identities may consist of a string containing RFC 822 E-mail addresses, bank account numbers, or any other kind of mutually accepted identifying information. Machines are identified by the address on which they are reachable in the transport infrastructure that is used to establish the connection.

The 'association block' in the Security Manager of the Da CaPo++ core system (cf. Section 6.2.6.1 on page 23) will verify identities, and note which protocols are associated with which applications and users, communicating this information to other endsystems, if needed and allowable. The Da CaPo++

user interface (which is used for user authentication in the first place, if not done via individual applications) can be used to force modifications in these associations, *e.g.*, if a user wants to force an immediate dissociation from an application which turned byzantine. It is to be remarked that the user interface is an application like any other, which just holds special knowledge of the inner workings of Da CaPo and communicates with the security manager through the standard API.

6.1.2 Specifying and Translating Security Requirements

To express privacy and authentication requirements, applications have to pass attributes to Da CaPo++. The attributes are hierarchically ordered in a generic sense, and may consist either of discrete values from a set of possibilities, or specify a range of acceptable values. The attributes specifying security requirements are generally handled exactly like any other QoS attribute. This allows to employ the standard attribute passing and protocol configuration mechanisms of Da CaPo for the building of secure protocols (see also Section 6.1.4 on page 19). Special treatment is scarcely needed, *e.g.*, for an attribute containing keying material or connection setup authentication requirements.

The security requirements needed for the configuration of secure protocols span a very wide range. To allow for a more transparent (and algorithm independent) handling in the application, the concept of requirement translation is introduced in Da CaPo++. An application specifying only generic application requirements (AR) will accept the defaults that the translation mechanism concludes as being corresponding concrete QoS parameters (PAR). An application may still specify as many detailed parameters as wanted, but may thus create a set of requirements which the system can not fulfill. In that case, no communication can be established. The results of such a translation depend on the available algorithms and machine power, and on the state of the art in cryptography. If the translation process is kept up to date, and the application uses generic security requirements, they will ‘support’ adequate cryptographic mechanisms not only at the time of creation, but in the future too.

The ‘attribute block’ doing the translation actually resides in the A module of each ‘secure’ protocol, and receives the AR by the way of the lower API, together with the non-security related attributes. As the attributes are not parsed by the API, but passed on transparently, no extension thereof is needed for new attributes.

6.1.3 Protocol Management, Reconfiguration and Keying

A secure protocol, which has been configured, includes modules performing cryptographic operations. These may be of symmetric nature, *e.g.*, DES, IDEA, RC4 for encryption, and MD5, SHA for authentication support, or asymmetric, *e.g.* , RSA, DH or El Gamal. Additionally to traffic encryption and authentication, they will allow for key exchange if rekeying is an issue and may allow for the receipt and processing of tokens providing sender- and receiver nonrepudiation functionality.

The ‘key management block’ (see below) of the security manager provides access to the database containing public and private keys, as the generation of authenticated keying material has been delegated to the communication system. Key changes in the running protocol can thus automatically take place, the ‘protocol control block’ of the Security Manager does ‘asynchronous’ key changes. The only way for an application to change the properties of a secure protocol is to initiate a reconfiguration.

On the other hand, if the application has chosen to provide keying material (as will be possible later), a reconfiguration of the protocol is necessary. This may be a very cheap process (called small reconfiguration), if the change does not require different functionality, *e.g.*, when the application chooses not only to change traffic keys, but likes to change employed algorithms also.

6.1.4 Security Assurance at Runtime

The configuration process provides secure protocols, if working correctly and receiving requirements from the application that do request this. At a later point of this project, the ‘security assurance block’, located within the module instantiation process, can verify the compliance with requirements, by checking precalculated and certified module properties, the integrity of the employed modules themselves, and the validity (in terms of security requirements) of the created protocol. The same holds if an unilateral

reconfiguration downgrades a communication protocol. The communicating peers (Da CaPo++ systems) will be notified of the reconfiguration that took place and when instantiating the new module graph, have to decide if they accept this change. If they can not accept the change under the current requirements, they will have to notify the application to change the requirements or refuse the change.

At runtime, the ‘security assurance block’ monitors the usage of keying material, and takes track on how much data, and for how long a traffic key has been in use. A special event will be issued to the ‘protocol block’ when this happens, and is sent to the application if rekeying is necessary. This is another aspect of runtime security assurance.

6.1.5 Keys and Certificates

The key and certificate database is used to store certificates of modules and their properties, public and private keys of users, applications and machines, and is accessed by the ‘key management block’ to provide keying material to various parties.

6.2 Discussion of Components

Introducing security has an impact on nearly all parts of the Da CaPo core system. This section identifies the parts that will be created/modified, and states coarse assumptions on data structures and points of interaction. The elements are:

- API
- QoS Parameters
- C-Modules
- Protocols
- Key Database
- Security Manager

Additionally, the connection manager may be changed to use a protocol that insures privacy. How this is to be done will have to be evaluated after modules, protocols, and parts of the security manager (and the existing connection manager) have consolidated. These changes will mainly result in the connection manager using a different (secure) module graph.

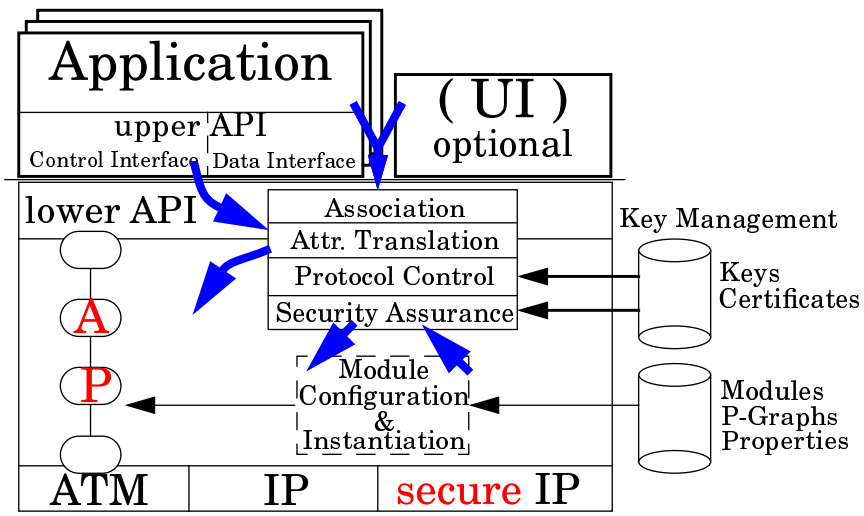


Figure 6Security Architecture in Da CaPo++

The system architecture is presented in Figure 6. Secure IP represents a transport infrastructure that already offers security. Future T modules will have the intelligence needed to understand the existence of such a service, and will thus optimize the protocol configured by the core system.

6.2.1 API

The API will transparently forward application requirements which an A-module can translate into QoS parameters. See Section 6.2.2 on page 21 for a list of ARs. Additionally, the API handles the identification/authentication issues, and provides for a way to forward events to the application. A transparent ‘control channel’ will be available, through which the user interface or other specialized applications may communicate with the security manager, *e.g.,* to access the key manager for the purpose of generating, storing, retrieving, certifying keys and certificates.

The upper API has to process and forward the following information upon establishment of a controlling connection between core and application: local user name, process id, global user name, global application identifier, user key ID and passphrase. This is passed on to the security manager (association block) by the lower API and verified there. Afterwards the lower API receives a clearance or denial from the security manager and acts accordingly. As secure protocols and keys can be defined and changed using the generic (re-)configuration mechanisms, no addition to the API is needed for this purpose.

For end-to-end authentication with non-repudiation, the API offers two sets of functions, which allow this extended security protocol to perform using slightly different semantics. When a flow is created or reconfigured to use a receiver-non-repudiation protocol, a proof of receipt will be generated for each received message. Below the API, a message may be limited by arbitrary bounds, defined by START/STOP pairs in the control flow. For the application level, the concept of a ‘message’ has to be introduced, or, alternatively, synchronous end-to-end authentication/return receipt requests can be initiated by one of the peers. Although continuous media has no fixed boundaries, they can be added artificially by the application, *e.g.,* by requesting a return-receipt after each frame, or each second.

6.2.2 QoS Parameters

Expected application level requirements encompass at least:

- S-Privacy-generic (time*money)
- S-Key-ID (for access by database)
- S-Secret-Enabling-Key

Additionally, all lower layer requirements may be specified directly.

TABLE 8. Parameter for Privacy

Privacy [OFF, 1, 5, 10, 50, 100, max.]			
Precondition: [JPEG | MPEG | H.261 | CellB | ULaw | G.721 | error-free | ordered | none]			
Algorithm: DES, IDEA, 3DES, RC4, RC5, Blowfish, Safer-K			
Keysize, Rekey-Interval, Rekey-Volume			
Blockcipher:	Rounds	Blocksize	ECB, CBC, OFB
Partial:	Space	Time	Control
	Variance		
Postcondition:	Delay increase	CPU x Bandwidth	Blocking factor

They have to be translated into a number of QoS parameters. This subject will be discussed in more detail within the detailed design phase. A preliminary set of information has been compiled in Table 8 on page 21 and Table 9 on page 22.

TABLE 9. Parameter for Authentication

Authentication [OFF, 1, 5, 10, 50, 100, max. | SYM, ASYM, ASYM-R]			
Precondition: [JPEG | MPEG | H.261 | CellB | ULaw | G.721 | error-free | ordered | none]			
Algorithm: MD5, SHA. Tandem-DES, Tandem-IDEA, RSA, El_Gamal			
Keysize, Rekey-Interval, Rekey-Volume			
Blockcipher:	Rounds	Blocksize	ECB, CBC, OFB
Partial:	Space	Time	Control
Transaction	Frame	n Frames	nth Frame
Postcondition:	Delay increase	CPU x Bandwidth	Blocking factor

6.2.3 C-Modules

The provided C modules for introducing security into the data transfer are:

- ECB: Provides for DES, IDEA and RC5 in electronic cookbook mode
- CBC: Same in cipher block chaining mode
- CBC_ORDER: Same, but depends on ordered and lossless data transfer
- RC4: Stream cipher encryption module
- MD: Provides MD4 and MD5 message digest algorithms
- DS: Provides asymmetric (RSA) and symmetric MAC (message authentication code) for the signature of a message
- DH: Provides Diffie-Hellman for establishment of a shared secret at runtime (ephemeral traffic keys), is used by the other cryptographic modules.

They behave like normal Da CaPo modules, although they use advanced features like intra-module communication, with one exception. For the purpose of internal rekeying, and user driven (not application driven) security control, they have an additional interface directly linked with the protocol control block of the security manager, and announce themselves to the association block on initialization.

6.2.4 Protocols

The provided protocols will be:

- Encryption
- Authentication
- Encryption+Authentication
- Encryption+full non-repudiation (sender&receiver)

6.2.5 Key Database

The key database later residing in the GMS (Group Management System), interfaces directly with the Security Manager, respectively its key management block, to provide for public keys upon request, and otherwise keep them in persistent storage. This component is invisible for the rest of the Da CaPo core

system although it will be accessible by the application layer through a transparent channel in upper and lower API, if and when access to the database will be provided to the application. The overall features and behavior of the key database are very similar to PGP.

6.2.6 Security Manager

The concepts behind the Security Manager have been discussed in Section 6.1 on page 18 The functionality can be separated into the following building blocks:

- Association
- Attribute translation
- Protocol control consisting of
  - module rekeying
  - event propagation
  - reconfiguration
  - Key management

The security manager is implemented as a part of the Da CaPo core system which owns its own thread, which may eventually be delegated to lower API. The user interface (connecting through the transparent channel in the API) can induce actions like rekeying, switching security for one particular graph on or off, generally controlling behavior of owned protocols, and will provide for user authentication functionality. The goal is an experimental and prototypical access to the core, to allow for debugging and testing.

6.2.6.1 Association Block

As soon as an application establishes a connection with the Da CaPo core and sets up some secure flows, the security manager needs to know which flow is owned by which application, and for which user the application is running. Flow- and session specific information is collected by the lower API, which passes on a handle to this information and additional authenticating data to the association block. The association block verifies authenticity of the provided information, and allows or denies access. In the case events are generated, keying material is missing or other actions are required, the controlling owner is retrieved via the association block, and the message forwarded via the lower API (cf. the detailed Design Document later).

6.2.6.2 Attribute Translation Block

Attribute translation is only conceptually part of the security manager. It is realized as a set of functions integrated into the A-modules of all security-aware protocols, and needs to understand all application requirements pertaining to the security mechanisms, and which have to be mapped to QoS parameters in this particular protocol.

6.2.6.3 Protocol Control Block

As the name say, the protocol control block handles all security related issues that influence communication behavior. It is the switchboard that receives requests from the cryptographic modules for new keying material, eventually then retrieves key-IDs from the association block, and gives new keying material to the modules.

This may also lead to the generation of an event, if, *e.g.*, the actual key has expired. Other possible events (generated by the protocol itself, or the protocol control block) are

Rekey request / confirm: Request is issued if the keying material is provide by the application at a later point in time. The confirmation is sent always when a rekeying occurs, but may be safely ignored by the application.

Remote Reconfiguration: The remote peer (or the creator in the case of a multicast flow) initiated a reconfiguration, which completed successfully. As the runtime security assurance provides for a control of sufficient security, this should never lead to fatal conditions.

Return-Receipt Request/Confirm: Before actually delivering a return-receipt to the remote Da CaPO system, the application in charge of the particular flow will be asked whether it wishes to give a confirmation. If yes, Return-Receipt Confirm delivers the proof-of receipt to the application originating the data.

Failed Authentication: Issued if received data is not authentic

Key Expired: The certificate in the Da CaPo key database expired, and communication has been stopped, pending announcement of a new key.

As the security part of Da CaPo allows for a ‘small’ reconfiguration (*e.g.*, change of keys, switching security on/off), this is done in the protocol control block. If a change in the status is required, protocol control stops the lift, accesses the involved security C-modules via their announced interface, and changes their behavior. They communicate the change to their receiving peers using intra-module communication. The peers forward the event to the remote protocol control block, and on the sending side the lift is restarted.

6.2.6.4 Key Manager Block

This is the access point to the key database. It provides for fetching keys and certificates from the database, generates random (ephemeral) keying material, and provides it to the C-modules. Although the key manager block currently provides only functionality to the Da CaPo core system, it may be visible to the application via the upper API, and provide for a limited key management functionality. (Retrieval of keys and certificates, storage of new keys, check of signatures, etc.) At a later point in time it will use the GUA, and the key database will disappear.

6.2.7 Runtime Security Assurance

Runtime security assurance will be provided at a later point in time. It will interface with the module instantiation part of Da CaPo, to verify the correctness (authenticity) of modules. Additionally, instantiated in the monitoring part of Da CaPo, it interacts with the module graph to check if the actually achieved security conforms to the local application requirements.

6.3 Open questions:

A number of open questions remains due to an exact use of core system functionality. This will be decided within the detailed design phase, finally.

- Exact use of transferred attributes/requirements in the monitor to assure qos,

  - Re-study events, formalize

  - Give 2-level QoS structure

  - Define names of applications (‘service’ does not equal‘application’)

  - Connection Manager provides privacy. No authentication through DH/RC4 modules, clogging is no issue

- How is dacapo-dacapo authentication done? It will use the Connection Manager interface.
- How to generate ‘random’ keying material?
- Use of security aware T modules.
- Authority is checked by the application.
- What security functions are visible in the upper API?

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.

Da CaPo implements a very basic multicast model where the creator of a multicast flow is the only sender. Participants are therefore always receivers, although a back path is always available. 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 of the protocol and module graph. This means that the protocol configuration is based on the creator's application requirements only.

7.1 Multicast-capable Connection Manager

The Connection Manager (ConMan) guarantees that a compatible protocol in terms of all modules is used inside a Da CaPo++ session for communicating participants. The ConMan triggers local configuration and reconfiguration, distributes the resulting mechanism graph and starts/stops the protocols. The actions taken by the connection manager are initiated either by the local application or by a remote connection manager. There is one ConMan per session.

The protocol stack used by the connection manager is defined statically. There is one protocol stack per transport infrastructure and per session. Four transport infrastructures are planned: UDP, UDP multicasting, ATM, ATM multicasting. Figure 7 on page 25 shows the protocol stack of the connection manager.

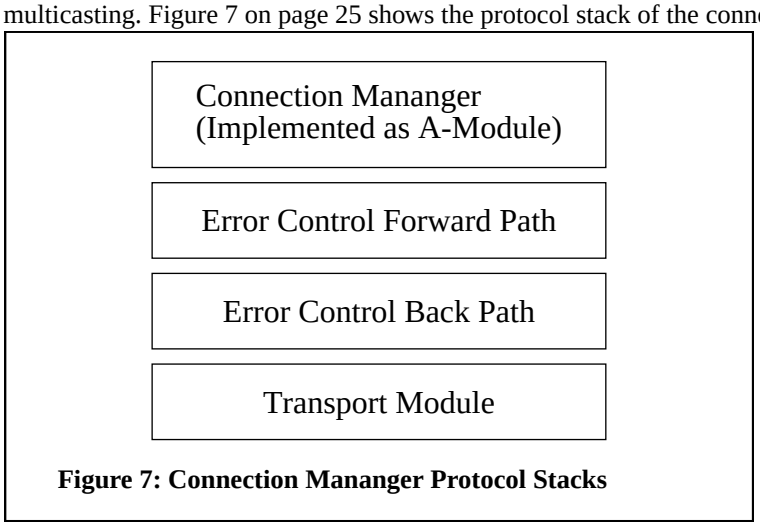


Figure 7: Connection Mananger Protocol Stacks

The control over a multicast session lies solely at the creator. This means that only the application entity that created the session is allowed to reconfigure, start and stop the multicast session. Only local reconfiguration is supported, the application requirements of the participants are ignored, a (re-)configuration is based entirely on the creator's application requirements. Participants are not allowed to reconfigure a multicast session, *e.g.*, an application's reconfigure request has no effect. Start and stop at a participant's site leads to a stop of the local execution of the lift. This means that no data packets are delivered to the participant's application which means that data loss is inevitable. Therefore, the stop operation should only be executed by applications that tolerate data loss like audio or video applications.

The connection manager in the multicast case is built according to the client-server principle. The ConMan of the creator is the server, the ConMans of the participants are clients. The ConMan at the creator distributes the mechanism graphs for the sessions and starts and stops the session. The other ConMans request the mechanism graph and send start/stop events to the creator's ConMan.

7.1.1 The Creator's ConMan

The creator's ConMan receives the following requests and events:

- Application requests:
  - Create
  - Configure
  - Start
  - Stop
  - Destroy
- Events from participants ConMans:
  - Send Graph
  - Remote Start
  - Remote Stop
  - Remote Destroy

The semantics of these requests is shown in Table 10 on page 26.

TABLE 10. Creator's Connection Manager Requests

Create	The application creates the session. For each protocol inside a session, the default modules are initiated. The ConMan of the session is started and initialized.
Configure	The application triggers a local configuration. The configuration is based on local application requirements only. After the configuration is done, the ConMan distributes the resulting mechanism graph to all participants and instantiates the new protocol.
Start	The local lifts are started as soon as at least one participant is available. The ConMan sends a 'remote start event' to all participants.
Stop	The local lifts are stopped. The ConMan sends a 'remote stopped event' to all participants.
Destroy	The local lifts are stopped and all modules are deallocated. The ConMan sends a 'remote destroy event' to all participants.
Send Graph	A new participant wants to join the group. The ConMan answers with the current mechanism graph. Additionally, the ConMan forwards this event to the lower API.
Remote Start	A participant signals that its application has triggered a start event. If this is the first 'remote start event' that the creator obtaines, then it starts its lifts also.
Remote Stop	A participant signals that its application has triggered a stop even. If no participants are running any more, then the ConMan also stops its local lifts.
Remote Destroy	A participant signals that its application has terminated the session. The ConMan informs the lower API that the participant has left. If all participants have left, then the local lifts are stopped, too.

7.1.2 Participants' ConMan

The participant's ConMan handles these events:

- Application events

- Create
- Configure
- Start
- Stop
- Destroy

- Remote events
  - GetGraph
  - Remote Start
  - Remote Stop
  - Remote Destroy

TABLE 11. Participant´s Connection Manager Requests

Create	The application creates the session. For each protocol inside a session, the default modules are initiated. The ConMan of the session is started and initialized. At the same time, the ConMan sends a ‘getgraph event’ to the creator.
Configure	The application triggers a local configuration, which has no effect whatsoever.
Start	The local lifts are started if the creator has also started. The ConMan sends a ‘remote start event’ to the creator’s ConMan.
Stop	The local lifts are stopped. The ConMan sends a ‘remote stop event’ to the creator’s ConMan.
Destroy	The local lifts are stopped and all modules are deallocated. The ConMan sends a ‘destroy event’ to the creator’s ConMan.
Get Graph	A new (or the first) mechanism graph is sent by the creator. The ConMan instantiates the new mechanism graph according to the parameters of this event.
Remote Start	The creator has started its lifts. If the application already issued a start event, then the lifts are also started.
Remote Stop	The creator has stopped its lifts. Stop the lifts, too.
Remote Destroy	The creator has terminated the session. Destroy the session, too. Send an event to the application.

The semantics of these requests is shown in Table 11 on page 27.

7.1.3 ConMan Error Control Protocol

The connection manager needs reliable data transfer. For a multicast capable session, a multicast error control is needed as well as unicast error control. The principal flow of connection manager data is shown in the Figure 8 on page 28.

In order to simplify the implementation, the connection manager uses TCP for the bidirectional unicast flows, even if ATM is used as a an underlying transport infrastructure. The control protocol then uses the modules shown in Figure 9 on page 28. Depending on the data which needs to be sent, either the multicast error control or the unicast error control (TCP) is used. The connection manager uses an address field in the header of the data to specify the recipient of the messages. Since this header field has to be interpreted by the multicast error control module as well as the transport module, Da CaPo headers are not suitable for this task. Instead, the header fields are placed inside the Da CaPo data packets. This has

the effect that all used modules are no longer universal and can only be used inside the connection manager.

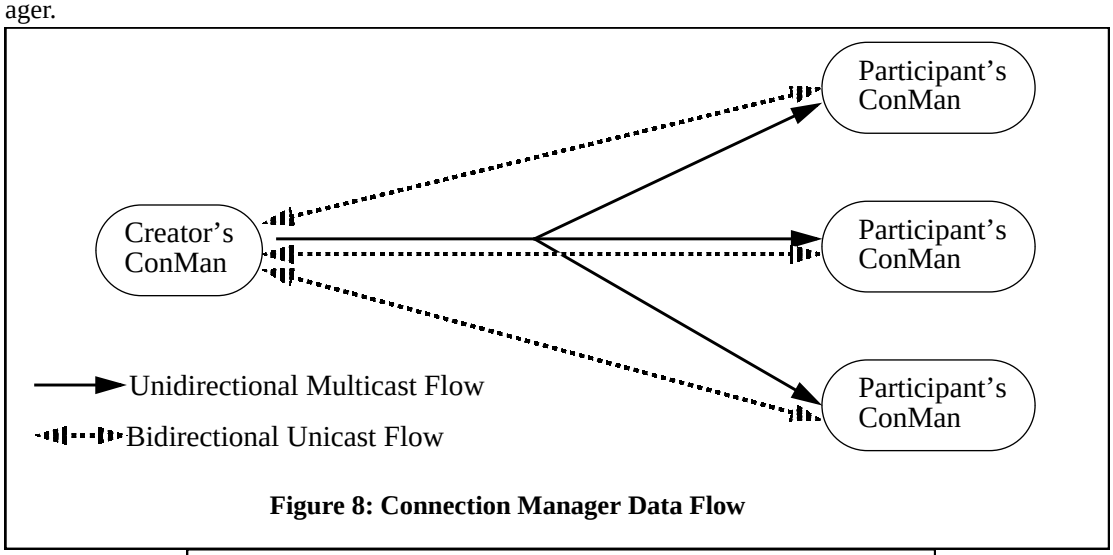


Figure 8: Connection Manager Data Flow

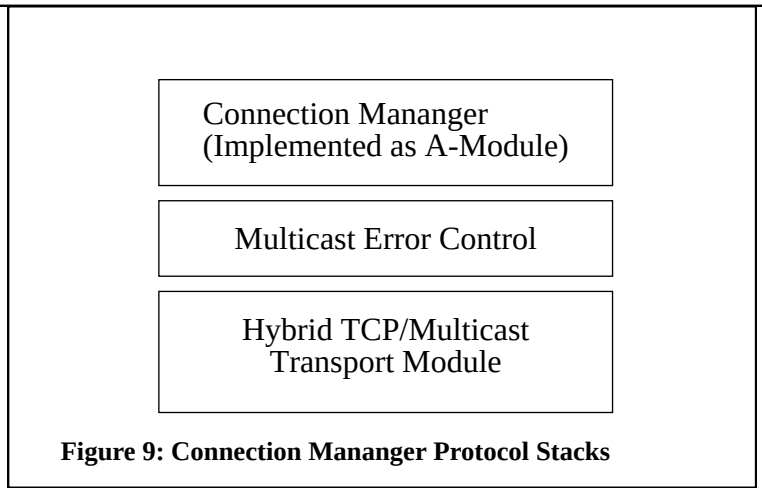


Figure 9: Connection Mananger Protocol Stacks

7.1.4 Finite State machines

7.1.4.1 Finite State Machine of Creator's ConMan

The creator's ConMan uses the finite state machine shown in figure Figure 10 on page 29. The ConMan changes its state upon receiving events. Events are received either from the application (for example 'Create') or from the remote connection manager. Remote events are shown with a leading arrow (<-). An arrow to the right (->) indicates an event that is sent to the creator's connection manager.

In order not to overload the figure, not all states and state transitions are shown. The following comments have to be made:

The 'Exit' state is reached from every other state upon receiving a 'destroy event' from the application.

The states 'peer known', 'peer started' and 'running' keep track of the number of participants. For example, the transition between 'peer known' and 'local configured' occurs only when the last participant sends a destroy message. The same holds true for the other states.

A 'configure' event from the application leads to a local reconfiguration, irrespective of the state. After the configuration, the new protocol is instantiated and a 'sendgraph' event is distributed to all participants.

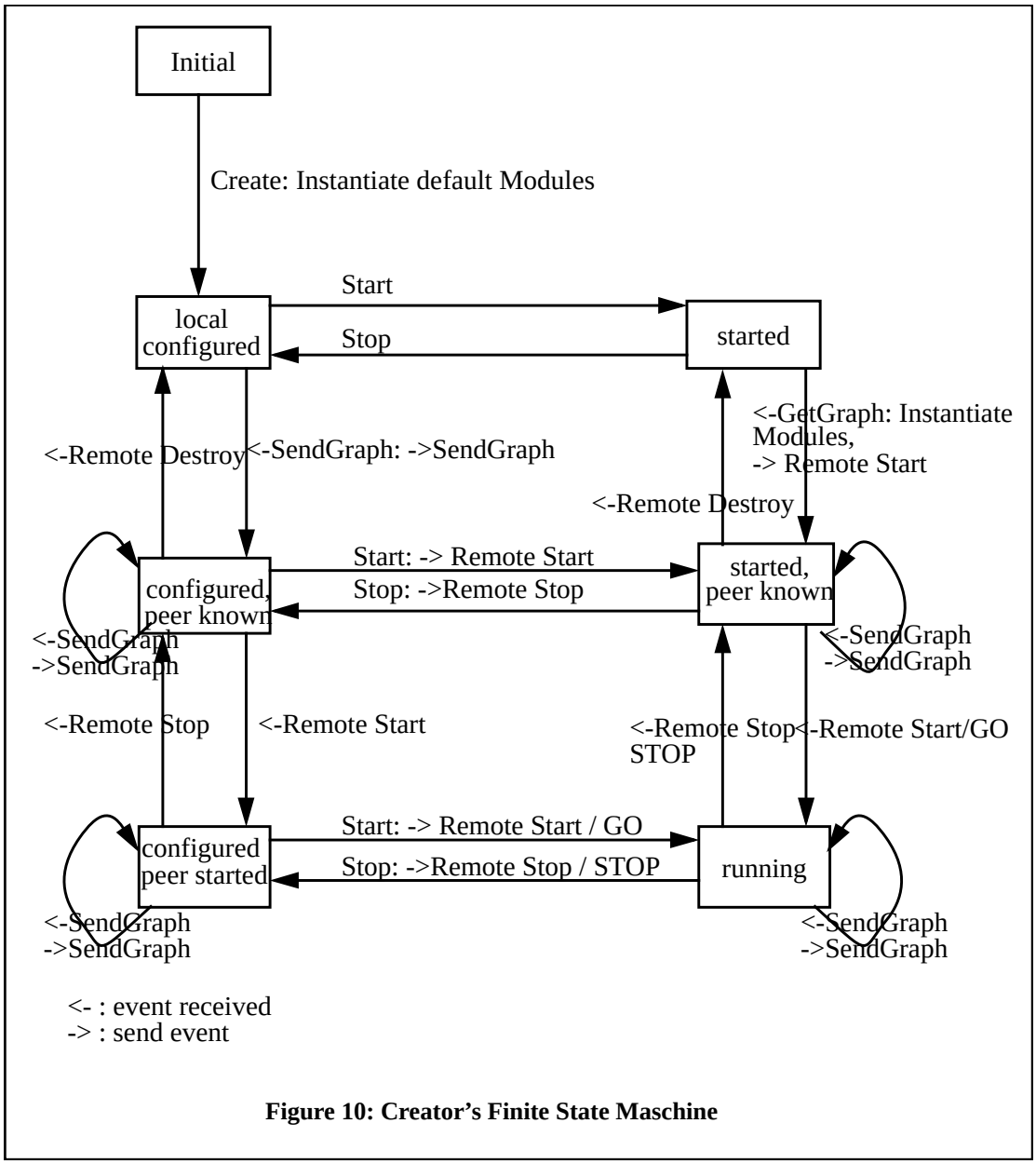


Figure 10: Creator’s Finite State Maschine

7.1.4.2 Finite State Machine of Participant’s ConMan

The finite state machine of the participant consists of seven states. Figure 11 on page 30 shows the state machine without the ‘Exit’ state. The ‘Exit’ state is reached when either the application wants to destroy the session or when the creator’s ConMan sends a destroy event. In the case where the application triggers the destruction, a destroy event is sent to the creator’s ConMan.

If the state ‘ready to run’ is reached, data loss may occur, because the creator’s lifts are running while the local lifts are stopped.

7.2 Multicast Transport Protocols

Basically, there are only two different multicast transport protocols: A reliable point-to-mulitpoint protocol and a simple point-to-multipoint protocol for audio and video. Both protocols are based on a multicast capable network layer. This is either IP multicasting or ATM multicasting with extensions.

