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

1. **Introduction and Goals** **1**
 1.1 Brief Survey of Da CaPo 1
 1.2 Structure of this Architectural Document 2
2. **Table of Contents** **3**
3. **Listof Figures** **5**
4. **List of Tables** **6**
5. **Application Programming Interface (API)** **7**
 5.1 Design 7
 5.2 API Components 7
 5.3 Upper API 7
  5.3.1 Manager Objects 8
  5.3.2 Session and Flow Objects 8
   5.3.2.1 Variant A: 9
   5.3.2.2 Variant B: 10
   5.3.2.3 Variant C: 10
 5.4 Lower API 13
 5.5 IPC between Application and Da CaPo 14
  5.5.1 Data and Control Interfaces 15
 5.6 A-Module 16
  5.6.1 Concurrency for control and data information 16
  5.6.2 QoS Mapping 16
 5.7 Buffer Management 16
  5.7.1 Structure 16
  5.7.2 Functionality 17
6. **Security Aspects of Da CaPo++** **18**
 6.1 Security Design Overview 18
  6.1.1 Assuring Authenticity: Associations and Identities 18
  6.1.2 Specifying and Translating Security Requirements 19
  6.1.3 Protocol Management, Reconfiguration and Keying 19
  6.1.4 Security Assurance at Runtime 19
  6.1.5 Keys and Certificates 20
 6.2 Discussion of Components 20
  6.2.1 API 21
  6.2.2 QoS Parameters 21
  6.2.3 C-Modules 22
  6.2.4 Protocols 22
  6.2.5 Key Database 22
  6.2.6 Security Manager 23
   6.2.6.1 Association Block 23
   6.2.6.2 Attribute Translation Block 23
   6.2.6.3 Protocol Control Block 23
   6.2.6.4 Key Manager Block 24
  6.2.7 Runtime Security Assurance 24
 6.3 Open questions: 24
7. **Multicast Aspects of Da CaPo++** **25**
 7.1 Multicast-capable Connection Manager 25
  7.1.1 The Creator's ConMan 26
  7.1.2 Participants' ConMan 26

7.1.3 ConMan Error Control Protocol 27
7.1.4 Finite State machines 28
7.1.4.1 Finite State Machine of Creator's ConMan 8
7.1.4.2 Finite State Machine of Participant's ConMan 29
7.2 Multicast Transport Protocols 29
7.2.1 Reliable Multicast Protocol 30
7.2.2 Simple Multicast Protocol 31
7.2.3 Changes in the Existing Da CaPo Kernel – Attributes 31

8. Application Framework of Da CaPo++ 32

8.1 Applications 32
8.1.1 Picture Phone (PP-APP) 32
8.1.1.1 Design 32
8.1.2 Video Conference (VC-APP) 33
8.1.2.1 Video Conference Setup 33
8.1.3 Extended WWW Browser and Server (EWB-APP) 35
8.2 Application Components 36
8.2.1 File Server 36
8.2.1.1 Server Control Component 36
8.2.1.2 File Control Component 36
8.2.1.3 Da CaPo++ Video File Server 37
8.2.1.4 Class Design 37
8.2.2 File Client 38
8.2.2.1 Connection Control Component 38
8.2.2.2 Client Control Component 39
8.2.2.3 File Control Interface 40
8.2.2.4 Da CaPo Video File Viewer 40
8.2.2.5 Class Design 40
8.2.3 Group Management Comfort (GMC) 41
8.2.4 Multicast Support (MCS) 42
The Multicast Support Object Model 42

9. Error Handling 44

9.1 Data Transmission and Error Levels for File Server and Client 44
9.1.1 File Data 44
9.1.2 Control Data 44
9.1.3 Connection Link 44
9.1.4 Error Messages 44

3. List of Figures

Figure 1	API’s main components	7
Figure 2	Upper API Objects	8
Figure 3	Instantiable Flow Classes	9
Figure 4	Data and Control Interfaces	15
Figure 5	Buffer Management Structure	17
Figure 6	Security Architecture in Da CaPo++	20
Figure 7	Connection Manager Protocol Stacks	25
Figure 8	Connection Manager Data Flow	28
Figure 9	Connection Manager Protocol Stacks	28
Figure 10	Creator’s Finite State Machine	29
Figure 11	Participants’ Finite State Machine	30
Figure 12	Example of a Video Conference application setup	34
Figure 13	Extended WWW Browser	35
Figure 14	Da CaPo++ File Server	36
Figure 15	Da CaPo++ Video File Server	37
Figure 16	Class Design	37
Figure 17	Da CaPo++ File Client	38
Figure 18	Da CaPo++ Video File Viewer	40
Figure 19	Class Design	41
Figure 20	Multicast Support Component	42

4. List of Tables

TABLE 1. Advantages and Disadvantages for Variant A 10
TABLE 2. Advantages and Disadvantages for Variant B 10
TABLE 3. Advantages and Disadvantages for Variant C 11
TABLE 4. API Functionality 12
TABLE 5. Session Table Structure 14
TABLE 6. Flow Table Structure 14
TABLE 7. Buffer Management Interface... 17
TABLE 8. Parameter for Privacy 21
TABLE 9. Parameter for Authentication 22
TABLE 10. Creator‘s Connection Manager Requests... 26
TABLE 11. Participant´s Connection Manager Requests...................................... 27

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.

