Project Da CaPo++, Volume III:
Performance Evaluations

TIK-Report No. 42

Burkhard Stiller, Germano Caronni, Christina Class, Christian Conrad, Bernhard Plattner, Marcel Waldvogel
Computer Engineering and Networks Laboratory (TIK)
ETH Zürich, Gloriastrasse 35, CH – 8092 Zürich, Switzerland
Phone: +41 1 632 7016, FAX: +41 1 632 1035
E-Mail: <last-name> @ tik.ee.ethz.ch

Abstract

Performance evaluations of advanced communication subsystems and their applications are necessary methods to prove that a certain level of communication functionality demanded and a required minimal processing power of end-systems (workstations) and of intermediate systems (networks and routers) have been achieved. The communication middleware package Da CaPo++ provides a modern communication platform that supports flexible communication services, in particular for end-systems. Based on an implementation of Da CaPo++ on workstations (Sun SPARCStations and Sun UltraSPARCs) a performance evaluation has been carried out. Specifically, the performance for relevant communication tasks is identified and overhead required for providing various degrees of communication service flexibility is illustrated.

1. General Information

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

This report is structured in one single part and contains the documentation on performance evaluations performed within Da CaPo++, which is based on the architectural and detailed design as described in TIK-Report No. 28 and the implementation documentation as described in TIK-Report No. 29.

This page has been left blank intentionally.

2. Table of Content

1. General Information **1**

2. Table of Content **3**

3. List of Figures **4**

4. List of Tables **4**

5. Introduction **7**

6. Performance Evaluations **8**

6.1 General and Particularly Communication-related Issues 8
6.2 Special Features of Da CaPo++ 9

7. Performance Measurements of Da CaPo++ **9**

7.1 Technical Environment 9
7.1.1 Networks 9
7.1.2 Workstations – Hardware and Software 9
7.1.3 Da CaPo++ Thread and Process Model 10
7.1.3.1 Da CaPo++ Server 10
7.1.3.2 Applications 11
7.1.4 Measurement Method and System 12
7.2 Measurement Scenarios 12
7.2.1 Da CaPo++ Core: Lift 13
7.2.2 Da CaPo++ Core: C-modules 14
7.2.2.1 Overview figures 14
7.2.2.2 Detailed Analysis 14
7.2.2.3 Sending Side 15
7.2.2.4 Authentication 17
7.2.2.5 Encryption 19
7.2.2.6 Receiving Side 19
7.2.2.7 Overview of Measurement for Every C-module 21
7.2.3 Da CaPo++ Core: A-modules 23
7.2.3.1 AudioFile A-module 23
7.2.3.2 SunVideoFile A-module 29
7.2.3.3 RawAPI A-module 42
7.2.4 End-to-End Issues: Protocols, Connection Manager, and Security Manager 46
7.2.4.1 Protocol Times for Audio and Video 49
7.2.4.2 Protocol Times for an Unreliable and Reliable Data Transmission Protocol 50
7.2.4.3 Protocol Times for Secured Protocols 52
7.2.4.4 Connection Manager and Security Manager 52
7.2.5 API Measurements 52
7.2.6 Application Framework 60
7.2.6.1 The WWW Application Scenario 60

8. Summary **63**

9. References **64**

3. List of Figures

Figure 1. Architecture of the Da CaPo++ Middleware Package 7
Figure 2. Thread Model of the Da CaPo++ Core 11
Figure 3. Sender Call Graph Overview 17
Figure 4. Sender Initialization 18
Figure 5. Sender Processing 19
Figure 6. Receiver Call Graph Overview 21
Figure 7. Comparison of Some Security C-modules 23
Figure 8. Some Security Module Combination Measurements 23
Figure 9. The Function Hierarchy in the AudioFile A-module (Sender) 25
Figure 10. The Function Hierarchy in the AudioFile A-Module (Receiver) 29
Figure 11. The Function Hierarchy in the SunVideoFile A-module (Sender) 31
Figure 12. The Function Hierarchy in the SunVideoFile A-module (Receiver) 40
Figure 13. The Function Hierarchy in the RawApi A-module (Sender) 43
Figure 14. The Function Hierarchy in the RawApi A-Module (Receiver) 46
Figure 15. The Measurement Scenario for the Audio and the Video Protocol 48
Figure 16. Video Protocol Processing Times 50
Figure 17. Audio Protocol Processing Times 51
Figure 18. The Measurement Scenario for the Data Transmission Protocols 51
Figure 19. API Measurement Process 54
Figure 20. Measurement Scenarios 55
Figure 21. Throughput Measurements Application-to-application 59
Figure 22. Throughput Measurements With Variable Packet Size 60
Figure 23. The Measurement Scenario for the Entry Session 62
Figure 24. Measurement scenario for the response time of commands 63

4. List of Tables

Table 1. Lift Overhead [in µs] 14
Table 2. Lift and Resource Manager Overhead [in µs] 14
Table 3. Security C-module Measurements in [sec] 15
Table 4. Sender Initialization Measurements 16
Table 5. MD5 Measurements 22
Table 6. MD4 Measurements 22
Table 7. DES Measurements 22
Table 8. IDEA Measurements 22
Table 9. RC5-12-16 Measurements 22

Table 10. Algorithm Costs 24
Table 11. Quantify Values for the Sender (User Time) 25
Table 12. Detailed Measurements of asIndiAudio (Real Time) 27
Table 13. Quantify Values for the Receiver (User Time) 29
Table 14. Detailed Measurements on arRequAudio (Real Time) 30
Table 15. Video Data Characteristics 30
Table 16. Detailed Measurements of the Sender of SunVideoFile (Real Time and Total Costs for Functions) 32
Table 17. Detailed Measurement of asIndiVideoFile (Real Time) 32
Table 18. Detailed Measurements of the Out-of-band Communication (Real Time) 33
Table 19. OOB Commands Separately (Real Time) 33
Table 20. Data of Sender_OoBCommand 34
Table 21. Data of the Control Commands 34
Table 22. Measurements for the Control Commands in asIndiVideoFile (Real Time) 37
Table 23. Detailed Results for asIndiVideoFile After the 2nd Frame in the CIS 38
Table 24. The Out-of-band Commands (Real Time) 39
Table 25. Detailed Measurements of the Receiver of SunVideoFile (Real Time) 41
Table 26. Detailed Measurement of arRequVideoFile (Real Time) 42
Table 27. Quantify Values for the Sender (User Time) 44
Table 28. Detailed Measurements of data_back (Real Time) 44
Table 29. Detailed Measurements on source_requ (Real Time) 45
Table 30. Quantify Values for the Receiver (User Time) 46
Table 31. Detailed Measurements on sink_requ (Real Time) 47
Table 32. The SunVideoFile Protocol 49
Table 33. The AudioFile Protocol 50
Table 34. Unreliable Data Transmission Protocol 52
Table 35. Reliable Data Transmission Protocol 52
Table 36. DaCaPoClient Creation Measurements 56
Table 37. Session Creation Measurements 56
Table 38. Session connect/listen Measurements 57
Table 39. Session activate/deactivate Measurements 57
Table 40. Session close Measurements 58
Table 41. Session Sending and Receiving User Data Measurements 59
Table 42. Throughput Measurements With Variable Packet Size 60
Table 43. Measurements for the Entry Session 62
Table 44. Entry Session (Evaluation) 62
Table 45. Response Time of Commands 64

5. Introduction

Da CaPo++ has been designed for providing flexible communication services for advanced multimedia applications /SBCC97a/. Its overall abstract architecture is depicted in Figure 1. Based on a variety of available network services, such as unicast and multicast ATM (Asynchronous Transfer Mode), unicast or multicast IP-based (Internet protocol) networks, or even secure IP, the Da CaPo++ communication subsystem – or Da CaPo++ core for short – is located. This core is viewed as a low-level middleware for communications. It provides communication services that are supported by communication protocols have been configured of single protocol functions according to current application demands. Each of these protocol functions are implemented in a separate module providing a specific task, such as network access, compression, error-control, or security functionality. An application programming interface (API) is offered on top of the Da CaPo++ core to enable efficient and straight-forward multimedia application development. Due to a logical and implementatory boundary between the Da CaPo++ core and applications the API has been divided into two separate parts. The lower API is provided once per Da CaPo++ core which is available on a networked workstation once as well. It abstracts away communication-specific details of the protocol configuration, single protocol functions and modules, and the variety of network services. In addition, it offers a well-known access for applications to the generic service interface of the Da CaPo++ core that is capable of providing a variety of communication services.

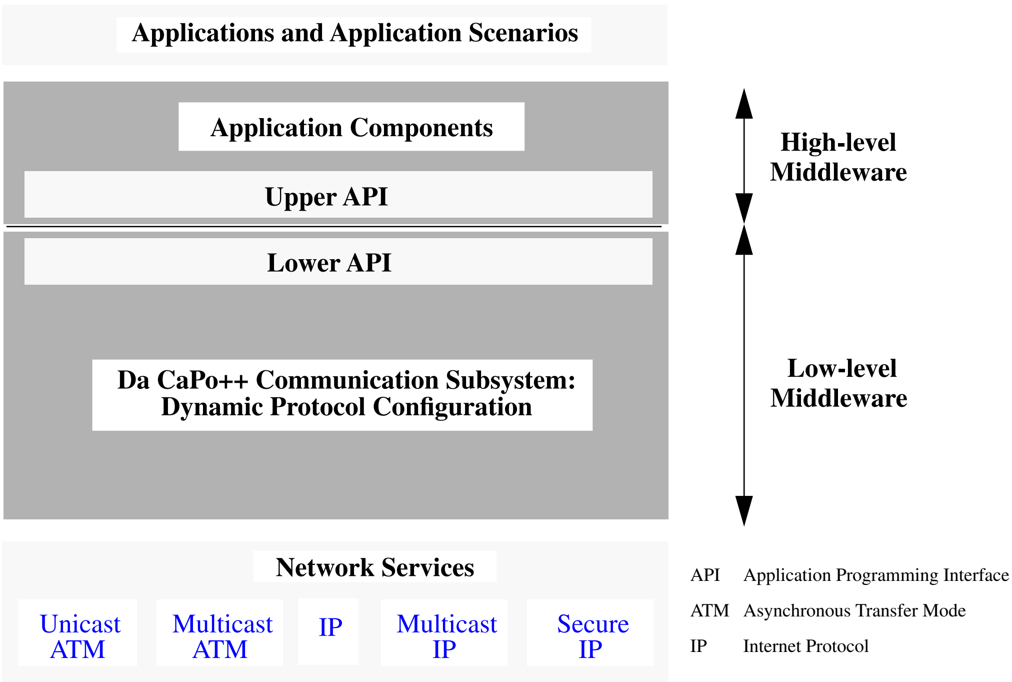

Figure 1. Architecture of the Da CaPo++ Middleware Package

The high-level middleware of the Da CaPo++ approach consists of the upper API and application components. The upper API communicates with the lower API as user data or control data have to be exchanged. This exchange is required initially to bind an application with the Da CaPo++ core and initialize a communication association. User data transfers are done subsequently. Upper APIs have to be linked with every single application, in particular the application component to be able to use Da CaPo++ communication services. Since multimedia applications always show a number of similar functions to be performed, application components comprise these functions in practical building blocks, such as an audio and video component. Particularly, certain access modules (A-modules) within the Da CaPo++ core utilized for a special communication protocol provide features to deal with multimedia devices, such as cameras, microphones, and speakers. The allow for the bypassing of user data, which is the covers the direct sending and receiving to and from devices via the Da CaPo++ core under control of the application. In this case, user data are not passed between the upper and lower API.

Finally, application and application scenarios may be defined utilizing the high-level middleware. These upper domain issues are handled within a defined application framework offering re-usable elements for modular application development /Stil96/. In any case, low-level middleware has to be accessed by the lower API. This architecture of the Da CaPo++ middleware package is suitable of supporting multimedia applications that require advanced protocol and network characteristics.

6. Performance Evaluations

Communication services and protocols are the key issue in determining performance of communications. Emerging multimedia applications require certain means by which their success can be measured. Performance evaluations provide numbers that allow for the identification of advantages and drawbacks of current settings and scenarios. Furthermore, the protocol processing environment, the run-time system, is evaluated to determine its influence.

Unfortunately, a communication subsystem, such as Da CaPo++, is a very complex system with many points of interactions. Therefore, the following performance measurements are based on detailed investigations of the entire system, identifying a sufficient number of spots that clearly express the strength of the communication subsystem, the communication protocols employed, the application programming interface, and the networks utilized. In addition, a number of applications have been implemented on top of the middleware package. Therefore, an end-to-end performance of a variety of advanced multimedia applications has been investigated.

6.1 General and Particularly Communication-related Issues

According to the basics of performance evaluations of running systems /Tane96/ the following aspects have to be clearly taken care of:

- Use a sample size that is large enough to obtain an almost certain mean value and a significant standard deviation.
- Make sure that the samples collected are representative.
- Identify the resolution of clocks applied to interpret the measured times correctly.
- Set up a scenario that does limit system activity at the time of measurement to tasks only that you are interested in.
- Identify clearly all influences of a certain end-system architecture, *e.g.*, the caching architecture, hardware support for audio/video processing, or network interface boards, that will effect the performance measurements undertaken.
- Understand what you are measuring to obtain the figures you are looking for in the end.
- Do not extrapolate results thoughtlessly without considering some theory behind queues, drivers, and other factors.

In addition to these general aspects that are valid not only for communication measurements, but comprise principles for performance measurements in a broad sense, communication performance needs to particularly investigate the following issues:

- Separate end-system or platform-dependent figures, *e.g.*, including CPU speed and memory bandwidth, from network-dependent numbers, such as network delay and network bandwidth.
- Specify usage of operating system functions for handling communication tasks.
- Reduce the protocol overhead per data unit to be transmitted.
- Minimize context switches and define an applicable process/thread model
- Minimize copying operations.
- Define phases for connection management purposes (initialization, set-up, operation, and tear-down) and distinguish performance measurements between them explicitly.

Finally, performance figures measured require a unique specification of their semantics. In addition to system-inherent parameters, such as the number of data units to be processed or network interface access delays, a number of application or user expressive parameters are required, such as end-to-end delay, throughput achievable, or quality of transmission.

6.2 Special Features of Da CaPo++

As the communication subsystem Da CaPo++ covers mainly end-system issues /SBCC97a/, the network prerequisites are required to be stated in advance. Therefore, two different scenarios are to be distinguished, since two types of very different network architectures and technologies are applied. On one hand, this covers the traditional Local Area Network (LAN) in terms of an Ethernet. On the other hand, the high-speed Asynchronous Transfer Mode (ATM) is utilized, particularly offering certain levels of guarantees for network-internal performance.

Quality-of-Service (QoS) parameters, so-called Da CaPo++ attributes, are used to specify application requirements /PPVW92/.

7. Performance Measurements of Da CaPo++

Performance measurements of Da CaPo++ are carried out within three different levels. The first level comprises the separated observation of single, almost atomic units of the Da CaPo++ core system. These units encompass single protocol modules, such as A-, C-, or T-modules, the application programming interface, and application units. Within the second level of measurements a specific communication relevant view is taken, that delivers statements on the performance of a certain communication service, including particular features such as security, multicasting, or reliability. Therefore, communication protocol aspects and end-to-end issues in general are covered. The distinction in phases (cf. Subsection 6.1) becomes very important on this level. Finally, the third level of measurements comprises applications and their networking demands.

7.1 Technical Environment

The environment concerns besides the two above mentioned networks a number of workstations. In addition, the thread and process model for Da CaPo++ is important, but based on the technology available for workstations.

7.1.1 Networks

For Da CaPo++ two networks have been applied. That is the IP-based Ethernet and the Asynchronous Transfer Mode (ATM). For ATM-access the Adaptation Layer Protocol Type 5 (AAL 5) is used. As described within /SBCC97b/, the structure of the network and its details are hidden within an according T-module that provides transparent access to the particular network. In addition, the Ethernet is accessible by two different T-modules, one implementing a TCP (Transmission Control Protocol) and the other one a UDP (User Datagram Protocol). Further details of the networks are not visible to any application or Da CaPo++ core unit.

7.1.2 Workstations – Hardware and Software

The workstation environment in use for the current implementation of Da CaPo++ comprises of Sun SPARCStations SS10/20 and Sun UltraSPARC 170E. As the current operating system version Solaris 2.5.1 has been adopted. They are connected by ATM networking cards from Fore Systems (SBA-200E) to the local ATM-switch and offer Ethernet connectivity through their built-in Ethernet card. In addition, workstations are equipped with a Multimedia Kit, including a Sun camera, the Sun Video Subsystem (SunVideo card), microphones, and optional speakers.

The SunVideo card /SunV94/ offers a real-time capture and compression for digital video, such as NTSC, PAL, and S-Video. Via the S-Bus-Video interface the workstation´s S-Bus is accessed. Additionally, it includes the video capture subsystem, the compression engine, and a frame storage. In addition, a software is provided that supports the capturing, digitization, and compression of video. The XIL Imaging Library provides required functions for image processing, compression, and decompression. If the SunVideo card and the XIL are utilized, XIL takes advantage of the extra hardware on the SunVideo card to accelerate the processing, capturing and compressing, only. Decompression of video is processed always on the workstation. The SunVideo card uses DRAM to store code, intermediate results of compressions, and captured images. The number of buffers available effects the frame achievable and the latency observable. In general, a higher number of buffers guarantees the highest frame rate performance, but at the cost of increased latency per frame. The compression scheme in use for the SunVideo card are CellB, JPEG, MPEG-1, and low-level UYVY. For the Da CaPo++ approach and its application framework only two schemes are utilized: CellB /HeMi81/ and JPEG (Joint Photographic Experts Group) /JPEG90/.

Audio support is provided by standard Sun Audio devices, only. There is no particular hardware required.

Observed behavior for a polling process on an otherwise idle system (Solaris 2.5.1 in default configuration on a Ultra 170E): The process receives time slices of 9.4 ms, if the system is otherwise undisturbed, and between time slices for a mean of 0.1 ms pauses are observed. As soon as two competing processes are running or substantial disk I/O (or other kernel activities such as network I/O) occurs, scheduling behavior changes substantially. Delays of up to 10 ms are the rule. For three processes, delays are up to 16 ms, with peaks up to 30ms. I/O times, such as paging of programs that are executed from NFS mounted devices, can range between 1-30 ms, depending on network load. Semaphores that are shared between processes can induce up to 100 ms for control flow switches. Due to the statistical properties of the scheduling algorithm, the actual variance between these values is larger than 100%.

7.1.3 Da CaPo++ Thread and Process Model

Two processes are generally in use whenever Da CaPo++ communication is taking place. On one side, there is the application, which is generally not involved in high-performance data movement, but just controls the data input, transmission, and output. On the other side, we have the Da CaPo++ server process, which comprises all the functionality for efficient management of high-throughput data.

7.1.3.1 Da CaPo++ Server

On every workstation capable of offering Da CaPo++ communication services, a Da CaPo++ server has to be run. This server, known as the Core, is running in a single process, containing multiple internal threads.

Da CaPo++ applications are implemented in C++. The upper API is included into applications by the class interface provided by the upper API. Applications instantiate the Da CaPo++ client class once and the session class for each session.

Services of the Da CaPo++ core are accessed via the lower API. Therefore, the Da CaPo++ Server is located in the lower API as a single thread, always listening on a unix domain socket for new applications (*API Listen* in Figure 2). For every connecting application, a new thread is started within the lower API to communicate with this application, or. This thread implements the main finite state machine of the Da CaPo++ Server /SBCC97a/.