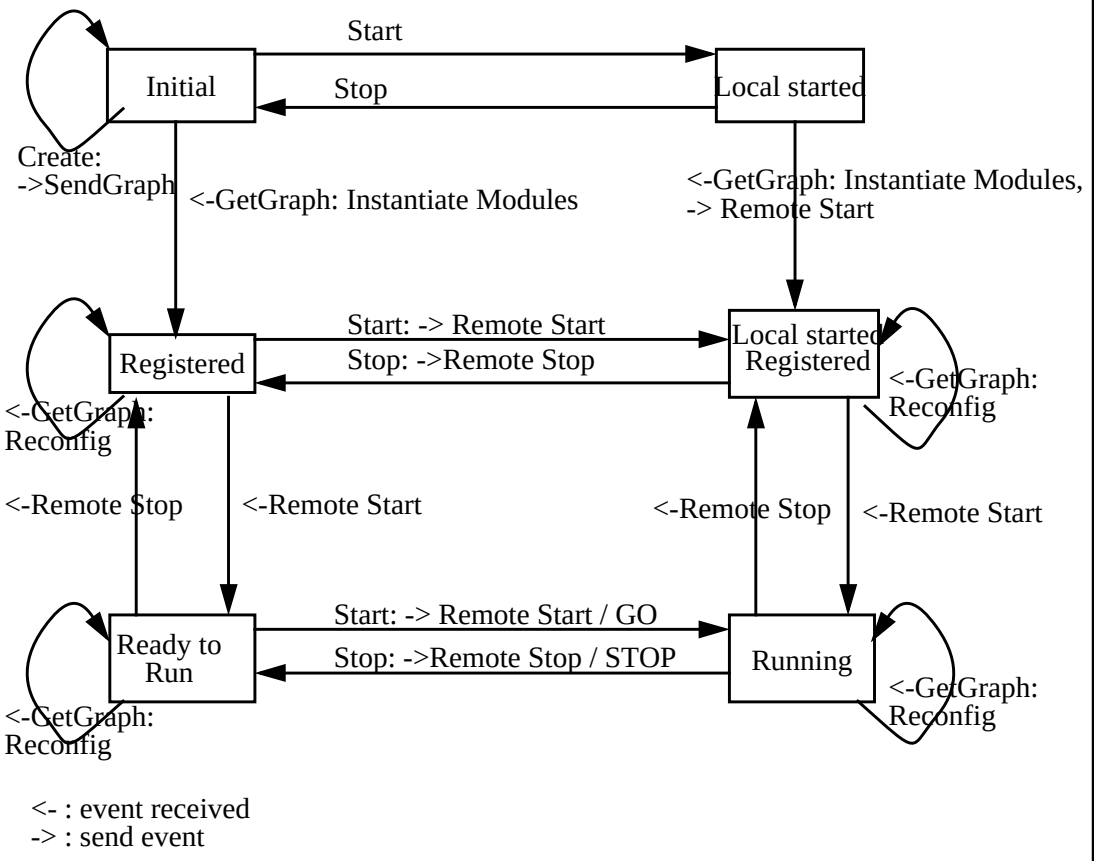


Figure 11: Participants' Finite State Maschine

7.2.1 Reliable Multicast Protocol

The reliable multicast protocol uses an error control mechanism based on retransmissions for assuring the correctness of the data transport. In order to avoid a packet implosion at the sender, a negative acknowledgment scheme is used. The receivers detect loss of data by comparing the sequence numbers in the arriving packets with the expected sequence numbers. When a gap in the numbers is detected, the packets are requested from the sender. If the sender has no data to send, it regularly sends a so-called heartbeat packet that contains the last sequence number only. The heartbeat packet enables the recipients to detect packet losses. Heartbeats are sent in predictable intervals. Retransmissions which are requested from one of the receivers are multicast to the group. Duplicates are detected by the receivers and thrown away.

As seen from the basic description above, the protocol can be separated into a sender's part and a receiver's part. The sender's part has the following tasks:

- Multicast datagrams to the group and store the datagrams for retransmission.
- Send heartbeat packets when no datagrams are available to send, use a timer for the heartbeat messages.
- Answer retransmission request by re-multicasting the packet to the group.
- Send a leave message to the receivers when leaving.

The tasks of a receiver are the following:

- Maintain the actual sequence number for the sender. The initial sequence number is statically defined, i.e. the first packet has the sequence number 0.

- Maintain a timer for the sender. The timer is reset whenever a datagrams or a heartbeat packet arrives. If the timer goes off, either a datagrams or a heartbeat packet has been lost.
- Check for corrupted, lost or duplicated datagrams. Issue a retransmission request for lost and corrupted datagrams, throw away duplicates.
- Sort the datagrams according to their sequence numbers.

The protocol also features a segmentation and reassembly module which is placed on top of the error control module.

7.2.2 Simple Multicast Protocol

The simple multicast protocol is used as a transport protocol for audio and video services. The protocol is very simple. It consists of a segmentation and reassembly mechanism and a transport mechanism. Error detection is optional, since both IP multicasting and ATM AAL5 already provide error detection methods. This protocol only makes sense if used in conjunction with a specialized A-module that directly feeds or gets the data from a multimedia device such as the parallax video board or the audio device.

7.2.3 Changes in the Existing Da CaPo Kernel – Attributes

Both multicast protocols need a multicast capable connection manager in order to run properly. When used with the unicast connection manager, the sender can only transmit data to a single receiver. In addition to the multicast capable connection manager, new Da CaPo attributes have to be introduced:

- aiGroupAddress
- aiGroupService

When using in conjunction with an IP multicasting transport module, the following attributes are needed for full application control:

- aiTTL‘Time to Live’: used for controlling the distribution of multicast packets.
- aiLoopback Specify whether packets should be delivered back on the same machine.
- aiMcastInterface Interface on which multicast packets are issued.

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, telebanking, teleseminar, teleshopping, teleteaching. Due to a set of well-defined differences between these applications, this spectrum of applications looks quite unstructured. For instance, a teleseminar 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 controling the handling and manipulation of audio and video data. Finally, a teleseminar 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 controling 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 teleseminar or teleteaching 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.

8.1 Applications

8.1.1 Picture Phone (PP-APP)

In the Da CaPo++ terminology, the Picture Phone application is a unicast 1:1 application including live audio and video data exchange. The multicast case (n:n) is referred to as the Video Conference application. In general, the unicast case can be seen as a specific subset of the multicast application.

8.1.1.1 Design

The current design state for the Picture Phone application is as follows:

- The unicast Picture Phone is today available and can be demonstrated with little effort. However, it is a very basic application with regards to the user interface and the protocol functionality (only two A-modules and one T-module).

Basically, the needs of the Picture Phone application concerning Da CaPo is on one hand the audio/video flows objects (with corresponding A-modules and audio/video protocol graphs) and on the other hand (application components) the corresponding A/V components to make video/audio data visible/audible to the user.

There is already in the design Section of the API (cf. Section 5 on page 7) a simple example on how to declare necessary flow/session objects to meet the functionality of a 1:1 Picture Phone application. Basically it consists out of 2 synchronized sending flows for both audio and video and of their corresponding receiving flows. The A/V Presentation component must be created as an object and bound to both receiving flows. Finally, the communication may start us intended.

8.1.2 Video Conference (VC-APP)

The current design state for the multicast n:n Video Conference application is as follows:

- The multicast Video Conference still requires some thoughts on how to make it compliant with the current state of the Da CaPo core system (especially with the connection manager).

In the current state of the connection manager in Da CaPo, it is not possible to have in a single session several multicast flows issued from different senders (cf. Section 7 on page 25, connection manager). Therefore it is not possible to consider a unique session at the participant's site, as each receiving flow, either audio or video, would have to "register" to a different sender.

To alleviate this limitation, one session per participant is considered which consists in a multicast sending flow to all others and the corresponding receiving flows. In the case of n participants, at each participant's site there will be one multicast sending session (consisting in 2 multicast sending flows for both audio and video) and (n-1) receiving sessions (consisting in 2 receiving flows for both audio and video).

The connection manager is built in a client-server manner. This obliges each potential receiving session to explicitly require the protocol mechanisms from the corresponding sender (this is performed through the ConnectSession() function of the upper API). This property motivates the set up of an application protocol, so each potential receiver can be informed when it has to create a new session and who the creator (sender) is.

At this point it is either possible to implement a more sophisticated connection manager or to design a group management application component (in the upper API) which would have its counterpart in the Da CaPo kernel system. The first solution leads to modifications in the kernel system, and the new planned functionality may be too specific for the purpose of Da CaPo communication system. The second solution would allow to tailor dedicated solutions for various CSCW applications (as not only the Picture Phone application is likely to need such an "application" protocol).

8.1.2.1 Video Conference Setup

In a simple example one creator (C) and 2 participants (P1 and P2) as depicted in Figure 12 is considered.

The proposed protocol is based on a session creator (C) which acts like a master to distribute notifications to all participants. Therefore it is a strongly centralized management. The setup process can be decomposed in the following steps:

1. Object creation:

   An McSession object is instantiated with the list of all participants addresses at the creator, and only the creator address at the participants.

2. Connect phase:

   A multicast Da CaPo control connection is set up between the creator and all participants. This control connection must be bidirectional for the purpose of the application protocol. Either one unicast Da CaPo control connection is set up between each participant and the creator, or the out-of-band signalling functionality of the connection manager can be used (in this case, the API must be extended to also offer this functionality to the application).

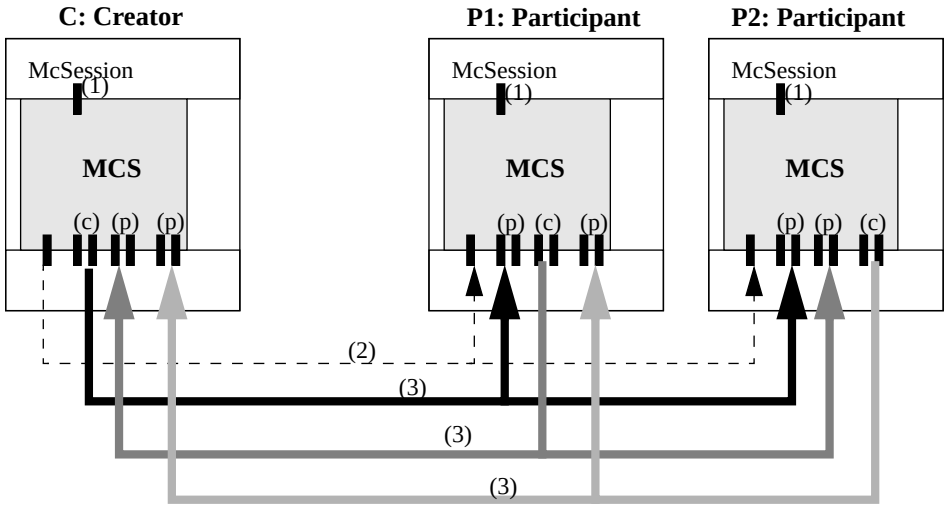


Figure 12Example of a Video Conference application setup

3. Data flows setup:

   Once the control connections are set up, it is possible to start with the requested application protocol (this protocol can be changed according to the needs of the distributed application). In a VideoConference application, it would look as follows (sites are referred as C, P1 and P2; "`x -> y : task`" means station x sends a notification to y with command "task"):

   ```
   C->P1 : "Create a receiving session for A/V flows coming from C"
   C->P2 : "Create a receiving session for A/V flows coming from C"
   ```

   On receiving these notifications, P1 and P2 will create the requested flows and register themselves by C. The multicast A/V flows are now set up between C and P1, P2.

   ```
   C->P1 : "Create a sending session for A/V flows"
   C->P2 : "Create a receiving session for A/V flows coming from P1"
   ```

   C and P2 will now create the requested flows and register themselves by P1 (being master, C knows which flows it has to create). The multicast A/V flows are now set up between P1and C, P2. When this is performed, C takes control again:

   ```
   C->P2 : "Create a sending session for A/V flows"
   C->P1 : "Create a receiving session for A/V flows coming from P2"
   ```

Finally, the multicast session is properly set up. By keeping an internal table with the status of each connection (involving all participants who are known at the beginning), the Creator MSC can issue an ActivateSession() on all his sessions as soon as all multicast connections are properly set up. As all participants have already executed an ActivateSession(), this enables the Creator to get a synchronized start with all participants (this property may not be necessary for a typical VideoConference application with live audio/video transfer, but it has to be provided for other applications, *e.g.*, where the Xwedge component is used).

An interesting property of this application component McSession object is that the creator of such an object is actually both creator and participant for several traditional Session objects. In a similar way, the participant of a McSession object is both creator and participant for several traditional Session objects. This situation is illustrated in Figure 12 on page 34 where (c) and (p) denote the actual creator and participant of traditional Session objects.

This mapping of the McSession object in all necessary Session objects is done transparently by the application component. Therefore the programmer has no access to the dynamically created traditional Session objects. The only way how to access them must be offered at the McSession object layer (through the available methods).

8.1.3 Extended WWW Browser and Server (EWB-APP)

The Extended WWW Browser enables the WWW user to receive file data via a Da CaPo++ link. The data is sent, received and presented by using a Da CaPo++ File Server and a Da CaPo File Client (cf. Section XXX below, application components). The Da CaPo++ File Client has to be started automatically on the client's side. It has to be informed with information needed to establish a Da CaPo++ link to the corresponding server and to receive the requested data file.

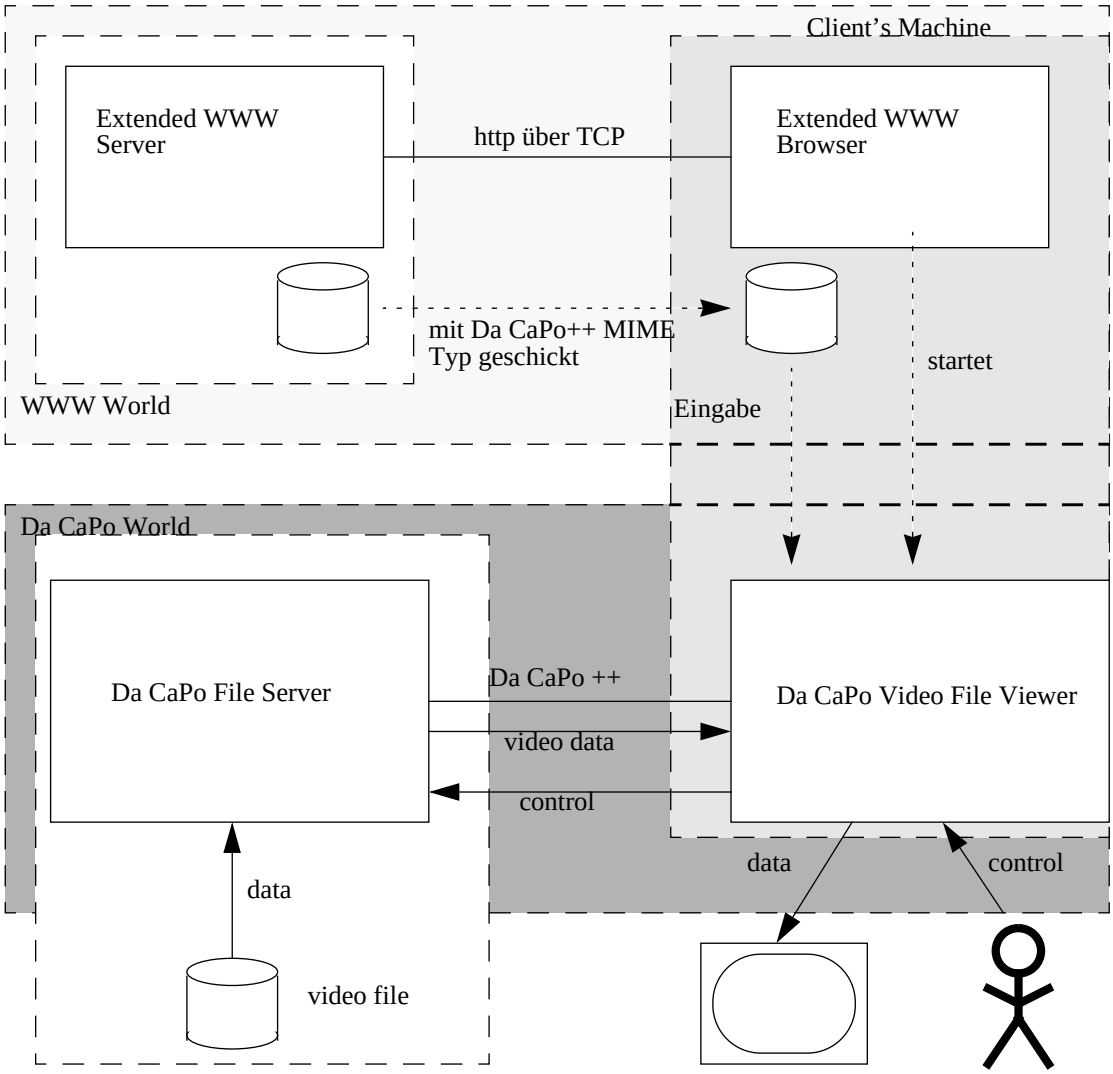


Figure 13Extended WWW Browser

This is done as follows. All information needed for the Da CaPo++ File Client to be initialized to retrieve the specified video data is written into a connection and video file specification file. This file gets the "user-defined" MIME type *application/x-dacapo* which has been assigned.

When the user clicks on a hyperlink within an html page to retrieve Da CaPo++ video file data, its machine receives via the http connection the connection and video file specification file. The Extended WWW Browser recognizes the Da CaPo MIME type and automatically starts the Da CaPo++ File Client

and hands over the newly received file as input file. The Da CaPo++ File Client then establishes the Da CaPo++ link and presents the video data on the screen as illustrated in Figure 13.

8.2 Application Components

8.2.1 File Server

The Video File Server was renamed to Da CaPo++ File Server, as it can easily be used to transmit audio or other data that are stored in a file to the client's site. The components shown in Figure 14 on page 36 will be explained in the following paragraphs.

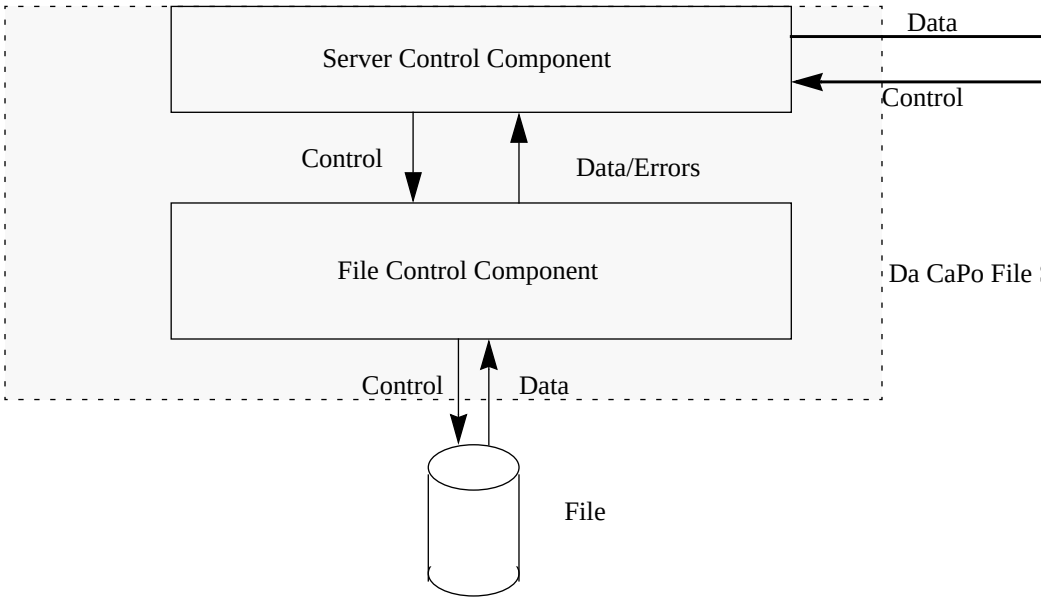


Figure 14Da CaPo++ File Server

8.2.1.1 Server Control Component

The server control component controls the Da CaPo++ link to the client and correctly transmits the file control information from the client to the file control component and the data from the file control component to the client.

Thus, the component has the following functionality:

- A Da CaPo++ communication link is established between the client and the Da CaPo++ File Server. This link satisfies the requirements (QoS parameters) specified by the client.
- The client transmits the name and type (audio, video, text) of the file to be read via the newly installed Da CaPo++ communication link to the server control component. Depending on the file type an appropriate file control component is instantiated, *e.g.*, the video file control.
- The server control component forwards the file control information the client sends to the server to the file control component as well as the file data obtained from the file control component via the Da CaPo++ link to the client.
- The Da CaPo++ link is closed properly.

8.2.1.2 File Control Component

The file control component controls the retrieval of file data. The data read is transmitted via the server control component and the Da CaPo++ link to the client.

The component has the following functionality:

- It obtains the name of the file to be read. This file is opened. Possibly occurred errors are transmitted to the client via the server control component.
- The control commands obtained by the client are performed. Such control commands may be, *e.g.*, “pause data transmission” or in case of a video file “fast forward”.
- Errors are handled within this component or appropriate error messages are created that are transmitted to the client.
- The file data being read is transmitted to the server control component to be sent to the client’s site.
- The file is closed on command.

8.2.1.3 Da CaPo++ Video File Server

The Da CaPo Video File Server consists out of the following components as depicted in Figure 15.

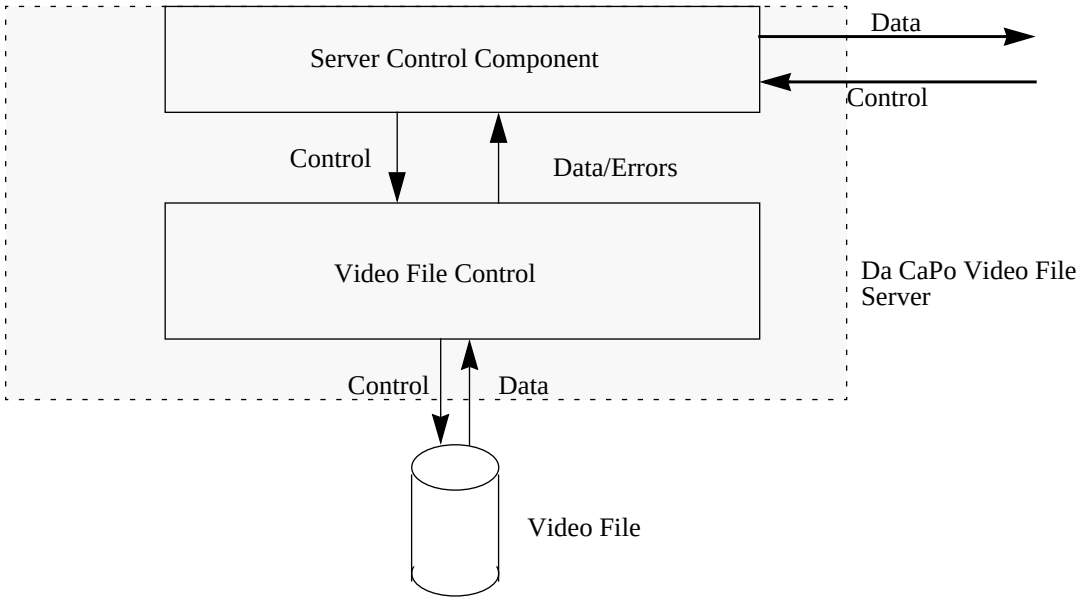


Figure 15Da CaPo++ Video File Server

The video file control is a file control component that provides the specified tasks to read a video file. The commands that have to be provided are play, pause, stop, fast forward, fast rewind, etc.

8.2.1.4 Class Design

To implement the Da CaPo File Server the following class design was chosen (cf. Table 16 on page 37):

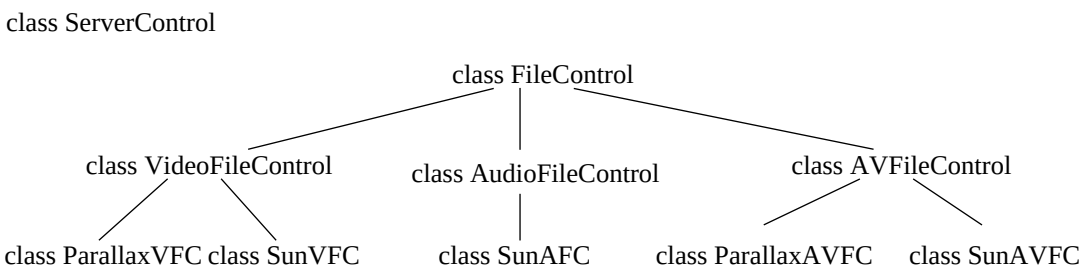


Figure 16Class Design

The classes can briefly be described as follows:

- class **ServerControl:** this class provides methods to control the Da CaPo++ link between the Da CaPo File Server and the Da CaPo File Client, to transmit incoming file control data to the instantiated file control component, and to send the read file data in form of a data stream to the API. Depending on the file type the Da CaPo File Server receives as input, an appropriate file control component is to be chosen, *e.g.*, an appropriate FileControl class is to be instantiated.
- class **FileControl:** this class provides methods to open and close the file to be read, to provide a pause in the data transmission, and to provide the error handling. It is the base class for all possible file control classes, i.e. file control components.
- class **VideoFileControl:** this class is the base class of all video file control components. Video file specialized methods are provided within this class, as Fast Forward, Fast Rewind, Play.
- class **AudioFileControl:** this class is the base class of all audio file control components, providing methods of play and fast forward and fast rewind the audio.
- class **AVFileControl:** this class is the base class of all file control components for files, in which audio and video data are stored together. Methods to perform fast forward, fast rewind, play are provided in this class.
- class **ParallaxVFC, SunVFC, SunAFC, ParallaxAVFC, SunAVFC:** these classes are inherited from the corresponding file control classes. Methods are overloaded depending on the technical environment of a computer system like the video card if necessary.

8.2.2 File Client

On the client's site a Da CaPo++ File Client is installed, that controls the Da CaPo++ link to the Da CaPo++ File Server, that gets the user input and that provides the output of the file data.

The structure of the Da CaPo++ File Client is shown in Figure 17 on page 38.

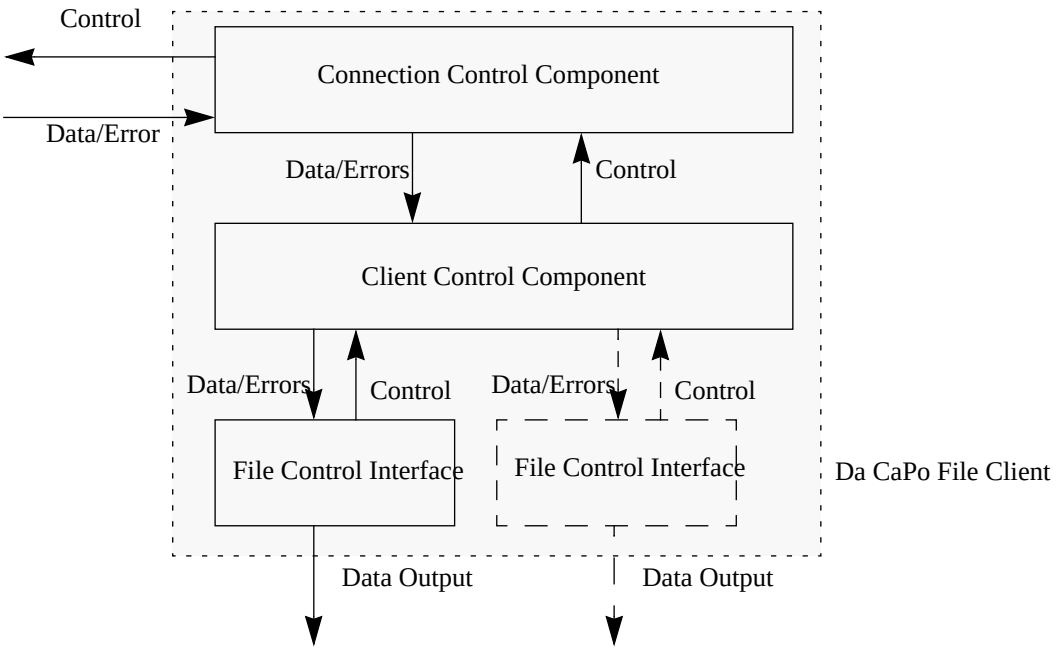


Figure 17Da CaPo++ File Client

The components shown in Figure 17 on page 38 will be explained in the following paragraphs.

8.2.2.1 Connection Control Component

This connection control component controls the Da CaPo++ link between the Da CaPo File Client and the Da CaPo File Server. The Da CaPo++ link is initiated by the client, i.d. the connection control component of the Da CaPo File Client.

The connection control component gets an input that specifies the address and name of the Da CaPo++ File Server, the requirements of the Da CaPo++ link (i.e. the values of the QoS parameters) as well as the name and the type of the file to transmit. The connection control component establishes a Da CaPo++ link to the specified Da CaPo++ File Server. This link satisfies the demanded requirements. The data transmitted by the server's site will be given to the client control component, as well as all control commands, not concerning the Da CaPo++ link from the client control component will be transmitted via the Da CaPo++ link to the server's site.

The connection control component, thus, has the following functionality:

- A Da CaPo++ link to a Da CaPo File Server is established. The server as well as the characteristics of the link are specified as input.
- All incoming file data from the server are transmitted to the client control component.
- All incoming control commands from the client control component are transmitted via the Da CaPo++ link to the other side, as long as the control commands do not concern the Da CaPo++ link itself.
- The Da CaPo++ link is closed properly.

The connection control component does not care about the non Da CaPo++ control commands it receives from the client control component nor about the data or non Da CaPo++ error messages it receives from the server's site. Therefore the same connection control command can be used together with different client control components providing different tasks.

8.2.2.2 Client Control Component

The client control component takes care about the Da CaPo++ connection between server and client. It's the only component that really knows the partner of the link and the connection parameter values. The file control interface, on the other hand, provides the file in- and output (for the moment, though, it is not planned to implement a Da CaPo++ File Client creating or changing files on the server's site). It directly interacts with the user. The file control interface is not aware of the location (local or remote) of the file. The same file control interface, therefore, could be used to control a local file.

The main focus of the client control component is to assure this transparency. It provides the following functionality:

- Depending on an input file type, an appropriate file control component will be instantiated.
- All file data being transmitted from the server's site and received from the connection control component are transmitted to the file control component. The file data though could also come directly from a local file.
- Control commands from the file control component are transmitted to the connection control component to be sent to the server's site. These commands could also be directly translated into commands for direct file access, if the file to be read is a local file.