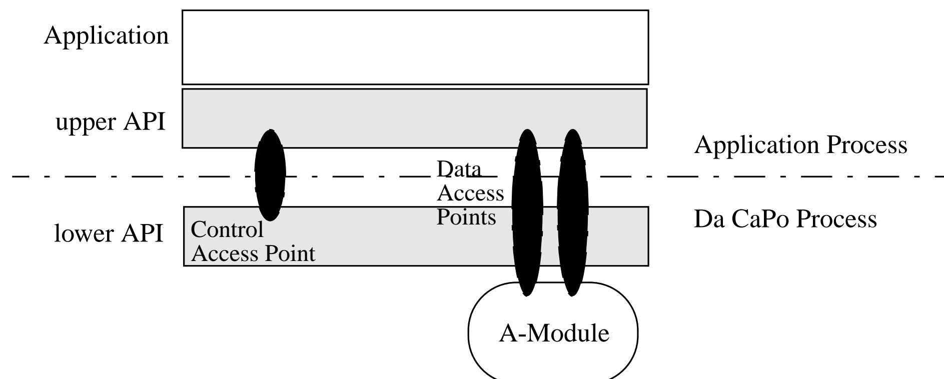
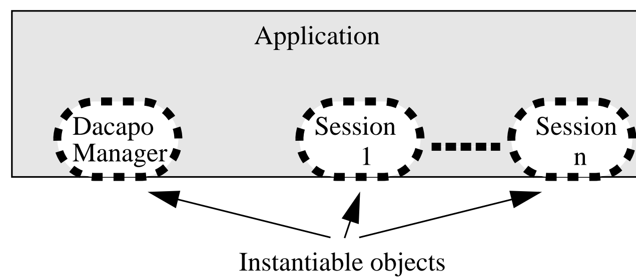


Figure 1API's main components

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.).

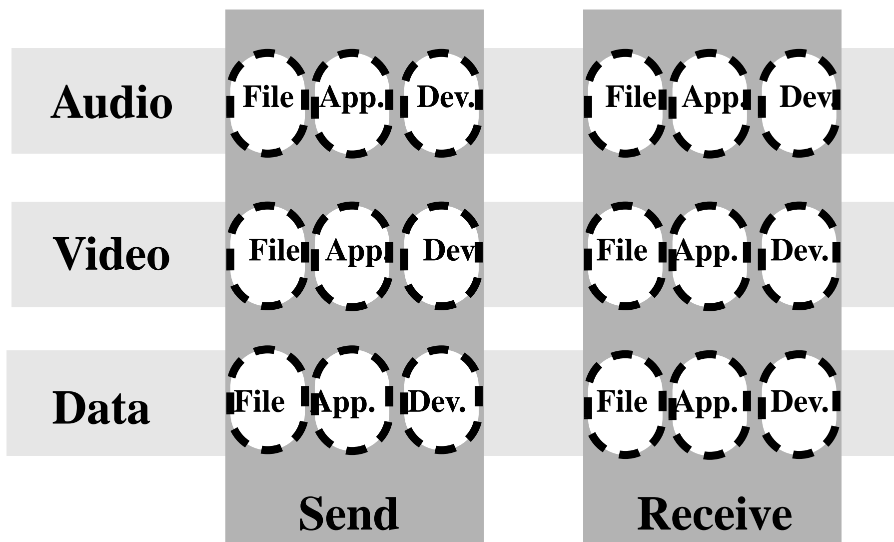


Figure 3Instantiable Flow Classes

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 *configurationFile, 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

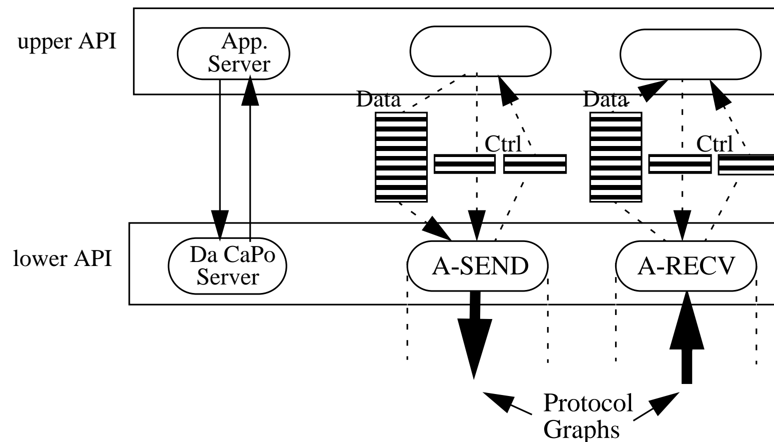


Figure 4Data and Control Interfaces

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:

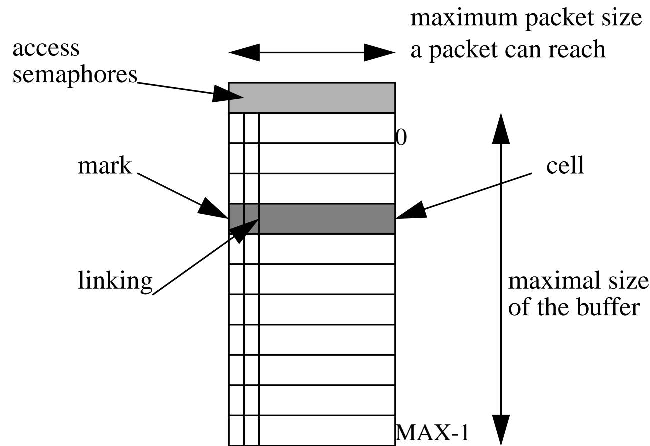


Figure 5Buffer Management Structure

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.

