Modules as Building Blocks for Protocol Configuration

Thomas Plagemann
Martin Vogt
Bernhard Plattner
Thomas Walter

Swiss Federal Institute of Technology Zurich
Computer Engineering and Networks Laboratory

Abstract

Da CaPo (Dynamic Configuration of Protocols) provides an environment for the dynamic configuration of protocols. Configuration is done with respect to application requirements, properties of the offered network services and the available resources in the end systems. The goal of the configuration is to provide a service with minimal necessary functionality for each request, i.e., to diminish protocol complexity and so to increase protocol performance. Modules serve as basic building blocks for the protocol configuration. Common software engineering principles like encapsulation and information hiding as well as a unified module interface allow the unconstrained configuration of modules to protocols. The Da CaPo runtime environment links modules to protocols in one UNIX process and realizes an efficient data transport inside the end systems because performance reducing operations like data copying or process switches are minimized.

1. Introduction

It is a well known fact that the bottleneck in the performance of distributed applications in modern high speed networks is located in the end systems. Reasons for this 'slow-software-fast-transmission' bottleneck are the insufficient processing power of the end systems, the overhead in the end system protocols and the insufficient embedding of the communication subsystem in the operating system [1].

The aim of the Da CaPo (**D**ynamic **C**onfiguration of **P**rotocols) project is to overcome the communication speed bottleneck by configuring end system protocols. Protocols will be optimally adapted to the requirements of the applications, to the offered network services and to the available resources in the end systems. The properties of network services and end system resources as well as the application requirements are described in a common syntax. The configuration process uses these property descriptions and an abstract protocol specification to estimate the property and the complexity of possible configurations. It selects a protocol which fulfills the application requirements and uses a minimum amount of resources. This protocol represents a light-weight protocol, as all unnecessary protocol functions are excluded [2]. Naturally, the reduction of protocol complexity increases its performance. Multi-media applications requiring high performance and tolerating unreliability up to a certain extent are of special interest in this context.

Besides this performance aspect Da CaPo enforces the implementation of reusable building blocks and may be used as a flexible test- and measurement environment. New building blocks can be easily integrated, tested and compared with other building blocks implementing the same functionality.

In this paper we mainly concentrate on the design of basic building blocks for protocol configuration and their implementation in Da CaPo. The next section gives an overview of Da CaPo, its foundation and its components. The main part of the paper (Section 3) discusses protocol functions and their implementing modules. Based on this discussion we show design guidelines for modules and how to use them as building blocks for protocol configuration. The implementation related aspects of our module design, the embedding into the runtime environment and the integration into the UNIX operating system is described in Section 4. Section 5 discusses related work and Section 6 gives some conclusions.

2. Overview of Da CaPo

The three layer model for dynamic configuration of light-weight protocols [3] is the foundation of Da CaPo. The model is realized by three co-operating entities which perform the following tasks:

- the configuration process determines the most suitable protocols,
- the connection manager dynamically negotiates the protocol configuration, and
- the resource manager provides a run-time environment for the configured protocols.

0-8186-3670-X/93 $03.00 © 1993 IEEE

A database stores information of general interest and is mainly used by the configuration process.

2.1 Three layer model

Da CaPo is based on a three layer model which splits communication systems into the layers A, C and T (Figure 1). End systems communicate with each other via layer T, the transport infrastructure. The transport infrastructure represents the existing and connected communication infrastructure, its service is named T service[1]. The T service is different from the OSI transport service. The service of the layer T is a generic service that can be the service of a MAC layer of a MAN such as DQDB, an ATM adaptation layer or the service of a transport protocol as TCP or XTP. In layer C the end-to-end communication support adds functionality to the T services in such a manner that at the AC-interface services are provided to run distributed applications (layer A).

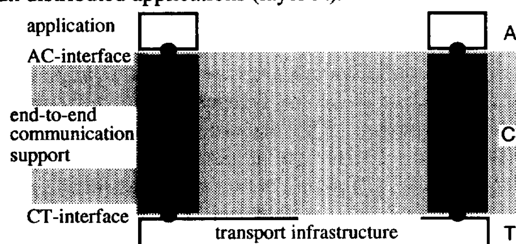

Figure 1: Three layer model

Layer C services are decomposed into a set of protocol functions. Each protocol function encapsulates a typical protocol task like error control, flow control, de- and encryption, presentation coding, etc. Data dependencies between the protocol functions, arranged on top of a T service, define a partial order on the protocol functions and are specified in a *protocol graph.* Independence between protocol functions, i.e., the possible parallel execution of these protocol functions, is directly expressed in the protocol graph. A protocol graph is an abstract protocol specification which has to be defined by a protocol engineer (see example in section 2.5). If multiple T services can be used, there is one protocol graph for each T service.