8.2.2.3 File Control Interface

The file control interface presents the file data on the monitor or another device and gets the user input concerning the file control. Such user control commands may be: get data, open file, close file. The file control interface provides, in general, a graphical user interface to enable the file control.

The functionality of the file control interface consists in the following points:

- The incoming file data are properly presented on the output device.
- The user input is read and translated to control commands that are transmitted to the client control component.
- A graphical user interface is provided to the user.

Such a file control interface may be provided for video files, audio files, files where video and audio data are combined.

8.2.2.4 Da CaPo Video File Viewer

The Da CaPo Video File Viewer is a Da CaPo File Viewer that serves to present a video file on the monitor. Its components are shown in Figure 18 on page 40.

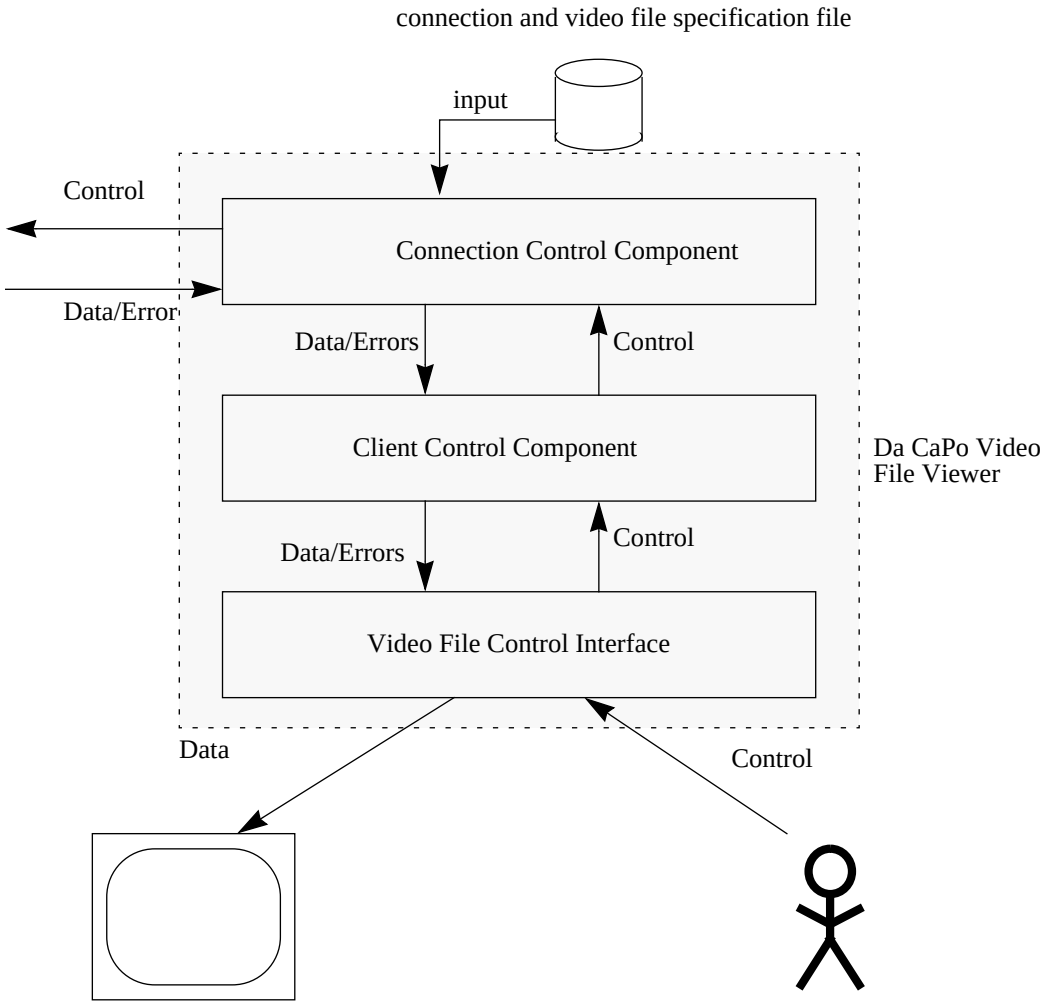


Figure 18Da CaPo++ Video File Viewer

To implement the Da CaPo++ File Viewer we need the following specialized components:

- A connection control component that gets its input data from an input file. The file is named: connection and video specification file and contains the name and address of the server, the name of the video file, the file type “video” and all Da CaPo++ requirements for the link being able to correctly transmit the requested video data.
- A client control component that knows the connection control component and a video control interface providing a control interface suited for video file data to the user.
- A video file control interface that provides a graphical user interface suited for video file data. Such a GUI will provide control commands for play, pause, stop, fast forward, fast rewind, resize window. The output of the video data is provided on the monitor screen.

8.2.2.5 Class Design

The following class design (cf. Figure 19) was chosen to implement the Da CaPo File Client.

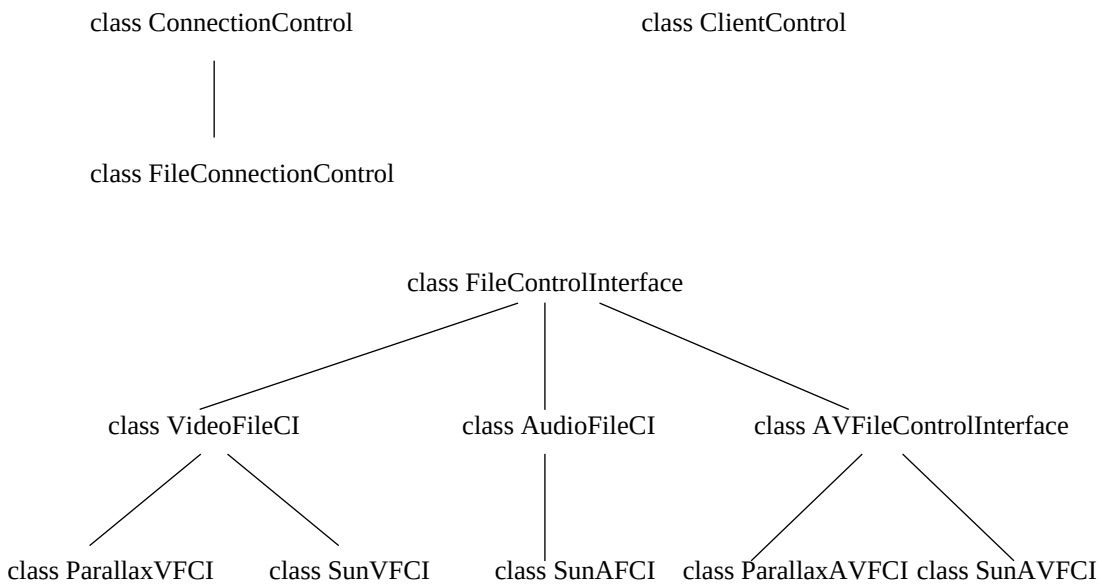


Figure 19Class Design

The classes can be described as follows:

- class **ConnectionControl:** the class ConnectionControl is the base class of the connection control components. It provides all methods to establish a Da CaPo++ link to a specified server, to properly close an existing Da CaPo++ link, to transmit user data via this link to the server, and to transmit data coming from the server to the appropriate client control component.
- class **FileConnectionControl:** the class is inherited of the class ConnectionControl. The input data specifying the Da CaPo File Server, the file to be read, and the connection parameters are given as an input file.
- class **ClientControl:** this class is the implementation of the client control component that provides all the functionality described for this component.
- class **FileControlInterface**: this class is the base class for all file control interface classes. It provides a standardized graphical user interface with functions as open file, close file, start data retrieval, stop data retrieval.
- class **VideoFileCI:** this class provides the file control interface for video files. Hereby additional control commands as resize video window, fast forward, fast rewind, pause, are provided.
- class **AudioFileCI:** this class provides the file control interface for audio files. The additional functionality provided in this class is pause, fast forward, fast rewind, change volume, change height.
- class **AVFileControlInterface:** this class provides an interface for video and audio data coming from a file. It offers the functionalities: fast forward, fast rewind, pause, resize video window, change volume, change height.
- class **ParallaxVFCI, SunVFCI, SunAFCI, ParallaxAVFCI, SunAVFCI:** these classes are inherited from the corresponding file control interface classes. If necessary methods are overloaded to be implemented according to the constraints of the technical environment as video and audio cards.

8.2.3 Group Management Comfort (GMC)

Ideally this component would be based on a Group Management System (GMS) design. A centralized server would inform each potential participant on the currently available sessions and how to join them. Notifications would also be emitted to all already established participants to let them know about the joining participant.

Another possibility would be to propose a less ambitious solution based on the current connection manager and a dedicated application component which would hide all complexity generated by this application protocol. A very coarse design is proposed in the following section.

8.2.4 Multicast Support (MCS)

The introduction of this new application component can be viewed in Figure 20 on page 42.

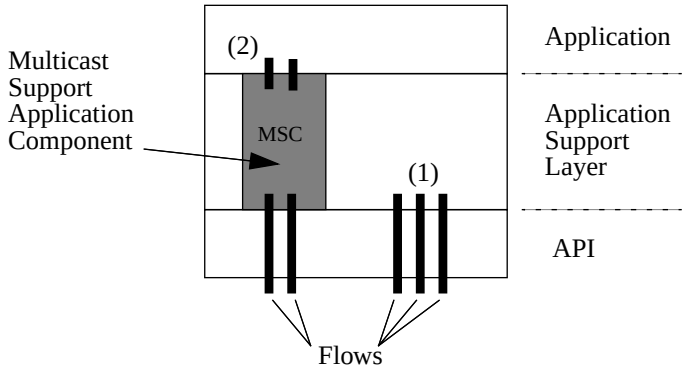


Figure 20Multicast Support Component

A new layer is introduced, namely the application support layer, between application and API. Purpose of this new layer is to offer to any application support for n:n multicast communication. Basically, the above mentioned application protocol should be hidden in this new component. Naturally the programmer still has the possibility to directly address the API as in (1), for additional functionality however, it can use the extended API offered in (2).

A new session object (McSession, more explanations in Section 8.2.4.1 on page 42) is instantiated in the upper API which issues a control communication with its peer multicast support MSCs for notification exchanges. All group management activity (join of new participants, ...) is performed by these MSCs over Da CaPo control connections. The MSC behaves like a server, waiting for notifications relevant to the current application, and creating if necessary new sessions in the current VideoConference application (in the current example, new receiving sessions are created when a new participant wishes to join the VideoConference).

Doing this new application component intelligent enough would enable to re-use it for different CSCW applications (and not only symmetric ones as for the current VideoConference). The main advantages of this solution are to meet requirements of the PicturePhone application and to keep connection manager as it is (no change necessary in core system).

8.2.4.1 The Multicast Support Object Model

The already mentioned multicast object is an instance of the following class:

```
class McSession {
     McSession(char *ConfigurationFile);
     ~McSession();
             // the ConnectMcSession method is used by both a
             // participant known at the beginning and by a late
             // “joiner”, in both cases, the Creator address
             // has to be provided
     ConnectMcSession(CREATOR|PARTICIPANT, ...);
     ConfigureMcSession();
     ActivateMcSession();
     ...
     LeaveMcSession()
     ...
}
```

The purpose of this McSession object is to provide additional functionalities at the application-application component layer interface with regards to the one offered by the upper API. This new session object is responsible to hide all necessary application protocols that is used to set up a Video Conference application (or, more general, a CSCW application).

It is important to remember that the McSession object is part of an application component and not part of the upper API as the traditional Session objects. Thus, an McSession object may encompass several traditional Session objects, which can be dynamically created or destroyed according to the application needs (either local and remote applications).

9. Error Handling

9.1 Data Transmission and Error Levels for File Server and Client

According to the hierarchical structure of the design we can distinguish between different kinds of data and errors. The following paragraph gives a brief overview. We will distinguish the conceptual level and the implementation level, as Da CaPo++ A modules may provide some data in- and output.

9.1.1 File Data

The data stored in and read from the file conceptually is known within the file control on the server side and within the control interface on the client side. The idea of the design is that all underlying components do not know which type of file data is to be transmitted and presented. For all other components these data are data streams of any type. As an appropriate A-module has to be instantiated to transmit specific data over a Da CaPo++ link, this conceptual transparency of the data type can not really be implemented. The API has to know which data type has to be transmitted.

9.1.2 Control Data

The file control commands (control data) come from the user and influence the retrieval of data from the file. Conceptually these data are only known in the file control component in the server resp. the file control interface in the client. But as the retrieval of file information to be transmitted via a Da CaPo++ link is done within an A-module, the components directly have to address the A-module. We can state, that the way to transmit and retrieve file data within Da CaPo++ leads to consequences that soften the strict transparency provided by the design presented above.

9.1.3 Connection Link

The server control component on the server side resp. the connection control component on the client side are the only components that have information on the connection link and that create and receive control data concerning the network connection (Da CaPo++ connection).

9.1.4 Error Messages

We can distinguish two different kinds of errors that can occur. Some errors concern the file to be read. These errors have to be treated within the file control component on the server side. If they cannot be handled properly, appropriate error messages have to be created for the user that should be presented within the file control interface level. Other errors concern the connection link level. If they are not handled directly, in Da CaPo++ the error handling has to be provided within the server control component on the server's side resp. the connection control component on the client's side.

Also publ. as: Computer Engineering and Networks Laboratory (TIK),
Swiss Federal Institute of Technology Zürich (ETH), 1997, TIK-Report No. 28

Detailed Design:
KWF – Da CaPo++ – Project

Daniel Bauer, Germano Caronni, Christina Class, Christian Conrad, Burkhard Stiller, 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 Goals

The research project KWF–Da CaPo++ is based on the project Da CaPo (Dynamic Configuration of Protocols) at the ETH. The extended system of Da CaPo++ shall provide 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.

One main goal for Da CaPo++ includes the provision of a real-life application framework. A variety of different applications has to be managed modularly. Therefore, the selection of special services, application components, applications, and application scenarios results in the transparent handling of communication relevant tasks. Specifically, details on the type of network to be used or the functionality of the applicable communication protocol is hidden completely form the user´s perspective.

Another main goal of the extended Da CaPo++ system is to provide privacy and authentication of transferred data. Therefore, in the banking environment a configurable degree of security is supported. The range of parametrizable security functionality includes varying degrees of authentication and privacy. A set of Quality-of-Service (QoS) parameters, attributes in the Da Capo++ terminology, allows for the specification of various security algorithms to be used or time to live boundaries for cryptographic keys to be specified.

Furthermore, the design of Da CaPo++ is 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), will be supported.

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.

Currently, Da CaPo supports one single application, including multiple protocols for different data streams, *e.g.*, a Picture Phone handles an audio and a video data stream separately by two different logical communication protocols which are represented by four different flows (sending and receiving audio and video) and supported by four different instantiated communication protocols in the Da CaPo core. A specific run-time system supports on a module basis a variety of tasks. Every module used offers a unique interface, including control and user data manipulations.

The Da CaPo++ core is responsible for handling 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.

The core system is internally structured into eight components. The attribute translation accepts the application requirements and translates them to a structure suitable for CoRA. CoRA calculates the appropriate module graph. The module graph is locally instantiated by the data transport component and distributed to peer systems by the connection manager. The security manager validates users and applications and assures that the necessary modules are contained within the module graph.

1.2 Structure of this Detailed Design Document

This document contains a discussion of necessary changes and extensions to Da CaPo and describes key functionality that has to be added to different elements of Da CaPo. Furthermore, the application framework is presented, including the number of designed elements, such as application components and applications. 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. Section 5 on page 8 includes the architectural 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.

Section 6 on page 25 contains security aspects. In detail the specification and translation of security requirements is discussed in addition to keying and assurance of security at run-time. The Security Manager as the main Da CaPo core component for dealing with security issues is introduced and discussed.

Furthermore, Section 7 on page 43 covers relevant aspects of multicasting. This is on one hand the adaptation of a Da CaPo core component the Connection Manager to multicast requirements. Additionally, the protocol functionality for handling multicast connection in the transport level is presented.

Finally, the Application Framework of Da CaPo++ is extensively provided in Section 8 on page 54. Applications (Picture Phone, Video Conference, Extended WWW Browser) and application components (File Server, File Client, Multicast Support) are presented. Application scenarios have not been included due to project partner responsibilities.

Section on page 3 delivers the table of contents and additionally lists figures and tables afterwards.

1.3 Remarks

This document covers work packages done at ETH Zürich, in more detail 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 architectural design phase of the project, all issues are subject to change in more detail. 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.

2. Table of Contents

3. List of Figures

4. List of Tables

5. Application Programming Interface (API)

This Section intends to introduce the architectural design of the API (it was initially planned to bind the API IPC mechanisms with the buffer management strategy in the Da CaPo++ project, however, this has been separated for the first approach). All function names, arguments, table fields that are proposed just give hints on how to implement desired functionality and are thus subject to change. Final versions of these information will be given within the detailed design phase.

5.1 Design

In order to make the management of resources easier, it was decided to use only one Da CaPo process on a machine. Thus applications have their own processes and communicate with the Da CaPo server via 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

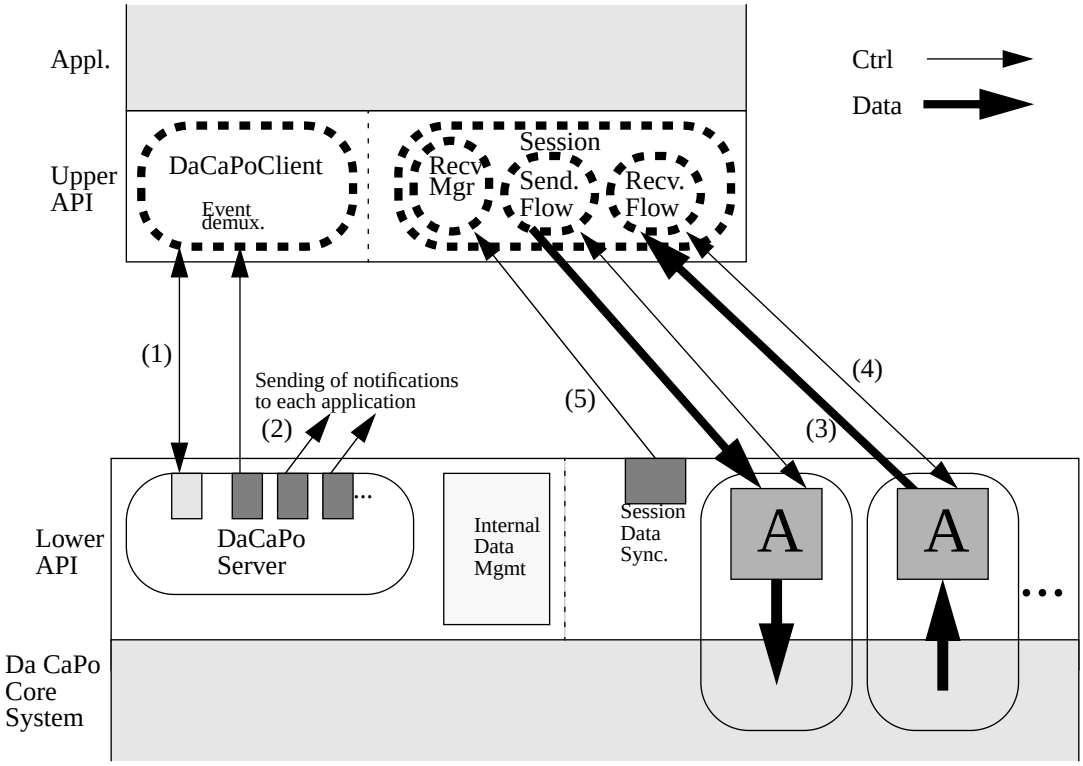


Figure 1 API Components

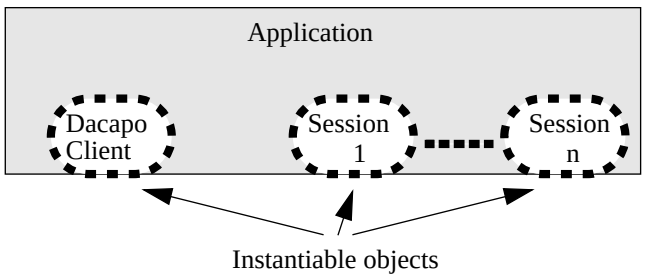
Basically, it consists of an upper API (described in Subsection 5.2), of a lower API (described in Subsection 5.3)and finally of an IPC mechanism to deal with the sending and receiving of both data and control information between the two separate processes (see Subsection 5.4). All components of this figure will be intensively considered in the following sections.

5.2 Upper API

The upper API part is linked to the application and represents exactly what the application sees from the Da CaPo core system. To enable the use of abstractions and thus to facilitate the developer's job, it was decided to write both application and upper API in an object-oriented programming language (actually C++). On the other hand, the Da CaPo core system remains pure C code. For some already written applications in C, either a "dedicated" C-C++ interface has to be provided, or the upper API has to also be available in C (this can be done automatically with translation tools, however, this will not be considered in this document).

5.2.1 The Upper API's Object Model

Following sections introduce the most important objects and components that build the object-oriented interface to the Da CaPo core system, namely the Da CaPo client objects on the one hand, and the session objects on the other hand (cf. Figure 2).



NOTE:

In this figure, only the directly visible objects are represented (cf. explanations in Paragraph 5.2.1.2)

Figure 2 Upper API Objects

5.2.1.1 The DaCaPoClient Object

Before being able to work with Da CaPo, the user has to be properly registered and authorized. This is done by creating a DacapoClient object which will send a request to the core system security manager with all security relevant parameters[1] such as the application identifier, the user identifier (either a password, a passphrase, an RFC 822 identifier, ...), a public or secrete key (this parameter list is not exhaustive, and may not be transmitted in all cases, cf. security document).

The DaCaPoClient object can have only one instance in an application. Moreover it is the only available interface for sending control information to the Da CaPo core system. Thus, a reference of this object's instance has to be provided each time a transaction is initiated between upper and lower APIs.

The DaCaPoClient object has also to consider incoming events from the Da CaPo core system. This task is provided by an "Event Demultiplexing" component. This EventDemux component is implemented in a thread. Each time a notification arrives (containing the session identifier it belongs to), it calls the event upcall function which was delivered by the corresponding session when it was created.

The attributes and methods of this object are illustrated in Table 1. The mapping of this information on the concrete C++ classes can be performed straightforward. This table representation allows to avoid considering specific C++ programming issues, such as the public and private parts, and the interclass relations such as friendship, ... (these issues will be addressed in the code header files).

The complete process of the creation of a DaCaPoClient object is depicted in Figure 5 on page 20.

5.2.1.2 The Session Object

A session is a collection of flows that can be synchronized. To enable the automatic generation of session objects by parsing of a configuration file and to keep the possibility of late joining/removing of flows in a session, it was decided to make flows objects not directly visible to the application programmer. Thus, session objects have to provide all desired functionality relative to both session and flows. The session object's methods are illustrated in Table 2 (this table also defines the API's functionality that is visible to the application programmer). The attributes are shown in Table 3.

5.2.1.3 The Flow Objects

The flow object model is illustrated on Figure 3

1. For convenient reasons, all security relevant information is considered as a single string (char *) throughout this document. This coarse approach will be refined with help of the security design document.

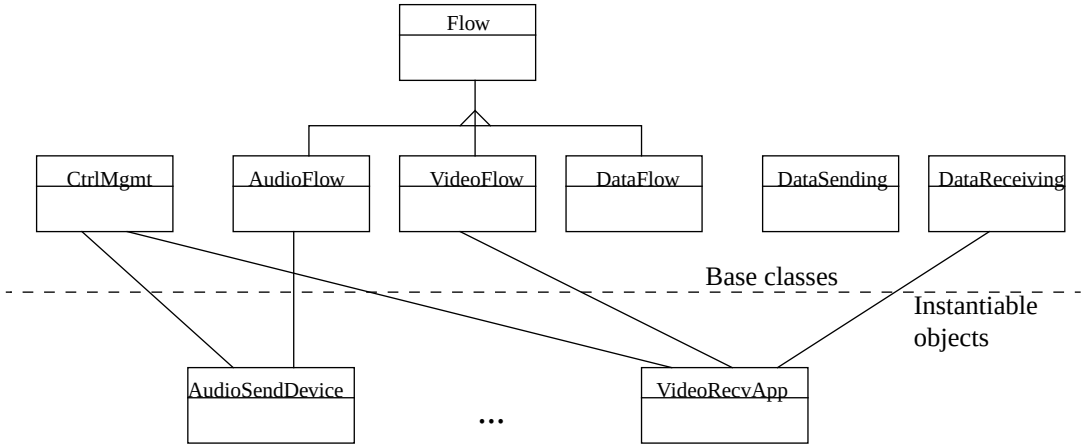


Figure 3 Flow Object Model

TABLE 1. The DaCaPoClient Object

Class	Attributes	Methods
`DaCaPoClient`	• `int Status;` This status information tells if the user is authorized to work with Da CaPo. Possible values are either NOT_AUTHORIZED or AUTHORIZED • `char *DaCaPoAccessPoint;` This contains the well-known address of the lower API's control access point. • `char *LocalAccessPoint;` This contains local information for the set-up of the bidirectional control data connection. • `char *LocalEventAccess;` Internal Da CaPo events will be received and demultiplexed at this address.	• `DaCaPoClient(char *securityInfo);` Object constructor, this is the only visible method for the programmer. All following methods are exclusively for internal use. • `SendCtrl(char *request, char *indication);` Control data is sent to the Da CaPo Server component, the call is blocking and returns an indication from the Da CaPo core system. • `StartEventDemux();` A thread is created to process incoming events from the Da CaPo core system. Only one such thread is created in an application. The thread "listens" on the `LocalEventAccess` point. • `AddSessionUpFunc (int sessionId, UpcFunc *f);` This function enables to map incoming session events to the right event handler (through an upcall function).

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 the 6 base classes, 3 specify the type of data that is transmitted over the protocol graph. The remaining 3 base classes are related to the sending/receiving of data/ctrl information (NOTE: the sending and receiving of control information are encapsulated in the CtrlMgmt base class, as there must always be a bidirectional control connection between each flow object and the corresponding A-module). Attributes and methods of these base classes can be found in Figure 3

In all flow objects, there is the list of all application requirements that may be activated by the programmer for this flow. This allows to perform a first "sanity" check on the attributes before propagating them to the lower API, where the A-module will not be able to interpret them.

TABLE 2. Session Object Methods

Method	Description
Session(char *configurationFile, DaCaPoClient *client);	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
Listen(char *grpAddress, char *grpService, char *localAddress, char *localService, char *localInterface, EventAction *upcFunc)	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.
Connect(char *grpAddress, char *grpService, char *remAddress, char *remService, char *localAddress, char *localService, char *localInterface, EventAction *upcFunc)	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.
Configure()	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.
Activate()	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.
Deactivate (int stopWay)	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.
Close()	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.

TABLE 2. Session Object Methods

Method	Description
`GetFlowDescriptor (char *FlowName)`	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.
`SetReqFlow(int Flow-Descriptor, “QoS value”)`	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
`GetReqFlow(int Flow-Descriptor, “QoS value”)`	The GetReqFlow() function returns the actual configured values of required attributes, they may be identical to those set by SetReqFlow().
`SendDataFlow(int FlowDescriptor, char *data)` `RecvDataFlow(int FlowDescriptor, funcPtr *upcFunc)`	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.
`SendCtrlFlow(int FlowDescriptor, char *ctrl)` `RecvCtrlFlow(int FlowDescriptor, funcPtr *upcFunc)`	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.

TABLE 3. Session Object Attributes

Class	Attributes
Session	• `Flow *FlowList;` This is the list of all flows that belong to the session. • `int NbFlows;` Number of flows in the session. • `char *SessionName;` Logical name for the session. • `int SessionId;` This identifier is unique for each session on a machine, it is returned by the DaCaPo Server on the lower API. • `int Role;` Either CREATOR or PARTICIPANT

TABLE 4. The Flow Object

Class	Attributes	Methods
`Flow`	• `char *FlowName`	• `Flow();` The flow is registered in the lower API flow table • `SetAR();` The requirement, its value and weight function are sent to the lower API. An internal check is performed before sending the AR to lower API.
`AudioFlow`	List of all application requirements that are accepted by this flow	• Specific control functions for audio: `Play(), FF(), Rewind(), FastRewind, Stop(), ChangeSoundLevel(), ...`
`VideoFlow`	List of all application requirements that are accepted by this flow	• Specific control functions for video: `Play(), FF(), Rewind(), FastRewind, Stop(), ResizeWindow(), ...`
`DataFlow`	List of all application requirements that are accepted by this flow	
`DataSending`	• `char *shdMem;` • `sema_t *freePlace;` • `sema_t *newData;` It basically consists in waiting for `freePlace` and then signalling a `newData` over the shared mem.	• `SendData(char *data);`
`DataReceiving`	• `char *shdMem;` • `sema_t *freePlace;` • `sema_t *newData;` It basically consists in waiting for `newData` and then signalling a `freePlace` over the shared mem.	• `callback_function();`
`CtrlDataMgmt`	• `char *shdMemCtrl, *shd-MemData;` • `sema_t *freePlaceD, *newDataD, *freePlaceC, *newDataC;`	• `SendCtrl(char *data);` • `callback_function();`

5.2.1.4 The ReceiveManager Object

This object receives notifications from the lower API each time either a control or a data information has been sent by an A-module. The main reason why such a component is introduced is to reduce the number of necessary waiting threads in the upper API.

Each session object starts a ReceiveManager thread. This component acts then like an event demultiplexor, and calls the upcall functions of the corresponding flow objects. Each ReceiveManager object keeps an internal table where all upcall functions for all flows are listed. If this table is not complete, no Activate() can be performed on the session.

5.2.2 Session Configuration

To allow late joining or removing of a flow in a session, and to make the session structure more readable for an application programmer, it was decided to use a session configuration file. The contents of this file is transmitted as a parameter to the constructor of a session object. A dedicated component will then automatically parse this data, create the necessary flow objects, send the corresponding application requirements to the lower API (this process is described in Figure 6).

5.2.2.1 Setting of Application Requirements

An application requirement is composed of an attribute identifier, minimum and maximum values and a weight function.

Each application programmer is free to define his own attributes, under the condition he also implements the corresponding A-modules. Pro A-module, there must be a list of known attributes. These attributes must be then mapped in a human-readable way by the upper API flow objects.

All attributes can be identified by a string such as “FPS”, “DELAY”, “SECURITY”, ... According to the attribute type, the A-module knows then how to interpret these values.