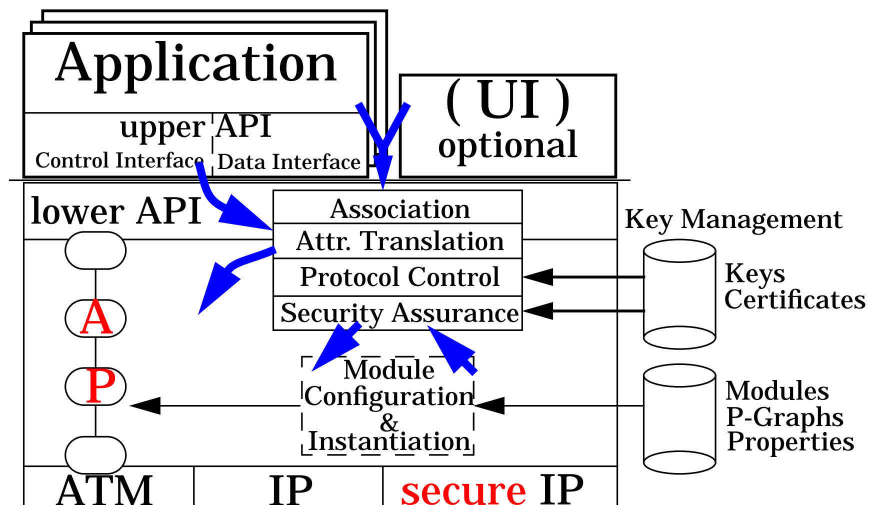


Figure 6Security Architecture in Da CaPo++

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.

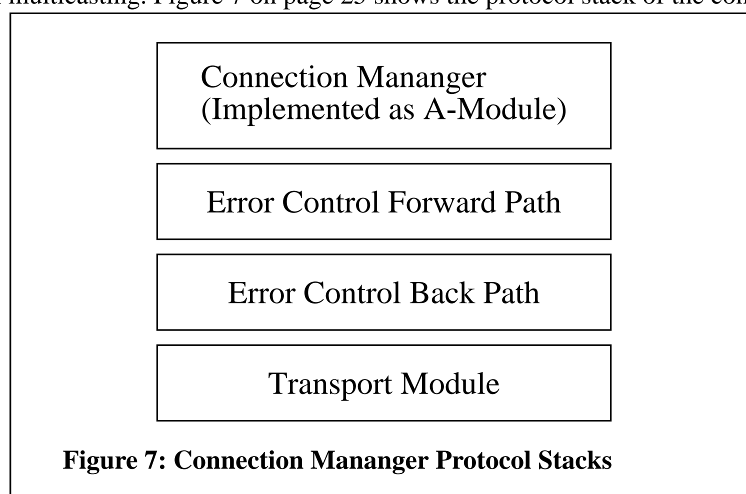


Figure 7: Connection Mananger Protocol Stacks

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.

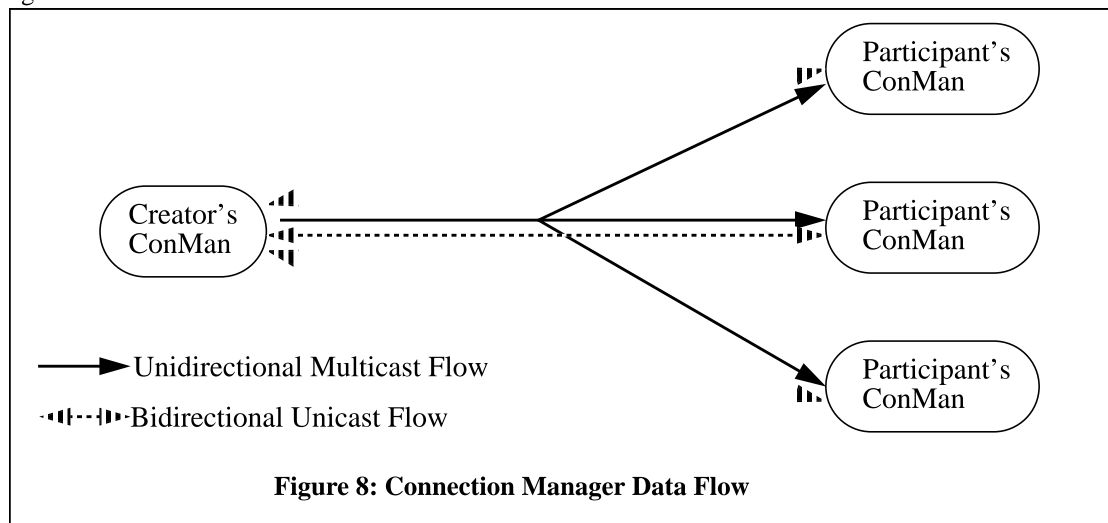


Figure 8: Connection Manager Data Flow

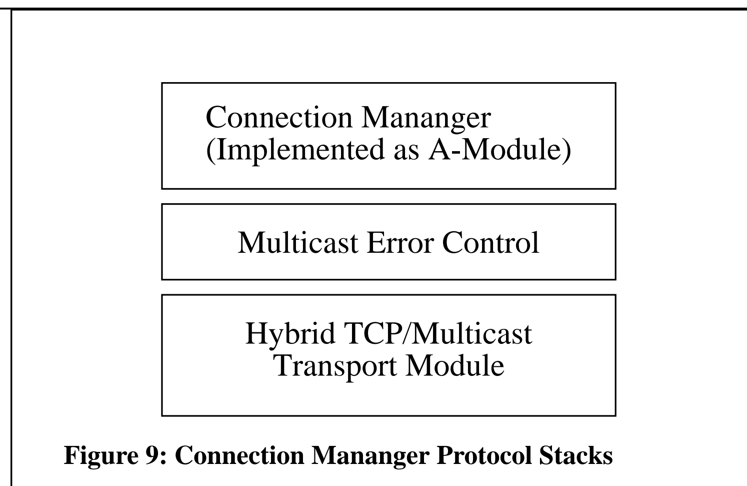


Figure 9: Connection Mananger Protocol Stacks

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.