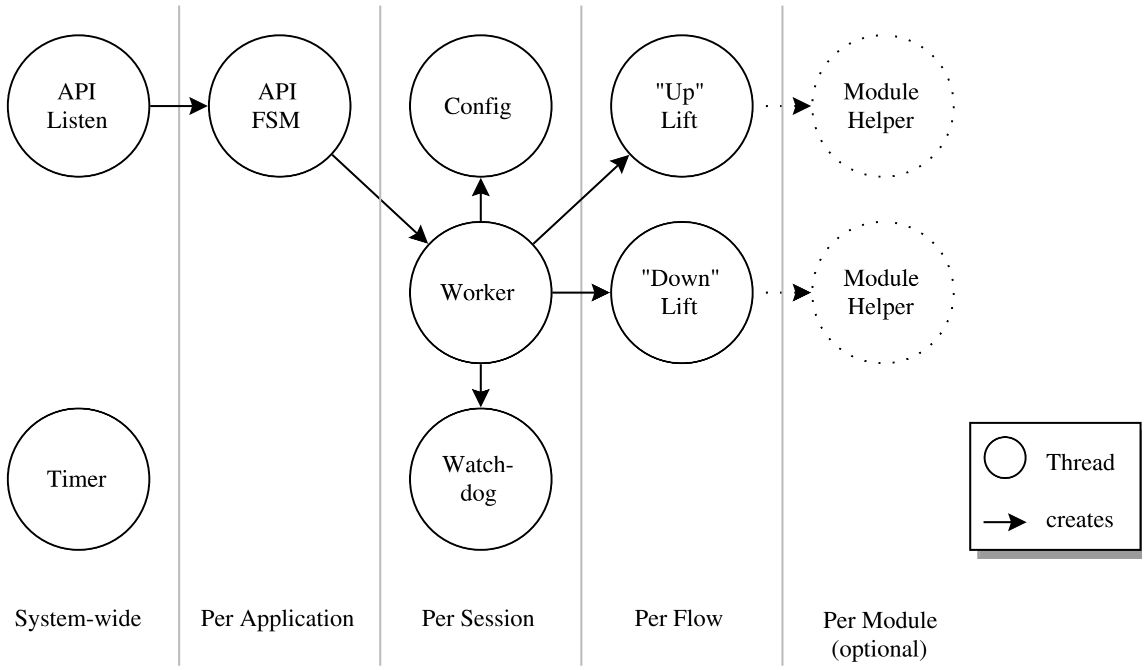

Figure 2. Thread Model of the Da CaPo++ Core

For each service requested by the application, it creates a *Worker* thread, which receives all commands directed at the session and executes them, often issuing further commands to some or all of the *Lifts*. A Flow is made up of a pair of Lifts: one for the main or forward path, transporting data from the sender to the receiver, and one for the back path, carrying protocol control information if necessary. A small number of modules create an additional *Helper* threads themselves, usually to overcome system limitations. *E.g.*, the Audio Mixer A-module handles mixing of the incoming audio data in a separate thread, and the ATM Multicast T-module joins new receivers independent of the module's data transfer activities.

Each Worker creates a *Watchdog* thread which regularly checks for unexpected behavior in the connection setup process and also keeps track of the different command queue lengths, sending a notification message to the console if commands do not complete for an unusually long time. This approach was selected to have a streamlined, maintainable Connection Manager, avoiding cluttering it with time-out and exception handling code. The Connection Manager is always running as the first Flow, hidden from the user's view. Thus, data transfer is only happening in the Flows.

There is also the *Config* thread, which checks the local and remote configuration as transmitted by the Connection Manager for compatibility on behalf of the Session and signals when the connection setup phase has been completed successfully. This is not done by the Connection Manager itself, since it might include chances in the Connection Manager's configuration also, since it is just another protocol.

For the asynchronous command execution and – if requested – completion notification used between the different threads, a sophisticated priority queueing and multiple execution system has been implemented. Another system-wide thread, the *Timer* manager, is still unmentioned. There actions can be registered for timely execution. The reason for the putting all the work into different threads is to make all the data transmissions independent of each other. Also, each module (especially A and T modules) may be coded independently of the other modules without the need to have a central distribution loop. The simplicity of the programming model and the efficient use of multiple processors generally makes up for the slight overhead of the lightweight context switching.

7.1.3.2 Applications

In general, applications reside in a separate process. Depending on the explicit application scenario, many processes may be required. In any case, communication services in terms of sessions and their included flows are requested via the upper API which is being linked to every application or process that requires sessions. For each session instantiated in the upper API (or, correspondingly, a single Da CaPo++ service in the core), a thread is created to deal with incoming notifications from the lower API. In addition, for each application, a thread is created to process asynchronous events from Da CaPo++. This thread handles all events destined to the particular application, *e.g.*, events for different sessions, by invoking the registered handler / SBCC97c/.

Consequently, the following formula for the number of threads is valid. Assume that A defines the number of applications communicating with Da CaPo++ on the same machine, that Si equals the number of sessions in application i, that Fij determines the number of flows of session j of application i, and that Mij is the number of special modules, in need of their own threads. Then the number of threads in the upper API of an application i is calculated to:

$$1 + Si \tag{EQ 1}$$

In addition, the number of threads in the lower API and the Core proper is computed to:

$$2 + A + 5 * \sum Si + 2 * \sum Fij + \sum Mij \tag{EQ 2}$$

7.1.4 Measurement Method and System

The measurement method applied is based on an implicit instrumentation of source code at compile time. This is automated by a tool and evaluation system called Quantify /Pure93/. This software monitor influences the to be measured program, however the falsification remains minimal. These influences are compensated by the measuring software, especially, if the pure CPU usage of a function is calculated. The only visible influence on the program is a reduced operation speed due to write access for the collected measurement data.

During run-time, the time spent in functions, routines, or procedures is measured. Collected timing data deliver the basis for evaluating the percentage of time spent for certain tasks or the absolute amount of CPU cycles, and recalculated into milli- or microseconds, utilized.

7.2 Measurement Scenarios

Measurements are carried out in three different levels. Each of this level is discussed in this subsection. In particular, the identification of modules, protocols, and applications is described. Furthermore, the number of relevant performance parameters to be measured are identified. They encompass the following ones, while specific ones are used additionally to determine particular behaviors:

- Processing times per module during saturated conditions (mean, max);
- Number of data units processed per time unit;
- Delays introduced; and
- Playout times.

Details on general platform-specific details (caches or network interface boards), on operating system calls (Solaris 2.5.), on memory and buffer management operations (operating system dependent and Da CaPo++ specific), Da CaPo++ core functionality, and the “real” functionality (such as error correction or video playout) have to be distinguished at all times. Basics that remain unchanged over the period of measurement have been enlisted above in Subsection 7.1. This approach allows for a comprising and comprehensive prove of Da CaPo++´s performance and behavior. The workstation Sun UltraSPARC 170E has been used in the following.

Firstly, the following list of Da CaPo++ core units is identified, being a basic building-block for separated measurements. They provide the important task of protocol processing, supported by the lift algorithm for data transport within the Da CaPo++ core and relevant protocol modules, implementing the transport access (T-module) to the networking infrastructure, the communication protocol processing (C-modules), and the application access to protocols (A-modules).

Secondly, exemplarily, Da CaPo++ protocols are evaluated.

Thirdly, a close look into the WWW scenario is taken as developed in the application framework, to provide an insight into end-to-end issues in Da CaPo++.

7.2.1 Da CaPo++ Core: Lift

To give an overview over the performance of the Lift as a transport mechanism and the current time overhead incurred by the Resource Manager to allocate the necessary packets is shown in Table 1 for a selection of module counts. For these measurements, the Da CaPo++ core was compiled using ‘make quick’, and all tests were done using 1000 Lift runs on a Sun UltraSPARC 170E workstation which was as idle as possible. All data was measured in real time and the modules in the graph were measurement modules, having a measurement overhead of 0.6µs each.

This results in a overhead of 9µs for the packet allocation and per-run Lift overhead, plus 0.4µs for each Lift step. The high maximum numbers stem from occasional context switches, which cannot be avoided on a non-real-time multitasking system.

Table 1. Lift Overhead [in µs]

Module count	Minimum	Average	Maximum	Standard deviation
3	11	11	48	1
5	14	14	53	2
10	19	20	88	6

The total run-time overhead depending on the amount of memory requested for a flow with 3 modules is shown in Table 2. As we can see, the whole memory management (request from the pool, returning it to the pool) takes 5µs, except for the first run, which takes additional 60µs.

Table 2. Lift and Resource Manager Overhead [in µs]

Memory requested	Minimum	Average	Maximum	Standard deviation	First run
0	11	11	45	1	15
1	16	16	54	2	76
10	16	16	47	2	75
100	16	16	49	2	77
1000	16	16	53	2	77
10000	16	16	52	2	80
100000	16	16	54	2	80

The overall maximum value occurs the first time a buffer is requested, all future requests are handled in a quick manner. The first run also includes some inherent locking overhead. The buffers are not reserved in advance, because there is currently no way to tell how many messages will be stored in the modules. The maximum value is due to context switches. As we can see, the required time is independent of the requested buffer size. Only the initial setup time increases with buffer size, but almost unremarkable.

This results in the following formula for Lift overhead calculation where M is the number of active modules, B is the number of buffers requested per run, and t is the resulting time in µs:

$$t = 9 + 0.6*M + 5*B \qquad \text{(EQ 3)}$$

7.2.2 Da CaPo++ Core: C-modules

This first section gives an short overview of the encryption-related performance of the Da CaPo++ C-modules. Afterwards, certain protocol elements are analyzed in closer detail according to their further properties. For these measurements, the Da CaPo++ core was compiled using ‘make DEBUG_LEVEL=6 OPTIMIZATION=6 MINUS_G=-g’, and all tests were done on an otherwise idle Sun UltraSPARC 170E workstation.

7.2.2.1 Overview figures

To get an overview of the system performance, 10000 packets holding 1000 Byte each were sent using the TCP T-module over Ethernet. Key changes for the symmetric algorithm occurred every 100 packets, while an RSA operation including encrypt/decrypt took place every 500 packets. Table 3 below delivers some typical figures in seconds. Depending on network and system load, these values can vary widely (+400%), but the given values represent a good range if the environment is favorable. The ‘User’ and ‘Sys’ columns represent corresponding CPU time consumed, while ‘Elapsed’ gives an overall time estimate for the data transfer.

Table 3. Security C-module Measurements in [sec]

Sending Side					**Receiving Side**				
Application			**Core**		**Core**		**Application**		
User	Sys	Elapsed	User	Sys	User	Sys	User	Sys	Elapsed
Data encrypted with DES and authenticated by MD5									
0.32	0.42	24.43	13.15	1.72	14.52	1.57	0.48	0.43	24.61
0.34	0.41	25.34	12.86	1.66	14.31	1.65	0.50	0.45	25.33
0.33	0.35	24.97	13.25	1.78	14.42	1.64	0.44	0.44	24.37
Data encrypted with IDEA and authenticated by MD5									
0.39	0.40	33.06	19.04	1.72	22.67	2.02	0.43	0.47	33.18
0.28	0.38	31.93	19.10	1.77	22.44	1.71	0.49	0.40	32.02
0.40	0.38	33.05	18.53	1.87	22.71	1.86	0.45	0.39	32.81
Data encrypted with RC5 (12 rounds, 128 bit key), authenticated by MD5									
0.31	0.37	26.10	9.33	1.49	13.05	2.08	0.43	0.33	26.36
0.48	0.31	26.16	9.60	1.51	13.08	1.89	0.50	0.39	25.95
0.25	0.43	26.51	9.25	1.84	12.88	1.92	0.46	0.63	26.70
No security									
0.34	0.37	10.79	1.46	1.27	1.28	1.83	0.46	0.35	10.80
0.30	0.28	10.27	1.55	1.29	1.19	1.93	0.59	0.47	9.94
0.43	0.40	10.64	1.80	1.26	1.31	2.19	0.45	0.37	10.68

7.2.2.2 Detailed Analysis

The detailed analysis was made by using quantify. Quantify instruments the object code and records the number of calls to procedures, and gives elapsed times or CPU consumptions for basic blocks. Da CaPo++ built for this purpose used ‘DEBUG_LEVEL=6 OPTIMIZATION=6 MINUS_G=-g CC=”quantify gcc”’ and runtime options were set to ‘QUANTIFYOPTIONS=

“-measure-timed-calls=user -windows=no -save-thread-data=stack,composite -max_threads =40 -thread_safe_locks=no“‘. This samples data for each thread separately and indicates User CPU consumption.

7.2.2.3 Sending Side

Figure 3 includes an overview of involved procedures as given in the form of a call-graph. Relevant procedures are prefixed by ‘md_’ for message authentication and ‘cbc_’ for encryption respectively.

During *sender initialization* the functions observed are called upon instantiation and initialization of the module graph for the sending side of the protocol. This happens before the application can use the protocol. An additional initialization overhead is the first exchange of keying information, before a data packet can be sent to the peer. For this, see sender processing. For involved and relevant functions see Figure 4. The time consumption for the initialization functions are called only once. The most expensive functions in terms of elapsed time are sm_GetPublic Key and sm_GetSecretKey, because they contain lookups to an external (file) database, and induce context switches. Table includes the measurements, given in milliseconds.

Table 4. Sender Initialization Measurements

Samples	Min	Avg	Median	Max	Variance
90	137	151	150	312	3769

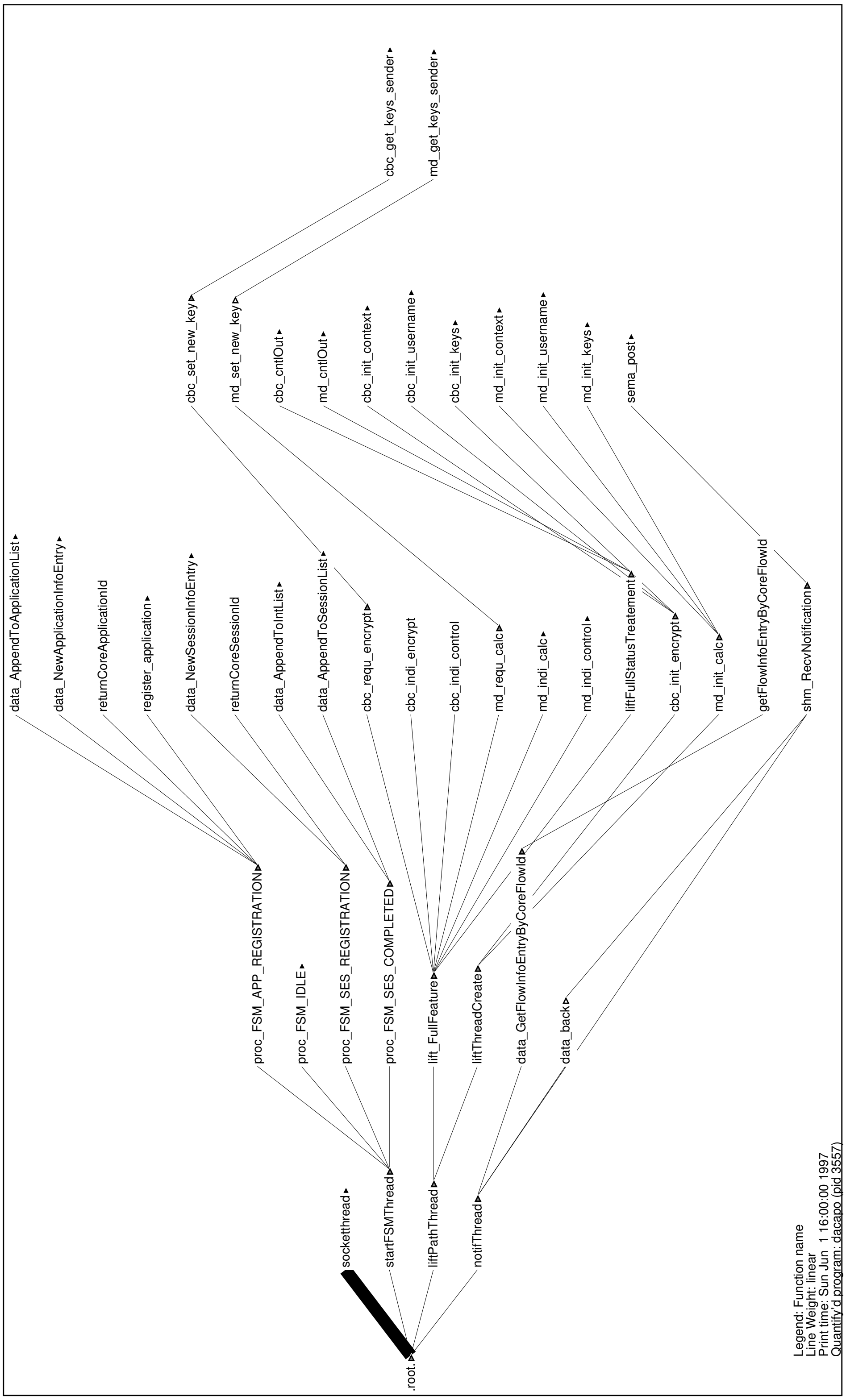

Figure 3. Sender Call Graph Overview

If verification of key signature authenticity were enabled, a constant 170 ms initialization delay would have to be added.

The registration with the security manager in sm_Register can be neglected. If module integrity were to be checked, another approximately 85 ms were to be added.

The overall crypto-initialization *User CPU Usage* is 33.06 ms on average. This figure is composed of searching the database for the public and secret key both for an encryption and an authentication module. Acquiring the secret key and decrypting it – by the user-provided passphrase – takes about 25 ms, acquiring the public key takes about 6 ms. The initialization overhead and CPU consumption is negligible, and thus no further details are given.

The *sender processing* is composed of 3 parts:

- Bulk-data authentication
- Bulk-data encryption
- Out-of-band communication of encrypted and signed keys.

The first and the second part vary depending on the employed algorithm. The third part is constant, the change induced by different session key sizes, *e.g.*, 8 vs. 16 Byte of session key data per key change, is negligible. The graph of dependencies is depicted in.

7.2.2.4 Authentication

As seen in the overview of Figure 3 and Figure 6, *time and CPU consumption* to transmit large blocks of data has a high variance. Reasons for this are the multitasking environment and the variable network bandwidth that is available during an experiment. For this reason, an ideal system will be assume, and only User CPU consumption will be further investigated.

Message Digest MD5 CPU Usage

md_requ_calc changes MAC session keys if needed and calculates MD5. The calculation of the MD5 checksums accounts for 97.6% of the CPU usage of this function. 2.1% of the time were used in md_get_keys_sender. The observed data set included 999 data packets, 1000 Byte each, 2 (RSA-based) key-

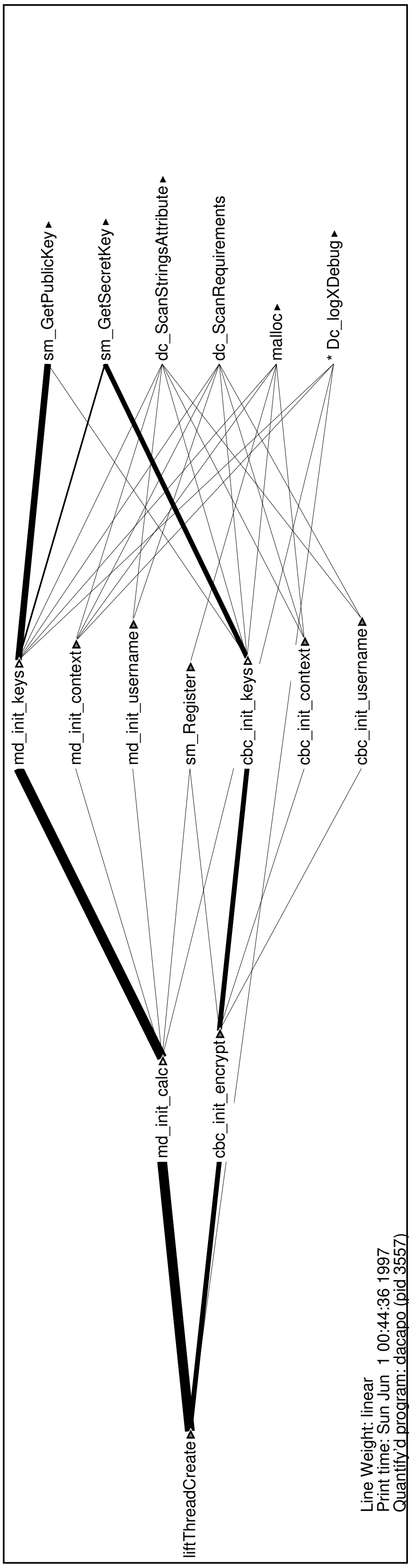

Figure 4. Sender Initialization

changes took place. The per-packet CPU usage without key change is (88.47-1.93)/ 999=0.086 ms. This corresponds to a theoretical throughput of 92 Mbit/s.