For the transfer of the weight functions, either a predefined function can be used (identified through a string, just as the attributes, e.g., “LINEAR”, “EXP”, ...) or the programmer has the possibility to transmit a string such as “3x^2-5x+2”.

5.2.2.2 Script and Configuration Language

The definition of such a script language is made using the EBNF notation according to the definition below. The list of application requirements is not complete (only an example subset is provided). Some application requirements only require a single value (e.g., ASN.1 for the presentation coding), this is not reflected in this “simple” specification. The weight functions are in a first step addressed through three function identifiers (in a further step, the programmer will have the possibility to define his own weight functions).

```
session      = session_role session_type;
               “SESSION” name;
               flow_specification {flow_specification};
               [flow_synchronization {flow_synchronization};]
               “END SESSION”;
flow_specification =  “FLOW” flow_type name
                      app_requirement {app_requirement}
                      “END FLOW”
flow_synchronization = “SYNCHRONIZE” name “WITH” name
session_role =  (“CREATOR” | “PARTICIPANT”)
session_type =  (“LOOPBACK” | “UNICAST” | “MULTICAST”)
flow_type =     (“AUDIO_SEND_DEVICE” | “AUDIO_RECV_DEVICE” |
                 “AUDIO_SEND_FILE” | “AUDIO_RECV_FILE” |
                 “AUDIO_SEND_APP” | “AUDIO_RECV_APP” |
                 “VIDEO_SEND_DEVICE” | “VIDEO_RECV_DEVICE” |
                 “VIDEO_SEND_FILE” | “VIDEO_RECV_FILE” |
                 “VIDEO_SEND_APP” | “VIDEO_RECV_APP” |
                 “DATA_SEND_DEVICE” | “DATA_RECV_DEVICE” |
                 “DATA_SEND_FILE” | “DATA_RECV_FILE” |
                 “DATA_SEND_APP” | “DATA_RECV_APP”)
app_requirement = req_name req_val req_val weight_function
req_name = ( “THROUGHPUT” | “DELAY” | “DELAY_JITTER” |
             “RESIDUAL_ERROR_RATE” | “PACKET_LOSS” | “PACKET_ORDER” |
             “PRESENTATION_CODING” | “DATA_COMPRESSION” | “COST” |
             “AUTHENTIFICATION” | “SECURITY” | “FPS”)
req_val          = string | numeric
weight_function  = (“WF_CONST” | “WF_LIN” | “WF_EXP”)
name             = string
string           = character {character}
numeric          = ... any numeric value ...
character        = ...
```

An example of such a session specification is provided below:

```
CREATOR MULTICAST;
SESSION PicturePhone;

FLOW VIDEO_SEND_DEVICE VideoOut;
```

```
     FPS      10   15       WF_LIN;
     DELAY    0.1  0.15     WF_CONST;
     COLOR    NO;
END FLOW;

FLOW AUDIO_SEND_DEVICE AudioOut;
     DELAY_JITTER 0.1  0.2      WF_EXP;
     DELAY        0.1  0.25     WF_LIN;
END FLOW;

SYNCHRONIZE VideoOut WITH AudioOut;

END SESSION;
```

5.3 Lower API

The purpose of the lower API is to manage the communications between several applications and a single Da CaPo++ core system, for both control and data transfers. General control data is processed by the DaCaPo Server (through the Service Access Point) whereas data is directly processed by the corresponding A-modules (through Data Access Points). Both access points are further described in the following sections.

5.3.1 Service Access Point

The DaCaPo++ Server component is implemented as a thread and permanently “listens” on a well-known port. On receiving a request from an application, it then redirects the request to the responsible Da CaPo core component (e.g., the security manager for admission control, the connection manager for a protocol reconfiguration, ...). There is however only one control access point, in other words, the DaCaPo Server can process only one application request at a time.

The necessary information for the management of several applications is kept internally by the DaCaPo Server in three different tables (application, session and flow tables). For efficiency, these tables are likely to be implemented as dynamic lists. The stored information may be used either internally by the lower API or by other Da CaPo core components (e.g., the security manager). An adequate interface has thus to be provided to reach stored information. Some of these access functions are provided in Table 8.

TABLE 5. Application Table Structure

Field	Description
`char *appliName`	`Name of the application, only valid on the local machine.`
`char *secInfo`	`All security relevant information that was transmitted during the application registration.`
`char *commChannel`	`Information on how to communicate with the application (done through a communication socket to transmit Da CaPo intern events to the application).`
`int appId`	`Application identifier (scope on the local Da CaPo system), not visible to the programmer.`
`int status`	`Status of the application: [IDLE | ACTIVE ]. To be ACTIVE, there must be at least one session in the application.`
`int *sessionList`	`List of all sessions belonging to the application.`

5.3.2 Data Access Points

Figure 1 on page 8 shows that each A-module has two access points, the one for either sending or receiving data, the other one to both send and receive control information. These operations and the A-module

TABLE 6. Session Table Structure

Field	Description
char *sessionName	Name of the session, only valid in the corresponding application
int sessionId	Session identifier (scope on the local Da CaPo system), not visible to the programmer.
int appId	Application identifier (scope on the local Da CaPo system), not visible to the programmer.
int status	Status of the session: [NOT_CONNECTED | CONNECTED]
int type	type of the session: [UNICAST | MULTICAST]
int *flowList	List of all flows belonging to the session.

TABLE 7. Flow Table Structure

Field	Description
char *flowName	Name of the flow, only valid in the corresponding application
char *commChannel ...	Information on how to communicate with the a corresponding flow object (shared memory area and access semaphores used by the A-module) for both data and control exchange.
int flowId	Flow identifier (scope on the local Da CaPo system), not visible to the programmer.
int sessionId	Session identifier the flow belongs to.
int type	Type of the flow (audio, video, data)
char *sync	List of all other flow identifiers it is synchronized to, and information on the kind of synchronization
int status	Status of the flow: [NOT_CONFIGURED | CONFIGURED | MODIFIED | READY_TO_START] MODIFIED checks if new application requirements were set for the flow and READY_TO_START checks that there is an upcall function for each receiving flow (if not, a start cannot be executed).
“Protocol graph access”	The modules being part of the corresponding protocol graph are made accessible through this information.
“QOS parameters”	All application requirements (or default values). These are made available for each Da CaPo core component.

TABLE 8. Interface to the Internal Management Tables

Function	Description
LAPI_NewAppEntry(char *appName, char *securityInfo, int *appId)	A new application is registered. The appId is returned
LAPI_NewSessionEntry(char *sessionName, int appId, int *sessionId)	A new session entry is created in the session list. The sessionId is returned.
LAPI_NewFlowEntry(char *flowName, int flowType, int appId, int sessionId, int *flowId)	A new flow entry is created. The flowId is returned.

TABLE 8. Interface to the Internal Management Tables

Function	Description
`LAPI_AddReq(int flowId, char *reqName, char *reqValue, char *weightFunc)`	`A new application requirement is appended to the flow. The status of the flow is set to MODIFIED.`
`LAPI_Get*By*(...)` `e.g., LAPY_GetAppIdByName(char *app-Name, int *appId)`	`Generic retrieval functions to access stored information (the desired functions are defined by the needs of Da CaPo core components.`

structure are further detailed in Figure 4 (NOTE: in the current section, send and receive always relate to the A-module's view and refer to interactions with the upper API).

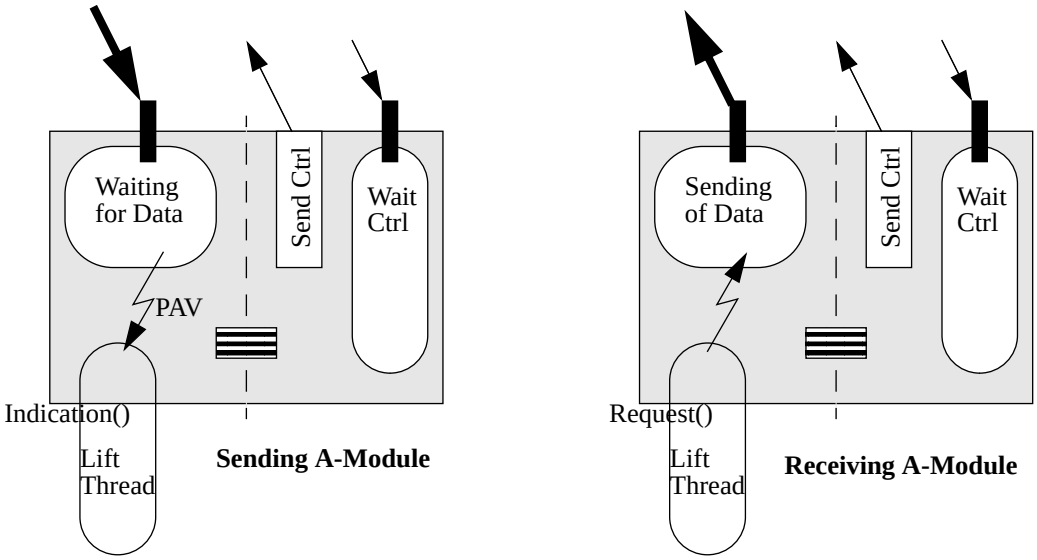


Figure 4 Internal Structure for Sending/Receiving A-Modules

5.3.2.1 A-Module Structure

As illustrated in Figure 4, a clear separation exists between "normal" data and control processing parts inside an A-module. This separation was made necessary due to the properties of the current lift algorithm (implemented in its own thread).

As several threads coexist and may access to A-module resources, special care must be taken during the concurrent implementation of an A-module (mutual exclusion and synchronization mechanisms may be necessary).

5.3.2.2 Sending or Receiving Data

In the case of a sending protocol, the lift thread invokes the indication procedure of the A-module to get data from the application (more precisely from the corresponding upper API Flow object). In this indication procedure, it is not possible to implement a blocking read on a communication channel (this would block the lift, preventing modules such as IRQ or sliding windows to perform their task).

As illustrated on the left side of Figure 4, a new thread is started to wait for incoming data from the application. On receiving data, this thread issues a packet available notification (PAV) to the lift algorithm and then the data is propagated in the module graph through the A-module's Indication() procedure.

5.3.2.3 Sending and Receiving Control Information

For convenient reasons, it was decided to separate the A-module in two distinct parts to process both data and control information (this makes it easier the processing of high-priority control information).

As sending/receiving of control information occurs in each A-module, the control parts for a sending and a receiving protocol are identical.

As for the data processing, a thread waits for incoming control information. On receiving such a control request, the necessary processing is performed in this thread.

5.3.3 Session Data Synchronization Component

This component illustrated in Figure 1 on page 8 is used for sending notifications to the upper API when information (either control or data) is set on the communication channels between upper and lower APIs. The main reason of this design choice is to reduce the number of waiting threads in the upper API. There is one such component for each session.

The corresponding component in the upper API is the ReceiveManager that acts as a demultiplexor to call the upcall functions for each flow.

5.3.4 QoS Mapping

Da CaPo offers a set of predefined attributes as throughput, delay, delay jitter, error-rate,... These basic attributes are however insufficient to characterize the behavior of some applications. In a video application context, the most significant Da CaPo attribute would be the throughput, though it is likely a programmer (or a user) would rather speak in terms of frames per second, color depth and image size. Actually the effective throughput can be computed by a combination of the three latter values (compression is ignored).

This operation is called QoS mapping and can be performed in each A-module. Each A-module has the necessary mapping functions to translate specific application requirements to its type (e.g.. video, audio or data). All original application requirements are stored in the flow table in lower API (see in Table 7). It is then the task of the A-module to process this application requirement[1].

5.4 IPC Mechanisms and Protocols

The goal of this section is on the one hand to give implementation details on all communication channels that are mentioned in the previous sections. On the other hand, it provides a description of the interactions between all components through “event-time diagrams” (?).

5.4.1 Communication Channels

All communication channels of Figure 1 on page 8 are now separately considered:

- Main control connection to Da CaPo core system (1)

  This connection is used when registering a new application and a new session. It is also used for sending general control information such as connecting, configuring, starting, stopping and closing a session.

1. A check has already been performed in the upper API to see if this application requirement can be interpreted by the corresponding A-module (this avoids for example to set a frames per second requirement for an audio flow)

It is implemented with a connection oriented UNIX domain socket[1]. This connection is not permanent, and has to be set up by each client application before sending data. Moreover the DaCaPo Server can only consider one client application at a time. The connection is closed by the lower API after the client request has completed and a return value as been returned to the application.

- Sending of notifications to the application (2)

  This connection is set up if the application could properly register. There is one such connection for each application and it is thus permanent. It is implemented with a connection oriented UNIX domain socket.

- Sending OR receiving of data to OR from the A-module (3)

  For efficiency, this connection is implemented with shared memory synchronized by 2 semaphores (for reading and writing). The size of the shared memory area has to be fixed during session configuration.

- Sending AND receiving of control data to AND from the A-module (4)

  These connections are implemented separately as 2 shared memory areas synchronized through 4 semaphores (same mechanism as for (3), but for each direction).

- Session data synchronization (5)

  Due to performance reasons, this connection is also implemented through shared memory and semaphores synchronization. Due to the small size of the transmitted packets (just the necessary information to specify which flow is concerned), one could also imagine a UNIX domain socket. This connection is permanent and one instance exists for each session.

NOTE: All shared memory based communication channel access functions are non-blocking. Sending data simply means writing data in a free buffer. As the access to these buffers is managed with semaphores, it can occur that a buffer is full. In this case, either the size of the buffer is automatically adjusted, or the writing process returns immediately with a failure code, or it can wait until free memory is available. This depends on the policy and can be set during session configuration.

5.4.2 Upper - Lower API Protocol

For all functions that build the API, the interactions between upper and lower API are presented in the following subsections (for some of them through time-sequence diagrams).

5.4.2.1 Creation of a DaCaPoClient Object

The process of creating a DaCaPoClient is illustrated on Figure 5 The parameters for the creation of such an object are all security relevant data concerning both the user and the application.

The DaCaPo Server just forwards this request to the Security manager which decides whether the new application can be authorized. If the answer is positive, then the DaCaPo Server updates all internal tables and sends a positive return code to the application.

After successful completion of this object creation, an event connection exists between the lower and the upper API. The DaCaPo Server releases the control connection and is ready to process another application request.

5.4.2.2 Creation of a Session Object

A session object cannot be created before a DaCaPoClient was successfully created. To reflect this, a reference to the DaCaPoClient object has to be given as a parameter to the Session object constructor.

1. The choice of a UNIX domain socket is justified as client and server processes are on the same machine. However for more specialized services (e.g., security queries, ...), it may be desirable to connect from another machine. This would require an INTERNET domain socket.

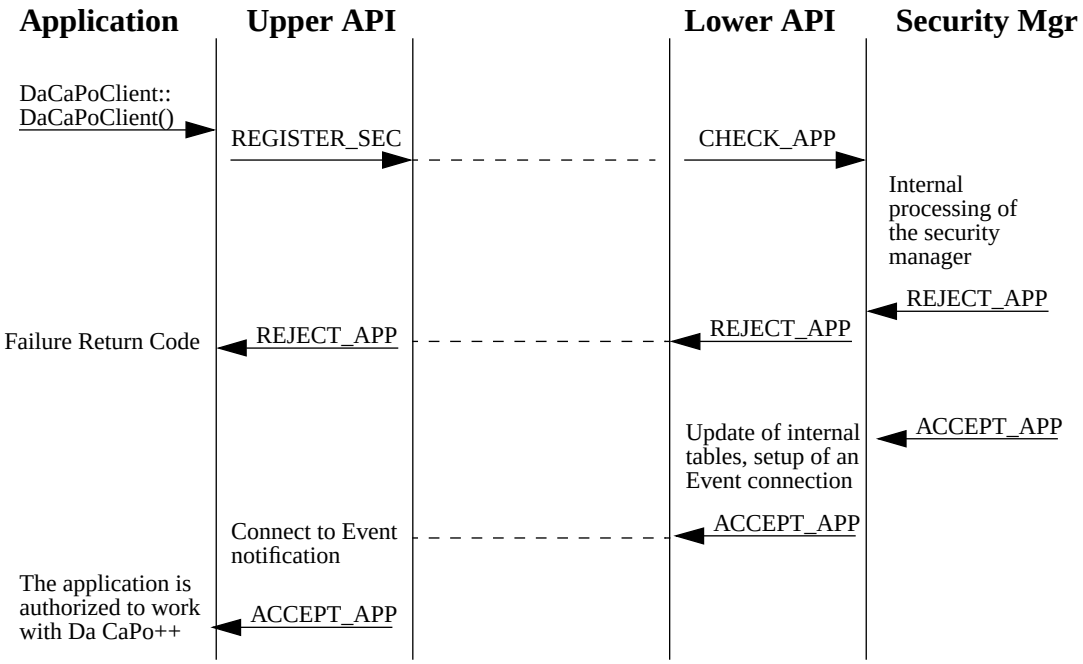


Figure 5 Creation of a DaCaPoClient Object

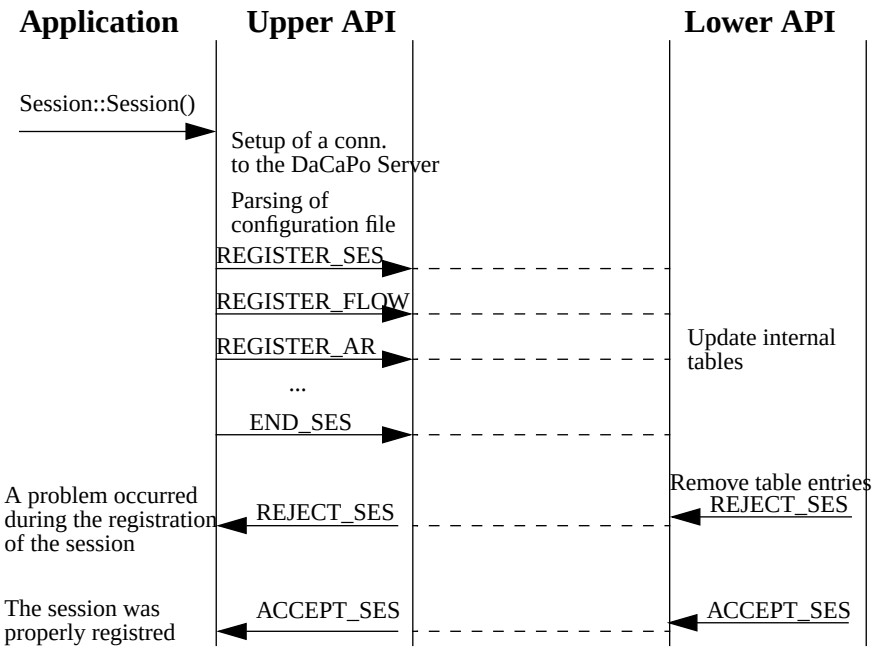


Figure 6 Creation of a Session Object

Over the DaCaPoClient object, a connection to the DaCaPo Server is set up, then a component in the upper API parses the configuration file and sends requests for registration of the session and of all flows (including the application requirements). The DaCaPo Server processes all incoming requests by updating the internal tables (the current application's status is ACTIVE, the session's status is NOT_CONNECTED and all flows' status are set to NOT_CONFIGURED).

By successful completion of the object creation, all session and flow structures are ready to receive a connect() from another application.

It was decided to split the object creation process (through C++ constructors) and the connect process. The object creation only involves upper and lower APIs, without any interaction with the network. On the other hand, the connect process involves remote applications.

5.4.2.3 Connecting a Session

The connect functionality of a session is a complex process which depends on the type of the session (either CREATOR or PARTICIPANT) and on the type of the desired connection (either UNICAST or MULTICAST). The unicast case will no longer be considered as it assumes that both partners know themselves (this solution is already implemented in the current Da CaPo system).

The main difference between a creator and a participant is that the creator waits for incoming join requests from a given set of participants[1]. The communication can then be started as soon as all participants have connected, or only a subset of them. During the connecting of a participant, the address of the corresponding creator must be known.

The API's Connect function cannot afford to be blocking (this would be done at the expenses of other client applications which may want a connection to the DaCaPo Server). Thus, the effect of a Connect is to send a notification to the connection manager and to update the status of the session in the lower API table. A return value is then sent to the application and the control connection is closed (the DaCaPo Server is ready to accept other client application requests).

This means that the connection manager must then be able to process remote participant's join requests and to update the session status in the lower API table. Finally, when a Start() is invoked by the application on this session, the status value in the session table is checked and if the session is not in the CONNECTED state, the Start() fails.

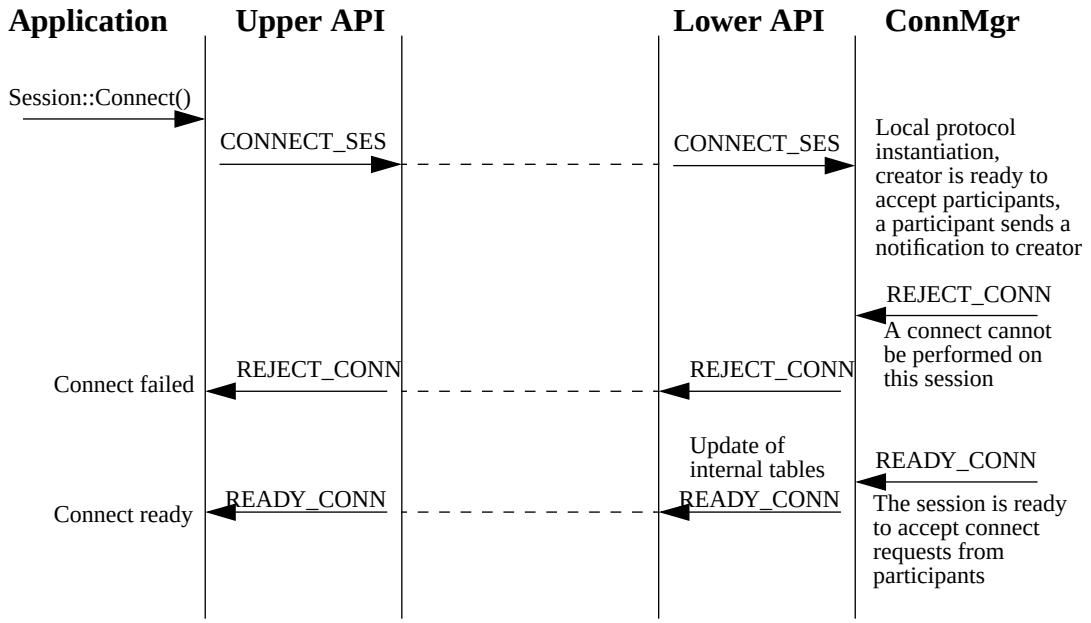


Figure 7 Connecting a Session, for both CREATOR and PARTICIPANT

5.4.2.4 Configuring a Session

Issuing a Configure() on a session is only required when the default configuration performed in the Connect() is not sufficient for the application's needs. Both initial configuration and further re-configuration are performed with this function call.

1. This set of participants can also be empty, in this case, the creator is free to accept or reject any participant

As it is unnecessary to re-configure a flow if no modification such as new application requirements occurred, the flow status is considered (if a change occurred, the value MODIFIED is set).

Configuration involves not only the computation of a new protocol graph, but also the transfer of the new graph to all other remote Da CaPo sites (through the connection manager component). Thus, between CREATOR and PARTICIPANT, and unicasting or multicasting, some restrictions regarding the use of this configuring functionality are set (cf. connection manager and multicast design documents).

As for the Connect() function, the Configure() function must not be blocking.

5.4.2.5 Activating a Session

The lift threads for all flows of the session are activated (e.g., they start either sending or receiving data).

Before starting a session, the upper API checks if upcall functions have been properly set up for each flow (2 upcall functions for the receiving flows where data is received by the application, only one in the other cases).

Then the lower API checks if the session was properly connected (by consulting the internal tables).

If all conditions are met, then the local activation of the lifts can be started. Before any data transfer can take place, the remote site(s), either CREATOR or PARTICIPANT, must also have issued a Start(). As the Start() cannot be blocking, the lower API returns an acknowledge stating that the start process has been properly registered. Then the application, through its upper API's component DaCaPoClient, could be blocked, waiting for a remote start event. When implementing this solution, additional functionalities must be provided to the application to break the Start() function if no participant issues the corresponding Start().

5.4.2.6 Deactivating a Session

All lift threads for all flows of a session are stopped. This deactivation can be either graceful or graceless.

In the graceful case, the sending A-modules no longer accept any new data coming from the application, but the data in the protocol graph is properly sent to its peer (without data loss).

In the graceless case, the lifts just stops, discarding data that was processed in the protocol graphs.

In both cases, the peer application receives a deactivate notification that the session was deactivated.

5.4.2.7 Closing a Session

Only a deactivated session can be closed. The peer application receives a remote close notification. All local resources are released.

5.4.2.8 Getting a Flow Descriptor

As flows are no longer directly accessible, the access is performed by the flow name through the session object. To avoid having to compare each time the flow name with all session's flow names, a similar mechanism to the UNIX file descriptors is provided.

This operation only involves the upper API, thus, no IPC communication has to be set up.

5.4.2.9 Setting of New Application Requirements

An application can change its application requirements at any time, this is performed by establishing a connection to the DaCaPo Server (implicit over the DaCaPoClient object), and by sending all new requirements. The DaCaPo Server then updates its internal tables and sets the status to MODIFIED for all modifies flows.

5.4.2.10 Getting of Configured Attributes

After configuration, the attribute values are not likely to be exactly the same as those requested in the application requirements. A call to this function for a set of application requirements will return their actual configured values.

5.4.2.11 Sending Control or Data Information to a Flow

Each flow can directly address its corresponding A-module, for both data and control information, independently of any other component. This operation does not directly involve the lower API, as data is simply written on the corresponding shared memory area.

5.4.2.12 Receiving Control or Data Information from a Flow

The goal of this function is to provide the upper API's ReceiveManager object with the necessary upcall functions to process incoming data.

5.5 Buffer Management

As the buffer management will not be part of the first implementation, this section delivers a vague idea initially.

As already mentioned, the circular buffer which is used to transfer data from the application process to the Da CaPo A-modules (or vice versa) is also the core of the memory management in Da CaPo. Storage is thus allocated for each data packet in the upper API (or directly in the application if it is reliable enough) and then released when the packet leaves the T-module (this is valid for the sending direction, in the receiving direction, allocation is performed at the T-module and release in the upper API).

5.5.1 Structure

As illustrated in Figure 8 on page 23, the buffer management consists of a shared memory area. This buffer is structured in cells, which have the maximal size a packet of the corresponding flow can reach (this should be a property of the flow). The initial number of cells in the buffer is set up during protocol configuration, but it can always be adjusted if necessary.

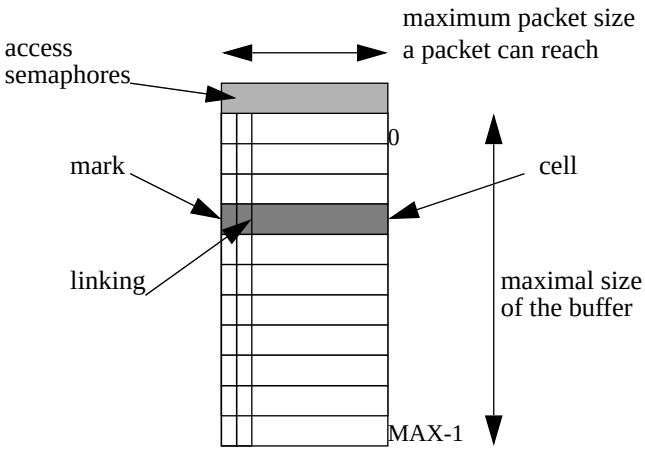


Figure 8 Buffer Management Structure

Two additional fields may be useful:

- mark:

  For an IRQ module, it is not possible to deallocate the data before receiving the corresponding acknowledge. In this case, the buffer is marked and cannot be removed when the corresponding data reaches the T-module

- link:

  To perform effective data compression, it may be necessary to gather several packets and to compress then a larger amount of data (compression ratios are likely to be better with more data). In this case, some cells can be linked to form a single large amount of data.

5.5.2 Functionality

The functionality of this buffer management is illustrated in Table 9.

TABLE 9. Buffer Management Interface

Function	Description
`char *allocateCell(int bufferId)`	`One single cell is allocated, returns a pointer on this cell`
`int freeCell(char *cellPtr)`	`The cell is released and can be allocated again for a new packet`
`int markCell(char *cellPtr)`	`The cell is marked, and may thus not be deallocated`
`int unMarkCell(char *cellPtr)`	`The cell is unmarked and may thus be deallocated`
`int linkCells(char **cellPtr)`	`All specified cells are linked to build a single large packet`
`int adjustBufSize(int bufferId, int delta)`	`The size of the buffer is modified`

Knowing that buffer may be allocated in the upper API (or application) and in Da CaPo kernel system, it is necessary to provide 2 interfaces for both programming languages C and C++ for some buffer management functions.

6. Security Aspects of Da CaPo++

Securing Da CaPo communications is achieved by defining protocols that include encrypting and authenticating modules. Depending on the security requirements that the application specifies, the configuration process will employ these modules, taking into account security which might be provided by lower level transport infrastructure. A static key and certificate database allows for the application-independent storage and recovery of public keys and related information. The actual control of security in Da CaPo is done by the Security Manager, which consists of several building blocks. This Section does not mention the assumptions related to trust that have been taken to allow for a realistic approach, as they have been already discussed earlier.

Four different areas are covered. First, users have to identify themselves to the Da CaPo core, and have to prove their identity. Second, applications that want to use Da CaPo in a secure fashion have to be identified and authenticated by Da CaPo. Another important area is the machine-machine authentication that allows two Da CaPo endsystems to communicate in an authenticated and secure manner even if no 'security aware' application or end-user is available. Finally, the fourth area covers the actual encryption and authentication of data that is transmitted over an unprotected network infrastructure.

The second and third area may actually be coalesced into one if user authentication is done through the application. Such behavior is not encouraged, as it leads to the necessity of a multitude of 'logins' for the user. The four areas show different behavior depending on whether a delegation of the respective identity to the Da CaPo system takes place. For the sake of simplicity, this is assumed to be the case throughout the following.