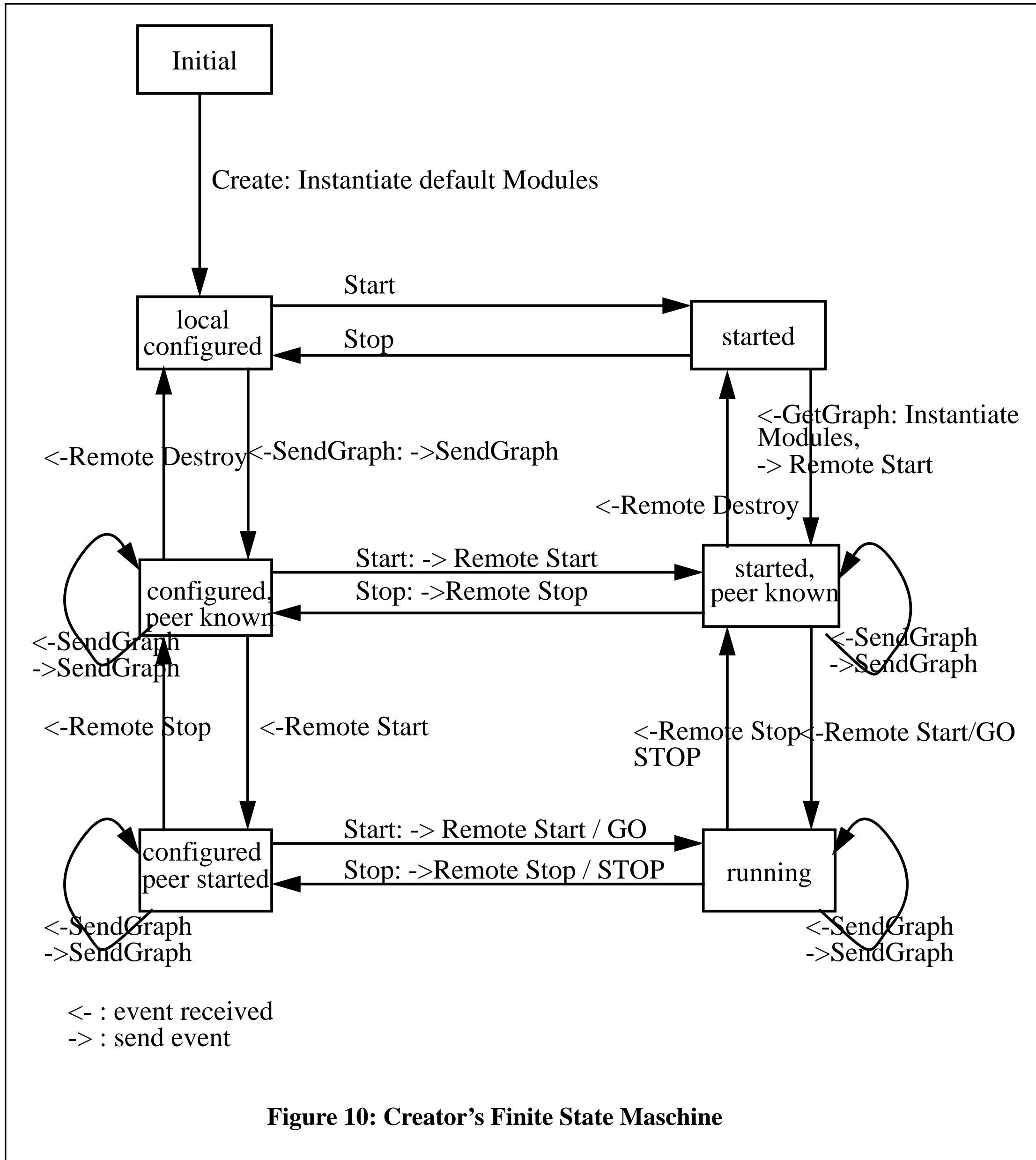


Figure 10: Creator’s Finite State Maschine

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.