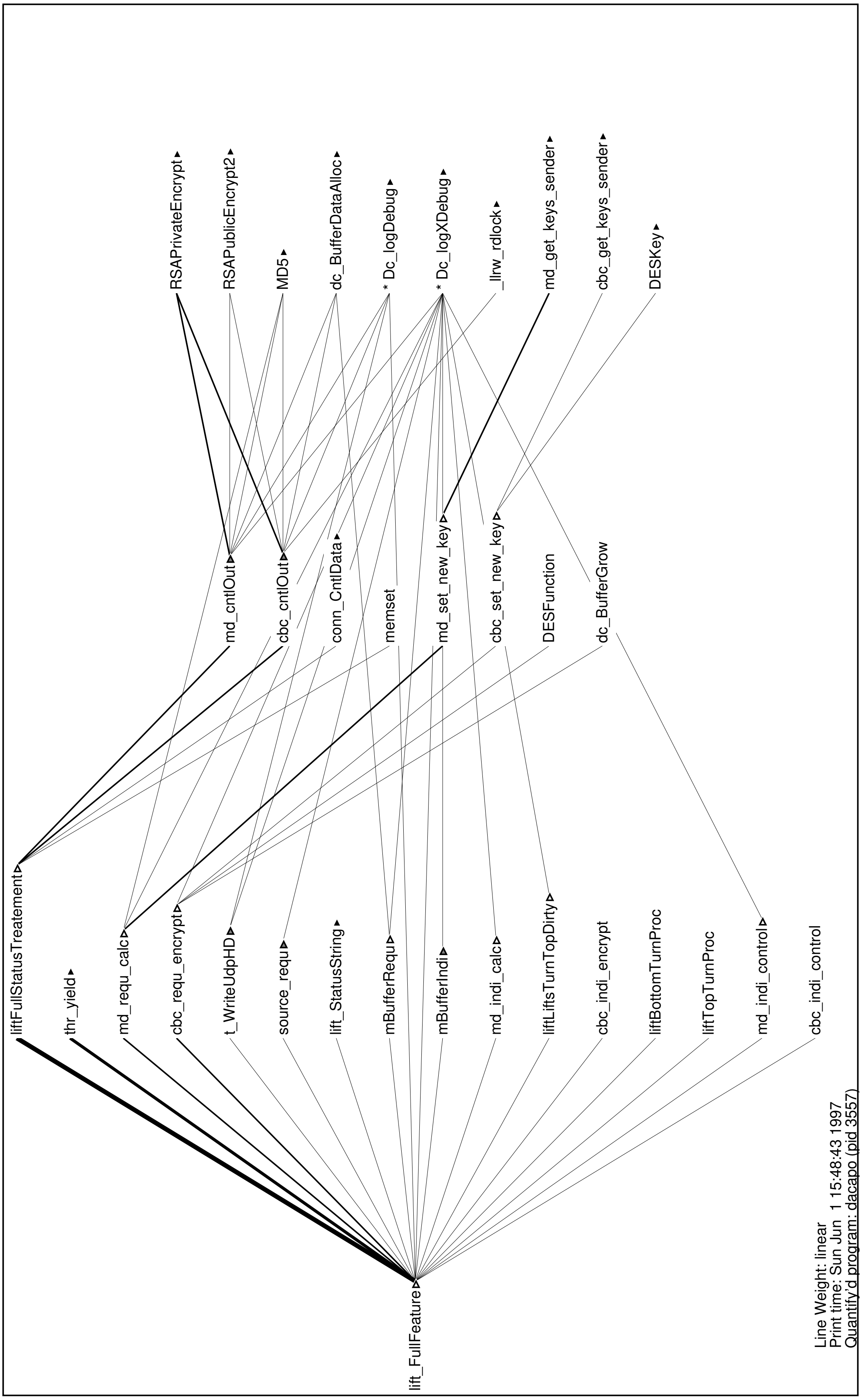

Figure 5. Sender Processing

md_get_keys_sender takes 0.965 ms per key generation, this time is consumed by hashing the pool with random data to extract new session keys in sm_MakeRandomKey. md_indi_calc has no significant contribution to the CPU consumption (0.04/997) ms.

md_cntlOut is used to signal OOB traffic to the lift and performs the encryption and certification of transmitted session keys. This takes 30.65 ms per key change, where the certification takes 93.48% of the time in RSAPrivateEncrypt. The encryption of the session key with the peers public key takes 6.47%. This shows impressively that operations using the public key are much cheaper than operations using the private key. The modular exponentiation with the value e=3 or 17, which is the typical RSA public exponent, uses much less time than the exponentiation with the large private value d. Modmult-speed depends on the number of bits set in the exponent.

After the session keys have been post-processed in cntlOut, they are transmitted to the peer via the reliable out of band data link provided by the connection manager. The actual transmission time from cntlOut to CntlIn of the peer varies wildly, due to unconstrained task and thread switching. md_exit_calc, md_stop are responsible for unregistering and removing of keying material and are not relevant.

7.2.2.5 Encryption

cbc_requ_encrypt behaves like md_requ_calc. DES CBC for 999 packets with 1000 Byte each takes 199.71 ms. This includes 10 DES key changes, costing 0.73 ms each, and two refills of the pool of session keys holding 5 keys at a time. This took 1.08 ms per refill. Additional time requirements as compared to md_get_keys sender are explained by the larger data block that has to be extracted from the random pool. 16 Byte have been used for md_get_keys_sender and 5*8=40 Byte for cbc_get_keys_sender. Per packet CPU usage without key changes is at 186.94 ms. This results in 0.187 ms per packet or a theoretical throughput of 42 MBit/s.

For other algorithms, see the security C-module summary in Paragraph 7.2.2.7

7.2.2.6 Receiving Side

Receiver initialization and *processing times* should algorithmically be identical to those of the sending side. However, the overview Figure 3 and Figure 6 tell a different story, indicating that the receiving side uses substantially more CPU power. The receiver initialization is independent from the sender initialization and happens in parallel.Time and CPU requirements are equal to the sending side.

CPU consumption of the crypto routines is the same on the sending and receiving side. The large difference in processing time is found in the receiving application module, where the library function mutex_unlock(), used for synchronization with the callback function of the application, consumes 98% of the additional CPU requirements, *e.g.*, 259 ms for a test run with 1000 packets.

:

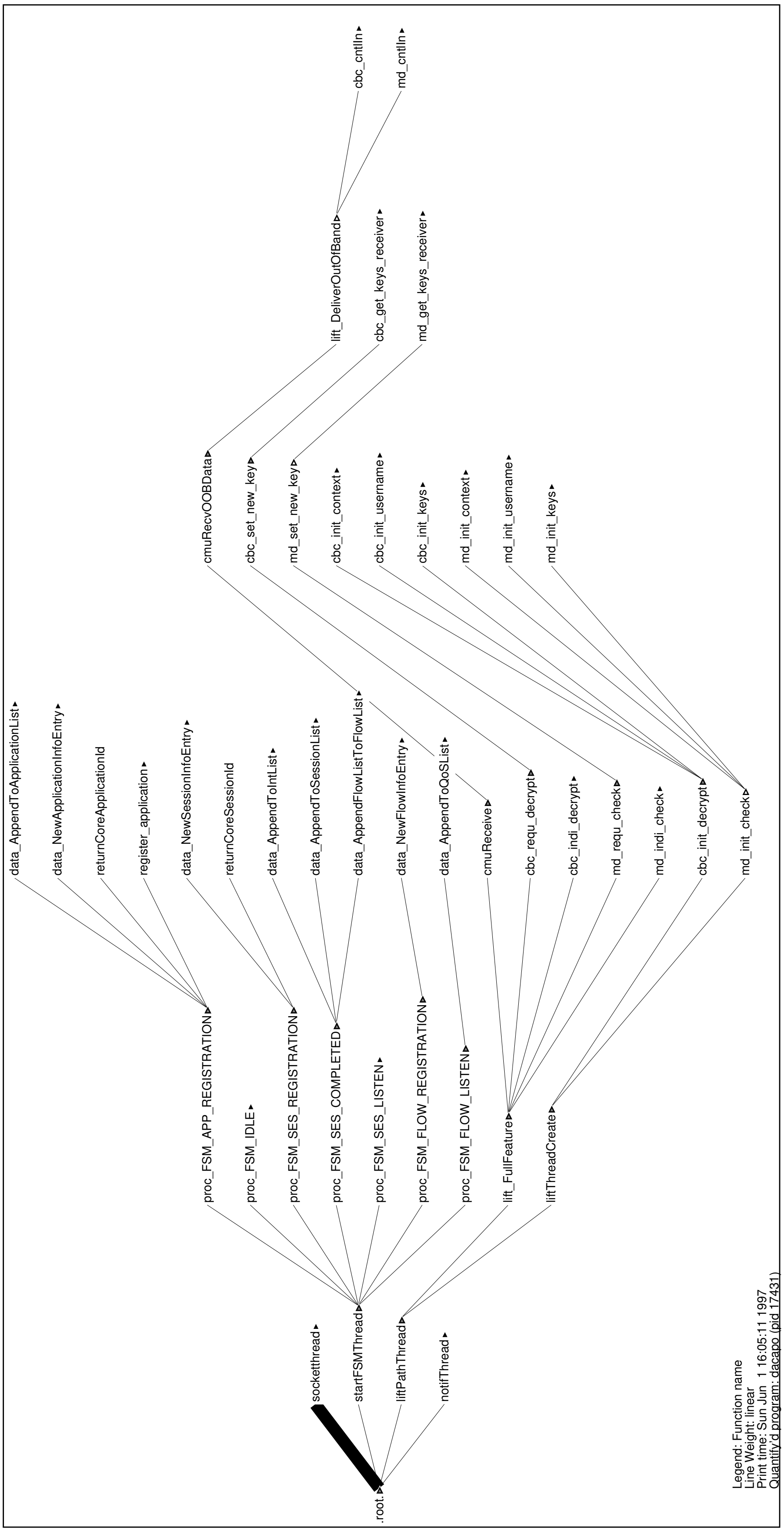

Figure 6. Receiver Call Graph Overview

7.2.2.7 Overview of Measurement for Every C-module

This section presents numbers for processing times (in ms) for all algorithms implemented in separate C-modules. Table 5 depicts MD5, Table 6 includes MD4, Table 7 shows DES,

Table 5. MD5 Measurements

Operation	Sender [ms]	Receiver [ms]
MD5 RNG acquire	0.965	N/A
MD5 alg. key change	N/A	N/A
MD5 RSA operations	30.15	56.73
MD5 per 1000 Byte packet	0.086	0.086
Overhead per packet	0.00020	0.00032

Table 6. MD4 Measurements

Operation	Sender [ms]	Receiver [ms]
MD4RNG acquire	0.965	N/A
MD4 algorithm key change	N/A	N/A
MD4 RSA operations	30.16	56.745
MD4 per 1000 Byte packet	0.059	0.059
Overhead per packet	0.00066	0.00034

Table 7. DES Measurements

Operation	Sender [ms]	Receiver [ms]
DES RNG acquire	1.13	N/A
DES algorithm key change	0.0073	0.073
DES RSA operations	29.175	28.86
DES per 1000 Byte packet	0.187	0.187
Overhead per packet	0.00990	0.00840

Table 8 presents IDEA, and Table 9 covers RC5 with 12 rounds and 128 bit keys. A graphical

Table 8. IDEA Measurements

Operation	Sender [ms]	Receiver [ms]
IDEA RNG acquire	1.30	N/A
IDEA algorithm key change	0.003	0.018
IDEA RSA operations	30.16	56.745
IDEA per 1000 Byte packet	0.304	0.292
Overhead per packet	0.01143	0.00987

Table 9. RC5-12-16 Measurements

Operation	Sender [ms]	Receiver [ms]
MD54RNG acquire	1.195	N/A
MD4 algorithm key change	0.01	0.01
MD4 RSA operations	30.15	56.73
MD4 per 1000 Byte packet	0.103	0.114
Overhead per packet	0.01364	0.01212

comparison of these values is shown in Figure 7. An excerpt of these numbers are included in the graphical representation of Figure 8.

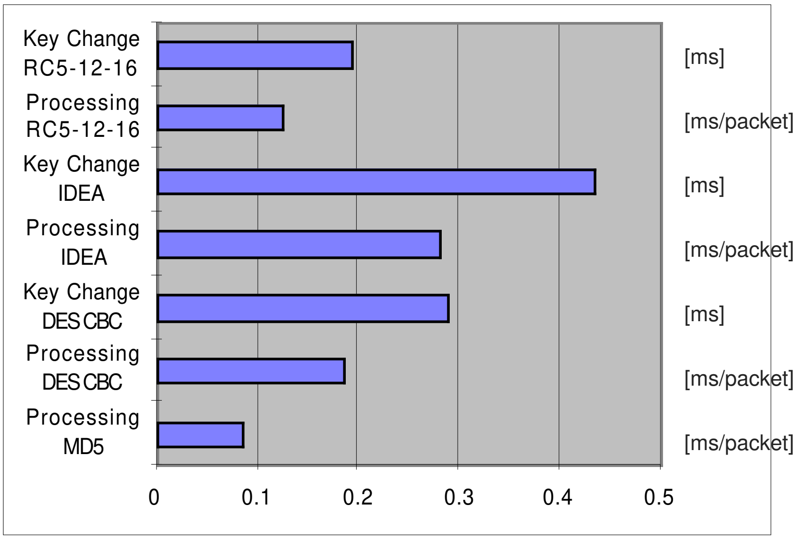

Figure 7. Comparison of Some Security C-modules

Random session key material is provided by parallel running threads, no direct influence to the communication behavior is observable. Assuming that key encryption, transmission, and decryption takes place in parallel to the actual bulk data processing, additional latency is not introduced by this. The cost of these algorithms is expressed as CPU-seconds per Megabit (CPU-s/Mbit). For simplification purposes, the calculated cost represents the maximum of the cost on the sending and the receiving side.

CPU-s/Mbit := #p* packet_cost + alg_cost + #k_RSA * RSA_cost * #key_bytes + #k * alg_keychange_cost + #k * RNG_cost.

Setting packet_cost overhead to 0.015 ms, RSA_cost to 3.6 ms per key-Byte and RNG_cost to a fixed value of 1.3 ms, the result depends on the number of packets per megabit, and the number of key changes and key encryption/exchanges per megabit. Assuming the number of bytes

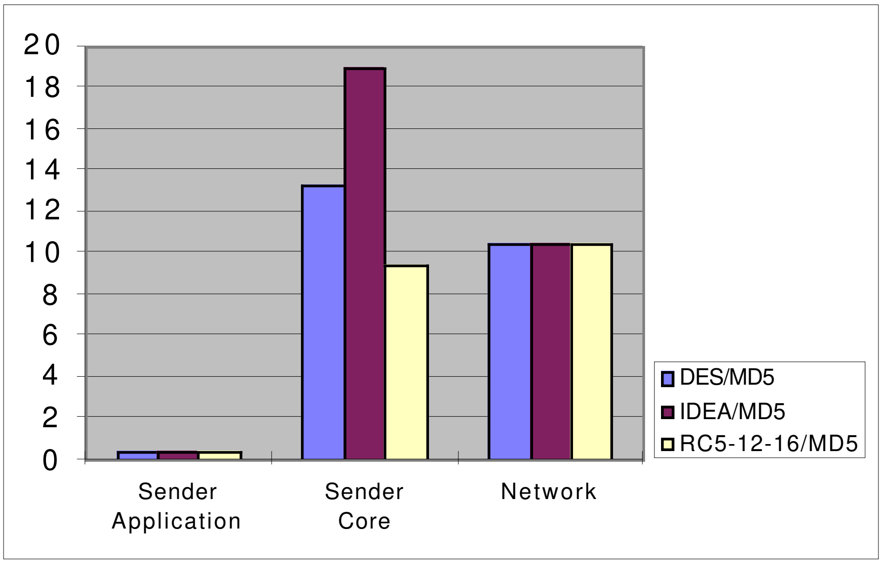

Figure 8. Some Security Module Combination Measurements

per packet to be 1000, and key changes to be performed every 100 kByte, RSA operations performed every 5 key changes, the following algorithm-dependent cost results:

CPU-s/Mbit := 125 * 0.000015 + alg_cost + 0.25 * 0.0036 * alg_keybytes + 1.25 * 0.0013 + 1.25 * alg_keychange_cost.

Table 10. Algorithm Costs

Algorithm	alg_cost	alg_key-Byte	alg_keychange _cost	CPU-s/Mbit	Mbit/s
MD5	0.010750	16	0	0.28650	35
MD4	0.007375	16	0	0.25275	40
DES	0.023375	8	0.00073	0.34988	29
3DES	0.070125	16	0.00219	0.09076	11
IDEA	0.038000	16	0.00018	0.05613	18
RC5-12-16	0.012875	16	0.0001	0.03090	32

To combine the costs for authentication and encryption, add the CPU-s/Mbit values.

7.2.3 Da CaPo++ Core: A-modules

7.2.3.1 AudioFile A-module

The AudioFile A-Module reads audio data from a file and sends it to the peer. The peer outputs the audio data to the speaker. The AudioFile A-module comprises the following functions, where the hierarchy of these functions is depicted in Figure 9:

- sender_start_audio,
- asInitAudio,
- asIndiAudio,
- asTimerAudio,
- start_receiver_audio, and
- arRequAudio.

Measurements with Quantify have been carried out. Firstly, the *sender side* has been evaluated. In Table 11 rows 3, 4, 5, 6, and 7 show quite constant values. The different ratio between *a_sender_OoBCommand* and *memcpy* in row 8 results from the high costs for the function *sender_start_audio* which is only called once to start the transmission (open the audio file and read the relevant information).

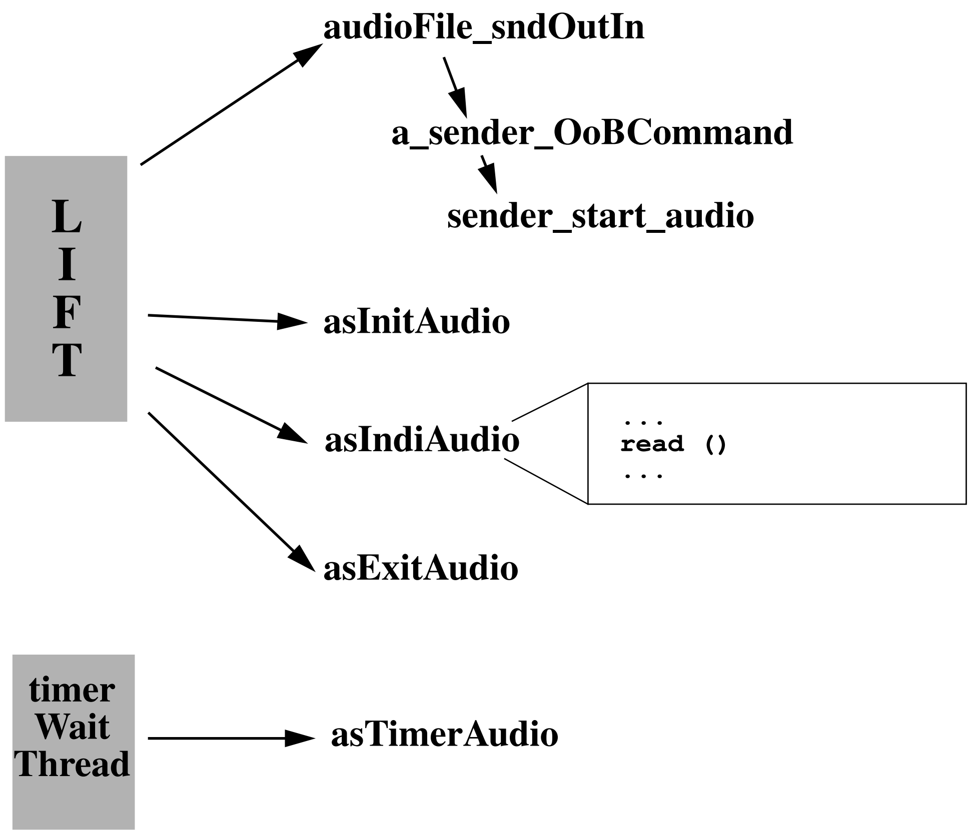

Figure 9. The Function Hierarchy in the AudioFile A-module (Sender)

The only function that is called regularly is *asIndiAudio*. For this reason the function was measured using the *gethrtime()* function (each call costs about 0.6 ms) to get the statistical details. As can be seen in Table 12 the function has a quite high variance resulting from the *read* call.