6.1 Fundamental Assumptions

To limit the complexity of this project, and allow for a reasonable result, severe limitations have been taken into account. This is not a complete security framework for applications, but only some excerpts from it that are related to communication issues. The following points elaborate on the particular axioms that have been defined for Da CaPo++:

1. Hardware (Computer and Firmware)
   The available hardware is assumed to be fundamentally secure. No possibility exists to detect or correct the non-fulfillment of this requirement. It is as incontrollable as a TEMPEST attack, or a bug in the keyboard that broadcasts keystrokes. The critical aspect concerning hardware is the fact that verifying computations are run on it, contrary to the 'passive' operations of a network. There exists no instance that can detect or control a maliciously misbehaving hardware.
2. Network
   All transmissions of data (or programs) are always assumed to be insecure (see below for exceptions). To counter this insecurity is the ultimate goal of the current project. Some of the attacks against which the communication has to be protected are: Man in the middle, replay of data, recovery of encryption and long-term keys, and so forth. Attacks leading to denial of service are not controllable in this environment, as the underlying communication infrastructure defines the communication paths. Transmissions that have to occur in a secure mode are: Public keys of participants (or the public key of a certifying authority), certificates for the operating system and the Da CaPo core, and addresses of trustworthy or certifying instances. This information has to be verified by out-of-band communication.
3. Operating System (Kernel) & Base algorithms
   Similarly to the underlying hardware, the operating system has to be available in not tampered form (loaded freshly from a CDROM just taken out of the safe). On the same CD, routines for the verification of certificates should be available. Again the same argument as above holds true: These are the means which allow the verification of correctness (integrity and authenticity). This excludes the possibility that they verify themselves or each other, if trust is not there in the beginning.
4. Da CaPo core system
   The core system has to be available, as it is the means for communication. The 'correctness' of the core system can be verified using a certificate, by using aforementioned cryptographic-routines. When the certificate verifies correctly, this only proves that the core system has not been tampered

with in the meantime. There still might be an error in the software, or a backdoor, which has been integrated before the certificate has been produced. This also reflects the amount of trust, that a user of a package has to invest into the certifying authority of it. Some of the questions a user should ask himself are: Can the certifying authority protect its own private key? Can the authority be trusted? If not, anybody can fake a certificate, or the authority can issue a certificate without knowing that the software is bogus. Or it could even issue a certificate for software that it knows to be bogus - for reasons of its own.

5. Da CaPo Modules and Applications
   The arguments of point 4 are valid. Components that are directly used or run by the core system, can have their certificate validated by Da CaPo itself.
6. Certificates and Key Data
   Certain key information (private key of the machine, is possible certified by the users of the machine without giving the users access to it) has to be available at runtime. The user can verify the correctness by checking his own certificate on the signature, and using a challenge- response protocol. The same holds true for certificates of applications. They can be 'insured' either by the machine key or some user keys.
7. User
   The user has to be able to access his own private key. This key can be stored on the machine itself (possibly encrypted) or on an external device such as a chipcard. External hardware (such as SecureToken or AuthentiBox) allows for the key to be indirectly accessible only, and limiting the window of opportunity for attackers.

6.1.1 Assuring Authenticity: Associations and Identities

Before employing a secure communication system, the participants have to be securely identified and their 'output' must be attributable to them in a reliable fashion. This assumption ignores issues like frequently changing identities and desires for anonymity, but is only relevant if authentication is required. In an extreme scenario, all trusted parties that are involved have to be mutually authenticated. These parties consist of the end-systems on which Da CaPo++ is running, the users involved in the communication, and/or the applications actually producing and consuming the data.

In the model employed by the Da CaPo++ communication system these three identities are ordered hierarchically. If no user authenticity can be provided, application authenticity, and failing that, machine authenticity will be provided. The instances participating in the communication can express their minimal requirements, and are notified upon connection establishment with whom they are actually communicating.

Before a communication can actually progress, users and/or applications involved are required to delegate their identities to the Da CaPo core system so that the core can authenticate data on their behalf, and prove their identity to the peer. This mainly consists in giving a secret to Da CaPo++ with which identities can be authenticated. The given secret need not be the secret that was originally employed to prove identities, and may be usable only by Da CaPo for a limited time span and/or for a limited amount of authentications. The typical case (and the one provided) will, nevertheless, be a full delegation.

Public key values and user/application identities are stored in a global key and certificate database, where application identities consist of arbitrary (but structured) strings identifying them, and user identities may consist of a string containing RFC 822 E-mail addresses, bank account numbers, or any other kind of mutually accepted identifying information. Machines are identified by the address on which they are reachable in the transport infrastructure that is used to establish the connection.

The 'association block' in the Security Manager of the Da CaPo++ core system (cf. Paragraph 6.2.6.1) will verify identities, and note which protocols are associated with which applications and users, communicating this information to other endsystems, if needed and allowable. The Da CaPo++ user interface (which is used for user authentication in the first place, if not done via individual applications) can be used to force modifications in these associations, *e.g.*, if a user wants to force an immediate dissociation from an application which turned byzantine. It is to be remarked that the user interface is an applica-

tion like any other, which just holds special knowledge of the inner workings of Da CaPo and communicates with the security manager through the standard API.

6.1.2 Specifying and Translating Security Requirements

To express privacy and authentication requirements, applications have to pass attributes to Da CaPo++. The attributes are hierarchically ordered in a generic sense, and may consist either of discrete values from a set of possibilities, or specify a range of acceptable values. The attributes specifying security requirements are generally handled exactly like any other QoS attribute. This allows to employ the standard attribute passing and protocol configuration mechanisms of Da CaPo for the building of secure protocols (see also Paragraph 6.2.4). Special treatment is scarcely needed, *e.g.*, for an attribute containing keying material or connection setup authentication requirements.

The security requirements needed for the configuration of secure protocols span a very wide range. To allow for a more transparent (and algorithm independent) handling in the application, the concept of requirement translation is introduced in Da CaPo++. An application specifying only generic application requirements (AR) will accept the defaults that the translation mechanism concludes as being corresponding concrete QoS parameters (PAR). An application may still specify as many detailed parameters as wanted, but may thus create a set of requirements which the system can not fulfill. In that case, no communication can be established. The results of such a translation depend on the available algorithms and machine power, and on the state of the art in cryptography. If the translation process is kept up to date, and the application uses generic security requirements, they will ‘support’ adequate cryptographic mechanisms not only at the time of creation, but in the future too.

The ‘attribute block’ doing the translation actually resides in the A module of each ‘secure’ protocol, and receives the AR by the way of the lower API, together with the non-security related attributes. As the attributes are not parsed by the API, but passed on transparently, no extension thereof is needed for new attributes.

6.1.3 Protocol Management, Reconfiguration and Keying

A secure protocol, which has been configured, includes modules performing cryptographic operations. These may be of symmetric nature, *e.g.*, DES, IDEA, RC4 for encryption, and MD5, SHA for authentication support, or asymmetric, *e.g.*, RSA, DH or El Gamal. Additionally to traffic encryption and authentication, they will allow for key exchange if rekeying is an issue and may allow for the receipt and processing of tokens providing sender- and receiver nonrepudiation functionality.

The ‘key management block’ (see below) of the security manager provides access to the database containing public and private keys, as the generation of authenticated keying material has been delegated to the communication system. Key changes in the running protocol can thus automatically take place, the ‘protocol control block’ of the Security Manager does ‘asynchronous’ key changes. The only way for an application to change the properties of a secure protocol is to initiate a reconfiguration.

On the other hand, if the application has chosen to provide keying material (as will be possible later), a reconfiguration of the protocol is necessary. This may be a very cheap process (called small reconfiguration), if the change does not require different functionality, *e.g.*, when the application chooses not only to change traffic keys, but likes to change employed algorithms also.

6.1.4 Security Assurance at Runtime

The configuration process provides secure protocols, if working correctly and receiving requirements from the application that do request this. At a later point of this project, the ‘security assurance block’, located within the module instantiation process, can verify the compliance with requirements, by checking precalculated and certified module properties, the integrity of the employed modules themselves, and the validity (in terms of security requirements) of the created protocol. The same holds if an unilateral reconfiguration downgrades a communication protocol. The communicating peers (Da CaPo++ systems) will be notified of the reconfiguration that took place and when instantiating the new module graph, have

to decide if they accept this change. If they can not accept the change under the current requirements, they will have to notify the application to change the requirements or refuse the change.

At runtime, the ‘security assurance block’ monitors the usage of keying material, and takes track on how much data, and for how long a traffic key has been in use. A special event will be issued to the ‘protocol block’ when this happens, and is sent to the application if rekeying is necessary. This is another aspect of runtime security assurance.

6.1.5 Keys and Certificates

The key and certificate database is used to store certificates of modules and their properties, public and private keys of users, applications and machines, and is accessed by the ‘key management block’ to provide keying material to various parties. In the case that private keys are stored in this database, they are encrypted by a user- or application- supplied passphrase prior to storage. This protects against abuse of the keys stored in that database.

6.2 Discussion of Components

Introducing security has an impact on nearly all parts of the Da CaPo core system. This section identifies the parts that will be created/modified, and states coarse assumptions on data structures and points of interaction. The elements are:

- API
- QoS Parameters
- C-Modules
- Protocols
- Key Database
- Security Manager

Additionally, the connection manager may be changed to use a protocol that insures privacy. How this is to be done will have to be evaluated after modules, protocols, and parts of the security manager (and the existing connection manager) have consolidated. These changes will mainly result in the connection manager using a different (secure) module graph.

The system architecture is presented in Figure 9 Secure IP represents a transport infrastructure that already offers security. Future T modules will have the intelligence needed to understand the existence of such a service, and will thus optimize the protocol configured by the core system.

6.2.1 API

The API will transparently forward application requirements which an A-module can translate into QoS parameters. See XXX for a list of AARs. Additionally, the API handles the identification/authentication issues, and provides for a way to forward events to the application. A transparent ‘control channel’ will be available, through which the user interface or other specialized applications may communicate with the security manager, *e.g.*, to access the key manager for the purpose of generating, storing, retrieving, certifying keys and certificates.

The upper API has to process and forward the following information upon establishment of a controlling connection between core and application: local user name, process id, global user name, global application identifier, user key ID and passphrase. This is passed on to the security manager (association block) by the lower API and verified there (see 6.3.2 for details). Afterwards the lower API receives a clearance or denial from the security manager and acts accordingly. As secure protocols and keys can be defined and changed using the generic (re-)configuration mechanisms, no addition to the API is needed for this purpose.

For end-to-end authentication with non-repudiation, the API offers two sets of functions, which allow this extended security protocol to perform using slightly different semantics. When a flow is created or reconfigured to use a receiver-non-repudiation protocol, a proof of receipt will be generated for each received message. Below the API, a message may be limited by arbitrary bounds, defined by START/STOP pairs in the control flow. For the application level, the concept of a ‘message’ has to be introduced, or, alternatively, synchronous end-to-end authentication/return receipt requests can be initiated by one of the peers. Although continuous media has no fixed boundaries, they can be added artificially by the application, *e.g.*, by requesting a return-receipt after each frame, or each second.

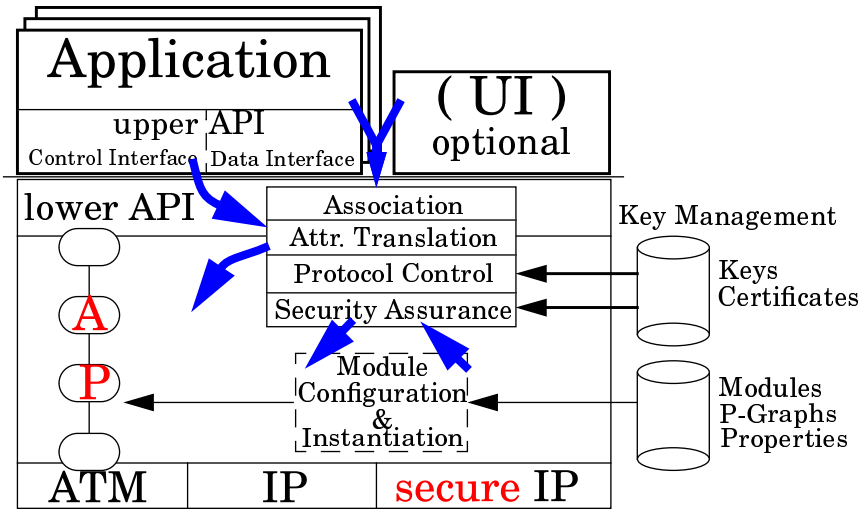


Figure 9 Security Architecture in Da CaPo++

6.2.2 QoS Parameters

The following three classes of QoS parameters (also named application requirements) exist:

- Abstract Application Requirements
  These are algorithm independent name-value pairs which define abstract properties. They may be of quantitative or qualitative nature, and are mainly used by the attribute translation to derive the low level requirements.
- Lower Layer (low level) Requirements
  indicate concrete algorithms, and specify parameters pertaining to them. They are either derived from the abstract AR through the attribute translation process, or directly specified by the application.
- Peer Authentication Requirements
  specify what minimal authentication is to be achieved with the peer side, such that a connection can be established successfully. The actual identity of the peer side has still to be passed up to the application, such that admission controlling application-dependent authorization can be performed.

6.2.2.1 Abstract Application Requirements (AAR)

These requirements pertain to three different classes. They describe the (algorithm independent) properties to be used for encryption purposes, for data authentication, or they provide access to per-protocol keying material and policy. The algorithm independent representation of encryption is name-value pair which expresses a quantitative amount of security to be achieved.

```
attribute abstract privacy = (NONE, 0.1, 0.5, 1..100, max.)
```

This security is expressed by assessing the strength of an algorithm to be used as currently known, and estimating the amount of years it would take an enemy to break this particular transaction, assuming he invests one million dollars per year and takes the best possible non-invasive approach at breaking it. It is obvious that this assessment depends on rapidly changing data, and the corresponding translation tables in the core system will have to be adapted regularly, to take newly discovered weaknesses of algorithms and price development of components into account.

At the same time, the encryption protocol can be optimized for the type of data to be transmitted. These are implicit preconditions which are evaluated by the protocol configuration process itself. The various preconditions could be:

```
preconditions set of values = JPEG, MPEG, H.261, CellB, ULaw, G.721,
Wavelet, error-free, ordered, none
```

This allows the deployment of encryption algorithms that are adapted to a specific kind of data, taking advantage of intimate knowledge of the inherent semantics to achieve a cost reduction for the encryption or authentication process. The currently supported preconditions are ‘ordered’ and ‘none’, as no special protocols have been designed up to now. Depending on above preconditions different modes of partial encryption/authentication are possible:

```
attribute partial processing = spacial axis, time axis, embedded con-
trol data only, variance mode
```

Here, variance indicates that only areas with ‘significant content’ are to be encrypted, e.g. areas in am image where ‘something happens’. For authentication purposes only, additional AARs exist. The first set defines the behavior of the protocol in respect to the non-repudiation issues:

```
attribute authentication type = none, symmetric, asymmetric, receiver-
nonrepudiation
```

Where ‘asymmetric’ is a synonym for sender-nonrepudiation mode. The actual signature from the sending side and/or proof of receipt are passed on to the respective applications via the event mechanism. (see there).

Again only for authentication, the following optimizing attributes may exist, depending on the valid precondition set of values:

```
attributes partial authentication = per frame, per n frames, Nth frame
only
```

They impose a sense of ‘transaction’ onto the underlying data stream.

```
attributes set of values = delay increase, CPU consumption increase,
throughput, blocking factor (delay jitter)
```

Above names represent application-settable attributes which usually define limiting values to be used by the whole protocol. Naturally, the cryptographic part of the protocol influences these values, and is at the same time constricted by the application requirements fixing resource consumption.

Keying material (*e.g.*, shared secret provided to sending and receiving side is provided by the protocol control block of the security manager. As an application may specify which public key may be used for this particular protocol, this results in application requirements that actually transmit data to the peer side:

```
attribute keying name = public key ID
```

Currently, the per-application keys are used to set up a shared secret between the communicating parties.

```
attribute keying control= keysize, rekeying interval, rekey volume
```

Finally, these attribute define the actual behavior concerning change of session keys.

6.2.2.2 Lower level requirements (LLR)

As the name implies, these requirements specify or depend on very concrete algorithm specific behavior. Applications accessing the LLR can influence the behavior of the core system concerning security very directly at the risk of not being always up-to-date on the actual strength of an algorithm. They are represented in tables in Paragraph 6.2.3.

6.2.2.3 Peer Authentication Requirements (PAR)

While the application requirements concerning the data transfer in itself are specified by the receiving and the sending side, PARs do not currently exist. The data transfer properties are actually checked on runtime, and compared to the application requirements. As soon as the specified limits are surpassed, the monitor is assumed to produce an appropriate event.

They are realized in the application itself, respectively in the upper API. Before accepting an communication peer, its authentication method (and identity) are passed up from the core system to the application. The application can then decide if the authentication method that was used is sufficient, or if the access is denied because end-to-end authentication has not been adequate to its requirements.

6.2.3 C-Modules

The provided C modules for introducing security into the data transfer are:

- ECB: Provides for DES, IDEA and RC5 in electronic codebook mode
- CBC: Same in cipher block chaining mode
- CBC_ORDER: Same, but depends on ordered and loss less data transfer
- RC4: Stream cipher encryption module
- MD: Provides MD4 and MD5 message digest algorithms
- DS: Provides asymmetric (RSA) and symmetric MAC (message authentication code) for the signature of a message
- DH: Provides Diffie-Hellman for establishment of a shared secret at runtime (ephemeral traffic keys), is used by the other cryptographic modules.

They behave like normal Da CaPo modules, although they use advanced features like intra-module communication, with one exception. For the purpose of internal rekeying, and user driven (not application driven) security control, they have an additional interface directly linked with the protocol control block of the security manager, and announce themselves to the association block on initialization.

The following table describes their attributes, which are settable as lower-layer requirements:

TABLE 10. List of Modules and Respective Parameters

Module Name	Parameter	Values	Default
ECB CBC CBC_Order	"CipherType"	*"des"* *"des3"* *"idea"* *"rc5-r/b"*[a]	*"idea"*
	"KeyChangeInterval"	-	*"100"*
Stream Cipher	"RC4_KeyLen"	-	16
	"KeyChangeInterval"	-	"100"
Message Digest	"MDType"	*"md4'"* *"md5"*	*"md5"*
	"MAC"	*"yes"* *"no"*	*"no"*
	"KeyChangeInterval"	-	*"100"*
Key Agreement	"dh_bitstringlength"	*"[new] size"*[b]	-

TABLE 10. List of Modules and Respective Parameters

Module Name	Parameter	Values	Default
Digital Signature	"DSType"	*"rc4" "rsa"*	*"rsa"*
	"SignInterval"	-	*"10"*
	"RC4_KeyLen"	-	*"16"*

a. r: Number of rounds (0, 1, ..., 255);
b: Number of bytes in the secret key (0, 1, ..., 255)

b. *new:* Diffie-Hellman shared secret is calculated anew;
size: Number of bytes for the shared secret

6.2.4 Protocols

The provided protocols will be (with more A-modules):

- Encryption (protocol function: Privacy)
  Other possible protocols are structured along the same lines as the one below, which represents a test protocol for the simple encryption-only module, setting a packet generator in top of it, and placing it over a UDP T-module. By adding this or other required protocol functions to any given protocol, video, audio or generic data transfer protocols can be converted to employ security.

```
db_RegisterGraph (&(DbGraph) {"DummyCryptoService", 4,
     (DbNode [4]) {
          (DbNode) {"pfDummy", NULL, "mcSingleDummyPacket", 1, 0},
          (DbNode) {"pfPrivacy", NULL, "mcCBC", 2, 0},
          (DbNode) {"pfKeyAgreement", NULL, "mcDH", 3, 0},
          (DbNode) {"pfTransport", NULL, "mcUdpSocket", 0, 0},
     },
});
```

- Authentication (protocol function: Auth)

```
db_RegisterGraph (&(DbGraph) {"AudioService", 5,
     (DbNode [5]) {
          (DbNode) {"pfAudio", NULL, "mcAudio", 1, 0},
          (DbNode) {"pfAuth", NULL, "mcMD", 2, 0},
          (DbNode) {"pfAsymAuth", NULL, "mcDS", 3, 0},
          (DbNode) {"pfKeyAgreement", NULL, "mcDH", 4, 0},
          (DbNode) {"pfTransport", NULL, "mcUdpSocket", 0, 0},
     },
});
```

- Encryption+Authentication

```
db_RegisterGraph (&(DbGraph) {"VideoService", 6,
     (DbNode [6]) {
          (DbNode) {"pfVideo", NULL, "mcVideo", 1, 0},
          (DbNode) {"pfAuth", NULL, "mcMD", 2, 0},
          (DbNode) {"pfAsymAuth", NULL, "mcDS", 3, 0},
          (DbNode) {"pfPrivacy", NULL, "mcCBC_order", 4, 0},
          (DbNode) {"pfKeyAgreement", NULL, "mcDH", 5, 0},
          (DbNode) {"pfTransport", NULL, "mcATM", 0, 0},
     },
});
```

- Encryption+full non-repudiation (sender&receiver)
  Equals the above, but demands communication of the received and signed digest values back to the sender, and a communication with the application on both sides.

6.2.5 Key Database

The key database later residing in the GMS (Group Management System) interfaces directly with the Security Manager, respectively its key management block, to provide for public keys upon request, and otherwise keep them in persistent storage. This component is invisible for the rest of the Da CaPo core system although it will be accessible by the application layer through a transparent channel in upper and lower API, if and when access to the database will be provided to the application. The overall features and behavior of the key database are very similar to PGP. The appendix describes the current interface to these key management functions of the security manager.

The planned data structure to be embedded into the GMS database is (to be defined):

```
Certificate::= --snacc isPdu:"TRUE" -- SET {
     certificateName       [0]     GmsObjectName,
     certificAttributes    [1]     CertificateAttributes,
     certificRelations     [2]     CertificateRelations
}
CertificateAttributes ::= SET {
     certificateType       [0]     CertificateType,
     nameType              [1]     NameType,
     validity              [2]     SEQUENCE {
         notbefore         [0]     UTCTime,
         notafter          [1]     UTCTime },
     name                  [3]     IA5String,
     data                  [4]     BIT STRING,
     signatures            [5]     BIT STRING
}
CertificateType ::= CHOICE {
     PGP                   [0]     NULL,
     X509                  [1]     NULL,
     NIS                   [2]     NULL,
     other                 [3]     IA5String
}
NameType ::= CHOICE {
     RFC822                [0]     NULL,
     E164                  [1]     NULL,
     IPv4                  [2]     NULL,
     other                 [3]     IA5String
}
CertificateRelations ::= SET {
     owner                 [0]     GmsRelationName
}
```

It represents mainly a container for various types of certificates, so that the currently used PGP like data structures will easily be integrable.

6.2.6 Security Manager

The concepts behind the Security Manager have been discussed in Subsection 6.1. The functionality can be separated into the following building blocks:

- Association
- Attribute translation
- Protocol control consisting of

  - module rekeying
  - event propagation
  - reconfiguration
  - Key management

The security manager is implemented as a part of the Da CaPo core system which owns its own thread, which may eventually be delegated to lower API. The user interface (connecting through the transparent channel in the API) can induce actions like rekeying, switching security for one particular graph on or off, generally controlling behavior of owned protocols, and will provide for user authentication functionality. The goal is an experimental and prototypical access to the core, to allow for debugging and testing.

6.2.6.1 Association Block

As soon as an application establishes a connection with the Da CaPo core and sets up some secure flows, the security manager needs to know which flow is owned by which application, and for which user the application is running. Flow- and session specific information is collected by the lower API, which passes on a handle to this information and additional authenticating data to the association block. The association block verifies authenticity of the provided information, and allows or denies access. In the case events are generated, keying material is missing or other actions are required, the controlling owner is retrieved via the association block, and the message forwarded via the lower API (cf. Paragraph 6.3.2).

During connection setup, the application sends information (via the upper and lower API) to the security manager, which allows for a reliable identification of the application in question and the user associated with it. For the exact content of each field, see Paragraph 6.3.2. Now follows a brief description of the information that is passed on:

- User ID
  represents the user that controls the application. If the application is a ‘daemon’, or the user wants to stay unknown, no user ID needs to be present, in that case the application assumes an anonymous identity, which is only associated with the machine on which it actually runs.
- User Public Key ID
  If the user wants to authenticate himself to the remote system, and have the session keys that the sender uses encrypted in a user dependent key, he provides the ‘name’ of his key. This corresponds to a PGP surname, or an actual PGP key ID. If no public key (and corresponding private key) is provided, the core system can only claim to the peer system that the user is indeed who he claims to be. Only if the asymmetric keying material is available, then can the system prove the users identity to the other side. For the public key to be usable, it has to be bound to the User ID with a signature that the user on the peer system can trust.
- User Query ID
  The far simplest (and probably most reliable) method to decide whether an application claiming to act on a certain user’s behalf consists in asking the user for confirmation. This can be done if there exists a path from the user to the Da CaPo core system which is not controlled by the application, and where the user can confirm his intentions to the system. A query usually contains all application related data (see below) the claimed identity, arbitrary information, as provided by the application, and additional information that makes the query unique. The response of the user is forwarded to the lower API, causing it to allow (or deny) access to the Da CaPo core system under the claimed identity. The user also may specify which actions the application may perform e.g. reconfiguration of another application, resp. sec on/off. This is useful for applications such as a generic session directory / GUA management application.
- User Private Key (Passphrase)
  An other method for the user to prove his identity is to provide the private key (which matches above mentioned public key) to the core system. Alternatively, only the passphrase, a key to the ‘superenciphered’ private key in the core system key database, can be provided. This method of identification assumes that the user trusts the correctness of the application, and is thus assumed to be weaker than above ‘User Query ID’.
- Application ID
  To identify which application is trying to establish a connection, they have to be provided with unique names. Those names must be generic enough to allow a quick identification of the application, and at the same time must provide for a way to differentiate between versions of the applications, and the operating system they are currently running on (cf. Paragraph 6.3.2.)

- Application Certificate
  This structure contains checksums for various instances (different versions and operating systems) of applications, together with signatures binding the name(s) of the applications to the signature(s). After successful verification of the signature (assuming that the signer is a trustworthy party, and his public key is known to the Da CaPo core system in question), the checksums can be compared with an actual one, that has been derived from the running application.

6.2.6.2 Attribute Translation Block

Attribute translation is only conceptually part of the security manager. It is realized as a set of functions integrated into the A-modules of all security-aware protocols, and needs to understand all application requirements pertaining to the security mechanisms, and which have to be mapped to QoS parameters in this particular protocol.

6.2.6.3 Protocol Control Block

As the name say, the protocol control block handles all security related issues that influence communication behavior. It is the switchboard that receives requests from the cryptographic modules for new keying material, eventually then retrieves key-IDs from the association block, and gives new keying material to the modules.

As the security part of Da CaPo allows for a ‘small’ reconfiguration (*e.g.*, change of keys, switching security on/off), this is done in the protocol control block. If a change in the status is required, protocol control stops the lift, accesses the involved security C-modules via their announced interface, and changes their behavior. They communicate the change to their receiving peers using intra-module communication. The peers forward the event to the remote protocol control block, and on the sending side the lift is restarted.

This may also lead to the generation of an event, if, *e.g.*, the actual key has expired. Other possible events (generated by the protocol itself, or the protocol control block) are described in section 6.3.4.

6.2.6.4 Key Manager Block

This is the access point to the key database. It provides for fetching keys and certificates from the database, generates random (ephemeral) keying material (for the employed method, see RFC1750), and provides it to the C-modules. Although the key manager block currently provides only functionality to the Da CaPo core system, it may be visible to the application via the upper API, and provide for a limited key management functionality. (Retrieval of keys and certificates, storage of new keys, check of signatures, etc.) At a later point in time it will use the GUA, and the key database will disappear.

6.2.7 Runtime Security Assurance

Runtime security assurance will be provided at a later point in time. It will interface with the module instantiation part of Da CaPo++, to verify the correctness (authenticity) of modules. Additionally, instantiated in the monitoring part of Da CaPo++, it interacts with the module graph to check if the actually achieved security conforms to the local application requirements.

6.3 Specification of Interfaces

The following paragraphs elaborate on the extent to which the security components are visible to ‘the rest of the world’ and how they can be accessed. After describing the data structure and procedures used between lower API and security manager, the events issued or filtered by the protocol control block are listed.

6.3.1 Application – upper API

The applications sees some elements of security through the upper API:

- Authentication and authorization upon connection establishment
- Access control for remote connections
- Protocols and attributes
- Change of protocol behavior at runtime
  - Reconfiguration
  - Key change
  - Security on/off
  - Terminate application
- Key Management (Da CaPo database)
- Authentication/acknowledgment (and receipt) of running traffic
- Events

6.3.2 Security Manager – lower API

The security manager understands the following data structures, as provided by the API:

- Connection Setup Data, provided by the upper API upon connection establishment.

```
struct auth_data {
     process_id;
     user_id;
     application-id;
     application_certificate;
     user_public_key_id;
     user_private_key_passphrase;
     user_private_key;
     query_string;
};
```

  The process ID is filled in as a hint by the upper API. A user ID consists of a simplified RFC822 address string <USER@MAIL.DOMAIN>, or one of a set of those. The application ID consists of a string with separated fields, e.g. <name = netscape_navigator; version = 2.0b; platform = sun4m; os = sunos5.4; author = foo@bar>. The user public key ID consists of either an RFC822 style PGP key ID, or the 64 low order bits of the public value. The private key corresponds to a (binary or ASCII) representation of a PGP key. The exact semantics are defined in XXX.

The security manager provides the following access points to the API:

```
void *register_application(void *api_handle, struct auth_data *);
```

Called upon first connect of application to Da CaPo. Validates and stores information, return NULL on failure, a handle on success.

```
void hash_buffer(u_char *buffer, int len, u_char result[16]);
```

Allows the lower API to generate a unique ‘cookie’ for each registered application, used for secure identification of local upper API communication partners.

```
void unregister_application(void *handle);
```

An application has left the Da CaPo system, the corresponding security relevant information is destroyed.

```
int secmgr_request(u_char *buffer, int len);
```

An application has connected to the lower API, and sents a security manager specific request, which is passed on transparently.

The lower API needs to allow the Security Manager the following operations:

```
flow *find_application_flows(void *api_handle);
```

Allows the security manager to find out, which flows an application holds. Gives (indirect) access to the dacapo core protocols an application holds.

```
api_handle *find_application_by_protocol(protocol *foo);
```

This functions tells the security manager to which application a particular dacapo core protocol pertains.

```
int destroy_application(void *api_handle);
```

Immediately aborts all communication from and to this application, sends an ‘security abort’ event to the application, and then effectively forgets its existence.