Protocol functions can be realized in multiple ways, by different protocol mechanisms, as software or hardware solutions. We call the implementation of a protocol function a module. Modules implementing the same protocol functions are characterized by different properties, e.g., different throughput figures or different degrees of error detection and correction. The protocol function error control can be implemented by a single parity bit or by the CRC-CCITT algorithm. While the computation of the parity bit is very simple and fast, the probability of error detection in a block is only 0.5. In contrast the CRC-CCITT is computationally expensive and reduces the throughput, but has much better error detection capabilities. To configure a protocol each protocol function must be instantiated by one of its modules.

2.2 Configuration process

The configuration process selects the most suitable modules depending on the application requirements and available T services [5]. The configuration process retrieves the protocol graphs of the invoked C service from the database and estimates the properties of possible protocol configurations. The instantiation of all protocol functions in a protocol graph with one of their modules creates a possible configuration, the resulting graph is called *module graph.* The configuration process starts with the properties of the T service and according to the protocol graph, estimates step by step the influence of the selected modules on the offered service of subordinated modules. For instance a selective retransmission module guarantees that all packets are received and introduces delay jitter while retransmitting a packet. Naturally, the empty module, which might be used for every protocol function, has no influence on the service. The description of this influence is part of the module's properties stored in the database. Furthermore, module properties indicate their availability (e.g., HW module) and their behavior in relation to the current load. We use a common syntax to describe the properties of modules, T services, protocols and application requirements. We call this language L [3]. All descriptions consist of tuples of attribute types and values. Furthermore, for each attribute the application requirements include a weight function to define the relative importance of the attribute with respect to the other attributes. Attribute values in the application requirements specify so-called *knock-out* values and indicate that the selected protocol must fulfill these requirements. If there are no threshold values, it is indicated by the *don't care* value '*'. This enables us to formulate contradictory requirements (e.g., high bandwidth - low costs) and to deal with a wide range of application requirements, mainly introduced by new multi-media applications. Table 1 contains the BNF definition of L.

The unified representation enables a direct comparison of application requirements and protocol properties. If a configured protocol fulfills the application requirements, i.e., do not violate any knock-out values, we say the protocol is in *compliance* with the application requirements. The protocol with the highest compliance degree is the

[1] Tschudin [7] calls T services "anchored instances".

best configurable protocol with respect to the application requirements.

RequirementsOrProperties ::=ApplicationRequirements | ModuleProperties | TransportProperties
ApplicationRequirements ::= "<" AppRequire ("," ApplicationRequirements |) ">" |
ModuleProperties ::= "<" ModuleProperty ("," ModuleProperties |) ">"
TransportProperties ::= "<" TransportProperty ("," TransportProperties |) ">"
AppRequire ::= "<" ARType "," (ARValue | "*") "," Weight ">"
ModuleProperty ::= "<" ARType "," Function ">"
TransportProperty ::= "<" ARType "," ARValue ">"
ARType ::= (* enumeration of application requirements *)
ARValue ::= (* denotation of a value *)
Weight ::= (* enumeration of functions *)
Function ::= (* enumeration of functions *)

Table 1: BNF definition of L

Protocol specifications define the functionality and the packet structure of a protocol. The module properties include the number and the length of header fields used by the module. The configuration process collects this information from all modules of the protocol according to the module graph and defines the header structure of the protocol. The configuration process delivers the description of the protocol in form of a module graph and the header description to the connection manager. Pre-defined configurations, which are compatible with standardized protocols [6], are also supported in Da CaPo by this way.

2.3 Connection manager

The connection manager assures that communicating peers use the same protocol for a layer C connection [7]. In detail the connection manager has the task to co-ordinate and perform

- the establishment of connections,
- the reconfiguration of existing connections,
- the exploitation of connections, and
- the handling of errors which cannot be treated inside a module.

In the connection establishment phase and in the reconfiguration phase the connection managers negotiate a configuration via a fixed 'protocol configuration' called *management support protocol.* Unreliable T services are enhanced by the management support protocol to offer a reliable service for the connection managers. The management support protocol is embedded in the run-time environment of Da CaPo in the same way as the application protocol.

Three different negotiation scenarios are maintained, and can be selected by the application requirements:

- In a *local configuration* the connection manager of the initiating system informs the peer connection manager about the selected protocol.
- In a *global configuration* the connection manager of the initiating system starts a local configuration and demands his peer to do the same. The initiating connection manager sends a list of the most suitable local protocol configurations to his peer, which compares the results with his own results, selects the best combination and informs his peer about the selected protocol.
- In the *combined configuration* the data exchange between the applications starts directly with a pre-defined protocol. Meanwhile the connection managers perform a global configuration 'out-of-band'.