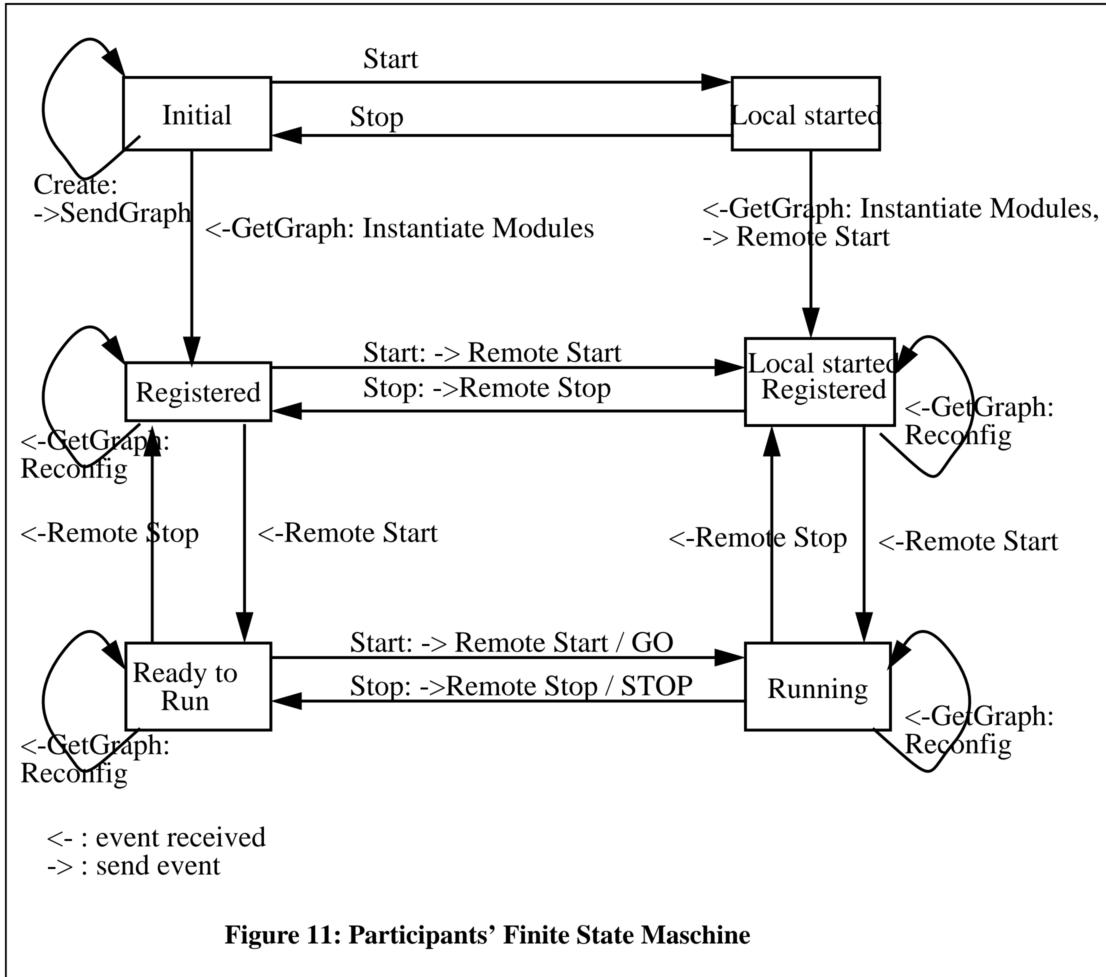


Figure 11: Participants' Finite State Maschine

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).

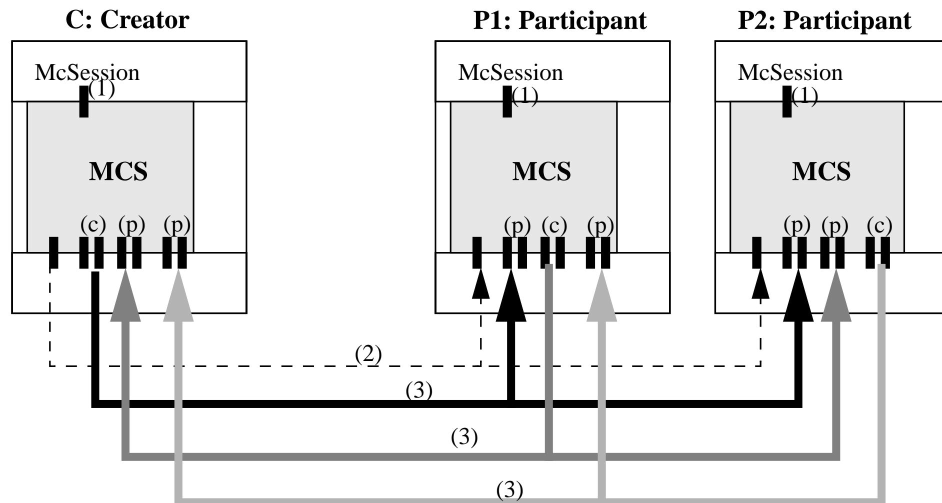


Figure 12Example of a Video Conference application setup

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.