Table 11. Quantify Values for the Sender (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
The differences in below measurements result from the fact, that the audiofile has not been on the same machine as the sender. For all further measurements the file was moved to the senders machine to avoid NFS effects.					
1	sender_start_audio	a_sender_OoBCommand	called 1 time 0.57 us Fct and Desc: 155.80 us (ca. 20%: _llrw_rdlock rsrc_TimerCreate rsrc_TimerContinue read open)	called 1 time 0.57 us Fct and Desc: 30035.8 us (66.49%: read 33.19% open 0.1%: _llrw_rdlock rsrc_TimerCreate rsrc_TimerContinue)	called 1 time 0.57 us Fct and Desc: 80035.80 us (62.43% read, 37.45% open)
The audiofile is now on the same machine as the sender.					

Table 11. Quantify Values for the Sender (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
2	asIndiAudio	lift_FullFeature	called 602 times avg: 0.32 us min: 0.08 us max: 0.37 us Fct: 191.61 us Fct and Desc: 15413.31 us (96.32% read [493 times] 1.50% rm_PacketDataFree [109 times])	called 381 times avg: 0.33 us min: 0.08 us max: 0.37 Fct: 126.49 us Fct and Desc: 10268.96 us (96.77% read [1385 times], 1.05% rm_PacketDataFree [51 times])	called 2376 times avg: 0.37 us min: 0.08 us max: 0.51 us[a] Fct: 881.66 us Fct and Desc: 83072.71 us (98.06% read [1381 times], 0.01% rm_PacketDataFree [2 times])
3	sender_start_audio	a_sender_OoBCommand	called 1 time 0.57 us Fct and Desc: 155.80 us (ca 20% read, _llrw_rdlock, rsrc_TimerCreate, rsrc_TimerContinue, open)	same value as 1st measurement	same value as 1st measurement
4	asInitAudio	liftThreadCreate	called 1 time 0.23 us Fct and Desc: 8.26 us (90.36% _llrw_rdlock, 5.07% lapi_RegisterAModule)	same as 1st measurement	same value as 2nd measurement
5	asExitAudio	liftThreadDestroy	called 1 time 0.13 us Fct and Desc: 60.65 us (99% close (2 operations))	same as 1st measurement	was not measured (the application run until Quantify took all the available memory space)
6	asTimerAudio	timerWaitThread	called 601 times avg = min = max: 0.11 us Fct: 64.78 us Fct and Desc: 19567.65 us (98.77%: lift_DataAvailable)	called 385 times avg = min = max: 0.11 us Fct: 41.50 us Fct and Desc: 12329.25 us (98.75% lift_DataAvailable)	called 2377 times avg = min = max: 0.11 us Fct: 256.20 Fct and Desc: 77642.66 us (98.77% lift_DataAvailable)

Table 11. Quantify Values for the Sender (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
7	a_sender_OoB Command	audioFile_snd OutIn	called 14 times avg: 0.32 us min: 0.16 us max: 1.98 us Fct: 4.57 us Fct and Desc: 174.97 us (89.04% sender_start_audio)	called 8 times avg: 0.44 us min: 0.16 us max: 1.98 us Fct: 3.50 us Fct and Desc: 169.34 us (92.00% sender_start_audio)	called 3 times avg: 0.87 us min. 0.16 us max: 1.98 us Fct: 2.63 us Fct and Desc: 164.17 us (94.90% sender_start_audio)
8	audioFile_sndO utIn	lift_DeliverOu tOfBand	called 14 times avg = min = max: 0.17 us Fct: 2.43. us Fct and Desc: 4206.47 us (95.69% memcpy, 4.16% a_sender_OoBComm and)	called 8 times avg = min = max: 0.17 us Fct: 1.09 us Fct and Desc: 2473.05 us (93.00% memcpy, 6.85% a_sender_OoBComm and)	called 3 times avg = min = max: 0.17 us Fct: 0.52 us Fct and Desc: 1028.06 us (83.90% memcpy, 15.97% a_sender_OoBComm and)

a. this is in case the audio file is at its end and the sender has to skip at the beginning of the file

Detailed measurements of *asIndiAudio* are depicted in Table 12.

Table 12. Detailed Measurements of asIndiAudio[a] (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
asIndiAudio measured with gethrtime as first command and just before return of asIndiAudio						
1	435	10 us	193 us	94 us	1318	36.300835
2	641	9 us	12948 us	194 us	955209	977.347793
3	1864	12 us	35668 us	371 us	3510924	1873.745926
asIndiAudio measured with gethrtime as first command and just before return of asIndiAudio. In addition a pair of gethrtime() inside the function measured the call of *read*						
4	504	9 us	16303 us	130 us	522483	722.829545
5	603	13 us	1769 us	101 us	5839	76.412933
6	1978	11 us	24460 us	248 us	1850783	1360.434999
these are the values of the **read** call in *asIndiAudio*						
4	433	79 ms	16296 us	143 us	605482	778.127489
5	519	83 us	1761 us	109 us	5312	72.885432
6	1977	22 us	24453 us	242 us	1851264	1360.611475

a. In the 1st and 4th measurement, the audio file has always been stopped 4 times (= 8 additional control commands). Each time, the timer calls asIndiAudio and playing is stopped, the function returns with a minimal time.
In the 2nd and 5th measurement, the audio file has been stopped twice.
During the 3rd and 6th measurement, the audio file was not stopped but it was played more than once (the file was rewound with the *lseek()* command)

asIndiAudio. The high variance of the time of one function call for *asIndiAudio* results from the time differences for the performance of one call of *read()*. This call reads the audio data from the audio file.

sender_start_audio. The function call seems to take a constant time, which is divided by the opening of the audio file, the reading of the relevant audio information (*read*), the creation and the starting of the timer as well as the internal function *_llrw_rdlock*. But overall the costs for these functions are not high (about 50 us)

asInitAudio. The most time consumed in this function is consumed by the internal function *_llrw_rdlock*. The registration of the A-Module in the lower API happens fast.

asExitAudio. This function is quite constant and just calls the 2 times a *close*, for the audio file as well as for the sample file.

asTimerAudio. This functions consumes a quite constant time. If divided by the number of calls, the time for Fct and Desc results in almost the same value (32.xxx).

a_sender_OoBCommand. The control command *OPEN* results in the call of the function *sender_start_audio* that is quite expensive. All other calls of control commands result in statements like *strcmp, strdup, lift_DataAvailable* and assignments, which all have constant costs.

audioFile_sndOutIn. The most expensive operation in this function is the memory copy of the out of band data packet itself.

Overall result. The most time consumed by the sending AudioFile A-Module is consumed by system calls, especially by *read* and *memcpy*. Moreover the call of *read* results in a high variance, but nevertheless in the measured scenarios one call remained under 25 ms, whereas the average was a lot smaller than 1 ms.
The overhead induced by the A-Module compared with a direct use of the audio device is small.

Conclusions for other measurements. As the measurement of the out of band functions showed the *memcpy* operation as the expensive part (consuming less than 1 ms) and as the out of band functions in the other A-Modules are implemented in the same way, they have not been measured there. The same conclusion was made for the timer function that is implemented in the same way in the SunVideoFile A-Module.

Secondly, the *Receiver side* has been evaluated. Figure 10 shows the function hierarchy of the measured functions in the receiver part of the AudioFile A-Module. Table 13 lists obtained Quantify measurements.

As the function *arRequAudio* is the only one that is called regularly, the function was measured in detail with *gethrtime()* which is shown in Table 14, presenting real time values.

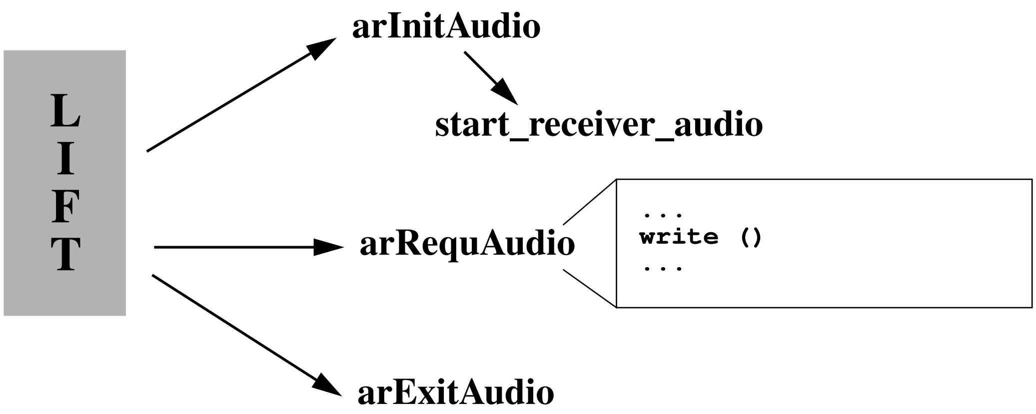

Figure 10. The Function Hierarchy in the AudioFile A-Module (Receiver)

Table 13. Quantify Values for the Receiver (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
1	start_receiver _audio	arInitAudio	called 1 time 1.94 us Fct and Desc: 68.50 us (51,05% _llrw_rdlock, 43.96% open)	same as 1st measurement	called 1 time 1.94 us Fct and Desc: 20008.46 us (99.83% _llrw_rdlock, 0.15% open)
2	arRequAudio	lift_FullFeature	called 599 times 0.29 us Fct: 172.17 us Fct and Desc: 28150.32 us (99% write)	called 471 times 0.29 us Fct: 135.38 us Fct and Desc: 14318.96 us (99% write)	called 1923 times 0.29 us Fct: 552.72 us Fct and Desc: 58461.50 us (99% write)
3	arInitAudio	liftThreadCreate	called 1 time 0.22 us Fct and Desc: 68.96 us (99% start_receiver_audio)	same as 1st measurement	called 1 time 0.22 us Fct and Desc: 20008.92 us (99.99% start_receiver_audio)
4	arExitAudio	liftThreadDestroy	called 1 time 0.08 us Fct and Desc: 30.19 us (99% close)	same as 1st measurement	same as 1st measurement

Table 14. Detailed Measurements on arRequAudio (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
arRequAudio measured with gethrtime as first command and just before return of arRequAudio						
1	671	88 us	3081 us	104 us	16634	128.972552
2	696	87 us	309 us	95 us	205	14.315758
3	2841	83 us	3250 us	97 us	6863	82.844052
arRequAudio measured with gethrtime as first command and just before return of arRequAudio. In addition a pair of gethrtime() inside the function measured the call of *write*						
4	1863	90 us	1629 us	101 us	1909	43.695518
5	614	88 us	590 us	104 us	968	31.114045
6	2676	89 us	2349 us	100 us	2229	47.207274
these are the values of the **write** call in *arRequAudio*						
4	1863	86 us	1624 us	95 us	1612	40.152539
5	614	85 us	438 us	99 us	620	24.898437
6	2676	86 us	2344 us	95 us	2097	45.798338

start_receiver_audio. This function is not expensive if the synchronization of threads does not block the functions (*_llrw_rdlock* in the 3rd measurement). In the 3rd measurement, the thread was blocked and the function time was much higher than in measurement 1 and 2, but nevertheless less than 21 ms (user time).

asRequAudio. The function *arRequAudio* has average costs of about 0.1 ms. The costs are up to 2.5 ms. The high variance results from the costs and variance in the *write* system call.

arInitAudio. The *arInitAudio* function just registers itself to the lower API and calls the *start_receiver_audio* function. Its execution time is mainly dependent from the latter function.

arExitAudio. The function's time is constant and just contains the system call *close* of the audio device.

Overall summary. The overhead induced by the AudioFile A-Module is very small.

7.2.3.2 SunVideoFile A-module

The SunVideoFile reads video data from a file and sends it to the receiving A-Module. The sending A-Module does not display the video data. The receiving A-Module decompresses the video frames and display them on the screen. For instance, the SunVideoFile A-Module does only handle video data that is compressed in the JPEG format. During the measurements the environment handled *Pseudocolr* as it was not yet truecolor able.

For all measurements a movie with the following characteristics has been used: 999 frames were captured and compressed in JPEG. The characteristics of frames are shown in Table 15. The movie has been sent and received with 10 frames per second.

Table 15. Video Data Characteristics

number of frames	minimum	maximum	average	variance	standard deviation
999	3076 Bytes	14282 Bytes	7353.027832 Bytes	9211907	3035.112354

Due to problems of the *XIL Library* together with *Quantify* (whenever a xil routine is called, quantify stops with a floating exception), the functions of the SunVideoFile A-Module have been measured directly with gethrtime(). The first 3 performance measurements just have been carried out with a gethrtime() statement at the beginning as well as at the end of the examined functions. Thus, for the SunVideoFile A-Module no *user data* could be measured.

As well as the AudioFile A-Module, the SunVideoFile A-Module uses out-of-band communication between sender and receiver. There are already 5 out-of-band commands being exchanged between sender and receiver before the receiver sees the first video frame. The necessary commands in the sender (**INIT, FPS, OPEN, START-TRANSMISSION** and **PLAY**) are measured in detail in the 2nd and the 3rd measurement block. In the 3rd scenario, the time to perform the out-of-band commands **FF, FWD, STOP, RWD, FR** is also measured. In the receiver only the out-of-band packet **X-WINDOW-INFO** is received, thus, this command is not separately measured. The receiver just is measured within two measurement blocks.

Firstly, the *sender side* of the VideoFile has been investigated. Figure 11 shows the function hierarchy of the measured functions in the sending SunVideoFile A-Module.

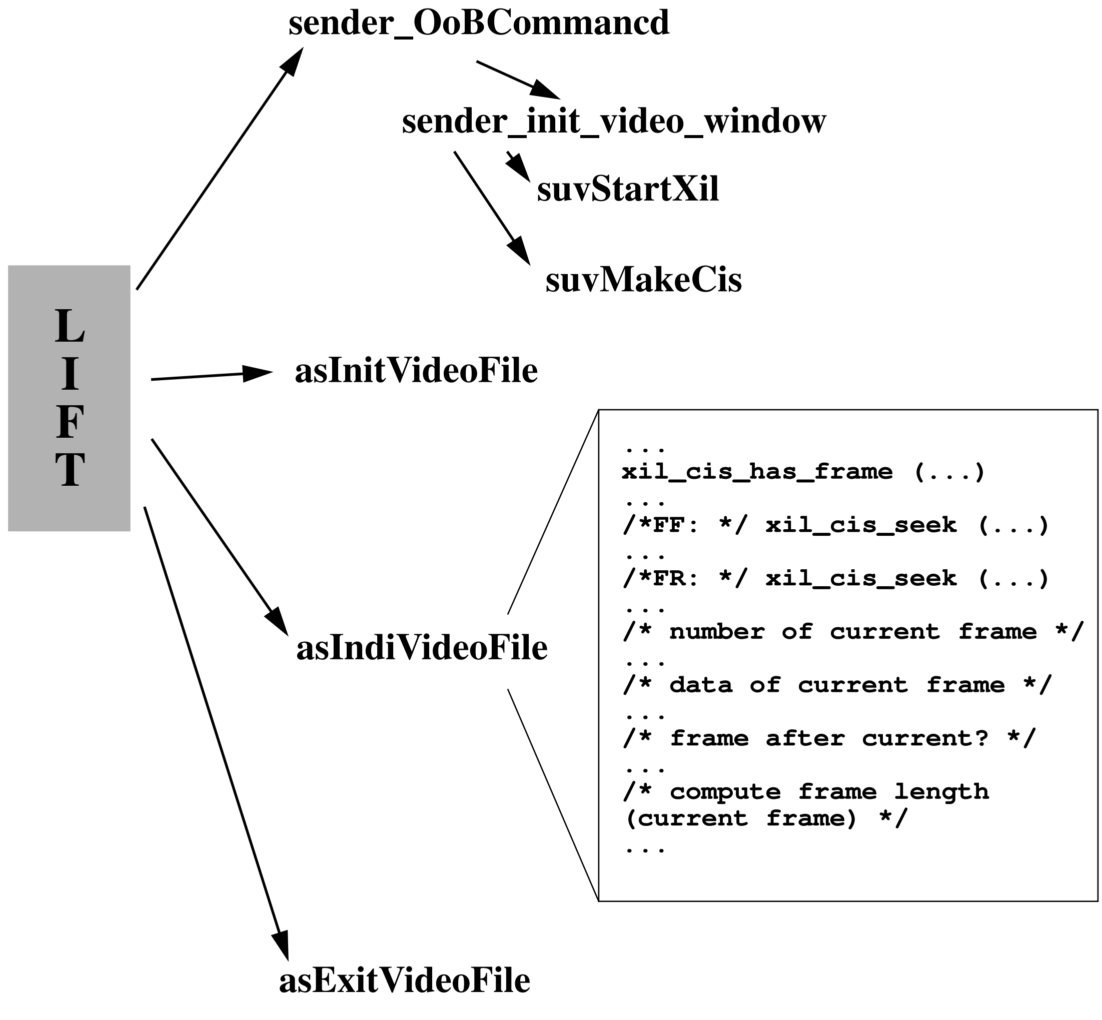

Figure 11. The Function Hierarchy in the SunVideoFile A-module (Sender)

The first block of measurements have been carried out without additional out-of-band communication, which is included in Table 16.

.

Table 16. Detailed Measurements of the Sender of SunVideoFile (Real Time and Total Costs for Functions)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
sender_OoBCommand						
1	5	4 us	118690 us	23811 us	2813123395	53038.885688
2	5	5 us	116746 us	23423 us	2721611786	52169.069243
3	5	3 us	117567 us	23588 us	2760067268	52536.342356
sender_init_video_window						
1	1	118562 us				
2	1	116633 us				
3	1	117441 us				
asInitVideoFile						
1	1	268 us				
2	1	265 us				
3	1	273 us				
asIndiVideoFile						
1	1018	7 us	1107850 us	1531 us	1204700016	34708.788744
2	1016	7 us	816567 us	1250 us	655597113	25604.630691
3	1011	7 us	1158151 us	1613 us	1325679260	36409.878600
asExitVideoFile						
1	1	6136 us				
2	1	6202 us				
3	1	6185 us				
suvStartXil						
1	1	77486 us				
2	1	76316 us				
3	1	76718 us				
suvMakeCis						
1	1	37415 us				
2	1	36849 us				
3	1	36985 us				

The second block of measurements have been carried out with out-of-band communication at the beginning and the asIndiVideoFile function, which is included in Table 18. In addition, Table 19 covers the out-of-band communication in real-time, while Table 19 shows out-of-band commands separately.

Table 17. Detailed Measurement of asIndiVideoFile (Real Time)

mea-sure-ment	num-ber of calls	minimum	maximum	average	variance	standard devia-tion
costs for the total function (for the measurement 7 pairs of gethrtime() have been added)						
4	1007	9 us	3090890 us	3525 us	9484423297	97388.003868
5	1010	9 us	1007413 us	1449 us	1003965931	31685.421426
6	1008	8 us	1199067 us	1648 us	1425347935	37753.780401

Table 17. Detailed Measurement of asIndiVideoFile (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
the call of *xil_cis_has_frame* to get the image						
4	999	13 us	184 us	79 us	1373	37.052020
5	999	12 us	664 us	77 us	1727	41.557833
6	999	14 us	170 us	80 us	1460	38.209466
in case of *fast_play == 1* the call of *xil_cis_seek()*						
4	there was no fast forward done in measurement 4 to 6					
5						
6						
in case of *fast_play == 2* the call of *xil_cis_seek()*						
4	there was no fast rewind done in measurement 4 to 6					
5						
6						
current_frame = xil_cis_get_read_frame() (number of current frame)						
4	999	1 us	197 us	2 us	38	6.171796
5	999	1 us	147 us	2 us	21	4.609365
6	999	1 us	138 us	2 us	19	4.318394
get data of current frame						
4	999	8 us	3089924 us	3235 us	9556315279	97756.407869
5	999	8 us	1006343 us	1143 us	1013473897	31835.104793
6	999	7 us	1198031 us	1337 us	1436391364	37899.754139
control if frame exists after current frame (*xil_cis_has_frame)*						
4	999	3 us	225 us	56 us	964	31.055040
5	999	3 us	216 us	58 us	1045	32.331489
6	999	2 us	7767 us	64 us	60491	245.948720
if yes, compute the length of current frame as difference						
4	998	5 us	389 us	122 us	5663	75.255101
5	998	4 us	373 us	124 us	6191	78.681358
6	998	5 us	392 us	121 us	5732	75.708083

Table 18. Detailed Measurements of the Out-of-band Communication (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
4	5	3 us	1166151 us	233305 us	271937917385	521476.670029
5	5	4 us	399799 us	80033 us	31953253969	178754.731318
6	5	5 us	400023 us	80085 us	31987777225	178851.271242

Table 19. OOB Commands Separately (Real Time)

measurement	command	time
4	INIT	us
5	INIT	29 us
6	INIT	31 us
4	FPS	17 us

Table 19. OOB Commands Separately (Real Time)

measurement	command	time
5	FPS	16 us
6	FPS	16 us
4	OPEN	1166151 us
5	OPEN	399799 us
6	OPEN	400023 us
4	START_TRANSMISSION	318 us
5	START_TRANSMISSION	318 us
6	START_TRANSMISSION	349 us
4	PLAY	3 us
5	PLAY	4 us
6	PLAY	5 us

The third block of measurements included several commands: PLAY, STOP, FF, FWD, FR, RWD. Measurement 7 to 9 measure the time of the commands **FF** and **RWD** as well as the time for *xil_cis_seek* in case of *fast_play == 1*. Measurement 10 to 12 measure the time of the commands **FR** and **FWD** as well as the time for *xil_cis_seek* in case of *fast_play == 2*. All measurements obtain some data on **PLAY** and **STOP as** well. Results are listed in Table 20 for Sender out-of-band commands, in Table 21 for control commands, and in Table 22 for the control commands in the function: **asIndiVideoFile**.