6.3.3 Security Manager – Connection Manager

The connection manager of the Da CaPo core system is impacted by the fact that there is security in the system in two different ways. First, the connection manager itself must employ a cryptographically enhanced protocol to talk to communication peers. This protocol itself needs to provide authentic and private data transfer, but itself depends not on the authenticity of the participants. If a peer wishes to connect,

6.3.4 Security-related Events

Events are generated by different sources within the Da CaPo core system, and ultimately passed up to the upper API or the application. The monitoring component of the core system compares the actual performance of a system against the application requirements. The protocol control block issues events in the case of problems with keying material, and passes on messages from the connection manager. Here a list of the possible events, and their semantics:

- Rekey request / confirm
  Request is issued if the keying material is provide by the application at a later point in time. The confirmation is sent always when a rekeying occurs, but may be safely ignored by the application.
- Remote Reconfiguration
  The remote peer (or the creator in the case of a multicast flow) initiated a reconfiguration, which completed successfully. As the runtime security assurance provides for a control of sufficient security, this should never lead to fatal conditions.
- Return-Receipt Request/Confirm
  Before actually delivering a return-receipt to the remote Da CaPo++ system, the application in charge of the particular flow will be asked whether it wishes to give a confirmation. If yes, Return-Receipt Confirm delivers the proof-of receipt to the application originating the data.
- Failed Authentication
  Issued if received data is not authentic
- Key Expired
  The certificate in the Da CaPo++ key database expired, and communication has been stopped, pending announcement of a new key.
- Security Reconfiguration
  This is analogous to ‘remote reconfiguration’, but is issued when an user (or another application that holds sufficient permissions) issues a ‘small’ reconfiguration for the security properties of a protocol, e.g. the change of keying material or the enabling/disabling of the processing by the security protocol modules.
- Peer connect request
  Contains the identity of the peer that connected, to be used for admission control by the application.
- Local Abort
  The security manager instructed Da CaPo++ to drop this application. This may be happen through the additional user interface or by another (controlling) application.

- Authentication tag
  Used for receiver-nonrepudiation if no sender-nonrepudiation is needed. The application can then store the tags provided by the sending peer system, to later prove the occurrence of this transmission. At the same time the application is responsible to store the data that is so validated.

6.3.5 Interface of the Current Security Manager

Im folgenden werden die einzelnen Funktionen und Datenstrukturen des Security Manager detailliert beschrieben. Dieser Anhang wurde aus der Systemdokumentation von Urs Hengartner übernommen.

Grundsätzlich gilt für alle Routinen, daß sie im Erfolgsfall 0 zurückliefern. Ist mit dem übergebenen Input etwas nicht in Ordnung, wird -1 zurückgeliefert.

Bei Routinen, die sowohl eine Key ID als auch eine User ID als Übergabeparameter verlangen, muß nur einer der beiden Werte gesetzt wird. Werden beide gesetzt, hat die Key ID Priorität. Ausnahmen von dieser Regel werden bei den entsprechenden Funktionen erläutert. Die Key ID ist ein 17 Byte langer Hex-String (z. B. “A43C01E257347FD1”). Die User ID ist ein beliebig langer String.

Da das API Datenaustausch nur via shared memory, aber nicht via Files vorsieht, werden z. B. vom Key Ring extrahierte Keys in Form eines Speicherblock übergeben und in den Key Ring einzufügende Keys müssen in einem solchen vorliegen.

```
int sm_Login(Dc_String username, Dc_String password)
```

Mit `sm_Login()` kann der Benutzer `username` sein Passwort `password` dem Security Manager übergeben. Die Routine liefert -1 zurück, wenn `username` schon eingeloggt ist. In diesem Fall muß sich `username` zuerst mit `sm_Logout()` ausloggen. Die Funktion `sm_Login()` nimmt keinerlei Authentifikation von `username` vor.

```
int sm_Logout(Dc_String username)
```

Mit `sm_Logout()` wird das Passwort von `username` gelöscht. Die Routine liefert -1 zurück, wenn `username` nicht eingeloggt ist.

```
int sm_GetPassword(Dc_String username, Dc_String *password)
```

`sm_GetPasswort()` liefert das Password `password` vom Benutzer `username`. Die Routine liefert -1, wenn `username` nicht eingeloggt ist.

```
int sm_MakeRandomKey(unsigned char* key, int keylen)
```

`sm_MakeRandomKey`() liefert einen zufälligen Schlüssel von `keylen` Byte Länge in `key` zurück. `key` muß schon alloziert sein. Sind keine echten Random Bits mehr vorhanden, werden Pseudorandom Bits verwendet, d. h. die Routine liefert immer 0 zurück.

```
int sm_MakeKeyPair(int keylen, Dc_String userid, Dc_String password,
         int flags, Dc_String username)
```

`sm_MakeKeyPair()` generiert ein Public/Secret Key-Paar der Länge `keylen` Bits. `userid` enthält die User ID des neuen Key-Paars und `password` dient als Passwort des Secret Keys. Ist das Flag auf `sm_SIGN_NEW_USERID` gesetzt, wird die Public/User ID-Kombination unterschrieben. `username` enthält den authentischen Namen des Benutzers und wird zur Bestimmung des Filenamens für den Secret Key benutzt. Der Public Key wird im Public Key Ring abgelegt. Bei Mißlingen wird ein negativer Wert zurückgeliefert.

```
int sm_GetPublicKey(Dc_String userid, char *keyID,
     R_RSA_PUBLIC_KEY *PubKey, Dc_String username)
int sm_GetSecretKey(Dc_String userid, char *keyID,
     Dc_String password,
     R_RSA_PRIVATE_KEY *PrivKey, Dc_String username)
```

`sm_GetPublicKey()` liefert in `PubKey` den Public Key mit User ID `userid` oder Key ID `keyID` zurück. `PubKey` muß schon alloziert sein. Ist `userid` `NULL`, wird `keyID` zur Identifi-

zierung des Keys benutzt, ansonsten `userid`, d. h. hier hat die User ID Priorität.[1] Wird mit `userid` auf den Schlüssel zugegriffen, wird in `keyID` dessen Key ID zurückgeliefert. `keyID` muß alloziert sein. `username` enthält den authentischen Namen des Benutzers. Bei Mißlingen wird ein negativer Wert zurückgeliefert.

Für `sm_GetSecretKey()` gelten die gleichen Aussagen wie für `sm_GetPublicKey()`. Als zusätzlicher Parameter wird das Passwort `password` des Secret Keys benötigt.

```
int sm_ViewKeyring(Dc_String userid, Dc_String ringfile, int flags)
```

`sm_ViewKeyring()` gibt von allen Keys in `ringfile`, die eine User ID mit `userid` als Teilstring enthalten, den Typ (`pub/sec`), die Länge, die User IDs und das Erzeugungsdatum aus. Ist das Flag `sm_SHOWHASH` gesetzt, wird für jeden Key ein Hashwert angezeigt (z. B. für die Authentifikation von Public Keys über eine Telefonleitung). `sm_SHOWSIG` bewirkt ein Anzeigen der vorhandenen Signaturen. Bei Mißlingen wird ein negativer Wert zurückgeliefert.

```
int sm_CheckKeyring(Dc_String userid, Dc_String ringfile, int flags)
```

`sm_CheckKeyring()` kontrolliert von allen Keys in `ringfile`, die eine User ID mit `userid` als Teilstring enthalten, die Unterschriften auf ihre Authentizität. Ist das Flag `sm_REMOVE_BAD_SIGS` gesetzt, werden ungültige Unterschriften entfernt. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_EditKey(Dc_String keyID, Dc_String userid, Dc_String newid,
     Dc_String oldpasswd, Dc_String newpasswd, int flags,
     Dc_String username)
```

Diese Funktion kann nur vom Besitzer des Public/Secret Key-Paars ausgeführt werden. Ist `newid` gesetzt, wird dem durch `keyID` resp. `userid` identifizierten Key-Paar eine neue User ID angehängt. Ist das Flag `sm_SIGN_NEW_USERID` gesetzt, wird die neue Key/User ID-Kombination signiert. `sm_ADD_PRIMARY_USERID` bewirkt, daß `newid` die erste User ID des Key Paars wird. `oldpasswd` enthält das Passwort des Secret Keys.

Ist `newpasswd` gesetzt, wird das Passwort des Secret Keys durch `newpasswd` ersetzt. Auch hier wird neben `keyID` resp. `userid` das alte Passwort `oldpasswd` benötigt. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_RevokeKey(Dc_String keyID, Dc_String userid,
     Dc_String password, Dc_String username)
```

Diese Funktion kann nur vom Besitzer des Public/Secret Key-Paars ausgeführt werden. Für den durch `keyID` resp. `userid` bezeichneten Public Key wird ein Revocation-Zertifikat ausgestellt. Dieses wird benötigt, wenn der Secret Key in falsche Hände geraten ist. Das Zertifikat kann mit `sm_ExtractPublicKey()` extrahiert werden und sollte an alle Benutzer des entsprechenden Public Keys verteilt werden. `password` enthält das Passwort des Secret Keys und `username` den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_DisableOwnPublicKey(Dc_String keyID, Dc_String userid,
         Dc_String username)
int sm_EnableOwnPublicKey(Dc_String keyID, Dc_String userid,
         Dc_String username)
```

Diese Funktionen können nur vom Besitzer des Public/Secret Key-Paars ausgeführt werden. Mit `sm_DisableOwnPublicKey()` wird der durch `keyID` resp. `userid` bezeichnete Public Key für eine weitere Benutzung im Public Key Ring gesperrt und mit `sm_EnableOwnPublicKey()` wieder zugelassen. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

---

1. Dies ist ein Widerspruch zur sonst geltenden Regel; da diese Routine aber nur im Initialisierungsteil eines Moduls aufgerufen wird, wo zuerst die Requirements nach einer Key ID durchsucht werden und bei Erfolg `sm_GetPublicKey()` mit `userid` gleich `NULL` aufgerufen wird, hat auch hier die Key ID Priorität.

```
int sm_ExtractPublicKey(Dc_String keyID, Dc_String userid,
     long *outlength, unsigned char **outfile)
int sm_ExtractSecretKey(Dc_String keyID, Dc_String userid,
          long *outlength,unsigned char **outfile,
          Dc_String username)
```

Mit `sm_ExtractPublicKey()` wird der durch `keyID` resp. `userid` bezeichnete Public Key aus dem Public Key Ring extrahiert und in einem `outlength` langen, durch `outfile` bezeichneten Speicherblock deponiert. Bei Misslingen wird ein negativer Wert zurückgeliefert. Für `sm_ExtractSecretKey()` gilt dasselbe, hier wird zusätzlich der authentische Name `username` des Benutzers benötigt, da diese Funktion nur vom Besitzer des Secret Keys aufgerufen werden kann.

```
int sm_RemovePublicKey(Dc_String keyID, Dc_String userid, int flags,
          Dc_String username)
int sm_RemoveSecretKey(Dc_String keyID, Dc_String userid,
     Dc_String username)
```

Diese Funktionen können nur vom Besitzer des Public/Secret Key-Paars ausgeführt werden (`sm_RemovePublicKey()` auch vom Systemadministrator). Mit `sm_RemovePublicKey()` wird der durch `keyID` resp. `userid` bezeichnete Public Key aus dem Public Key Ring entfernt. Ist das Flag `sm_SYSADM` gesetzt, muß der zu entfernende Public Key nicht auch als Secret Key im Secret Key Ring vorhanden sein. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert. Für `sm_ExtractSecretKey()` gilt dasselbe, hier kann nur der Besitzer des Secret Keys diesen entfernen.

```
int sm_RemoveUserID(Dc_String keyID, Dc_String userid,
     Dc_String rmuid, int flags, Dc_String username)
```

Diese Funktion kann nur vom Besitzer des Public/Secret Key-Paars und vom Systemadministrator ausgeführt werden. Mit `sm_RemoveUserID()` wird vom durch `keyID` resp. `userid` bezeichneten Public/Secret Key-Paar die User ID `rmuid` inkl. eventueller Signaturen in beiden Key Ringen entfernt. Ist das Flag `sm_SYSADM` gesetzt, wird nur im Public Key Ring die User `rmuid` inkl. eventueller Signaturen entfernt und der entsprechende Secret Key muß nicht im Secret Key Ring sein. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_RemoveSignature(Dc_String keyID, Dc_String userID,
     Dc_String sigkeyID, Dc_String siguserID,
     int flags, Dc_String username)
```

Diese Funktion kann nur vom Ersteller der zu entfernenden Signatur und vom Systemadministrator ausgeführt werden. Mit `sm_RemoveSignature()` wird vom durch `keyID` resp. `userid` bezeichneten Public Key bei der User ID `userID` (d. h. `userID` muß in jedem Fall gesetzt werden) die durch `sigkeyID` oder `siguserID` bezeichnete Unterschrift entfernt. Ist das Flag `sm_SYSADM` gesetzt, muß der Secret Key, mit dem die Unterschrift ausgestellt worden ist, nicht im Secret Key Ring sein. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_AddPublicKey(long inlength, unsigned char *infile)
int sm_AddSecretKey(long inlength, unsigned char *infile,
     Dc_String username)
```

Diese beiden Funktionen erlauben das Hinzufügen eines Public resp. Secret Keys in den entsprechenden Key Ring. `inlength` bezeichnet die Länge des Speicherblocks `infile`, der den Key enthält. `username` enthält den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_SignPublicKey(Dc_String keyID, Dc_String userID,
     Dc_String sigkeyID, Dc_String siguserID,
     Dc_String password, Dc_String username)
```

Mit sm_SignPublicKey() wird die User ID userID (d. h. userID muß in jedem Fall gesetzt werden) des durch keyID resp. userID bezeichneten Public Keys mit dem durch sigkeyID resp. siguserID bezeichneten Secret Key unterschrieben. password enthält das Passwort des Secret Keys und username den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_DisableCertificate(Dc_String keyID, Dc_String userID,
     Dc_String sigkeyID, Dc_String siguserID,
     Dc_String password, int flags,
     Dc_String username)
```

Mit sm_DisableCertificate() wird die User ID userID (d. h. userID muß in jedem Fall gesetzt werden) des durch keyID resp. userID bezeichneten Public Keys mit einem Disable Zertifikat versehen. Dafür wird der durch sigkeyID resp. siguserID bezeichnete Secret Key verwendet. Ist sm_DISABLE_WHOLE_KEY gesetzt, kann der Aussteller des Zertifikats überhaupt nicht mehr auf den entsprechenden Public Key zugreifen, ansonsten gelingt der Zugriff via keyID immer und der via userID nur, wenn die entsprechende User ID kein Disable Zertifikat vom Zugreifenden besitzt. password enthält das Passwort des Secret Keys und username den authentischen Namen des Benutzers. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_MakeCertificate(Dc_String keyID, Dc_String userID,
     Dc_String password,long inlength,
     unsigned char *infile, Dc_String infilename,
     char literal_mode,
     long *outlength, unsigned char **outfile,
     Dc_String username)
int sm_MakeSeparateCertificate(Dc_String keyID, Dc_String userID,
              Dc_String password,
     long inlength, unsigned char *infile,
     long *outlength,
     unsigned char **outfile,
     Dc_String username)
```

Mit sm_MakeCertificate() werden die im inlength langen, durch infile bezeichneten Speicherblock liegenden Daten mit dem durch keyID resp. userID bezeichneten Secret Key unterschrieben. password enthält das Passwort des Secret Keys, infilename den Namen des im Speicherblocks liegenden Files, literal_mode gibt den Typ der Daten an (MODE_TEXT/ MODE_BINARY) und username den authentischen Namen des Benutzers. Die signierten Daten werden zusammen mit der Signatur im outlength langen, durch outfile bezeichneten Speicherblock abgelegt. Bei Misslingen wird ein negativer Wert zurückgeliefert.

sm__MakeSeparateCertificate() funktioniert wie sm_MakeCertificate(), der zurückgelieferte Speicherblock enthält aber nur die Signatur. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_CheckCertificate(long inlength, unsigned char *infile,
          long *outlength, unsigned char **outfile,
     char *originalname)
int sm_CheckSeparateCertificate(long certlength,
     unsigned char *certfile,
     long datalength,
     unsigned char *datafile)
```

sm_CheckCertificate() kontrolliert ob, die im inlength langen, durch infile bezeichneten Speicherblock liegende Signatur die ebenfalls dort liegenden Daten authentisiert. Die Originaldaten werden im outlength langen, durch outfile bezeichneten Speicherblock zurückgeliefert. originalname liefert den Filenamen der signierten Daten zurück. Bei Misslingen wird ein negativer Wert zurückgeliefert.

sm_CheckSeparateCertificate() funktioniert wie sm_CheckCertificate(), nur wird hier die Signatur im certlength langen, durch certfile bezeichneten Speicherblock und

die zugehörigen Daten im datalength langen, durch datafile bezeichneten Speicherblock übergeben. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_AddCertificate(int nofids, Dc_String *userids,long inlength,
    unsigned char *infile)
```

sm_AddCertificate() fügt das im inlength langen, durch infile bezeichneten Speicherblock liegende Zertifikat (mit oder ohne signierte Daten) in den Zertifikat Ring ein. Dem Zertifikat werden zur Identifikation die nofids in userids übergebenen Strings vorangestellt. Diese Identifizierung muss eindeutig sein. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_ExtractCertificate(int nofids, Dc_String *userids,
    long *outlength, unsigned char **outfile)
```

Mit sm_ExtractCertificate() wird das durch die nofids Strings in userids bezeichnete Zertifikat in den outlength langen, durch outfile bezeichneten Speicherblock extrahiert. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_ViewCertring(int nofids, Dc_String *userids)
```

sm_ViewCertring() zeigt alle Zertifikate im Zertifikat Ring an, deren erste nofnames Strings den in userids übergebenen entsprechen. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_RemoveCertificate(int nofids, Dc_String *userids)
```

sm_RemoveCertificate() entfernt das Zertifikat aus dem Zertifikat Ring, das genau nofnames Strings besitzt und die alle den in userids übergebenen entsprechen. Bei Misslingen wird ein negativer Wert zurückgeliefert.

```
int sm_SecurityOn(char *buf, int socketdeskr,
    void(*callback)(int, Dc_String))
int sm_SecurityOff(   char *buf, int socketdeskr,
    void(*callback)(int, Dc_String))
```

Mit sm_SecurityOn() wird die Sicherheit in einem bestimmten Protokoll ein- resp. mit sm_SecurityOff() ausgeschaltet. buf enthält den Namen des Besitzers des Protokolls gefolgt vom Namen des Protokolls (C-Strings). Der Filedeskriptor socketdeskr und ein Antwortstring können der callback-Funktion übergeben werden und dieser wird an den Auslöser der Umschaltung zurückgeschickt. Bei Misslingen wird -1 zurückgeliefert.

```
int sm_SendSecurityOn(Dc_String hostname, Dc_String username,
    Dc_String prname)
int sm_SendSecurityOff(Dc_String hostname, Dc_String username,
    Dc_String prname)
```

sm_SendSecurityOn() resp. sm_SendSecurityOff() schicken einen entsprechenden Request an das auf hostname laufende Da CaPo-System, mit dem Ziel, das username gehörende Protokoll prname umzuschalten. Bei Misslingen wird -1 zurückgeliefert.