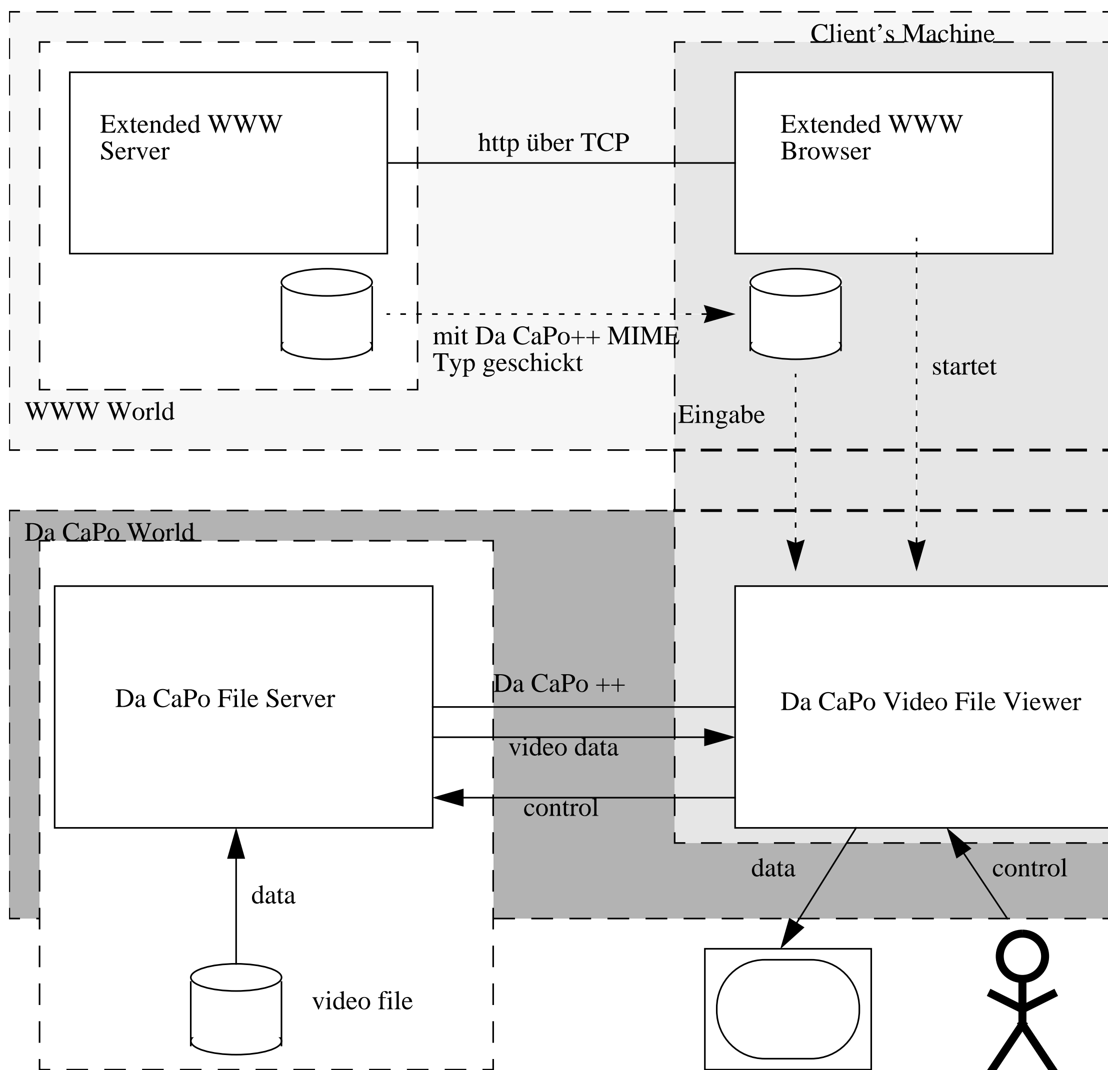


Figure 13Extended WWW Browser

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.

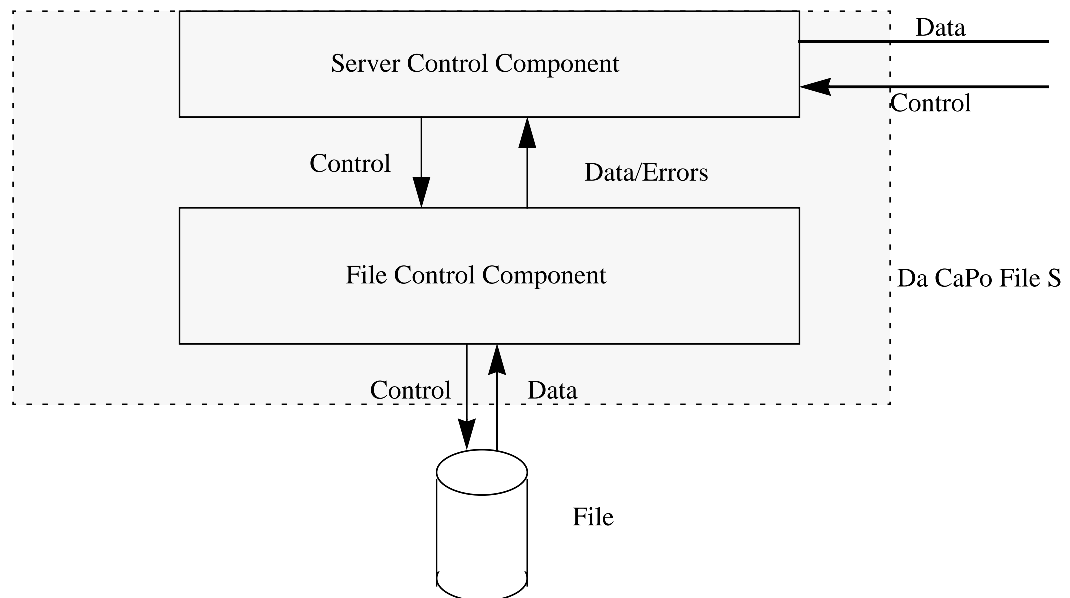


Figure 14Da CaPo++ File Server

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.