Table 20. Data of Sender_OoBCommand

measurement	number of calls	minimum	maximum	average	variance	standard deviation
7	19	3 us	1342451 us	70685 us	94846958140	307972.333400
8	20	3 us	529539 us	26506 us	14019005640	118401.881910
9	19	4 us	1356285 us	71412 us	96812033355	311146.321455
10	24	4 us	302648 us	12634 us	3815876406	61772.780460
11	24	4 us	1520826 us	63392 us	96368059165	310432.052412
12	24	4 us	608735 us	25389 us	15438618319	124252.236677

Table 21. Data of the Control Commands

measurement	command	time
measurement 7		
7	INIT	53 us
7	FPS	19 us
7	OPEN	1342451 us
7	START-TRANSMISSION	365 us
7	PLAY	4 us
7	FF	11 us
7	RWD	17 us
7	PLAY	3 us
7	PLAY	4 us
7	FF	10 us
7	RWD	18 us
7	PLAY	4 us

Table 21. Data of the Control Commands

measurement	command	time
7	PLAY	4 us
7	FF	10 us
7	RWD	16 us
7	PLAY	4 us
7	FF	10 us
7	PLAY	4 us
7	STOP	6 us
measurement 8		
8	INIT	41 us
8	FPS	19 us
8	OPEN	529539 us
8	START-TRANSMISSION	356 us
8	PLAY	3 us
8	FF	14 us
8	RWD	20 us
8	PLAY	4 us
8	PLAY	3 us
8	FF	14 us
8	RWD	22 us
8	PLAY	5 us
8	PLAY	4 us
8	FF	12 us
8	RWD	22 us
8	FF	11 us
8	PLAY	4 us
8	PLAY	4 us
8	PLAY	4 us
8	FF	12 us
measurement 9		
9	INIT	33 us
9	FPS	17 us
9	OPEN	1356285 us
9	START-TRANSMISSION	335 us
9	PLAY	4 us
9	FF	15 us
9	RWD	22 us
9	PLAY	4 us
9	FF	11 us
9	PLAY	4 us
9	RWD	23 us
9	PLAY	5 us
9	PLAY	6 us
9	FF	12 us
9	RWD	25 us

Table 21. Data of the Control Commands

measurement	command	time
9	PLAY	4 us
9	FF	12 us
9	PLAY	4 us
9	STOP	7 us
measurement 10		
10	INIT	31 us
10	FPS	20 us
10	OPEN	302648 us
10	START-TRANSMISSION	333 us
10	PLAY	4 us
10	STOP	8 us
10	FR	14 us
10	FWD	19 us
10	PLAY	5 us
10	PLAY	5 us
10	STOP	7 us
10	FR	12 us
10	FWD	17 us
10	PLAY	5 us
10	PLAY	5 us
10	STOP	7 us
10	FWD	18 us
10	FR	13 us
10	PLAY	4 us
10	PLAY	4 us
10	STOP	8 us
10	FWD	16 us
10	FR	13 us
10	PLAY	5 us
measurement 11		
11	INIT	31 us
11	FPS	20 us
11	OPEN	1520826 us
11	START-TRANSMISSION	349 us
11	PLAY	5 us
11	STOP	7 us
11	FR	12 us
11	FWD	19 us
11	PLAY	5 us
11	PLAY	4 us
11	STOP	7 us
11	FR	13 us
11	FWD	16 us
11	PLAY	5 us

Table 21. Data of the Control Commands

measurement	command	time
11	PLAY	4 us
11	STOP	7 us
11	FWD	17 us
11	FR	14 us
11	PLAY	4 us
11	PLAY	4 us
11	STOP	7 us
11	FWD	18 us
11	FR	12 us
11	PLAY	4 us
measurement 12		
12	INIT	31 us
12	FPS	19 us
12	OPEN	608735 us
12	START-TRANSMISSION	375 us
12	PLAY	4 us
12	STOP	7 us
12	FR	12 us
12	FWD	17 us
12	PLAY	4 us
12	PLAY	5 us
12	STOP	6 us
12	FR	13 us
12	FWD	16 us
12	PLAY	4 us
12	PLAY	4 us
12	STOP	6 us
12	FWD	16 us
12	FR	13 us
12	PLAY	5 us
12	PLAY	5 us
12	STOP	6 us
12	FWD	16 us
12	FR	13 us
12	PLAY	4 us

Table 22. Measurements for the Control Commands in asIndiVideoFile (Real Time)

mea-sure-ment	number of calls	minimum	maximum	average	variance	standard devia-tion
total costs for one function call						
7	732	7 us	3160331 us	4633 us	13641735306	116797.839475
8	721	7 us	3094354 us	4646 us	13277241113	115226.911411
9	786	7 us	3571647 us	4886 us	16226817581	127384.526460
10	779	8 us	2477596 us	3556 us	7877651720	88756.136237

Table 22. Measurements for the Control Commands in asIndiVideoFile (Real Time)

mea-sure-ment	number of calls	minimum	maximum	average	variance	standard devia-tion
11	760	6 us	3358243 us	4753 us	14836328658	121804.468956
12	909	7 us	1948116 us	2467 us	4173781496	64604.810161
in case of *fast_play == 1* the call of *xil_cis_seek()* ***(fast forward)***						
7	395	3 us	166 us	4 us	67	8.208522
8	393	2 us	169 us	3 us	70	8.393140
9	390	2 us	192 us	4 us	91	9.559329
in case of *fast_play == 2* the call of *xil_cis_seek() B* ***(fast rewind)***						
10	400	3 us	169 us	4 us	69	8.288891
11	400	3 us	177 us	3 us	76	8.712470
12	400	3 us	196 us	4 us	93	9.665767
get data of current frame						
7	451	9 us	3159335 us	7197 us	22129017793	148758.252856
8	483	7 us	3093390 us	6608 us	19809126175	140744.897508
9	499	9 us	3570607 us	7365 us	25546573111	159832.953771
10	521	19 us	2476645 us	4975 us	11770978054	108494.138338
11	478	18 us	3357301 us	7227 us	23577620663	153550.059143
12	542	20 us	1947186 us	3803 us	6993943436	83629.799927

The fourth block contains results for asIndi for the second CIS frame to the last one sent. This included in Table 23.

Table 23. Detailed Results for asIndiVideoFile After the 2nd Frame in the CIS

measure-ment	num-ber of calls	minimum	maximum	average	variance	standard devia-tion
total costs for one function call or *asIndiVideoFile.* Due to the high amount of information in the first frame of the cis, the first frame is excluded from this evaluation.						
1	999	64 us	786 us	451 us	31066	176.256118
2	998	56 us	814 us	454 us	30512	174.675751
3	998	59 us	911 us	473 us	29617	172.097014
4	998	65 us	945 us	459 us	30955	175.939498
5	998	68 us	1151 us	457 us	33455	182.906569
6	998	67 us	8046 us	463 us	89490	299.149001
7	689	7 us	923 us	335 us	77629	278.619156
8	685	7 us	917 us	372 us	78427	280.047616
9	776	7 us	910 us	346 us	84011	289.846189
10	769	8 us	949 us	380 us	87718	296.172371
11	739	6 us	843 us	343 us	81875	286.137955
12	893	7 us	1019 us	330 us	84492	290.674824

A summary of out-of-band commands can be found in Table 24. All measurements performed for the SunVideoFile A-Module use pairs of *gethrtime()* to determine the duration of functions. Thus, **no user time, but only real time** has been measured. This has to be taken into account while evaluating the results. Especially some high differences in minimum and maximum value for a given function for the same data result from the fact that the operating system may perform some of its tasks during the measured real time.

Table 24. The Out-of-band Commands (Real Time)

Command	num-ber of calls	minimum	maximum	average	variance	standard devia-tion
INIT	9	29 us	53 us	35 us	58	7.623064
FPS	9	16 us	20 us	18 us	3	1.615893
OPEN	9	302648 us	1520826 us	847384 us	239298548176	489181.508416
START-TRANS-MISSION	9	318 us	375 us	344 us	394	19.841735
PLAY	49	3 us	6 us	4 us	0	0.630166
FF	13	10 us	15 us	12 us	3	1.625123
RWD	9	16 us	25 us	21 us	9	3.004626
STOP	14	6 us	8 us	7 us	0	0.662994
FR	12	12 us	14 us	13 us	1	0.717741
FWD	12	16 us	19 us	17 us	1	1.164500

sender_OoBCommand. The costs as well as the variance is due to the performance of the **OPEN** command (see also Table 24). This commands calls the function *sender_init_video_window*.

sender_init_video_window. This function calls the functions *suvStartXil* and *suvMakeCis*. As all other calls do have quite constant costs (mostly assignments), the high variance that can be observed in the variance of the *OPEN* out-of-band command in Table 24 must result from high variances in the *sender_init_video_window* call, even if there have not been measured high differences in the first three measurements. This function is just performed once and calls itself the *suvStartXil* and the *suvMakeCis* functions. These functions consists out of Xil functions that do not directly perform on the data of the frames and, thus, are (after all measurements) assumed to have quite constant costs as well as 1 or 2 pairs of *MTLOCK, MTUNLOCK*. In addition in the function *sender_init_video_window* there are 4 more pairs of *MTLOCK, MTUNLOCK*. These pairs seem to be responsible for the high variance in the call of *sender_init_video_window* observed indirectly in Table 24.

asInitVideoFile. This function has constant costs below 0.3 ms.

asIndiVideoFile. The function *asIndiVideoFile* reads the current video frame, and puts its data into a packet given to the Da CaPo lift to be sent to the peer. To do so, this function calls different functions from the Xil library. The data structure of one *cis* is proprietary (*void* *). According to our measurements and experiences during the implementation phase, the first frame of the cis has relevant information of the cis assigned to. Thus the reading and transmission of this frame take a lot of time. The call of *asIndiVideoFile* takes for the first frame up to 3 seconds (see Table 16, Table 17 and Table 22), whereas the function time takes as average time 500 us and as maximum time 8 ms (see Table 23) after the second frame.

asExitVideoFile. The function costs constantly about 6 ms.

suvStartXil. The function costs constantly about 8 ms.

suvMakeCis. The functions costs constantly about 4 ms.

out-of-band commands. Besides the **OPEN** command (see above) the commands' cost are quite constant and small. The differences of costs for the **PLAY, STOP, FF** and **FR** call that do nothing but an assignments results from the different numbers of calls of the *strcmp* routine.

Overall Summary. The high cost and high variance that may partly be observed in *sender_init_video_window* as well as in *asIndiVideoFile* are due to system calls, calls of functions of the *xil* Library as well as to thread synchronization and the fact of real time measurements that may include time used by the operating system.

Figure 12 shows the function hierarchy of the measured functions in the SunVideoFile A-Module for the *receiver side*.

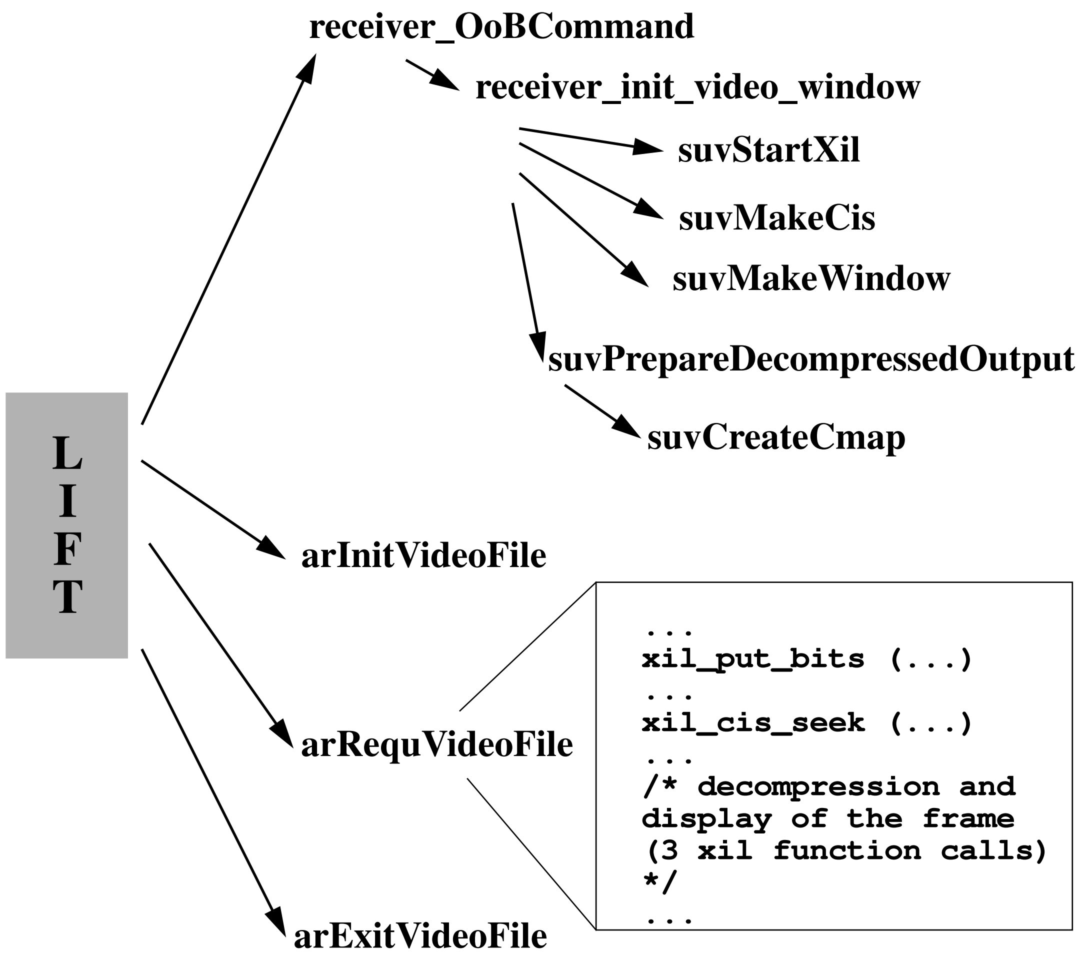

Figure 12. The Function Hierarchy in the SunVideoFile A-module (Receiver)

The first block of measurements of the receiving A-Module is depicted in Table 25, while the second block contains in Table 26 details on the ***arRequVideoFile*** function.

All measurements performed for the SunVideoFile A-Module use pairs of *gethrtime()* to determine the duration of functions. Thus, **no user time, but only real time** has been measured. This has to be taken into account while evaluating the results. Especially some high differences in minimum and maximum value for a given function for the same data result from the fact that the operating system may perform some of its tasks during the measured real time.

Table 25. Detailed Measurements of the Receiver of SunVideoFile (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
receiver_OoBCommand						
1	1	1412101 us				
2	1	728815 us				
3	1	1077110 us				
receiver_init_video_window						
1	1	1411980 us				
2	1	726053 us				
3	1	1076990 us				
arInitVideoFile						
1	1	1966 us				
2	1	270 us				
3	1	292 us				
arRequVideoFile						
1	1000	34839 us	103571 us	41982 us	24971934	4997.192578
2	1000	34875 us	108759 us	42128 us	25663003	5065.866435
3	1000	36158 us	109253 us	43699 us	28066823	5297.812992
arExitVideoFile						
1	1	44117 us				
2	1	41905 us				
3	1	40159 us				
suvCreateCmap						
1	1	7686 us				
2	1	7451 us				
3	1	7495 us				
suvStartXil						
1	1	77896 us				
2	1	80380 us				
3	1	77302 us				
suvMakeCis						
1	1	36403 us				
2	1	45888 us				
3	1	35984 us				
suvMakeWindow						
1	1	1286743 us				
2	1	588360 us				
3	1	953052 us				
suvPrepareDecompressedOutput						
1	1	8153 us				
2	1	7876 us				
3	1	7892 us				

Table 26. Detailed Measurement of arRequVideoFile (Real Time)

mea-sure-ment	number of calls	minimum	maximum	average	variance	standard devia-tion
arRequVideoFile with 4 pairs of gethrtime in it						
4	1000	48562 us	107120 us	57977 us	30080090	5484.531882
5	1000	36615 us	110234 us	44609 us	28625430	5350.273836
6	1000	33637 us	59399 us	41107 us	20706112	4550.396904
xil_cis_put_bits (the received Bits are written into the cis)						
4	1000	73 us	594 us	133 us	2040	45.163994
5	1000	74 us	287 us	129 us	1448	38.058434
6	1000	72 us	308 us	124 us	1276	35.715085
xil_cis_seek (the cis is set one frame back, so that the frame can be displayed)						
4	1000	5 us	1287 us	6 us	1668	40.846311
5	1000	4 us	144 us	5 us	20	4.417688
6	1000	4 us	147 us	4 us	20	4.519708
the **decompression** of the frame and its **display**						
4	1000	48420 us	106990 us	57823 us	29689703	5448.825815
5	1000	36517 us	110088 us	44459 us	28272505	5317.189566
6	1000	33534 us	59267 us	40962 us	20396011	4516.194327

receiver_OoBCommand. The costs result mainly from the function *receiver_init_video_window.*

receiver_init_video_window. This function calls different functions calling Xil and X functions. As it is just called once, the details of the high costs (up to 1.5s) have not further been investigated. The function leads to an initial delay, that hopefully will be accepted by the user. The highest costs in this function was provoqued by *makeWindow.* The initial delay in the performed measurement also has to do with the used window manager (fvwm) and the fact, that the user has to first click on the video window until it really is placed and the function is continued. Thus, the reaction time was part of the initial delay.

arInitVideoFile. the initialization of the SunVideoFile A-Module in the receiver has costs of about 0.3 ms. The costs of 2 ms in the 1st measurement are probably also due to the fact that real time was measured.

asRequFile. The time consumed to receive, decompress and display a single video frame varies from 35 ms to 110 ms, whereas the highest cost do not always occur at the same video frame (this results from the fact of measuring real time data and not user time data) Almost all time is used to decompress and display the frame (*xil* functions).

arExitVideoFile. The time for closing the window and destroying the cis is constant 40 - 44 ms.

suvCreateCmap. The creation of the color map requires about 8 ms.

suvStartXil. To start the *xil* takes about 8 ms.

suvMakeCis. Also the creation of the *cis* has quite constant costs of about 40 ms (36 - 45 ms).

suvMakeWindow. The creation of the *X Window* varies and takes from 560 ms to 1.3 s. This is due to real time measurements as well as due to the individual reaction time when the window came up (a mouse click was necessary to place is).

suvPrepareDecompressedOutput. The preparation of the decompressed output takes constantly about 8 ms.