```
int sm_Register(Dc_String username, Dc_ProtocolPtr protcolPtr,
    Dc_String ndInstanceName, int **state)
int sm_UnRegister(Dc_String username, Dc_ProtocolPtr protcolPtr,
    Dc_String ndInstanceName)
```

Mit sm_Register() kann sich ein Modul, das sich im Protokoll protocolPtr, das dem Benutzer username gehört, befindet, beim Security Manager registrieren. ndInstanceName bezeichnet den Namen des Moduls und ist fakultativ. state liefert die Adresse des Zustandsfeldes des Protokolls zurück. Mit sm_UnRegister kann sich ein Modul beim Security Manager abmelden.

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.

Da CaPo implements a very 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.

7.1 Multicast-capable Connection Manager

The Connection Manager (ConMan) guarantees that a compatible protocol is used inside a Da CaPo session. The ConMan triggers local configuration and reconfiguration, distributes the resulting mechanism graph and starts/stops the protocols. The actions taken by the connection manager are initiated either by the local application or by a remote connection manager. There is one ConMan per session.

The protocol stack used by the connection manager is defined statically. There is one protocol stack per transport infrastructure and per session. Four transport infrastructures are planned: UDP, UDP multicasting, ATM, ATM multicasting. Figure 10 shows the protocol stack of the connection manager.

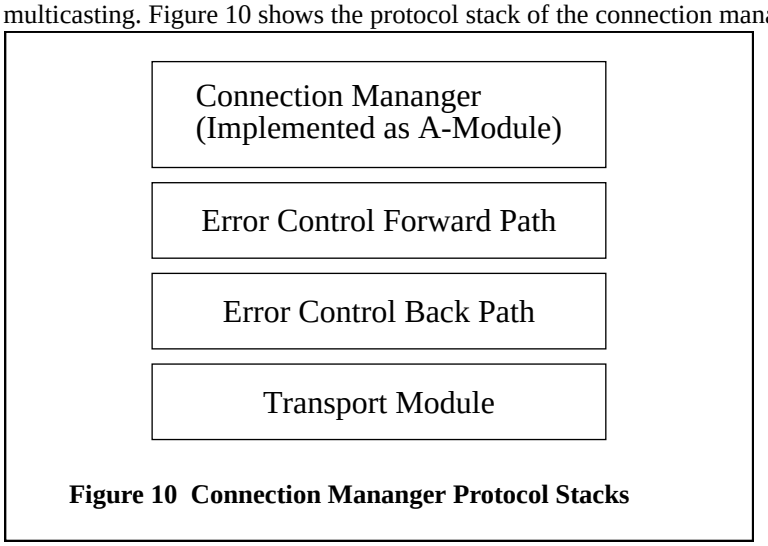


Figure 10 Connection Mananger Protocol Stacks

The control over a multicast session lies solely at the creator. This means that only the application entity that created the session is allowed to reconfigure, start and stop the multicast session. Only local reconfiguration is supported, the application requirements of the participants are ignored, a (re-)configuration is based entirely on the creator's application requirements. Participants are not allowed to reconfigure a multicast session, e.g. an application's reconfigure request has no effect. Start and stop at a participant's site leads to a stop of the local execution of the lift. This means that no data packets are delivered to the participant's application which means that data loss is inevitable. Therefore, the stop operation should only be executed by applications that tolerate data loss like audio or video applications.

The connection manager in the multicast case is built according to the client-server principle. The ConMan of the creator is the server, the ConMans of the participants are clients. The ConMan at the creator distributes the mechanism graphs for the sessions and starts and stops the session. The other ConMans request the mechanism graph and send start/stop events to the creator's ConMan.

7.1.1 The Creator's ConMan

The creator's ConMan receives the following requests and events:

- Application requests
  - Create
  - Configure
  - Start
  - Stop
  - Destroy
- Events from participants ConMans
  - Graph Request
  - Remote Start
  - Remote Stop
  - Remote Destroy

The semantics of these requests is shown in the Table 11.

TABLE 11. Creator's Connection Manager Requests

Create	The application creates the session. For each protocol inside a session, the default modules are initiated. The ConMan of the session is started and initialized.
Configure	The application triggers a local configuration. The configuration is based on local application requirements only. After the configuration is done, the ConMan distributes the resulting mechanism graph to all participants and instantiates the new protocol.
Start	The local lifts are started as soon as at least one participant is available. The ConMan sends a 'remote start event' to all participants.
Stop	The local lifts are stopped. The ConMan sends a 'remote stopped event' to all participants.
Destroy	The local lifts are stopped and all modules are deallocated. The ConMan sends a 'remote destroy event' to all participants.
Send Graph	A new participant wants to join the group. The ConMan answers with the current mechanism graph. Additionally, the ConMan forwards this event to the lower API.
Remote Start	A participant signals that its application has triggered a start event. If this is the first 'remote start event' that the creator obtaines, then it starts its lifts also.
Remote Stop	A participant signals that its application has triggered a stop even. If no participants are running any more, then the ConMan also stops its local lifts.
Remote Destroy	A participant signals that its application has terminated the session. The ConMan informs the lower API that the participant has left. If all participants have left, then the local lifts are stopped, too.

7.1.2 Participant's ConMan

The participant's ConMan handles these events:

- Application events

- Create
- Configure
- Start
- Stop
- Destroy

- Remote events
  - GetGraph
  - Remote Start
  - Remote Stop
  - Remote Destroy

TABLE 12. Participant´s Connection Manager Requests

Create	The application creates the session. For each protocol inside a session, the default modules are initiated. The ConMan of the session is started and initialized. At the same time, the ConMan sends a ‘getgraph event’ to the creator.
Configure	The application triggers a local configuration, which has no effect whatsoever.
Start	The local lifts are started if the creator has also started. The ConMan sends a ‘remote start event’ to the creator’s ConMan.
Stop	The local lifts are stopped. The ConMan sends a ‘remote stop event’ to the creator’s ConMan.
Destroy	The local lifts are stopped and all modules are deallocated. The ConMan sends a ‘destroy event’ to the creator’s ConMan.
Get Graph	A new (or the first) mechanism graph is sent by the creator. The ConMan instantiates the new mechanism graph according to the parameters of this event.
Remote Start	The creator has started its lifts. If the application already issued a start event, then the lifts are also started.
Remote Stop	The creator has stopped its lifts. Stop the lifts, too.
Remote Destroy	The creator has terminated the session. Destroy the session, too. Send an event to the application.

The semantics of these requests is shown in Table 12.

7.1.3 ConMan Error Control Protocol

The connection manager needs reliable data transfer. For a multicast capable session, a multicast error control is needed as well as unicast error control. The principal flow of connection manager data is shown in Figure 11

:

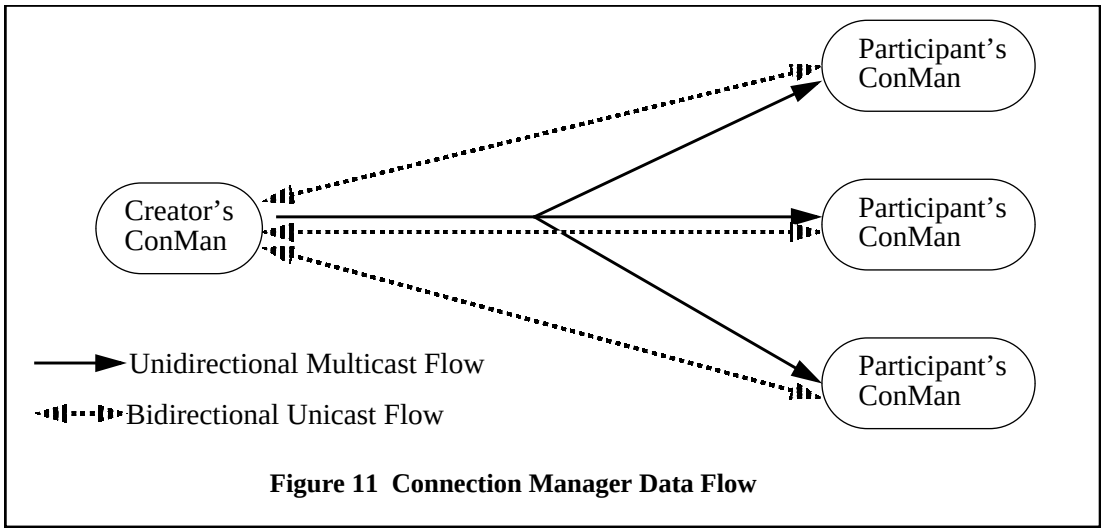


Figure 11 Connection Manager Data Flow

In order to simplify the implementation, the connection manager uses TCP for the bidirectional unicast flows, even if ATM is used as a transport infrastructure. The control protocol then uses the modules shown in Figure 12. Depending on the data which needs to be sent, either the multicast error control or the unicast error control (TCP) is used. The connection manager uses an address field in the header of the PDU to specify the recipient of the messages. This header field is also interpreted by the multicast error control and the transport module. The modules can therefore only be used in the connection manager protocol.

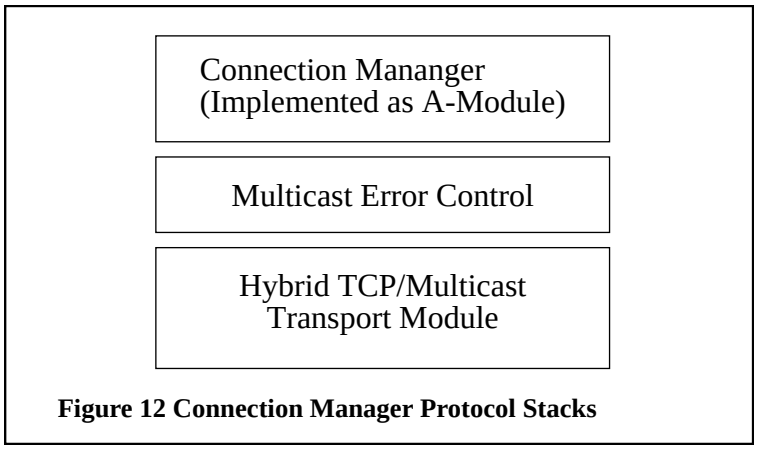


Figure 12 Connection Manager Protocol Stacks

7.1.4 Finite State machines

7.1.4.1 Finite State Machine of Creator's ConMan

The creator's ConMan uses the finite state machine shown in Figure 13 The ConMan changes its state upon receiving events. Events are received either from the application (for example 'Create') or from the remote connection manager. Remote events are shown with a leading arrow (<-). An arrow to the right (->) indicates an event that is sent to the creator's connection manager.

In order not to overload the figure, not all states and state transitions are shown. The following comments have to be made:

- The 'Exit' state is reached from every other state upon receiving a 'destroy event' from the application.
- The states 'peer known', 'peer started' and 'running' keep track of the number of participants. For example, the transition between 'peer known' and 'local configured' occurs only when the last participant sends a destroy message. The same holds true for the other states.

- A ‘configure’ event from the application leads to a local reconfiguration, irrespective of the state. After the configuration, the new protocol is instantiated and a ‘Graph Reply’ is distributed to all participants.

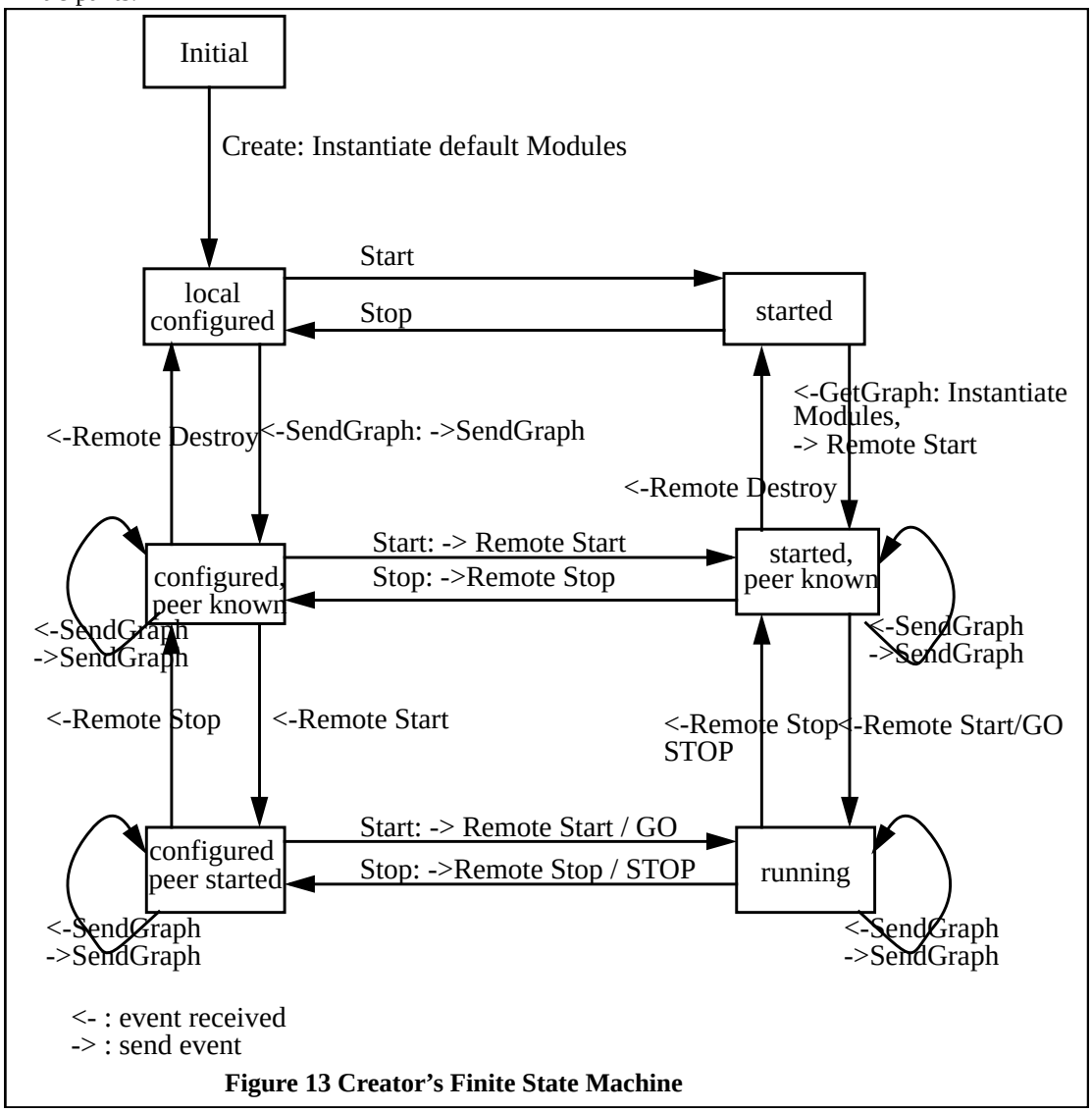


Figure 13 Creator’s Finite State Machine

7.1.4.2 Finite State Machine of Participant’s ConMan

The finite state machine of the participant consists of seven states. Figure 14 shows the state machine without the ‘Exit’ state. The ‘Exit’ state is reaches when either the application wants to destroy the session or when the creator’s ConMan sends a destroy event. In the case where the application triggers the destruction, a destroy event is sent to the creator’s ConMan.

If the state ‘ready to run’ is reached, data loss may occur, because the creator’s lifts are running while the local lifts are stopped.

7.1.5 Interfaces and Implementation Details

This paragraph describes the interface that is intended to be used and the structure of the implementation.

7.1.5.1 Connection Manager Interface

Both the unicast and the multicast connection manager are implemented as A modules. The lower API selects a connection manager protocol for each session and therefore decides which connection manager

will be used (now: connection manager protocol is still fixed). The interface to the Da CaPo++ runtime environment and to the modules is exactly the same for both unicast and multicast connection managers. The following functions are offered.

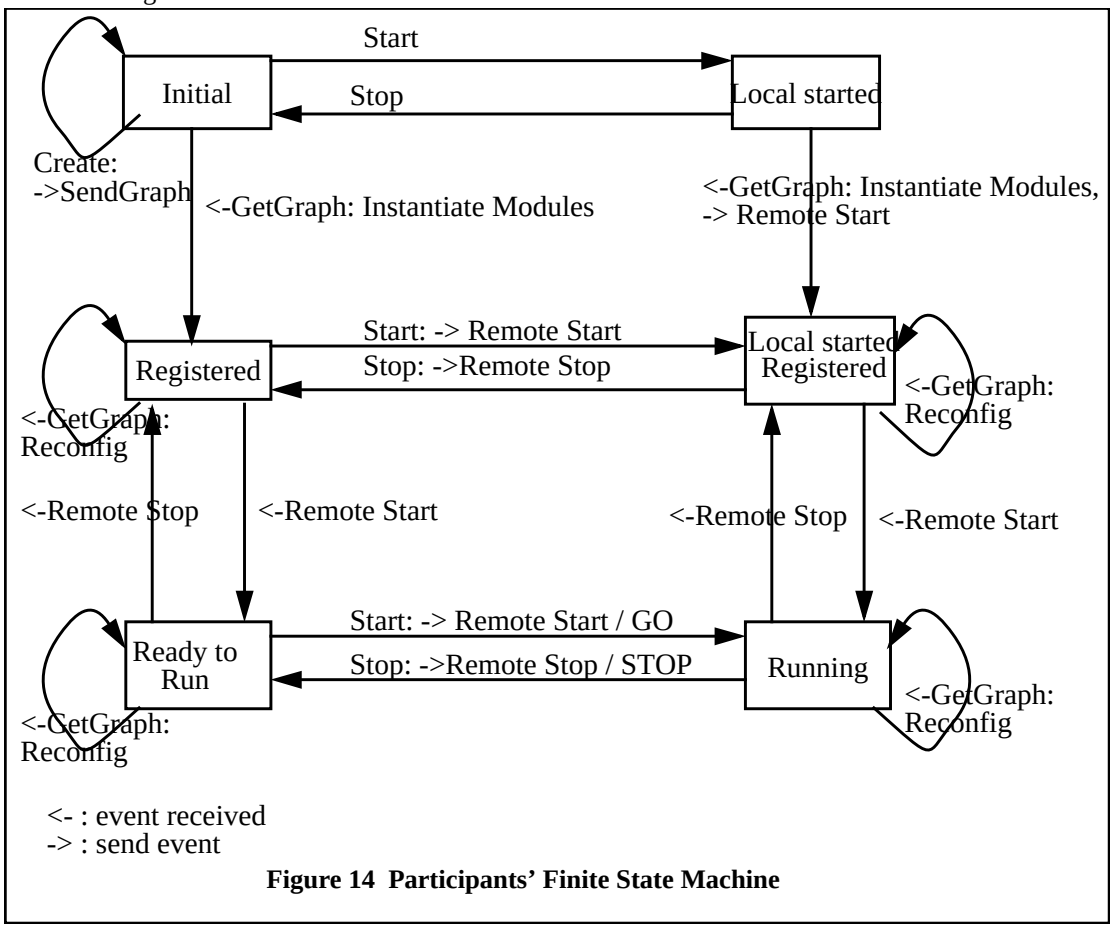


Figure 14 Participants' Finite State Machine

TABLE 13. Interface Functions

conn_Create(Dc_ServicePtr)	This function is called as a result to the applications call to Dc_Create. At the time conn_Create is called, the whole service has been set up.
conn_Configure(Dc_ServicePtr)	After the local configuration has been done, conn_Configure is called. The purpose of conn_Configure is to distributed the new mechanism graphs.
conn_Start(Dc_ServicePtr)	The connection manager is informed that the lifts are going to be started.
conn_Stop(Dc_ServicePtr)	The connection manager is informed that the lifts are going to be stopped.
conn_CntlData(Dc_ServicePtr,protocolNumber, StepNumber, Dc_BufferPtr)	Instructs the connection manager to transport the buffer to all peers and deliver the data to the step 'StepNumber' of the given protocol. This is used to transport 'Out-of-Band' data.
conn_Destroy(Dc_ServicePtr)	Informs the connection manager that the service is going to be deallocated.
conn_Init(void)	conn_Init is called when the Da CaPo system is initialized.
conn_Exit(void)	conn_Exit is called when the Da CaPo system exits.

A call to any of this functions is delegated to the connection manager in use. This is done by instructing the lift to call the indication function of the A module of the first protocol, which per definition is the connection manager. The indication function uses a simple command queue which is stored in the service structure. This command queue is also used to trigger commands caused by incoming events from remote connection managers. Figure 16gives an overview for both cases:

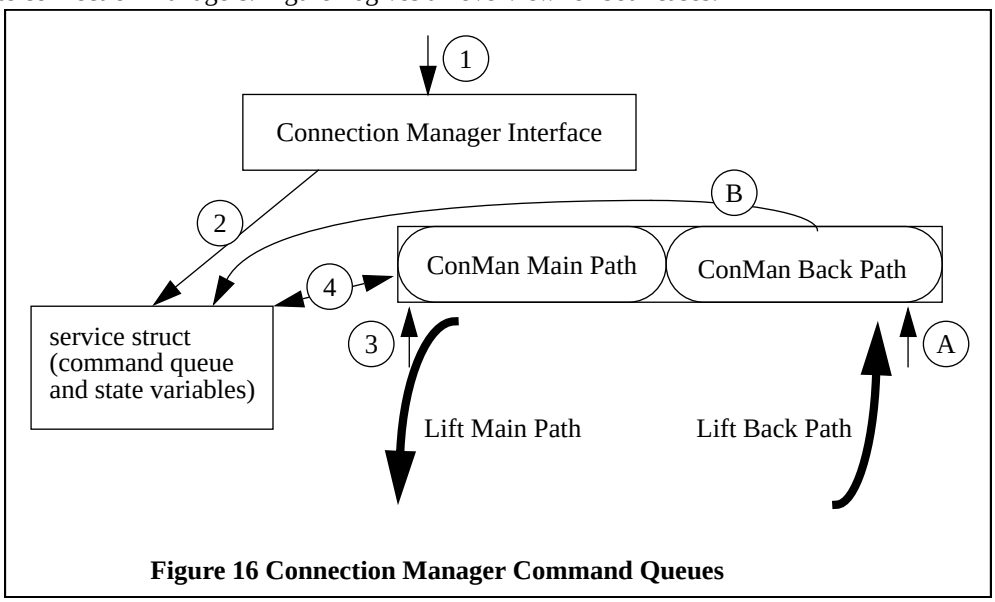


Figure 16 Connection Manager Command Queues

- The connection manager is called (1) or gets an event from the remote connection manager (A).
- The command (2) or the according reaction to the event (B) is stored in the command queue.
- In both cases, the lift is instructed to call the indication function of the connection manager (3).
- The indication function interprets the command queue, changes its internal state by setting variables in the service struct and forwards a packet to the remote connection managers (4).

Since the connection manager interface simply queues the commands, it does not need to know whether a multicast or a unicast connection manager is installed.

The connection manager uses Da CaPo events for the information exchange with the application. The Da CaPo event structure offers a “user pointer” which will be used to store the information that is passed to the application. Depending whether the application is creator or participant or whether a multicast session is used or not, different events are being used. Table 5 lists the events that can be received by the application that created a multicast session.

TABLE 14. Multicast Events Received by Applications

Event	Parameter	Description
joined	‘Address’ of the participant, e.g. value of the ‘aiOwn’ attribute. This attribute is set by the application. It usually contains a name such as ‘ktik8’.	The ‘joined’ event is generated when the connection manager receives a ‘Graph Request’. This event does not mean that the participant is ready to receive any data, it only means that a new participant is being attached to the session.
started		A participant has started its lift. This participant will receive data if the lift of the creator is also running.
stopped		A participant has stopped its lift and will thus no longer be receiving any data.
left		A participant has left the flow.

Events that are forwarded to the application on the participant's side are listed in Table 8. These events do not have any parameters.

TABLE 15. Multicast Events Sent to Applications

Event	Description
stopped	The creator has stopped and, as a result of this, the local lifts have stopped, too.
started	The creator has started the lifts. If the own lifts are running, then data will be transported.
terminated	The creator has terminated the session.

7.1.5.2 Implementation of the State Machine

The state machine will be implemented in the indication function of the connection manager. This function is called whenever an event either from the application or a remote connection manager needs to be processed. There are two indication functions, one for the creator and one for the participants. In a running connection manager, only one of these two functions is called by the lift. XXX gives an overview of the principal design. Both state machines use the same utility functions, such that the indication function

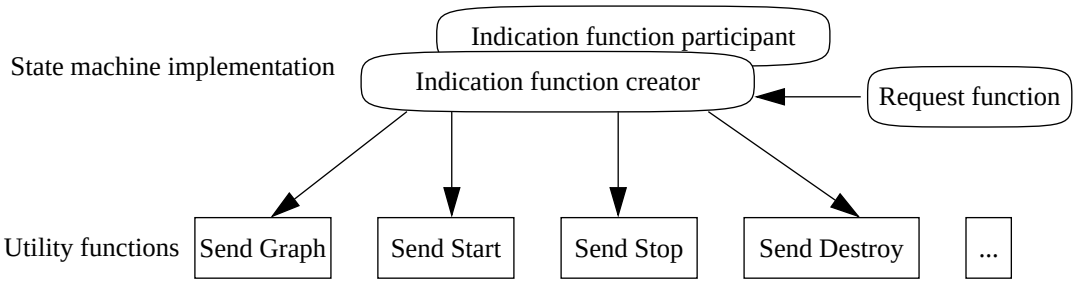


Figure 17 Overview of Connection Manager Functions

only has to keep track of the internal states. The request handling is done in the request function. The main purpose of this function is to put the appropriate 'command' into the queue and instruct the lift to call the indication function. Once the indication function is called, the state of the connection manager is changed and the appropriate action is taken.

7.1.6 Multicast Connection Manager PDUs

The PDUs sent by the connection manager consist of a header field and a data field. Unlike other Da CaPo protocols where modules only know their specific part of the header, each module in the connection manager knows the whole header. The connection manager modules thus can only be used inside the connection manager. The format of the header is shown in Table 8:

TABLE 16. Packet Data Unit Format

Started	Sender	Senders Service	Connection	Type	Length
1	16	16	4	1	4

- Started

  State of sender. TRUE if sender has started.

- Sender

  Senders machine address as specified by the application (*e.g.*, ktik8). This field is used for generating events that are forwarded to the application.

- Senders Service

  Senders service description, as specified by the application (*e.g.*, dacapo-cm)

- Connection

  Numerical identifier of the connection. ‘0’ means ‘multicast transport’. In the case of the creator, the packet is sent to all participants, in the case of the participant, the packet is sent to the creator where it is consumed by the multicast error control. A value bigger than 0 means a TCP connection to a single receiver. The participants use a value of ‘1’ to address the creator’s connection manager.

- Type

  Type of the packet. The different packet types are listed below.

- Length

  Length of the following data. This field is used for transport layers that do not preserve the message boundary, *e.g.*, TCP.

A number of PDUs (cf. Table 8) are exchanged between creator and participant.

TABLE 17. Exchanged PDUs

Type	Data	Description
Remote Started	-	Used by both creator and participant.
Remote Stopped	-	
Remote Destroyed	-	
Graph Request	Application Requirements	Sent by a participant that wants to join the session. The application requirements are ignored by the creator.
Graph Reply	Mechanism graphs for all flows inside the service.	Sent by the creator upon receiving a Graph Event or after reconfiguring.
QoS	QoS parameters that are valid for the active mechanism graph.	Immediately sent after a Graph Reply PDU.

7.2 Multicast Error Control Module

PDUs are passed from the connection manager to the multicast error control module. PDUs with a connection field other than 0 are passed to T-module. The rest of the PDUs are handled by the multicast error control module. A detailed description of the multicast error control module can be found elsewhere.

7.2.1 Transport Module

The transport module builds the interface to the network. It sends and receives data from TCP connections and multicast groups. The addresses of the TCP connections and the multicast group is specified by the application by setting the Da CaPo attributes.

The transport module has a different behavior, depending on whether it is used by the creator or by the participant. First, the tasks of the creator’s transport module are described:

- Administer TCP connections for each participant.
- Accept new TCP connection from new participants.
- Send data to a single participant using a TCP connection.
- Receive data from a TCP connection.
- Send data to all participants using a multicast group (currently only IP multicasting is supported).

- Receive control data for the multicast error control module.

Data is sent in the request function. Depending on the connection field of the header, the PDU is sent either over a TCP connection or to the multicast group.

Receiving data is done in the indication function. The receiving functions sleeps unless on of the following events occurs:

- A new receiver has connected to the listening socket. In this case, the call is accepted and the receiver is registered and numbered.
- Data has arrived on one of the TCP connections. The connection field is filled in and the data is passed on.
- Data has arrived on the multicast error control port. The connection field is set to 0, then the PDU is given to the multicast error control module.
- The participants transport module has the following tasks:
- Establish a connection to the creator's T-module using TCP.
- Send data over the TCP connection.
- Receive data from the TCP connection.
- Receive data on the multicast group port.
- Send data to the multicast sender.

The TCP connection to the creator's T-module is established during the initialization of the T-module. Afterwards, only the indication and request functions are used. Depending on the connection field, data is sent either over the multicast port or over the TCP connection. For the TCP connection, a connection field of '0' is being used. Receiving data is done analogous.

7.3 Multicast Transport Protocols

Basically, there are only two different multicast transport protocols: A reliable point-to-multipoint protocol and a simple point-to-multipoint protocol for audio and video. Both protocols are based on a multicast capable network layer. This is either IP multicasting or ATM multicasting with extensions.

7.3.1 Reliable Multicast Protocol

The reliable multicast protocol uses an error control mechanism based on retransmissions for assuring the correctness of the data transport. In order to avoid a packet implosion at the sender, a negative acknowledgment scheme is used. The receivers detect loss of data by comparing the sequence numbers in the arriving packets with the expected sequence numbers. When a gap in the numbers is detected, the packets are requested from the sender. If the sender has no data to send, it sends a so called heartbeat packet that contains the last sequence number only. The heartbeat packet enables the recipients to detect packet losses. Heartbeats are sent in predictable intervals. Retransmissions which are requested from one of the receivers are multicast to the group. Duplicates are detected by the receivers and thrown away.

As seen from the basic description above, the protocol can be separated into a sender's part and a receiver's part. The sender's part has the following tasks:

- Multicast datagrams to the group and store the datagrams for retransmission.
- Send heartbeat packets when no datagrams are available to send, use a timer for heartbeat messages.
- Answer retransmission request by re-multicasting the packet to the group.
- Send a leave message to the receivers when leaving.

The tasks of a receiver are the following.

- Maintain the actual sequence number for the sender. The initial sequence number is statically defined, i.e. the first packet has the sequence number 0.

- Maintain a timer for the sender. The timer is reset whenever a datagram or a heartbeat packet arrives. If the timer goes off, either a datagram or a heartbeat packet has been lost.
- Check for corrupted, lost or duplicated datagrams. Issue a retransmission request for lost an corrupted datagrams, throw away duplicates.
- Sort the datagrams according to their sequence numbers.

The protocol also features a segmentation and reassembly module which is placed in front of the error control module.

7.3.2 Simple Multicast Protocol

The simple multicast protocol is used as a transport protocol for audio and video services. The protocol is very simple. It consists of a segmentation and reassembly mechanism and a transport mechanism. Error detection is optional, since both IP multicasting and ATM AAL5 already provide error detection methods. This protocol only makes sense if used in conjunction with a specialized A-module that directly feeds or gets the data from a multimedia device such as the parallax video board or the audio device.

7.3.3 Changes in the Existing Da CaPo++ Core – Attributes

Both multicast protocols need a multicast capable connection manager in order to run properly. When used with the unicast connection manager, the sender can only transmit data to a single receiver. In addition to the multicast capable connection manager, new Da CaPo++ attributes have to be introduced:

- aiGroupAddress
- aiGroupService

When using in conjunction with an IP multicasting transport module, the following attributes are needed for full application control:

- aiTTL

  Time to Live': used for controlling the distribution of multicast packets.
- aiLoopback

  Specify whether packets should be delivered back on the same machine.
- aiMcastInterface

  Interface on which multicast packets are issued.

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-teaching, or tele-shopping. Due to a set of well-defined differences between these applications, this spectrum of applications looks quite unstructured. For instance, a teleseminar 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 teleseminar 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 teleseminar or teleteaching 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.

8.1 Applications

8.1.1 Picture Phone (PP-APP)

In the Da CaPo++ terminology, the Picture Phone application consists in the unicast 1:1 application including live audio and video data exchange. The multicast case (n:n) is referred to as the Video Conference application. In general, the unicast case can be seen as a specific subset of the multicast application.

8.1.1.1 Design

The current design state for both PicturePhone and VideoConference scenarios is as follows:

- The unicast PicturePhone scenario is today available and can be demonstrated with little effort. However it is a very basic application with regards to the user interface and the protocol functionality (only 2 A- and T-modules, no C-modules).
- The multicast VideoConference still requires some thoughts on how to make it compliant with the current state of the Da CaPo core system (especially with the connection manager core component). This document proposes the Multicast Support Component which is a solution to alleviate current limitations of the Da CaPo core system.

Basically, what the PicturePhone application needs from Da CaPo is on the one hand the audio/video flows objects (with corresponding A-modules and audio/video protocol graphs) and on the application side the corresponding A/V components to make video/audio data visible/audible to the user.

8.1.1.2 Unicast 1:1 PicturePhone

There is already in the design document of the API a simple example on how to write a configuration file to declare a session object to meet functionality of a 1:1 PicturePhone application (it basically consists of 2 synchronized sending flows for both audio and video and of their corresponding receiving flows). Then an AVPresentation object must be created and bound to both receiving flows. Finally, a graphical user interface must be implemented to offer the end-user the functionality the AVPresentation object provides. The Session's and AVPresentation's declaration are sufficiently explained and described in the API's and in the A/V Presentation's fine design documents. The exact design of the graphical user interface that will be made available has not yet been object to intensive study, but is likely to meet all functionality one can expect from such a PicturePhone application

8.1.2 Video Conference (VC-APP)

In the current state of the connection manager in Da CaPo, it is not possible to have in a single session several multicast flows issued from different senders (cf. connection manager design document). Therefore it is not possible to consider a unique session at the participant's site (as each receiving flow, either audio or video, would have to "register" to a different sender).

To alleviate this limitation, we consider one session per participant which consists in a multicast sending flow to all others and the corresponding receiving flows. If we have n participants, at each participant's site there will be one multicast sending session (consisting in 2 multicast sending flows for both audio and video) and (n-1) receiving sessions (consisting in 2 receiving flows for both audio and video).

The connection manager is built in a client-server manner. This obliges each potential receiving session to explicitly require the protocol mechanisms from the corresponding sender (this is performed through the ConnectSession() function of the upper API). This property motivates the set up of an application protocol, so each potential receiver can be informed when it has to create a new session and who the creator (sender) is.

At this point it is either possible to implement a more sophisticated connection manager or to design a group management application component (in the upper API) which would have its counterpart in the Da CaPo kernel system. The first solution leads to modifications in the kernel system, and the new planned functionality may be too specific for the purpose of Da CaPo communication system. The second solution would allow to tailor dedicated solutions for various CSCW applications (as not only the PicturePhone application is likely to need such an "application" protocol).

8.1.3 Extended WWW Browser and Server (EWB-APP)

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 18 on page 56 for the example of video data.

This figure serves as reference for all future discussions in this design document. The whole scenario will be introduced first before details will be explained.

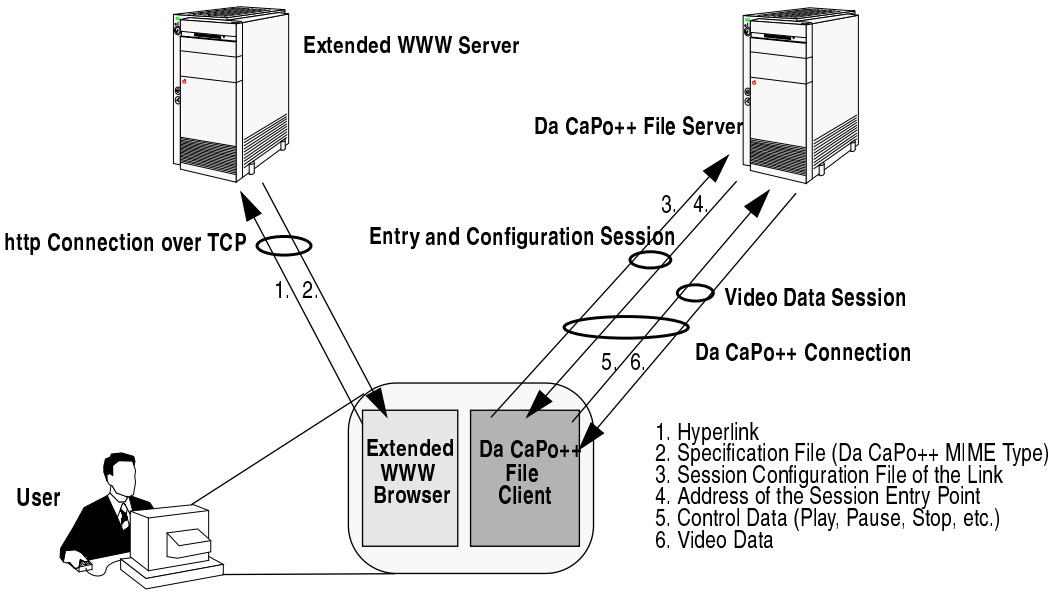


Figure 18 Extended WWW Browser in Case of Video Data Transmission

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, 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 types and start 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 programme can be started, referred to as *External Viewer*. To enable this the following changes have to be made.

8.1.4 The 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.

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.

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

8.1.5 The 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++ Video Viewer” -n
“Video   Viewer”   -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* programme on the client’s site. The *xterm* is not necessary but useful to see the program’s output like, e.g. warnings.

8.1.6 Multimedia File Client and Multimedia File Server

The programme 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. These will then be started as *External Viewers* in the Extended WWW Server and WWW Browser scenario.

8.1.7 Presenting Multimedia Data

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 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 kernel. 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 functionality and the design of the involved A-Modules, some technical details can not be avoided.

8.1.7.1 Communication in A-Modules

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

- **Sender:** The sending A-Module reads the data from a file and transmits it. Furthermore, all VCR[1] control the user initiates, are 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.

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

- **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.

Figure 19 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 overgives 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 function `asIndiA-Module-Name()` The return value for the Lift will be `sbDataOk | sbAgain`, indicating that there is data to transmit in the buffer and (in case of continuous data and `sbAgain`) that there is data to generate another packet. In case of `sbAgain` the Lift recalls the function `asIndiA-Module-Name()` to receive another data packet for transmission.

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 is required.

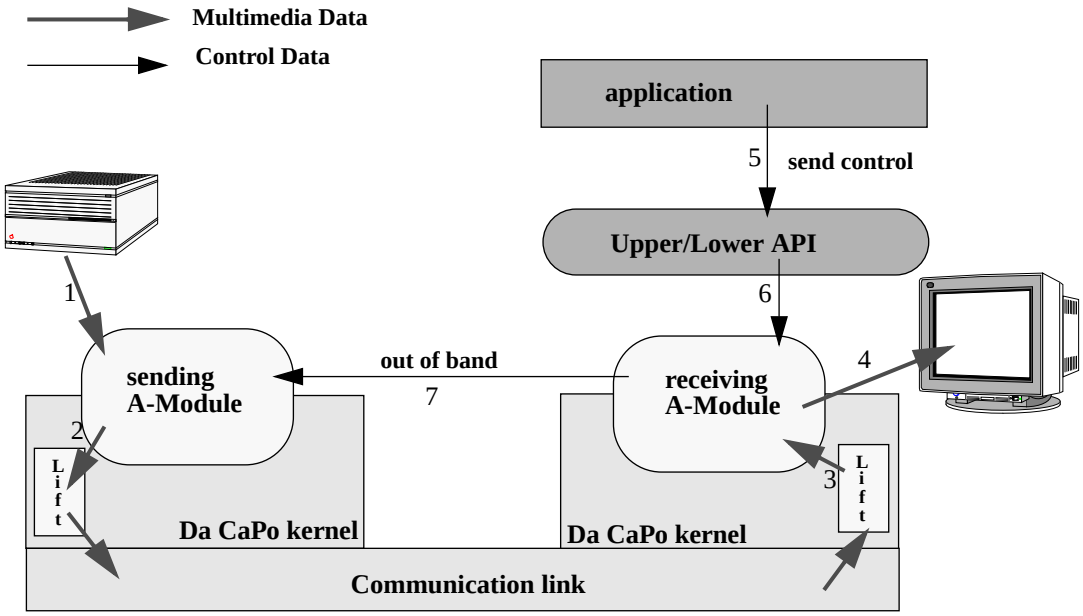


Figure 19 Scheme of the Communication Paths Between A-Modules

8.1.7.2 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 kernel documentation and is not scope of this document.

Figure 20 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.

8.1.7.3 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 kernel 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 flows 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 functions `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 process 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 kernel. 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 receiving out-of-band data.

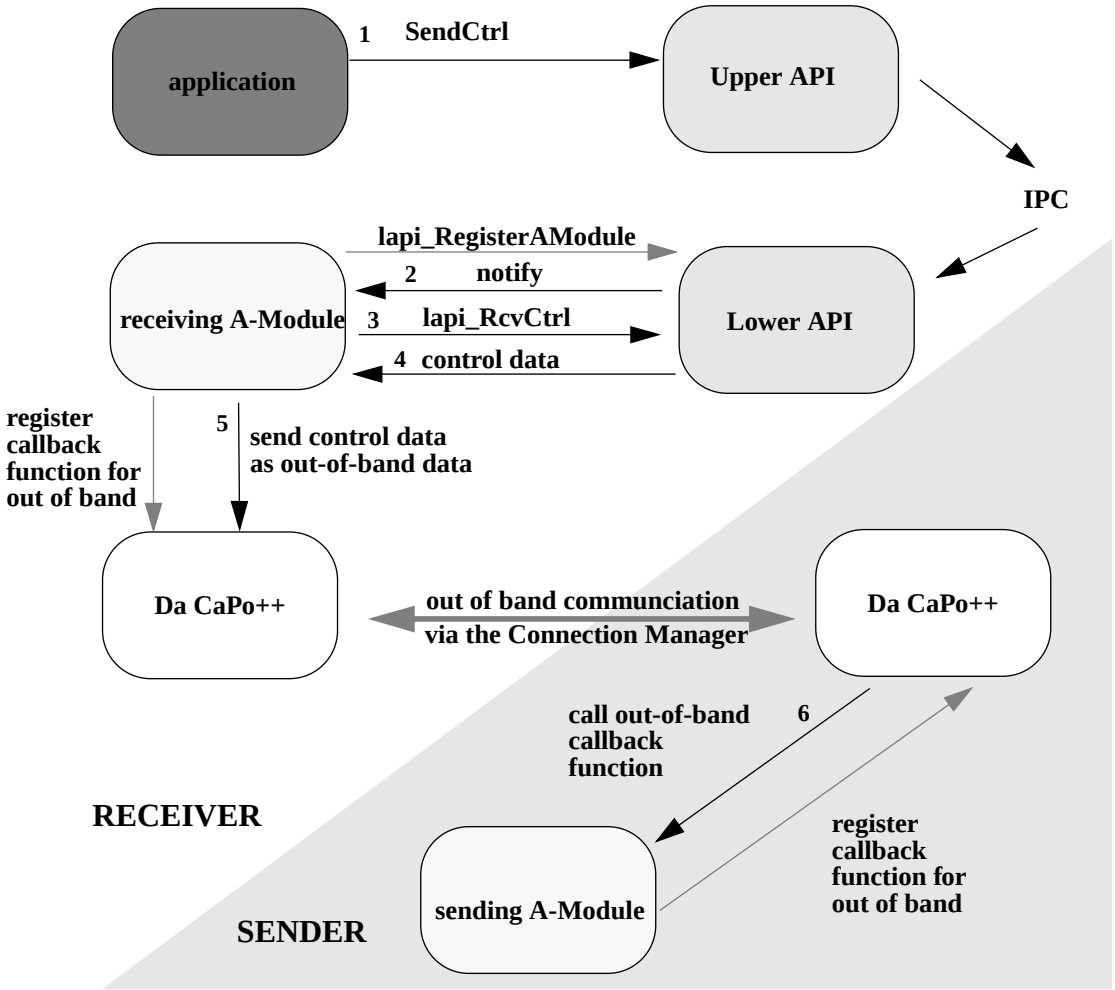


Figure 20 Sending Control Information Via the Out-of-band Communication

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.

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

Figure 20 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 control 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 (fast forward instead of normal speed playback).

8.1.8 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/sunvideofile/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).

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.

8.1.8.1 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 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.

---

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

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. This is planned for a future 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 10^{th} 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 11^{th} frame is played[1]. If the command *fast rewind* is called for a second time, the playback speed is reset to normal.

Commands that shall be provided in the future. The following commands are planned to be provided in the future.

- Forward: this command sets the last frame in the image sequence to the current frame.
- Rewind: this command sets the first frame of the image sequence to the current frame.
- Reverse: this commands provides playing of data in the reverse sequence with a normal playback speed.[2]

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

TABLE 18. 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
forward	FWD	skip to last video frame	not yet implemented
fast forward	FF	display each 10^{th} frame while displaying	implemented
rewind	RWD	go back to first video frame	not yet implemented
fast rewind	FR	display each -11^{th} frame	implemented
initialize	INIT; file_name	initialize the name of the file to be read to file_name	implemented but not yet used
open	OPEN	start playing	implemented

8.1.8.2 The 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.

1. This results in the same as skipping 9 frames in the normal reverse mode.
2. This command is not yet prepared in the implementation.

- `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 `out-of-band` field in the instance pointer and the out-of-band communication is invoked calling the function `liftFullStatusTreatement`.

- `void perform_cmd`

  This function is called in the sending A-Module. The content of the `out-of-band` field in the instance pointer is interpreted and the corresponding actions are performed. These actions consists mostly in setting the appropriate values for the variables `playing` and `fast_play`.

- `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 `out-of-band` 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 out-of-band field in the instance pointer structure and the above mentioned function `perform_cmd` is called.

- `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`. The cis as well as the X window are initialized.

  The file name of the file to read, the compression type and the maximum number of frames are hard-coded for instance. This is subject to change. Also the initialization of the X window will be eliminated in a future version of the A-Module. It is not sensible to display the multimedia data on the Server's site also, but for the current status of implementation and testing this is quite useful.

- `Lift_Status asExitVideoFile`

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

- `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|sdAgain` which indicates that there is no data to send but that the Lift remains active.

  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 10^{th} frame in case of 1, to the -11^{th} frame in case of 2). Then the video frame to send is computed by using two pointers as stated in the beginning of chapter 4.3.

- `Lift_Status asEmpty, asStartVideoFile` and `asStopVideoFile`

  These functions are called when the Lift is started and stopped respectively.

- `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 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.

- 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.

8.1.8.3 The 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.

8.1.8.4 The 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-Modules. 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.

8.1.8.5 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.

8.1.9 The (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++.

8.1.9.1 The 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