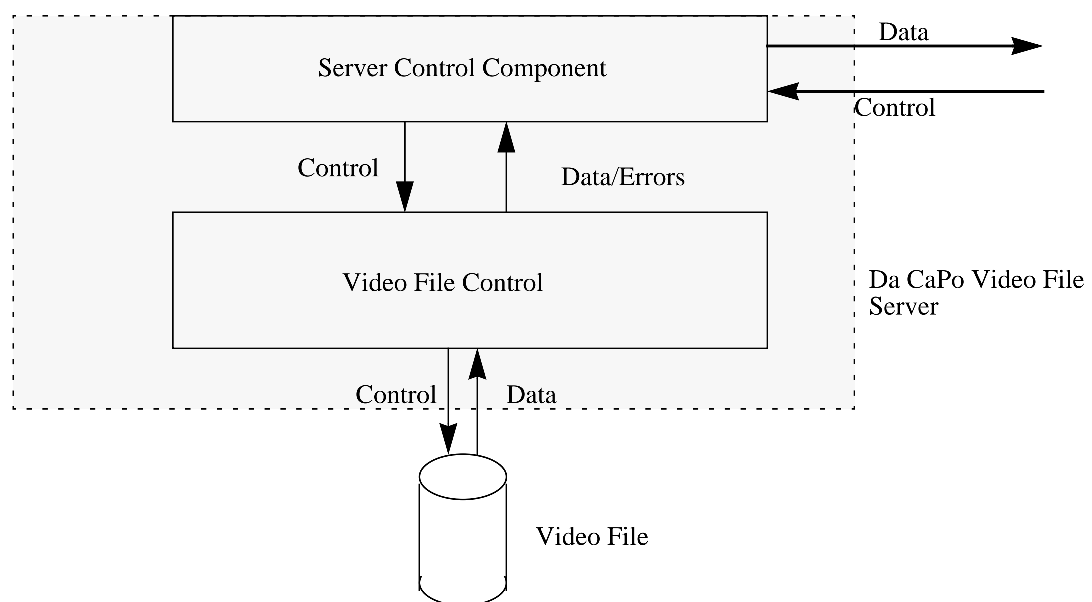


Figure 15Da CaPo++ Video File Server

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):

class ServerControl

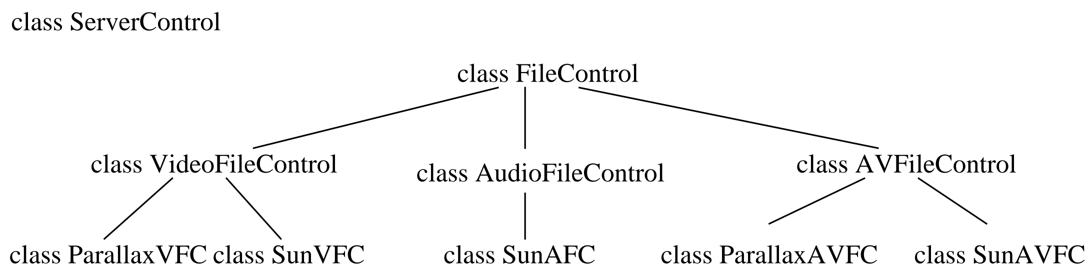


Figure 16Class Design

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.

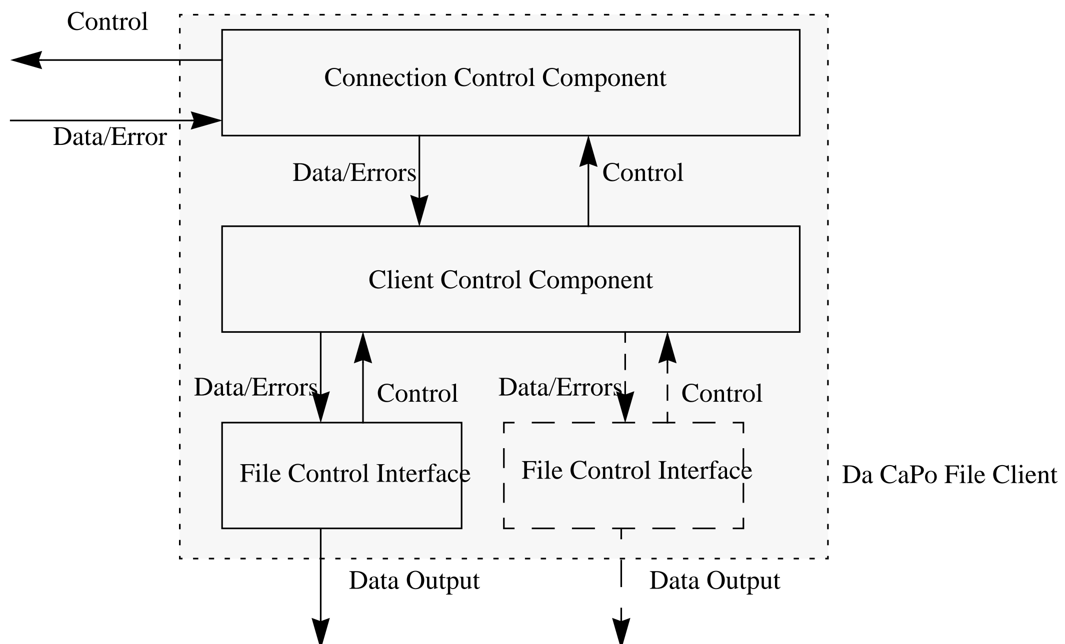


Figure 17Da CaPo++ File Client

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.