Overall Summary. Most processing time consumed in the SunVideoFile A-Module is consumed by the *xil* functions as well as some time which is consumed by functions of the *X Window* system. As even the average time for the decompression and the display of one single video frame is between 40 to 50 ms, a frame rate of more than 20 frames per second cannot be achieved on a normal ULTRASparc as receiver, and this frame rate **cannot** be guaranteed. This is due to the performance of the *xil* library.

7.2.3.3 RawAPI A-module

The RawApi allows the application of the sending side to send data to the application on the receiving side. To measure the A Module a test application has been written that sends 500 times a test packet to the other side.

Figure 13 shows the function hierarchy of the measured functions in the RawApi A-Module on the *sender side*.

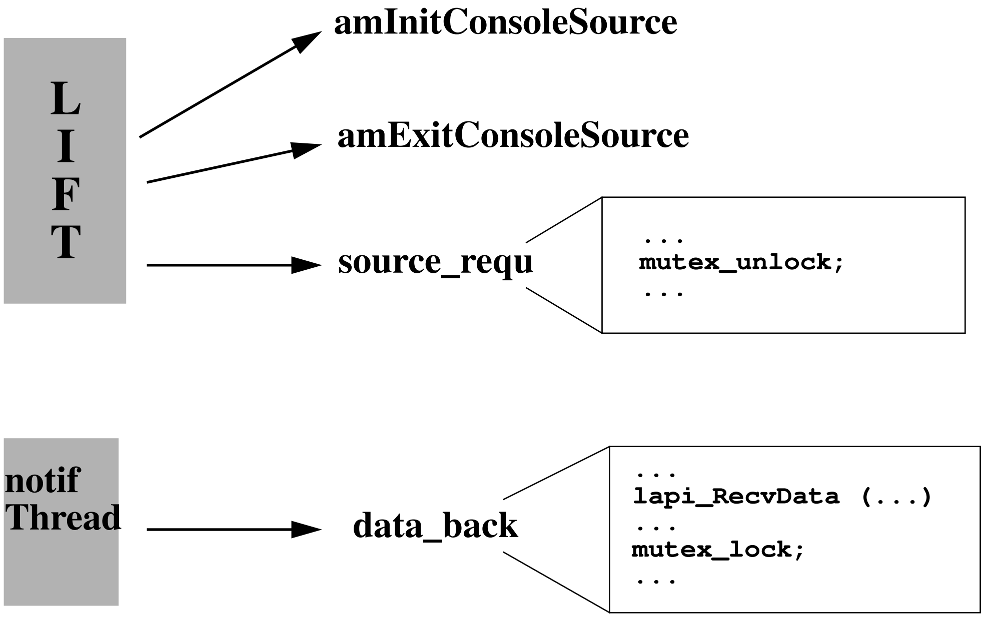

Figure 13. The Function Hierarchy in the RawApi A-module (Sender)

Quantify measurements are included in Table 27. As the functions *data_back* and *source_requ* are called for each packet, these functions are measured separately with the *gethrtime()* function, which is included in Table 28. The measurements of the function *source_requ* is included in Table 29.

Measurements 1to 6 have been performed with the sender sending one packet after the other. In measurement 7 to 9 the sender sleeps for 0.1 s (usleep(100000)) after sending one packet and before sending the next packet.

Table 27. Quantify Values for the Sender (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
1	data_back	notifThread	called 500 times 0.01 us Fct: 5.99 Fct and Desc: 15367.42 us (99.96% shm_RecvNotification)	called 500 times 0.01 us Fct: 5.99 us Fct and Desc: 35247.42 us (99.98% shm_RecvNotification)	same as 1st measurement
2	source_requ	lift_FullFeature	called 500 times 0.23 us Fct: 113.77 us Fct and Desc: 1223.38 us (47.14% mutex_unlock, 19% memcpy, 18% dc_Free)	same as 1st measurement	called 500 times 0.23 us Fct: 113.77 us Fct and Desc: 26415.48 us (97.55% mutex_unlock, 0.88% memcpy, 0.84% dc_Free)
3	amInitConsoleSource	liftThreadCreate	called 1 time 0.16 us Fct and Desc: 0.32 us (32% lapi_RegisterAModule, 17% mutex_init)	same as 1st measurement	same as 1st measurement
4	amExitConsoleSource	liftThreadDestroy	called 1 time 0.04 us no Desc	same as 1st measurement	same as 1st measurement

Table 28. Detailed Measurements of data_back (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
data_back measured with gethrtime as first command and just before return of data_back						
1	500	32 us	113125 us	812 us	34374912	5863.012237
2	500	28 us	89601 us	756 us	29586718	5439.367462
3	500	161 us	451716 us	1647 us	422088203	20544.785289
data_back measured with gethrtime as first command and just before return of data_back. In addition a pair of gethrtime() inside the function measured the call of *lapi_RecvData* and of *mutex_lock*						
4	500	34 us	247828 us	1308 us	146604855	12108.049195
5	500	165 us	384669 us	1519 us	318687727	17851.826990
6	500	34 us	246075 us	1026 us	128650258	11342.409702
7	500	59 us	387695 us	844 us	300510452	17335.237287
8	500	64 us	240201 us	553 us	115324191	10738.910148
9	500	62 us	322868 us	720 us	208393293	14435.833654
these are the values of the **lapi_RecvData** call in *data_back*						

Table 28. Detailed Measurements of data_back (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
4	500	16 us	309 us	19 us	248	15.736339
5	500	16 us	130 us	18 us	79	8.891046
6	500	16 us	128 us	19 us	95	9.761604
7	500	16 us	347 us	20 us	244	15.605807
8	500	16 us	226 us	20 us	118	10.844655
9	500	16 us	219 us	20 us	111	10.530842
these are the values of the **mutex_lock** call in *data_back*						
4	500	2 us	247792 us	1126 us	146165006	12089.872030
5	500	89 us	385634 us	1466 us	318566590	17848.433829
6	500	2 us	246039 us	876 us	128700339	11344.617180
7	500	2 us	387635 us	779 us	300516168	17335.402164
8	500	2 us	240152 us	484 us	115342890	10739.780715
9	500	1 us	322818 us	649 us	208419229	14436.731934

Table 29. Detailed Measurements on source_requ (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
source_requ measured with gethrtime as first command and just before return of source_requ						
1	500	5 us	8924 us	139 us	464682	681.675964
2	500	5 us	10079 us	231 us	1208285	1099.220070
3	500	5 us	9785 us	267 us	898209	947.738671
source_requ measured with gethrtime as first command and just before return of source_requ. In addition a pair of gethrtime() inside the function measured the call of *mutex_unlock*						
4	500	7 us	8728 us	86 us	176524	420.148134
5	500	6 us	9359 us	283 us	1011254	1005.611138
6	500	6 us	11165 us	189 us	579824	761.461603
7	500	9 us	193 us	11 us	150	12.265429
8	500	10 us	194 us	11 us	166	12.865329
9	500	8 us	198 us	10 us	184	13.562104
these are the values of the **mutex_unlock** call in *source_requ*						
4	500	1 us	8721 us	79 us	176425	420.029349
5	500	1 us	9353 us	277 us	1011131	1005.550291
6	500	1 us	11158 us	182 us	579835	761.468664
7	500	1 us	98 us	2 us	20	4.511418
8	500	1 us	98 us	2 us	19	4.318161
9	500	1 us	97 us	2 us	18	4.300198

data_back. This function has two calls with variant duration. The call of the *lapi_RecvData* function is not very expensive (less than 0.4 ms, with an average of 0.02ms) and has little variance whereas the *mutex_lock* system call is quite expensive (until 40 ms) and is of very variant length.

source_requ. The main costs as well as the variance in this function result from the *mutex_unlock* system call.

amInitConsoleSource. This function is of constant, negligible costs.

amExitConsoleSource. This function also is of constant, negligible costs.

Overall summary. The costs and the variances in the sender side of the RawApi A-Module result mainly from the system call *mutex_lock* and *mutex_unlock* and, thus, from the synchronization of the different threads in the Da CaPo++ core.

Figure 14 shows the hierarchy of the measured functions of the *receiver side* of the RawApi A-

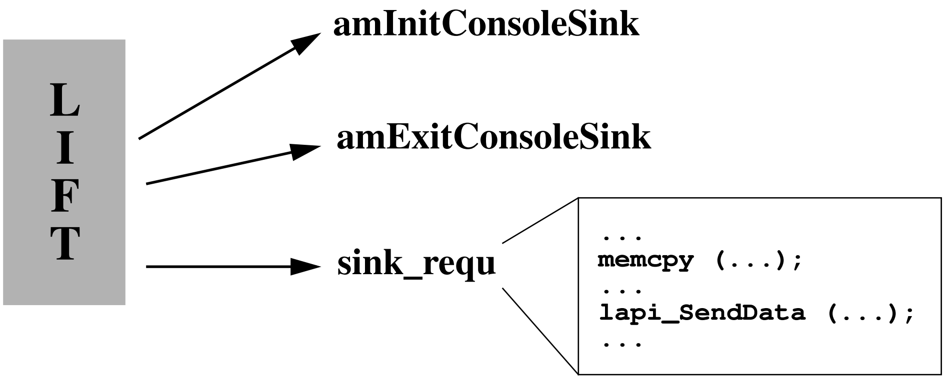

Figure 14. The Function Hierarchy in the RawApi A-Module (Receiver)

Module. Initial measurements with Quantify are included in Table 30. As the function *sink_requ* is called regularly, it is measured in detail included in Table 31.

In measurement 1 to 6 the sender just sends one packet after the other, whereas in measurement 7 to 9 the sender makes a pause of 0.1 s after sending one packet.

.

Table 30. Quantify Values for the Receiver (User Time)

	Function	called by	1st measurement	2nd measurement	3rd measurement
1	sink_requ	lift_FullFeature	called 500 times 0.02 us Fct: 8.98 us Fct and Desc: 1377.25 us (99.35% mBufferIndi)	same as 1st measurement	same as 1st measurement
2	amInitConsoleSink	liftThreadCreate	called 1 time 0.07 us Fct and Desc: 0.17 us (60.71% lapi_RegisterAModule)	same as 1st measurement	same as 1st measurement
3	amExitConsoleSink	liftThreadDestroy	called 1 time 0.04 us (no Desc)	same as 1st measurement	same as 1st measurement

Table 31. Detailed Measurements on sink_requ (Real Time)

measurement	number of calls	minimum	maximum	average	variance	standard deviation
sink_requ measured with gethrtime as first command and as last command						
1	500	25 us	56617 us	1516 us	14516644	3810.071410
2	500	25 us	111515 us	2042 us	35119066	5926.134162
3	500	25 us	71446 us	1798 us	19811517	4451.013022
sink_requ measured with gethrtime as first and last command. In addition the function calls *memcpy* and *lapi_SendData* are measured with gethrtime						
4	500	26 us	92594 us	1536 us	24712657	4971.182620
5	500	26 us	211223 us	1809 us	97858615	9892.351341
6	500	27 us	93002 us	913 us	22781456	4772.992369
7	500	62 us	656 us	69 us	1118	33.436939
8	500	39 us	638 us	77 us	3724	61.027800
9	500	62 us	630 us	70 us	1044	32.309567
the values for **memcpy**						
4	500	2 us	113 us	2 us	25	4.981053
5	500	2 us	151 us	2 us	44	6.645931
6	500	2 us	131 us	2 us	33	5.761309
7	500	3 us	157 us	3 us	47	6.886017
8	500	2 us	150 us	3 us	45	6.675719
9	500	2 us	160 us	3 us	49	7.025775
the values for **lapi_SendData**						
4	500	22 us	92517 us	1528 us	24688393	4968.741541
5	500	23 us	211217 us	1801 us	97859693	9892.405808
6	500	23 us	92996 us	905 us	22781849	4773.033541
7	500	53 us	263 us	58 us	420	20.486140
8	500	30 us	389 us	66 us	3087	55.560050
9	500	53 us	255 us	59 us	394	19.838114

sink_requ. The costs and their variance of this function call result from the call *lapi_SendData* that must write the data to a buffer for the IPC with the upper API.

amInitConsoleSink. This function has constant and negligible costs.

amExitConsoleSink. This function also has negligible, constant costs.

7.2.4 End-to-End Issues: Protocols, Connection Manager, and Security Manager

Communication protocols consist of a dedicated number of modules to provide a specific communication service. These services offer on an end-to-end basis the functionality, applications ask for. Therefore, the pure service performance is required to determine the quality of a communication system. In terms of Da CaPo++, the actual communication subsystem load is dependent on the communication protocol in use based on system-relevant restrictions and effects. Due to Da CaPo++ potential to support many different application flows of data in parallel, end-to-end issues rely on the particular application currently supported. However, a close-up view of some selected communication protocols, running isolated, is important to show the particular best-case behavior.

Therefore, some of the following protocols are exemplarily investigated separately:

- Audio protocol (live, file)
- Video protocol (live, file)
- Data transmission protocol (reliable, unreliable)
- Secured protocols

These protocols have been varied according to a subset of the following dimensions:

- unicast vs. multicast
- secure vs. insecure
- different devices and coding schemes.

The characterization of results is required to be depicted by QoS parameters, such as throughput (based on data unit sizes), frame rates, processing delays (vs. transmission delays), playout delays, error rates. Graphical representations encompass two-dimensional families of curves with varying parameters (such as number of receivers, size of data units, or network type).

7.2.4.1 Protocol Times for Audio and Video

Protocol times for Audio and Video Protocols have been measured with the scenario described in Figure 15. The Video Protocol consists in the SunVideoFile A-Module, the dummy C-Module and the ATM or UDP multicasting T-Module. The Audio Protocol consists in the AudioFile A-Module and the ATM or UDP multicast T-Module.

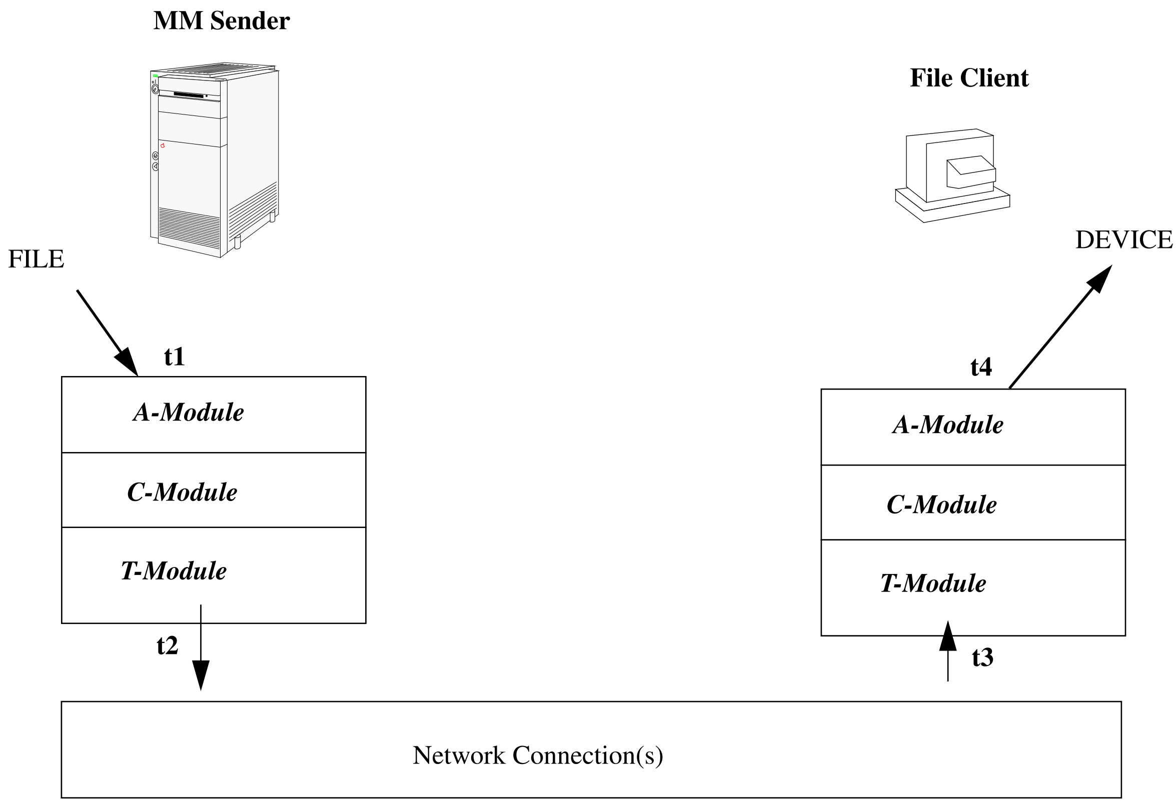

Figure 15. The Measurement Scenario for the Audio and the Video Protocol

The time points have the following meanings:

- **t1:** is the time when the data for one frame is read from the file
- **t2:** t2 indicates when the data for a file was put to the network for transmission
- **t3**: is the time the receiving T-Module receives the packet.
- **t4:** is the time the receiving A-Module displays the data on the screen/the audio device
- **t4 - t3:** indicates the time the protocol needs for one packet on the receiving side
- **t2 - t1:** indicates the time the protocol needs for one packet on the sending side

Table 32 and Table 33 comprise the obtained results.

Table 32. The SunVideoFile Protocol

mea-sure-ment	num-ber of calls	minimum	maximum	average	variance	standard devia-tion
Sender ATM						
7	587	46 us	317 us	66 us	360	18.967494
8	474	47 us	292 us	64 us	297	17.240946
9	496	47 us	296 us	65 us	286	16.906822
Receiver ATM						
7	587	36 us	220 us	44 us	137	11.701595
8	474	36 us	198 us	45 us	117	10.835221
9	496	43 us	197 us	50 us	100	10.023139
Sender UDP						
10	515	49 us	300 us	66 us	345	18.578827
11	494	46 us	297 us	65 us	292	17.078540
12	529	50 us	331 us	69 us	302	17.374756
Receiver UDP						
10	515	40 us	193 us	49 us	91	9.523914
11	494	39 us	193 us	49 us	75	8.676872
12	529	39 us	188 us	49 us	71	8.417578

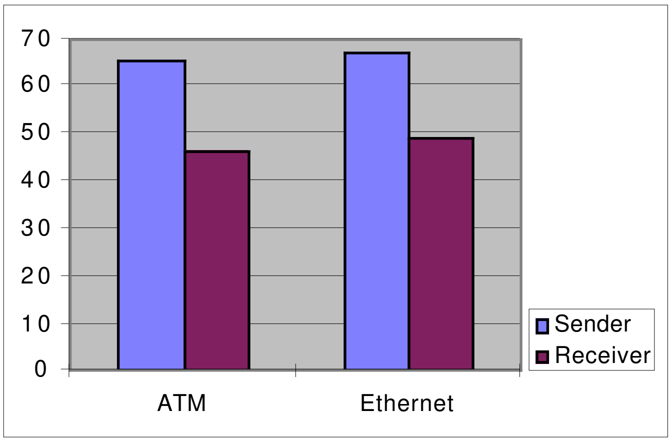

Figure 16. Video Protocol Processing Times

Table 33. The AudioFile Protocol

measurement	number of calls	minimum	maximum	average	variance	standard deviation
Sender ATM						
1	1151	15 us	367 us	19 us	136	11.653246
2	652	15 us	377 us	16 us	201	14.162735
3	638	15 us	354 us	16 us	184	13.580627
Receiver ATM						
1	1151	20 us	168 us	26 us	63	7.950880
2	652	23 us	176 us	28 us	98	9.877656
3	638	23 us	162 us	28 us	85	9.211709
Sender UDP						
4	462	13 us	300 us	15 us	177	13.316612
5	965	14 us	335 us	15 us	129	11.362796
6	397	13 us	1589 us	19 us	6507	80.664866
Receiver UDP						
4	462	25 us	215 us	30 us	181	13.436512
5	965	23 us	193 us	27 us	97	9.874052
6	397	25 us	188 us	29 us	157	12.534443

The graphical representation of these values Figure 16 and Figure 17 show that the protocol processing overhead for the run-time system of Da CaPo++, the lift algorithm, is very minimal. It shows additionally that the average overhead on a per module of these protocols is very low.