```
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[2]: the address, service and interface to connect to the server[3] and all sessions having to be invoked on the server
- 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 the flow is 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 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/config.Is.tmp` and

`/tmp/config.Ic.tmp`

for the server and the client, whereas `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[4] 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.

---

1. The language used in the specification file is defined in the document “The Specification File”.
2. The most general Multimedia File Server scenario consists in n File Servers which send data to m File Clients.
3. This address is valid for the *entry and configuration session* on the server only. The session entry points of the data sessions are transmitted from server to client during the *entry and configuration session*.
4. 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.

8.1.9.2 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 21 on page 66 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.

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

8.1.9.4 Instantiate 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 unicast 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, service, interface);
BEGIN_SESSION
SESSION_TYPE UNICAST;
SESSION video;
FILE name_of_file_to_read;
FLOW VIDEO_SEND_FILE VIDEO_RECV_DEVICE video_flow;
{
     /* all necessary application requirements */
}
END_FLOW;
END_SESSION;
END_ALTER_SPEC;
```

The Video Viewer is already realized. Its implementation is presented in this document.

The Audio Player. The Audio Player File Client type enables the client to receive and display one audiosstream 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, service, interface);
BEGIN_SESSION
SESSION_TYPE UNICAST;
SESSION audio;
FILE name_of_file_to_read;
FLOW AUDIO_SEND_FILE AUDIO_RECV_DEVICE audio_flow;
{
     /* all necessary application requirements */
}
END_FLOW;
END_SESSION;
END_ALTER_SPEC;
```

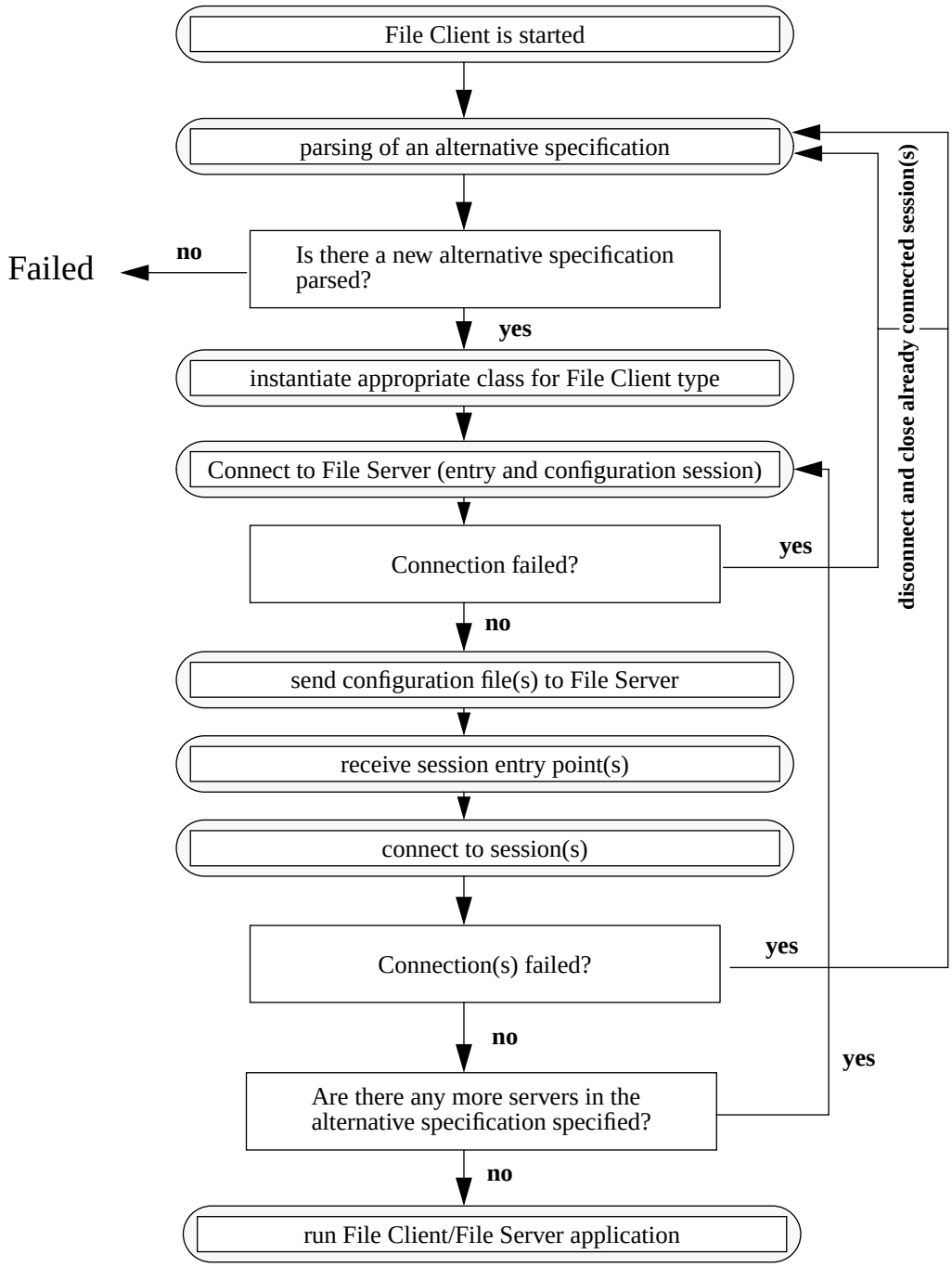


Figure 21 Flow Chart of the Initialization Process in the File Client

8.1.9.5 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.

8.1.9.6 Send 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.Is.tmp`) is (are) sent to the File Server. The server then creates session(s) with the specified configuration. E.g., in case of a Video Viewer the File Server creates one session containing one video flow.

8.1.9.7 Receive 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.

8.1.9.8 Connect 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.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.

8.1.9.9 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.

8.1.9.10 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 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 an 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.

8.1.10 The 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 following interface (whereas all methods are virtual and abstract).

TABLE 19. 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

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

8.1.11 The Interface of the File Client Type Classes

The current implemented Video Viewer does not provide but a very rudimentary GUI. The GUI can be seen (as a screen shot) in Figure 22 on page 68. 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 GUI 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.

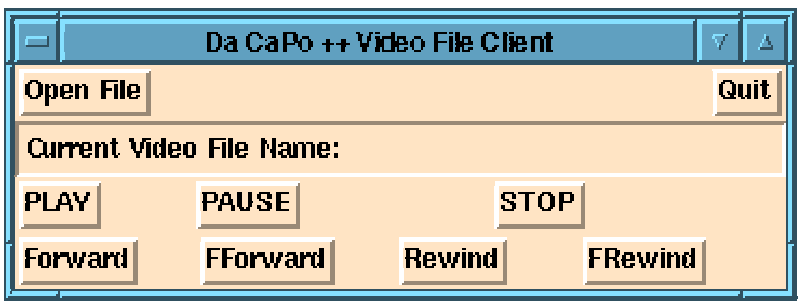


Figure 22 The Rudimentary GUI for the Video Viewer File Client

8.1.11.1 The Interface Class GenericFct

The class `GenericFct` was designed on account of the needs of Tcl/Tk. Pressing buttons like the “PLAY” button in Figure 22 on page 68 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.

8.1.11.2 The Class GenericFct

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

TABLE 20. 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	

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.

8.1.11.3 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:
     /*
      * instantiate class for the Video Viewer
      * let VideoViewer point to that instance
      */
     if (successful connection) {
```

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 is subject to be changed in August 1996, 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.

```
        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 File Client’s functionality. Not every File Client type class can call a given GUI function within its method `RunCtrlInterface` as the `CallFct` in the `GenericFct` may call itself functions not provided by the File Client type class.

8.1.12 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
   ```
   FileClient->CallFct(flag);
   ```
   whereas `flag` denotes the desired functionality.
2. A new class has to be derived from the already existing File Client type class which provides the following changes:

   TABLE 21. 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.

8.1.13 The 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 this document. The File Client type class implementing the Video Viewer is the class `VideoFileClCtrlComp` presented in the next section.

8.1.13.1 The 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
2. the class `GenericFct` defining the interface for the File Client.The class definition is given in Table 22 on page 71.

TABLE 22. 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 leas for Tcl/Tk)
	~VideoFileClCtrl-Comp	destructor		closes the session; calls the destructors of `GenericFct` and `ClCtrlComp`

TABLE 22. The Class VideoFileClCtrlComp

	name	type	inherited from	description + parameters
public	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 implemented 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. Will no longer be available after August (compare `Section 8.1.11.2 on page 69`).
pro-tected	Session *Video-FileSession	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
	char *read_file_name	attribute	GenericFct	this attribute will no longer be available after August (compare Section 8.1.11.2 on page 69).
	DaCaPoClient *DCmgr	attribute	ClCtrlComp	pointer to the Da CaPo Client

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

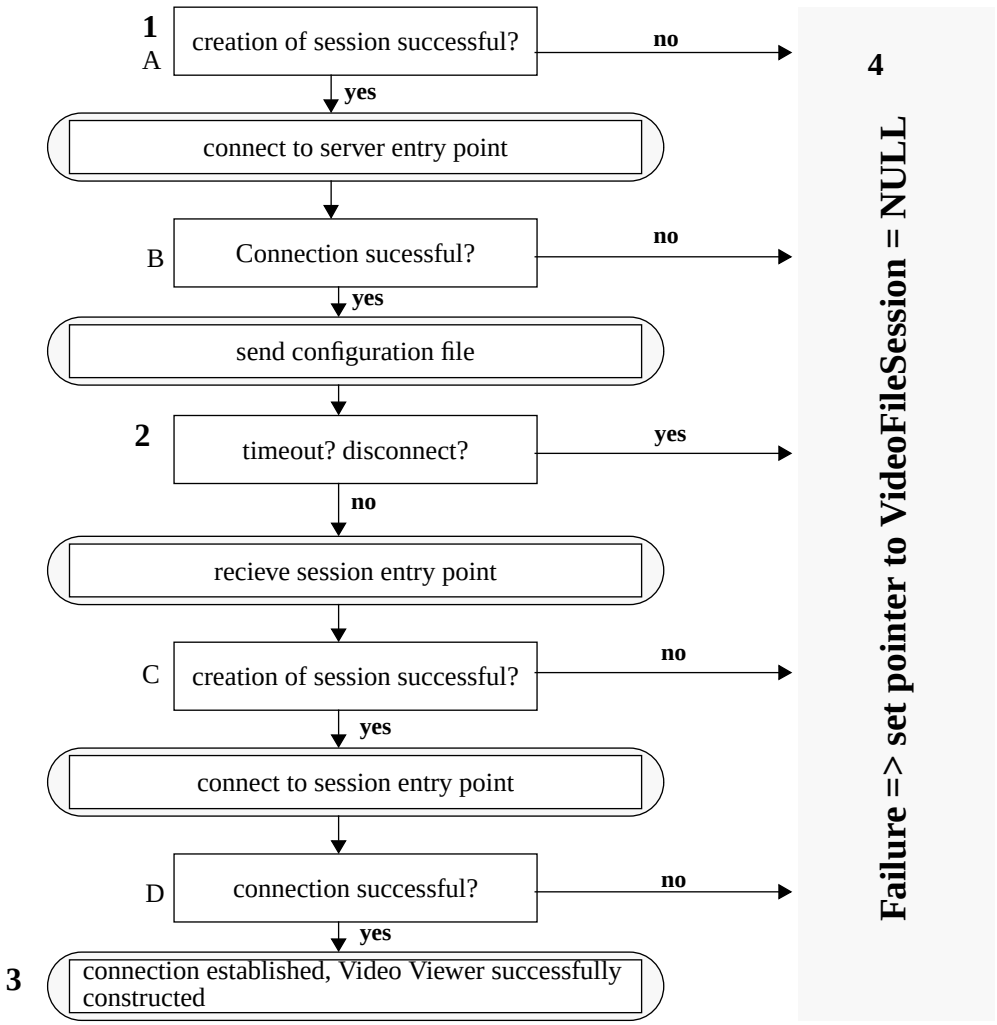


Figure 23 The Connection Establishment in the Video Viewer

8.1.13.2 The 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 23 on page 73 illustrates this process. In general the flow chart follows the process already shown partially in Figure 21 on page 66. 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 aa appropriate action in the application to avoid an a brute and improper aborting of the application.
- ad “timeout 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.

1. The disconnect event is generated when the File Server cannot create the demanded session and disconnects the entry and configuration session.
2. This may also include a faulty implementation of a File 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.[1]

8.1.13.3 The 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
- 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

8.1.13.4 The 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 23. Flags Defined for the Video Viewer

flag	meaning	control packet structure[a]
OPEN	send open command to A-Module	OPEN
PLAY	send play command to A-Module	PLAY
PAUSE	send pause command to A-Module	PAUSE
STOP	send stop command to A-Module	STOP
FWD	send forward command to A-Module	FWD
FF	send fast forward command to A-Module	FF
RWD	send rewind command to A-Module	RWD
FR	send fast rewind command to A-Module	FR
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

a. These packets are the same as defined for the SunVideoFile A-Module in table Table 18 on page 61

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 8.1.7.2 on page 58.

1. see also Table 18 on page 61.

8.1.13.5 The GUI of the Video Viewer

The graphical user interface of the Video Viewer is implemented in Tcl/Tk as can be seen in Figure 22 on page 68. 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): `PLAY, PAUSE, STOP, FWD, FF, RWD, FR, DISPLAY_FRAME, OPEN`

- `void RunVideoViewerGUI`

  This function implements the Tcl/Tk graphical user interface. The Tcl and the Tk command interpreters is 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 `src/video_demo/vfclient.tcl.`

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

8.1.14 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 needed 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 8.1.9.2 on page 65 the process of connection establishment in the File Server is not trivial, even more complex. As the data transmission is directly performed by 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.

8.1.14.1 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 ses-

1. The creation of the Tcl commands are already shown in the Section 8.1.11.1 on page 69.

sions and flow type information but also the port number of the session. The class for the user session needs the following methods:

TABLE 24. The Class for the User Session

	name	type	description
public	UserSession	constructor	
	get_name	method	returns name of user session
	get_first_session	method	returns first session
	get_next_session	method	return 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	
protected	session_list	Session-List[a]	points to the first element of the session list
	current	SessionList	points to the session to be retrieved from the list at next
	id	integer	number to identify the user session
	some user information		information required on 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 25. 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	UserSes-sionList	pointer to a list of user sessions
	current	UserSes-sionList	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.

8.1.14.2 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 26. 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
public	free_port	method	return value: void the functions adds a port to the flist 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

8.1.14.3 Connection Establishment and Closing

The processing of the Connection establishment for a user session is presented by the following state table.

TABLE 27. 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 27. 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
	timeout 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
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 adminis tration call destructor of U_i	END

TABLE 27. 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
	timeout of StC_j of U_i		disconnected 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
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

whereas

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

8.1.14.4 Error Cases in the File Server

Table 27 on page 77 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 closed
- error in the *sending of the SEP*:

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

8.1.15 Multicast Support (MCS)

The introduction of this new application component can be viewed in Figure 24 A new layer is introduced, namely the application support layer, between application and API. Purpose of this new layer is to offer to any application support for n:n multicast communication. Basically, the above mentioned application protocol should be hidden in this new component. Naturally the programmer still has the possibility to directly address the API as in (1), for additional functionality however, it can use the extended API offered in (2).

A new session object (McSession, more explanations in Section 8.1.15.1 on page 81) is instantiated in the upper API. which issues a control communication with its peer multicast support MSCs for notification exchanges. All group management activity (join of new participants, ...) is performed by these MSCs over Da CaPo control connections. The MSC behaves like a server, waiting for notifications relevant to the current application, and creating if necessary new sessions in the current VideoConference application (in the current example, new receiving sessions are created when a new participant wishes to join the VideoConference).

1. In this case the server is in State EC

Doing this new application component intelligent enough would enable to re-use it for different CSCW applications (and not only symmetric ones as for the current VideoConference). The main advantages of this solution are to meet requirements of the PicturePhone application and to keep connection manager as it is (no change necessary in kernel system).

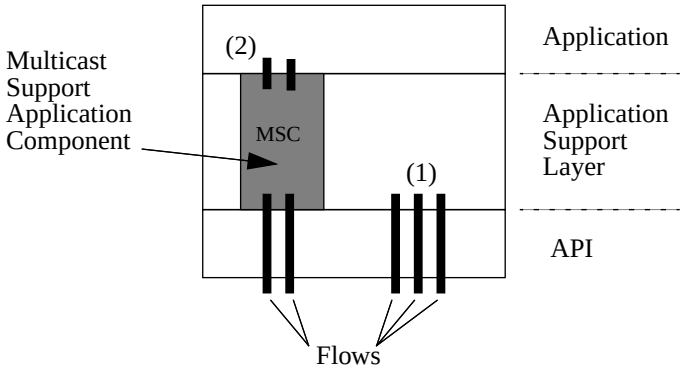


Figure 24 Multicast Support Component

8.1.15.1 The Multicast Support Object Model

The already mentioned multicast object is an instance of the following class:

```
class McSession {
     McSession(char *ConfigurationFile);
     ~McSession();
             // the ConnectMcSession method is used by both a
             // participant known at the beginning and by a late
             // “joiner”, in both cases, the Creator address
             // has to be provided
     ConnectMcSession(CREATOR|PARTICIPANT, ...);
     ConfigureMcSession();
     ActivateMcSession();
     ...
     LeaveMcSession()
     ...
}
```

The purpose of this McSession object is to provide additional functionalities at the application-application component layer interface with regards to the one offered by the upper API. This new session object is responsible to hide all necessary application protocols that is used to set up a Video Conference application (or, more general, a CSCW application).

It is important to remember that the McSession object is part of an application component and not part of the upper API as the traditional Session objects. Thus, an McSession object may encompass several traditional Session objects, which can be dynamically created or destroyed according to the application needs (either local and remote applications).

The constructor of the McSession object requires a reference on the application’s DaCaPoClient instance and a configuration file. This is similar to the traditional Session object’s constructor. The reference on the DaCaPoClient instance is the identical, however, the configuration file must internally reflect that there will not be an exact mapping between the objects at the Application-component and those in the upper API. For example, the number of receiving flows for both audio and video cannot be statically known, as it depends of the number of participants who join the VideoConference application. To reflect this, it is necessary to set up an extended script syntax for the configuration file, e.g, by providing a way to define generic flows that can then be instantiated several times according to the needs. A very coarse example of such a configuration file is provided below:

```
CREATOR MULTICAST
MCSESSION VideoConference;            // MCSESSION instead of SESSION
```

```
FLOW VIDEO_SEND_DEVICE VideoOut;       // direct mapping to flow
// several application requirements

FLOW AUDIO_SEND_DEVICE AudioOut;       // direct mapping to flow
// several application requirements

GENERIC VIDEO_RECV_DEVICE VideoRecv;   // may be several such flows
// several application requirements

GENERIC AUDIO_RECV_DEVICE AudioRecv;   // may be several such flows
// several application requirements

END MCSESSION
```

As for the 1:1 PicturePhone application, an AVPresentation object must be associated for each pair of receiving flows, this must be performed dynamically each time a new participant joins and is accepted in the VideoConference application. The graphical user interface contains two parts: a meta interface for the management of the “global” VideoConference application, and an independent user-interface associated to each AVPresentation object.

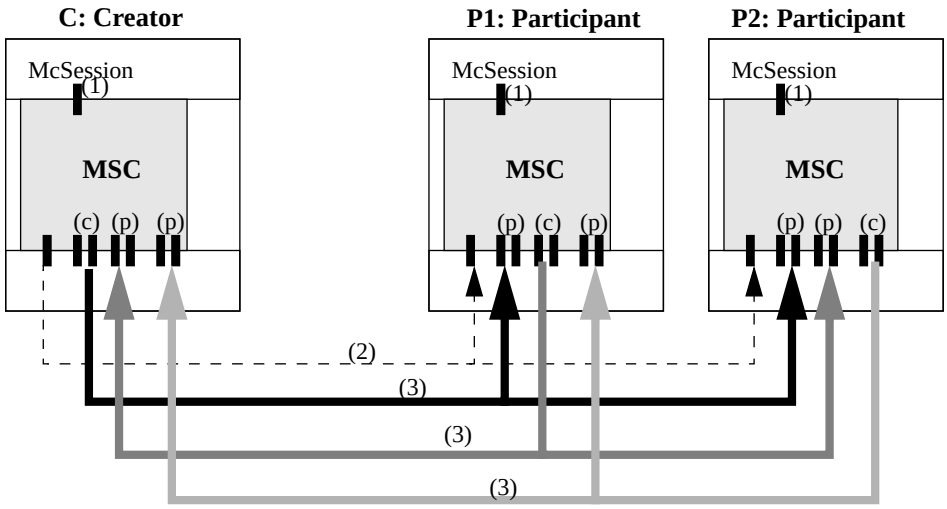


Figure 25 Example of a VideoConference application setup

8.1.15.2 Example: VideoConference Setup with the `McSession` Object

Let us consider one creator (C) and 2 participants (P1 and P2) as depicted in Figure 25

The proposed protocol is based on a session creator (C) which acts like a master to distribute notifications to all participants. Therefore it is a strongly centralized management. The setup process can be decomposed in the following steps:

1. Object creation:

   An McSession object is instantiated with the list of all participants addresses at the creator, and only the creator address at the participants.

2. Connect phase:

   A multicast Da CaPo control connection is set up between the creator and all participants. This control connection must be bidirectional for the purpose of the application protocol. Either one unicast Da CaPo control connection is set up between each participant and the creator, or the out-of-band signalling functionality of the connection manager can be used (in this case, the API must be extended to also offer this functionality to the application).

3. Data flows setup:

   Once the control connections are set up, it is possible to start with the requested application protocol (this protocol can be changed according to the needs of the distributed application). In a VideoConference application, it would look as follows (sites are referred as C, P1 and P2; "`x -> y : task`" means station x sends a notification to y with command "task"):

   ```
   C->P1 : "Create a receiving session for A/V flows coming from C"
   C->P2 : "Create a receiving session for A/V flows coming from C"
   ```

   On receiving these notifications, P1 and P2 will create the requested flows and register themselves by C. The multicast A/V flows are now set up between C and P1, P2.

   ```
   C->P1 : "Create a sending session for A/V flows"
   C->P2 : "Create a receiving session for A/V flows coming from P1"
   ```

   C and P2 will now create the requested flows and register themselves by P1 (being master, C knows which flows it has to create). The multicast A/V flows are now set up between P1and C, P2. When this is performed, C takes control again:

   ```
   C->P2 : "Create a sending session for A/V flows"
   C->P1 : "Create a receiving session for A/V flows coming from P2"
   ```

Finally, the multicast session is properly set up. By keeping an internal table with the status of each connection (involving all participants who are known at the beginning), the Creator MSC can issue an ActivateSession() on all his sessions as soon as all multicast connections are properly set up. As all participants have already executed an ActivateSession(), this enables the Creator to get a synchronized start with all participants (this property may not be necessary for a typical VideoConference application with live audio/video transfer, but it has to be provided for other applications, e.g., where the Xwedge component is used).

An interesting property of this application component McSession object is that the creator of such an object is actually both creator and participant for several traditional Session objects. In a similar way, the participant of a McSession object is both creator and participant for several traditional Session objects. This situation is illustrated in XXX, where (c) and (p) denote the actual creator and participant of traditional Session objects.

This mapping of the McSession object in all necessary Session objects is done transparently by the application component. Therefore the programmer has no access to the dynamically created traditional Session objects. The only way how to access them must be offered at the McSession object layer (through a set of available methods).

9. Data Transmission Levels and Error Levels

According to the hierarchical structure of the design we can distinguish different kinds of data and errors. The following Subsections give a brief overview. Distinguished are the conceptual level and the implementation level, as Da CaPo A-modules may provide data input and output.

9.1 The Conceptual Level

9.1.1 File Data

The data stored in and read from the file conceptually is known within the file control on the server side and within the control interface on the client side. The idea of the design is that all underlying components do not know which type of file data is to be transmitted and presented. For all other components these data are data streams of any type. As an appropriate A-module has to be instantiated to transmit specific data over a Da CaPo++ link, this conceptual transparency of the data type can not really be implemented. The API has to know which data type has to be transmitted.

9.1.2 Control Data

The file control commands (control data) come from the user and influence the retrieval of data from the file. Conceptually these data are only known in the file control component in the server resp. the file control interface in the client.

9.1.3 Connection Link

The server control component on the server side resp. the connection control component on the client side are the only components that have information on the connection link and that create and receive control data concerning the network connection (Da CaPo++ connection).

9.1.4 Error Messages

We can distinguish two different kinds of errors that can occur. Some errors concern the file to be read. These errors have to be treated within the file control component on the server side. If they can not be handled properly appropriate error messages have to be created for the user that should be presented within the file control interface level. Other errors concern the connection link level. If they are not handled directly in Da CaPo the error handling has to be provided within the server control component on the server's site resp. the connection control component on the client's site.

9.2 The Implementational Level

9.2.1 File Data

The file data only is known in the two A-modules on the two sides. This data is directly transmitted via the Da CaPo kernel to the other side. The Upper API as well as the application never sees this data. The server's A-module directly transmits the video data directly using the Da CaPo++ link to the peer. The file data will directly be presented on the monitor on the clients side. This is provided by the A-module and the used video card, without using any functioning the API or the application.

9.2.2 Control Data

The control data in this context are the control commands provided by the video file control interface after a user command occurred. This data is passed to the client control command. The client control command passes the data to the corresponding A-module. The A-module performs the commands, if it

can do so locally, otherwise the control command is transmitted to the other sides A-module where the command directly is performed. The command is not passed via the application.

9.2.3 Connection Link

The control of the connection link is done by the Upper APIs on both sides.

9.2.4 Error Messages

Occurring errors referring to file data are handled, as far as possible, within the A-modules. If an error can not be handled there, appropriate error messages have to be provided that are transmitted to the control component[1] that generates an error message for the user. When a file error occurs on the server site, an appropriate error message has to be created and transmitted via Da CaPo++ to inform the user.

All other errors are handled and error messages are created, if necessary, within the component where the errors occurred.

1.This is done by returning a non-zero value in the function where the error occurred.