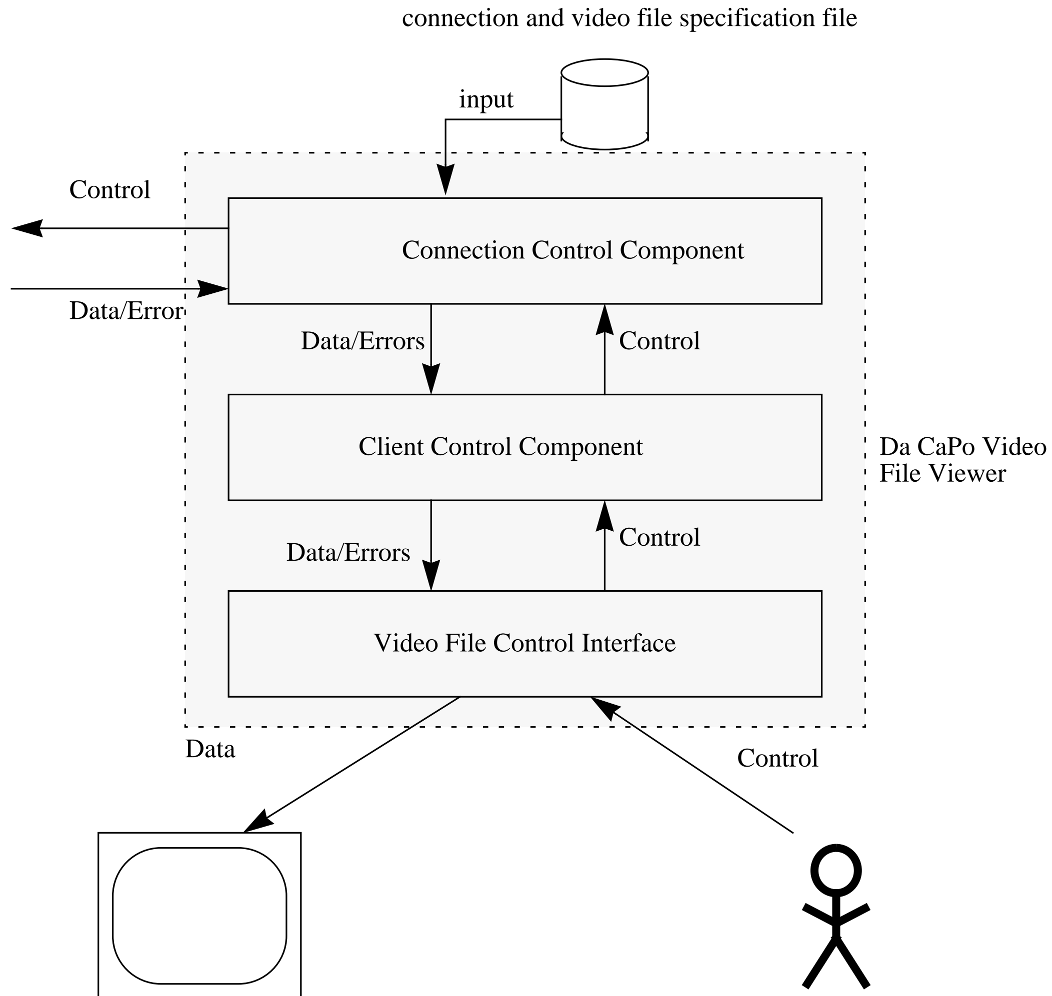


Figure 18Da CaPo++ Video File Viewer

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.

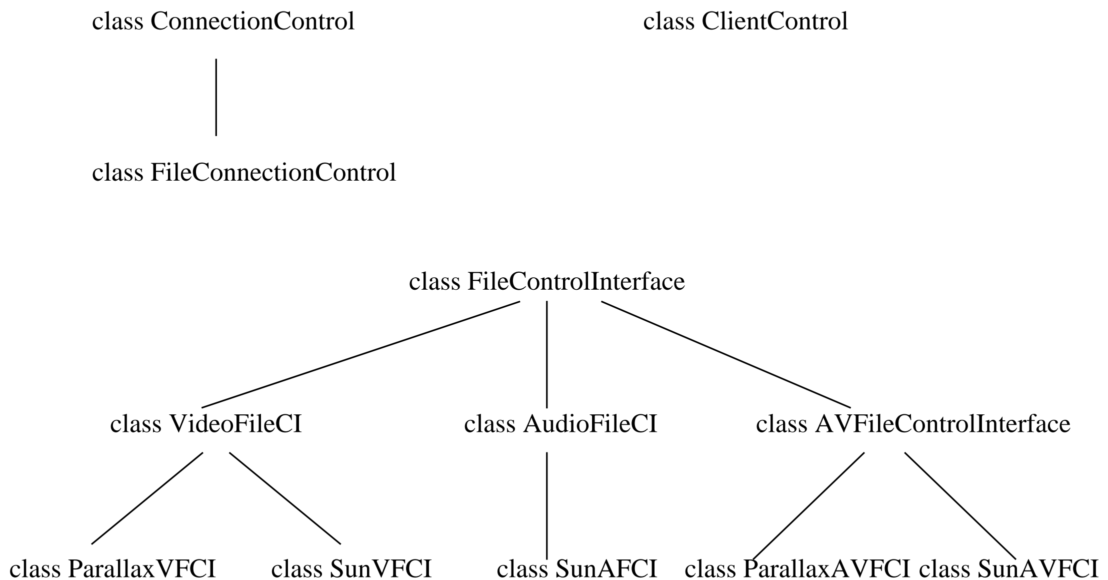


Figure 19Class Design

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.

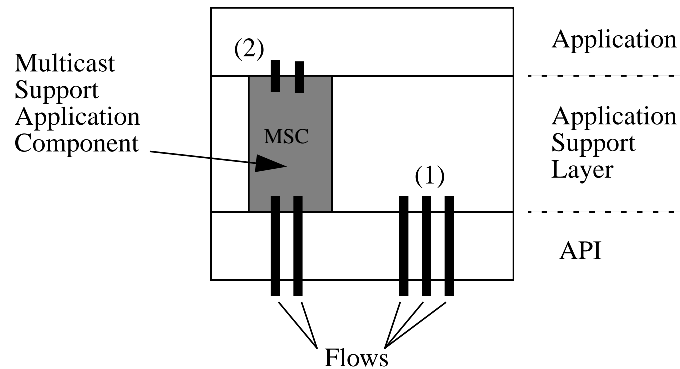


Figure 20Multicast Support Component

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.