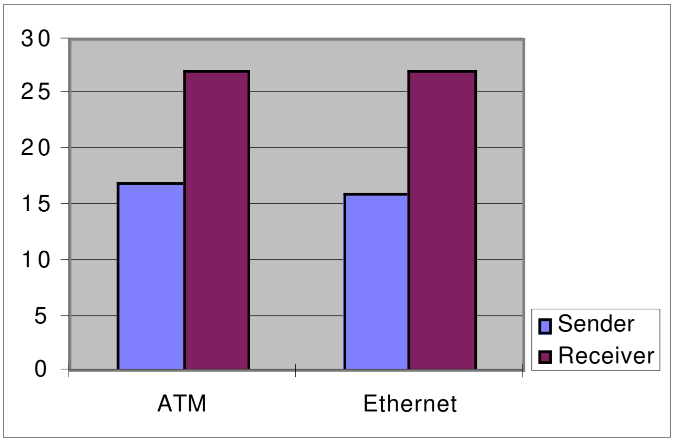

Figure 17. Audio Protocol Processing Times

7.2.4.2 Protocol Times for an Unreliable and Reliable Data Transmission Protocol

The measurement scenario for protocol times for an unreliable and a reliable data transmission protocol is depicted in Figure 18.

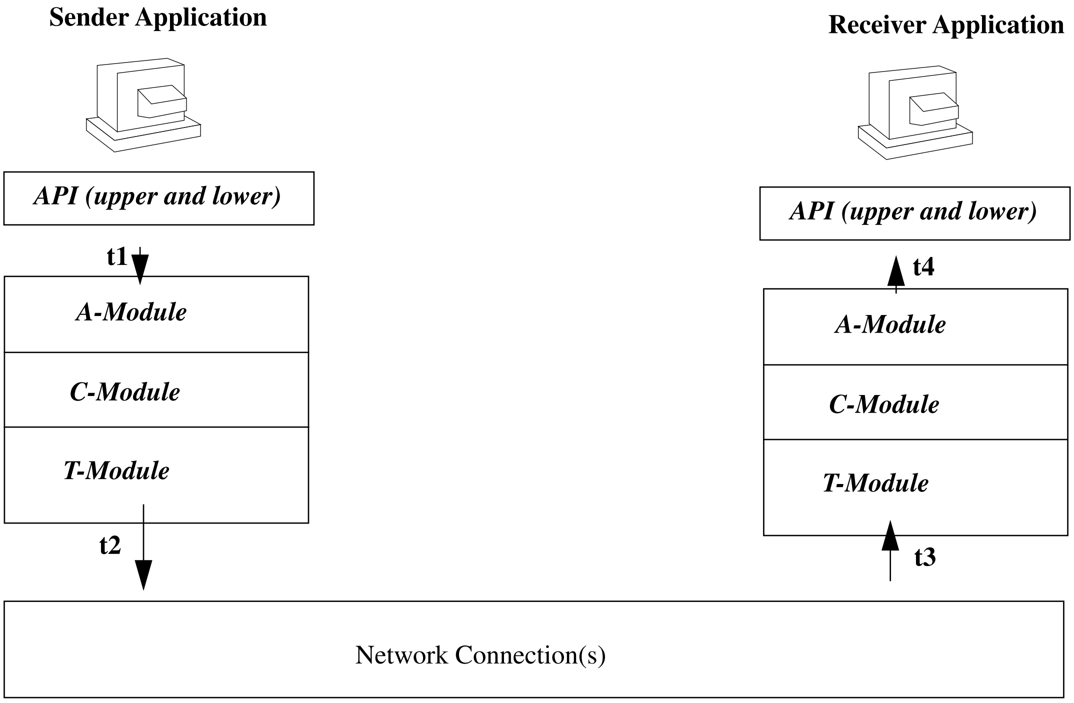

Figure 18. The Measurement Scenario for the Data Transmission Protocols

In this scenario the time points have the following meanings:

- **t1:** is the time when the data to be sent was checked but before it is copied to the Da CaPo++ data packet
- **t2:** indicates when data was put to the network for transmission

- **t3:** the receiving T-Module receives the data packet
- **t4:** is the time after the Da CaPo++ data packet has been copied to the data packet structure of the API
- **t4 - t3**: indicates the time the protocol needs for one packet on the receiving side
- **t2 - t1**: indicates the time the protocol needs for one packet on the sending side

The unreliable data transmission protocol stack comprised the UDP T-Module and the RawAPI A-Module, whereas reliable data transmission protocol stack comprised the TCP Multicast T-Module and the RawAPI A-Module.

The same test application as used for the measurements of the RawAPI A-Module has been used for these measurements.

For the measurements a *usleep()* of 0.1 s has been introduced after the sending of each packet. The *usleep*() has been removed for measurements 7 and 8.

Table 34. Unreliable Data Transmission Protocol

mea-sure-ment	**num-ber of calls**	**minimum**	**maximum**	**average**	**variance**	**standard deviation**
Sender						
1	500	16 us	272 us	18 us	182	13.496975
2	500	15 us	271 us	17 us	178	13.328012
3	500	15 us	273 us	17 us	181	13.448403
Receiver						
1	500	24 us	318 us	32 us	191	13.829144
2	500	26 us	323 us	30 us	187	13.668727
3	500	24 us	322 us	31 us	188	13.704055

Table 35. Reliable Data Transmission Protocol

mea-sure-ment	**num-ber of calls**	**minimum**	**maximum**	**average**	**variance**	**standard deviation**
Sender						
4	500	20 us	265 us	41 us	245	15.645957
5	500	18 us	284 us	20 us	191	13.818488
6	500	18 us	278 us	20 us	197	14.020310

Table 35. Reliable Data Transmission Protocol

measurement	number of calls	minimum	maximum	average	variance	standard deviation
7	500	16 us	859 us	242 us	8678	93.155924
8	500	15 us	877 us	172 us	16645	129.014465
Receiver						
4	500	19 us	279 us	21 us	178	13.328163
5	500	35 us	263 us	41 us	175	13.218126
6	500	35 us	443 us	40 us	397	19.921219
7	500	20 us	80058 us	184 us	12811104	3579.260276
8	500	24 us	278 us	27 us	249	15.785234

7.2.4.3 Protocol Times for Secured Protocols

Secured protocols may consist of one of the above mentioned ones, including a number of required C-modules as evaluated in Paragraph 7.2.2. Therefore, a particular end-to-end view comprises of the times required to deal with the unsecured protocol (cf. Paragraph 7.2.4.1) in addition to the processing required for the C-modules and their communication within the protocol (cf. Paragraph 7.2.2).

7.2.4.4 Connection Manager and Security Manager

Two specific end-to-end issues comprise the Connection Manager and the Security Manager of Da CaPo++. The first one being involved in the set up and tear-down of communication protocols and the provision of communication services only, requires a close investigation of its performance for different networks and different scenarios. The second is required, if secure communications have to be supported. Therefore, its functionality has to be evaluated separately. The involvement for multicast scenarios with the Security Manager has to be discussed.

The Security Manager has been designed and implemented such that unidirectional communication from the creator to all participants in a multicast case is well possible, as long as these participants use all the same key information. Communication from the participants to the creator is possible only, if Da CaPo++ parameters are set such that a key change – which may be precomputed – is issued for each packet sent.

7.2.5 API Measurements

Located between any application and the Da CaPo++ communication subsystem, the Application Programming Interface (API) plays an important role during both connection establishment and data transfer phases. The two main goals in the design of the API were on one hand to provide high efficiency, especially due to the compulsory interprocess communication (IPC), and on the other hand to encapsulate all Da CaPo++ components in high-level abstractions, such as session and flow objects, that can be used straightforward by applications.

The measurement strategy has two goals that are related to both control and data paths. Firstly, it aims at measuring the necessary time to setup a connection. Therefore, the delays to register an application, to instantiate a session object, to either listen or connect, to activate, to deactivate, and to close a session are examined. In all cases, the response times as viewed by the application under a zero-load machine are given. Secondly, the data path is further examined under throughput and timing aspects. As only API performances are relevant, the A-modules were slightly changed in order to behave either like a sink (data consumer) or like a data generator (data producer). Thus, the overhead of the whole Da CaPo++ communication subsystem is not further considered. Note that throughput measurements were not performed for the control path, as the amount of data is by far negligible in comparison with the data path.

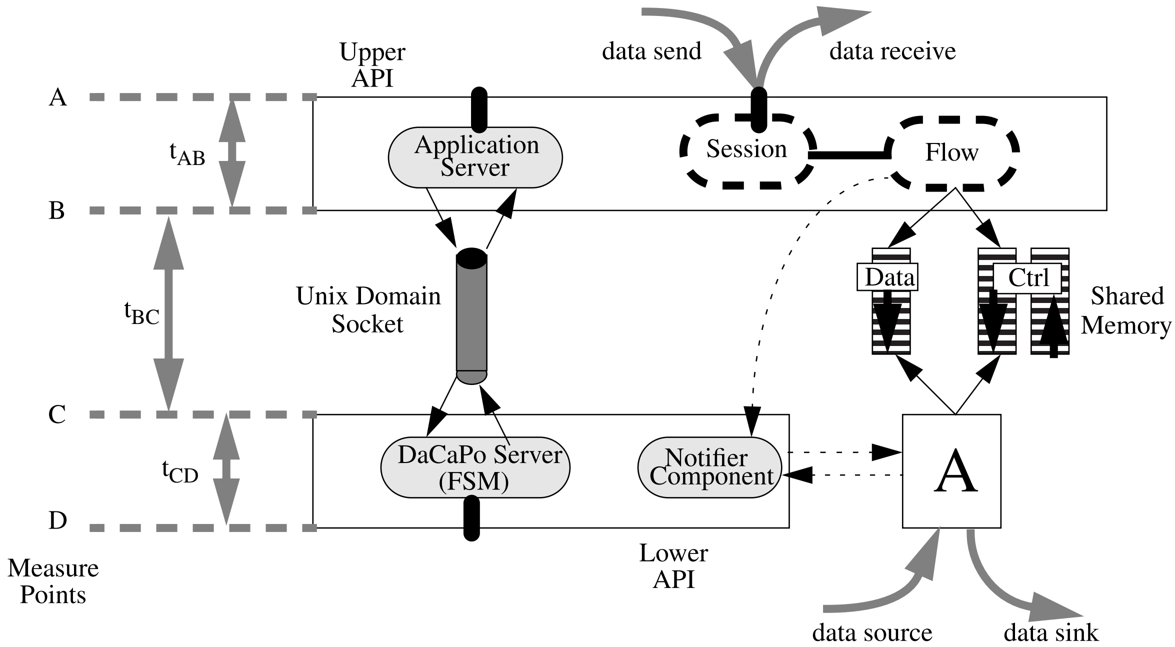

Figure 19. API Measurement Process

The overall environment of API measurements is depicted in Figure 19. Control data are exchanged between the application and Da CaPo++ over a unix domain socket. User data is either injected or received by a test application in the upper API. In the lower API, the A-module either generates new data or consumes incoming data. Four different measure points were identified, namely A, B, C, and D. When IPC mechanisms are used, it is not possible to obtain measures solely based on the Quantify tool. This is due to the fact that Quantify can only provide relative execution times (*e.g.*, the number of CPU cycles required by a given operation), and not absolute time intervals involving several processes. This can be seen on Figure 19 where the time interval t_{BC} cannot be measured via the Quantify tool. The adopted solution consists in using a not expensive system call (gethrtime() only requires 0.6 µsec pro execution) to get an absolute time reference expressed in nanoseconds. The returned time value has only a significance on the machine, and can thus be used straightforward with several UNIX processes. However, it must be kept in mind that measurements obtained with gethrtime() include all the overhead of the machine (especially the operating system), including context switches. This overhead may be a reason for the sometimes very large standard deviation observed.

The way the measure points A, B, C, and D are to be interpreted for various measurement scenarios is depicted in Figure 20. Control path measures are obtained by measuring on one hand the response time as viewed by the application (A-A noted), and on the other hand the Da

CaPo processing time (D-D noted). The delays due to the interprocess segments B-C and C-B are also expressed in relation to the response time. When data is sent, the response time tells when the application is ready to send the next packet, whereas the latency over the API indicates when the data is made available in the A-module. For the receiving of data, only the latency over the API is a significant value. In almost all cases, values obtained with the Quantify tool are provided, and can be compared with the measured elapsed times.

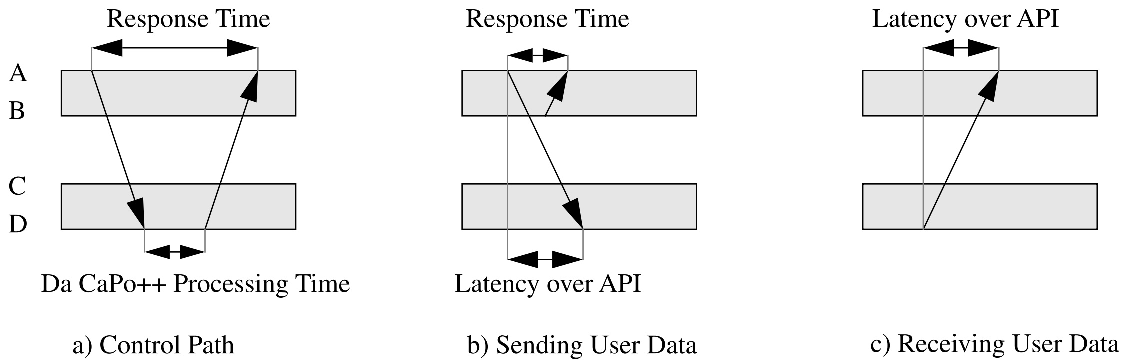

Figure 20. Measurement Scenarios

The results are illustrated in several tables according to the measured functionality. Two different machines were considered. Firstly a Sun UltraSPARC 170E, and secondly a Sun SPARCstation SS20. An attempt was done to keep consistency between all tables in order to facilitate both reading and result interpretation.

Table 36 lists measured results for the instantiation of a DaCaPoClient object (application registration process). As a notable difference for the Da CaPo++ processing time between CRE-

ATOR and PARTICIPANT was measured, values for both roles are reported in Table 36. The time spent in IPC (actually B-C and C-B) was estimated in comparison with the response time.

Table 36. DaCaPoClient Creation Measurements

DaCaPoClient::DaCaPoClient()		**ULTRA 170E [msec]**	**SPARCStation SS20 [msec]**
	response time (elapsed time)	mean=14.88, sdev=0.82, min=13.86, max=16.24, med=15.40	mean=31.70, sdev=1.35, min=29.96, max=33.54, med=32.10
	dacapo++ processing time (elapsed time)	CREATOR: mean=0.059, sdev=0.02, min=0.019, max=0.086, med=0.061 PARTICIPANT: mean=0.013, sdev=0.0, min=0.012, max=0.017, med=0.013	CREATOR: mean=0.127, sdev=0.0, min=0.125, max=0.128, med=0.127 PARTICIPANT: mean=0.031, sdev=0.0, min=0.030, max=0.032, med=0.031
	time spent in IPC (% of response time)	20-35%	50-65%
	real user time	upper API: 2.95msec lower API + DaCaPo: 5.10msec	(not experimented)

Table 37 illustrates the instantiation of a session object. As this operation requires several transitions between upper and lower API, only the response time (A-A) as viewed by the application is reported. The influence of the parsing operation of the configuration file is indirectly measured by instantiating sessions with different complexity levels. Firstly, a session consisting of a single flow without QoS requirements is measured, and secondly a session consisting of 10 flows with each 10 application requirements is setup.

Table 37. Session Creation Measurements

Session::Session()		**ULTRA 170E [msec]**	**SPARCStation SS20 [msec]**
small configuration file: 1 flow without QoS parameters	response time (elapsed time)	mean=44.54, sdev=5.66, min=34.63, max=53.64, med=44.27	mean=75.00, sdev=7.18, min=67.39, max=88.45, med=74.15
	real user time	upper API: 17.45msec lower API: 3.84msec	(not experimented)
large configuration file: 4 flows with each 10 QoS parameters	response time (elapsed time)	mean=141.09, sdev=26.20, min=120.35, max=193.33, med=127.50	(not experimented)
	real user time	(not experimented)	(not experimented)

Table 38 provides measurements for both connect and listen processes. A listen call issued by the application is then transformed in a listen at either the BSD Socket or at the ATM level.

Thus, response times for the listen call do not express anything, as the response time is highly dependent on the time the corresponding connect is invoked.

Table 38. Session connect/listen Measurements

Session::Connect()/Listen()		**ULTRA 170E [msec]**	**SPARCStation SS20 [msec]**
connect	response time (elapsed time)	mean=72.25, sdev=7.30, min=65.51, max=84.23, med=74.59	mean=148.09, sdev=23.17, min=127.89, max=173.38, med=143.01
	dacapo processing time (elapsed time)	mean=40.08, sdev=21.43, min=22.01, max=78.37, med=34.53	mean=135.38, sdev=10.97, min=123.66, max=145.41, med=137.08
	time spent in IPC (% of response time)	32-66%	3.5-15%
	real user time	upper API: 32msec lower API + Da CaPo: 3.73msec	(not experimented)
listen	real user time	upper API: 29msec lower API + Da CaPo: 28.8msec	(not experimented)

The activate and deactivate processes as illustrated in Table 39 show a very large standard deviation which is due to the overhead of the whole Da CaPo++ communication subsystem. Whenever it was possible, separate values for the CREATOR and PARTICIPANT are provided. Similarly, values obtained during the closing of a session and listed in Table 40 have a very large variance. Thus, these values only give an idea on how response times can look like, and are in no way to be interpreted as reliable statistical results.

Table 39. Session activate/deactivate Measurements

Session::Activate()/Deactivate()		**ULTRA 170E [msec]**	**SPARCStation SS20 [msec]**
activate	activate response time (elapsed time)	CREATOR: mean=1.39, sdev=1.37, min=0.66, max=3.43, med=0.79 PARTICIPANT: mean=8.67, sdev=0.47, min=8.23, max=9.18, med=8.31	CREATOR: mean=1.71, sdev=0.21, min=1.59, max=1.96, med=1.60 PARTICIPANT: mean=5.22, sdev=3.24, min=1.68, max=8.04, med=5.95

Table 39. Session activate/deactivate Measurements

Session::Activate()/Deactivate()		ULTRA 170E [msec]	SPARCStation SS20 [msec]
deactivate	deactivate response time (elapsed time)	CREATOR: mean=9.71, sdev=0.15, min=9.55, max=9.83, med=9.77 PARTICIPANT: mean=17.22, sdev=6.70, min=9.62, max=22.26, med=19.80	CREATOR: mean=3.72, sdev=0.07, min=3.64, max=3.78, med=3.73 PARTICIPANT: mean=4.05, sdev=3.03, min=2.26, max=7.55, med=2.35
activate/deactivate	dacapo processing time (elapsed time)	mean=0.71, sdev=1.51, min=0.031, max=5.41, med=0.087	mean=1.65, sdev=1.99, min=0.067, max=5.56, med=0.882
	time spent in IPC (% of response time)	(too variable to be significant)	(too variable to be significant)
activate	real user time	upper API: 0.23msec lower API + Da CaPo: 65usec	(not experimented)

Table 40. Session close Measurements

Session::Close()		ULTRA 170E [msec]	SPARCStation SS20 [msec]
Closing of a small session	response time (elapsed time)	mean=31.16, sdev=6.75, min=22.71, max=43.31, med=30.10	mean=51.76, sdev=53.91, min=3.06, max=131.17, med=45.32
	dacapo processing time (elapsed time)	mean=1.49, sdev=1.63, min=0.025, max=4.26, med=0.93	mean=2.69, sdev=2.52, min=0.037, max=6.84, med=3.00
	time spent in IPC (% of response time)	60-75%	70-85%
	real user time (upper + lower API execution times) (*)	upper API: 0.27 msec, lower API + Da CaPo: 1.25msec	(not experimented)

Table 41 lists performance values that can be expected when either sending or receiving data over the API. All measures were obtained with a similar scenario, namely the sending/receiving of 1000 packets of size 1000 Byte. Firstly, the raw throughput under saturation is measured over the API. The difference between the maximal sending throughput and the maximal receiving throughput is due to a necessary overhead in the A-module (data coming from the application is not made immediately available to the lift algorithm). In addition, Table 21 includes the graphic representation of the achieved application-to-application throughput.

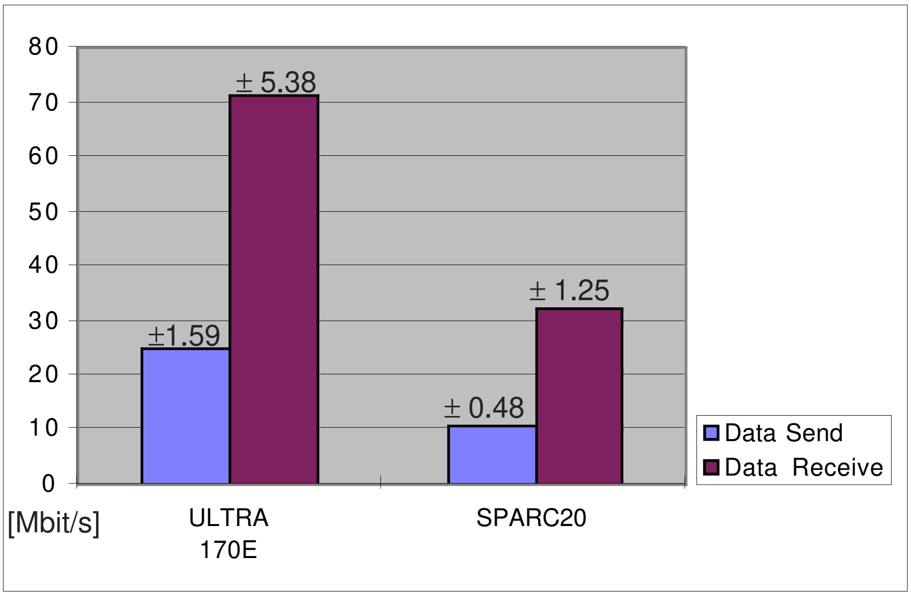

Figure 21. Throughput Measurements Application-to-application

The size of the shared-memory buffer was modified, however, no significant change in the obtained throughput was noticed.

Table 41. Session Sending and Receiving User Data Measurements

Session::SendData()/RecvData() (throughput + timing aspects) Send/Receive 1000 packets of 1000 Byte DEBUG_LEVEL set to 5		**ULTRA 170E [µsec] or [Mbps]**	**SPARCStation SS20 [µsec] or [Mbps]**
sending	maximal throughput under saturation [Mbps] (varying shared-memory size)	large shared buffer: mean=24.59 Mbps, sdev=1.59 Mbps one packet buffer size: mean=24.11 Mbps, sdev=2.68 Mbps	large shared buffer: mean=10.58 Mbps, sdev=0.48 Mbps
	response time for a single data packet (elapsed time)	mean=325, sdev=701, min=38, max=22159, med=236	mean=825, sdev=2079, min=54, max=65769, med=728
	latency over API for a single data packet (elapsed time)	mean=787, sdev=1514, min=316, max=23269, med=816	mean=3005, sdev=6067, min=435, max=65633, med=683
	real user time as given by the Quantify tool	upper API: 77.5 usec lower API: 60.7 usec	(not experimented)

Table 41. Session Sending and Receiving User Data Measurements

Session::SendData()/RecvData() (throughput + timing aspects) Send/Receive 1000 packets of 1000 Byte DEBUG_LEVEL set to 5		**ULTRA 170E [µsec] or [Mbps]**	**SPARCStation SS20 [µsec] or [Mbps]**
receiving	maximal throughput under saturation [Mbps] (varying shared-memory size)	large shared buffer: mean=70.99 Mbps, sdev=5.38 Mbps	large shared buffer: mean=31.93 Mbps, sdev=1.25 Mbps
	latency over API for a single data packet (elapsed time)	mean=700, sdev=249, min=167, max=2251, med=731	mean=872, sdev=844, min=151, max=3339, med=993
	real user time as given by the Quantify tool	(not experimented)	(not experimented)

Finally, the same measurements were performed in the sending direction, with packet sizes ranging from 256 Byte to 8 KByte. The results are listed in Table 42 and graphically depicted in Figure 22.

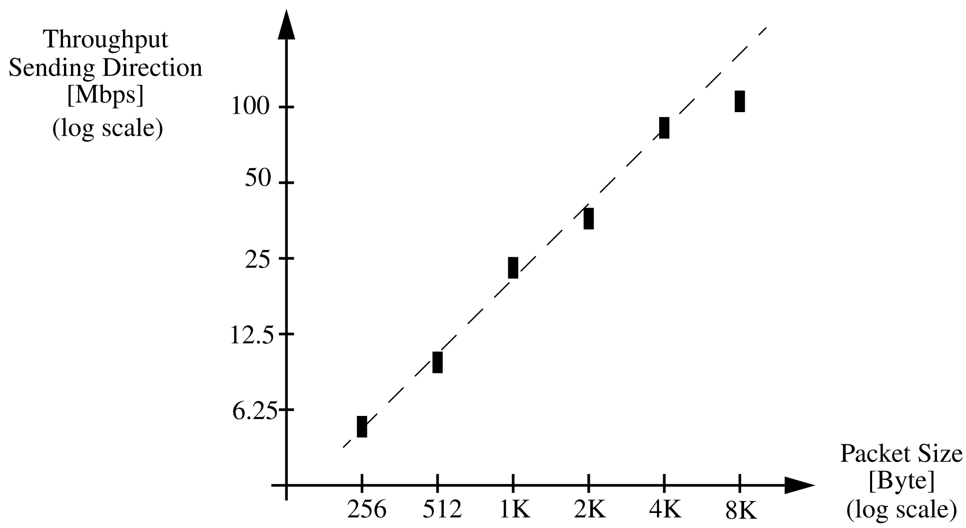

Figure 22. Throughput Measurements With Variable Packet Size

Table 42. Throughput Measurements With Variable Packet Size

Packet size [Byte]	**256**	**512**	**1 K**	**2 K**	**4 K**	**8 K**
Throughput [Mbps] Mean (sdev)	5.38 (0.32)	9.62 (0.97)	24.59 (1.59)	35.73 (2.32)	58.98 (3.21)	108.51 (4.75)

It can be seen that there is almost a linear relation between the packet size and the obtained throughput, with a saturation with 8 K packets and 110 Mbps. This can be explained by the relatively large overhead caused by the synchronization mechanisms on the shared-memory (semaphores). The necessary time to copy larger packets (via syscall memcpy()) is not significant in comparison with synchronization operations, thus, the throughput increases almost linearly with the packet size, until the saturation limit is reached.

7.2.6 Application Framework

Since applications are of main interest for communication systems and their practical behavior in real world scenarios, a number of examples has been implemented for Da CaPo++. These examples are performance evaluated on two different aspects. Firstly, it is the pure numerical performance obtained for applications and scenarios as done for Da CaPo++ core system units (cf. Paragraphs 7.2.1 to 7.2.3), end-to-end issues (cf. Paragraph 7.2.4), and the API (cf. Paragraph 7.2.5). Secondly, application programmer´s experiences made with the API and the Da CaPo++ core are discussed. Thirdly, user acceptance and user-perceived quality investigations should be performed in the future.

For the following list of applications have been implemented within the project:

- Picture Phone (1:1)
- WWW-based Video Server and Video Client (with varying numbers of participants)
- WWW-based Audio Server and Audio Client (with varying numbers of participants)
- Secure Applications

7.2.6.1 The WWW Application Scenario

There are several interesting points to measure in the WWW application scenario that are evaluated here.

Entry Session

As the WWW scenario implies the use of an entry session for the client to connect to the MMServer, transmit the configuration file, obtain the address of the MMSender and connect to the MMSender, the user has to wait for some time, until the MM data transmission is started. The time for the entry session was measured.

The measurement scenario, as depicted in Figure 23, is used to measure the time for the entry session, the following 6 time points are measured:

- **t1**: This time indicates the time the File Client connects to the Entry Session.
- **t2**: This is the time the receiver got the address of the media server and can now connect to the Server. The Entry Session is finished.
- **t3**: t3 indicates the time of all outgoing data packets in the Entry Session on the Client Side. (This is only the packet with the session file).
- **t4**: t4 indicates the time of all incoming packets of the Entry Session. (which is the packet with the session file)
- **t5:** t5 is the time, the MM Server send its address packet in the Entry Session.
- **t6**: At time t6 the File Client receives the packet with the address of the MM Sender.
- **t2 - t1:** indicates the duration of the entry session
- **t5 - t4:** indicates the processing time of the entry session in the MM Server.
- **t6 - t3***:* indicates the duration of network connection **and** processing time in the MM Server.
- **(t2 - t1) - (t6 - t3) + (t5 - t4):** indicates the processing time of the entry session without the time for the network connection.

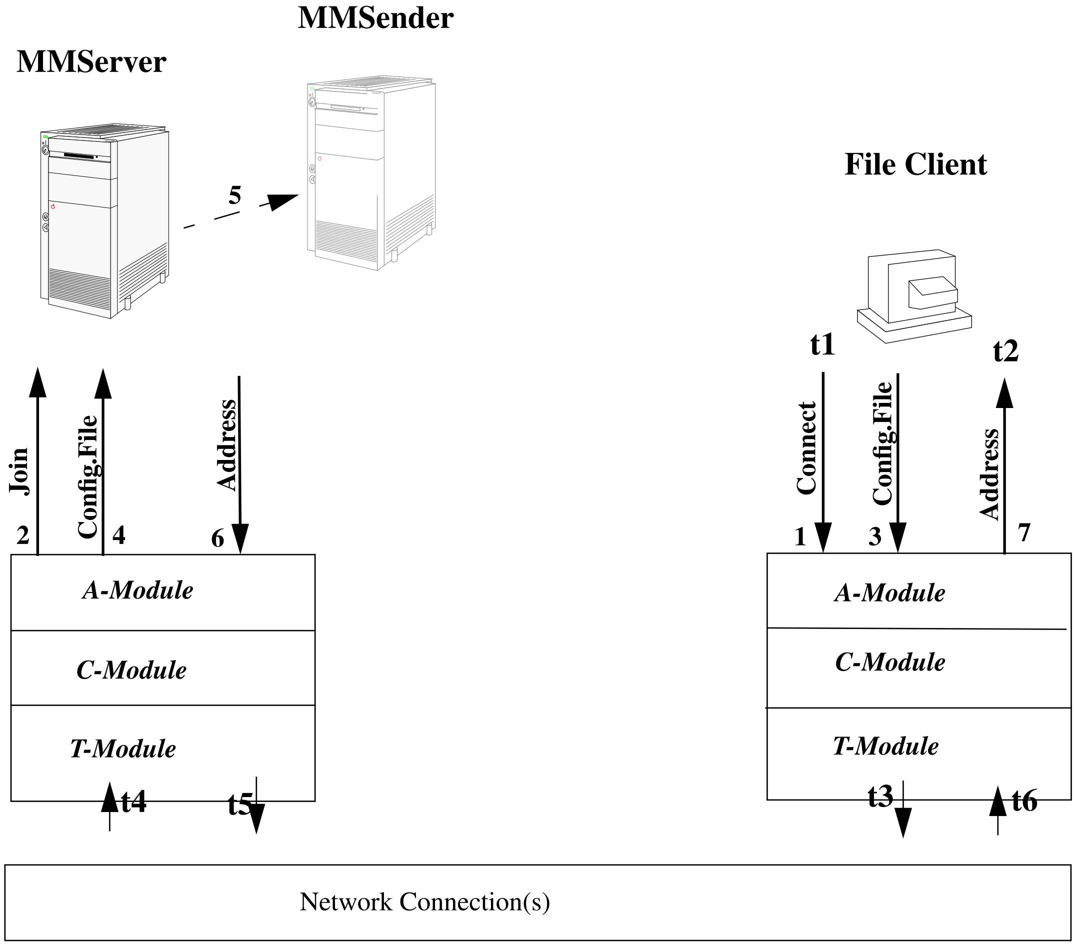

Figure 23. The Measurement Scenario for the Entry Session

All these times in Table 43 and Table 44 can just be measured with *gethrtime()* thus is, in the real time mode, as different processes and different machines are included in this scenario. Due to reliability reasons, the Entry Session has in all cases been implemented on the TCP/IP Protocol.

Table 43. Measurements for the Entry Session

measurement	t2 - t1	t5 - t4	t6 - t3	(t2 - t1) - (t6 - t3) + (t5 - t4)
1	257491 us	520 us	104176 us	153835 us
2	218346 us	4490 us	44395 us	178441 us
3	180875 us	1663 us	50373 us	132165 us
4	191567 us	670 us	86905 us	105332 us
5	185707 us	498 us	86414 us	99791 us
6	274670 us	584 us	101757 us	173461 us

Table 44. Entry Session (Evaluation)

value	number	minimum	maximum	average	variance	standard deviation
t2 - t1	6	180875 us	274670 us	218109 us	1578176587	39726.270747
t5 - t4	6	498 us	4490 us	1404 us	2480785	1575.050782

Table 44. Entry Session (Evaluation)

value	number	minimum	maximum	average	variance	standard deviation
t6 - t3	6	44395 us	104176 us	79003 us	657235907	25636.612621
(t2 - t1) - (t6 - t3) + (t5 - t4)	6	99791 us	178441 us	140504 us	1133450451	33666.755866

To measure the response time for commands, the command *PLAY* has been chosen. When the command is triggered, the time is measured with *gethrtime ()*. When the sender receives the command, the next packet sent has a 2 in its 1 Byte header. The receiving A-Module records the time, whenever a packet with the header byte 2 is received.

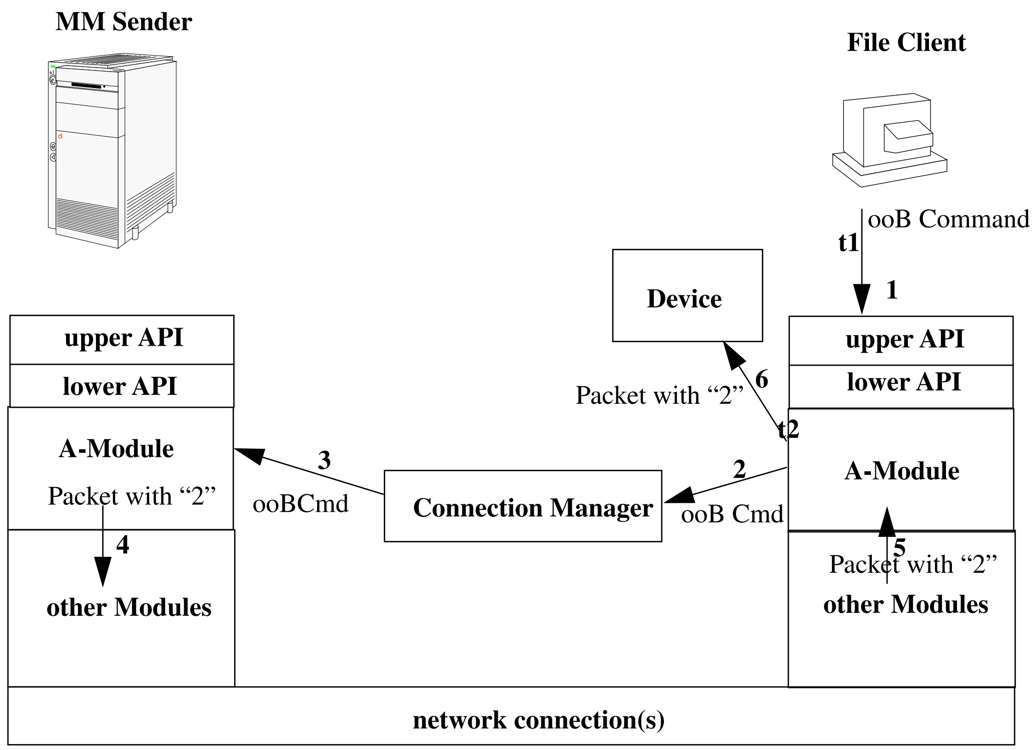

Figure 24. Measurement scenario for the response time of commands

As Figure 24 shows the measured response time for commands in the WWW scenario is not independent from:

- the time the Upper and the Lower API need for their inter process communication,
- the time, the connection manager takes to transmit this command,
- the time, the sending A-Module takes to respond to the command, which also depends on the frame/sample rate at packets are sent periodically and not as soon as the play command has been triggered,
- the network transmission time and
- the processing time in the receiving A-Module.

For this reason the following measured values in Table 45 just serve as indicators how long users might have to wait for a command to be performed.

Table 45. Response Time of Commands

measurement	number of calls	minimum	maximum	average	variance	standard deviation
udp, audio16						
1	16	84005 us	161860 us	102512 us	318551434	17848.009232
2	21	72443 us	167013 us	97295 us	462812611	21513.079997
3	12	76472 us	161657 us	101503 us	541349932	23266.927858
atm, audio						
4	12	8783 us	54721 us	27266 us	259689382	16114.880771
5	10	10794 us	51181 us	33194 us	318296662	17840.870545
6	9	16828 us	53373 us	30434 us	190816054	13813.618414
udp, video						
7	13	104117 us	1132907 us	210105 us	77581878617	278535.237658
8	19	106063 us	1090689 us	193725 us	47984930003	219054.627897
9	17	109423 us	2168197 us	276179 us	238252850204	488111.514107
atm, video						
10	9	14633 us	1285537 us	194029 us	168377862266	410338.716508
11	13	13438 us	940490 us	120422 us	61218780331	247424.292120
12	11	10310 us	1331298 us	177682 us	146962542184	383356.938354

As the sending A-Module sends data controlled by a periodic timer, the packet with the header of 2 is also sent after a timer alarm was triggered. Thus, the response time for commands depends also on the value for the periodic timer which is higher in the Video File case than in the Audio File case.

8. Summary

As the number of performance investigations and evaluations determine, Da CaPo++ is capable of supporting multimedia applications on standard workstations in a sufficient manner. Protocol processing tasks can be performed efficiently within the run-time system (Da CaPo++ kernel). The modules being implemented show a comparable performance for software-based systems and are applicable in real-life scenarios. This has to be emphasized especially for the rich number of security modules being implemented und utilized.

The Application Programming Interface (API) determines the visible interface to application programmers utilizing Da CaPo++. As it provides a data stream bypass functionality it is well suited for multimedia data streams, heading for devices or coming from them. Even though, the performance obtained for data crossing the API is suitable for applications running on standard workstations.

The Da CaPo++ Middleware provides by many means an efficient and flexible platform for supporting a broad range of multimedia applications on standard UNIX workstations. As the overall protocol performance numbers determine, a high number of concurrent streams can be supported and protocol functionality can be selected flexibly by applications or users.

9. References

/HeMi81/ D. J. Healy, O. R. Mitchell: *Digital Video Bandwidth Compression Using Block Truncation Coding;* IEEE Transactions on Communications, Vol. 27, No. 9., September 1979, pp 1335 – 1342.

/JPEG90/ ISO: *Joint Photographic Expert Group;* ISO-CD 10918.

/PPVW92/ T. Plagemann, B. Plattner, M. Vogt, T. Walter: *A Model for Dynamic Configuration of Light-Weight Protocols*; 3rd IEEE Workshop on Future Trends of Distributed Systems, Taipeh, Taiwan, April 1992, pp 100 – 106.

/Pure93/ Pure Software: *Quantify User´s Guide;* Pure Software Inc., Sunnyvale, California, U.S.A., 1993.

/SBCC97a/ B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel: *Communication Support for Distributed Applications,* Institut für Technische Informatik und Kommunikationsnetze (TIK), ETH Zürich, Switzerland, TIK-Report No. 25, January 1997.

/SBCC97b/ B. Stiller, D. Bauer, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Vogt, M. Waldvogel: *Project Da CaPo++, Volume I: Architectural and Detailed Design;* Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich, Switzerland, TIK-Report No. 28, July 25, 1997.

/SBCC97c/ B. Stiller, G. Caronni, C. Class, C. Conrad, B. Plattner, M. Waldvogel: *Project Da CaPo++, Volume II: Implementation Documentation;* Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich, Switzerland, TIK-Report No. 29, August 25, 1997.

/Stil96/ B. Stiller: *An Application Framework for the Da CaPo++ Project;* 5th Open Workshop on High Speed Networks, ENST, Paris, France, March 20–21, 1996, pp 4-17 – 4-24.

/SunV94/ Sun Microsystems: *SunVideo User´s Guide;* Sun, Mountain View, California, U.S.A., 1994.

/Tane96/ A. Tanenbaum: *Computer Networks;* 3rd Edition, Prentice Hall, London, England, 1996.