After the successful negotiation the connection managers request their resource managers to establish the selected protocol. In the combined configuration the pre-defined protocol is substituted by the result of the global configuration; this corresponds to a reconfiguration. Further reasons for reconfigurations are changes in the properties of the T service or in the end systems offending the application requirements. The resource manager detects the changes and requests a reconfiguration by the connection manager. Da CaPo supports two strategies for the reconfiguration, first with possible data loss and second without data loss but with additional delay. The application has the possibility to select the reconfiguration strategy by specifying it in the application requirements.

2.4 Resource manager

The resource manager provides an efficient run-time environment for Da CaPo protocols [8]. The resource manager performs the following tasks:

- linking modules according to the module graph,
- initialization of modules,
- synchronization of parallel modules,
- monitoring the properties of a protocol,
- monitoring the available resources,
- dynamic reconfiguration of protocols, and
- release of modules and resources after connection release.

In contrast to other architectures, which need a process per data packet [9], Da CaPo uses only one process per protocol. This process awaits data from either the application or the T service, like the UNIX System V Streams [10] from head and driver. It then guides the data packet from one module to the next until the data packet reaches its local destination (T service or application) or is blocked by a module, e.g., to enforce a rate control. In both cases the process continues to obtain a new data packet from the application or the T service.

This algorithm allows the execution of a protocol within a single process. It is also possible to integrate the application into this process as the application can be designed to have the same interface as the modules. This way, the application can directly benefit from the buffer management provided by the resource manager. Thus, unnecessary copy operations on data are avoided. On a single processor machine the modules are sequentially executed, while on a multi-processor machine or on a machine with specialized hardware the modules are started in parallel and synchronized by the resource manager.

2.5 Example

We illustrate the concepts and goals of Da CaPo with a simple example. Figure 2 shows two protocol graphs for a video transfer service. Both graphs are instantiated by different modules (gray boxes) according to application requirements and properties of T modules, (a) ethernet and (b) ATM adaptation layer 1 (AAL1). Protocol graph (a) includes the protocol functions de-/encryption, segmentation/reassembly, sequencing, error handling and flow control. Protocol graph (b) includes de-/encryption, sequencing and error handling. Segmentation/reassembly and flow control is performed in AAL1.

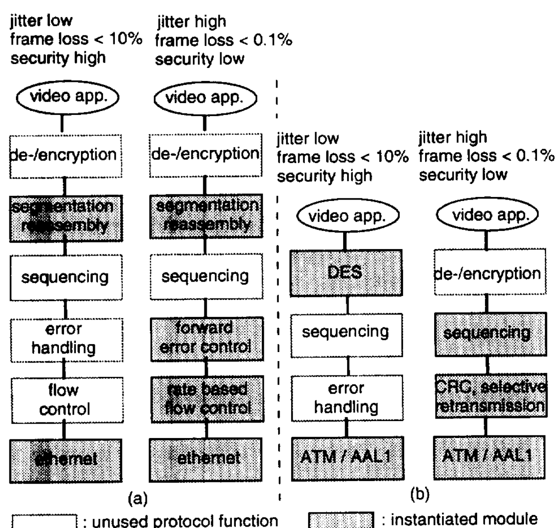

Figure 2: Protocol graph for video

Application requirement 1 (AR1) demands low delay jitter, frame loss lower than 10 % and high security. Application requirement 2 (AR 2) tolerates high delay jitter, 0.1% frame loss and low security.

We assume that the properties of a low loaded ethernet meet AR 1 directly: the ethernet introduces low delay jitter, the average frame loss is lower than 10% and it is secure because it is used in-house. Only segmentation/reassembly modules are needed in this case (the maximum transfer unit of ethernet is smaller than the size of video frames). The properties of AAL1 are similar, frame loss is lower than 10%, no additional delay jitter is introduced, but security is not guaranteed. Therefore, the configuration for AR 1 on AAL 1 consists of modules performing de-/encryption according to data encryption standard (DES) to realize secure video transfer.

To assure the low frame loss in AR 2 both T services are enhanced by error handling modules. We placed forward error control and rate based flow control on top of ethernet to reduce the number of losses. AAL 1 is extended with CRC and selective retransmission to guarantee reliability. Selective retransmission increases delay jitter and reorders packets, the high delay jitter is tolerated from the application but sequencing modules are needed for this configuration.

3. Modules as building blocks

In this section we discuss the design of modules to use them as building blocks for protocol configuration in Da CaPo.

3.1 From protocol functions to modules

Protocol functions describe typical communication tasks of protocols, e.g., error control, flow control, presentation coding, en- and decryption, etc. In general, protocol functions can be realized by different protocol mechanisms. For example, presentation coding can be performed by applying basic encoding rules (BER), external data representation (XDR), JPEG or MPEG. Protocol mechanisms specify the rules supporting the data transfer from one service user to its peer and they specify the data handling. Communicating entities have to use the same protocol mechanisms, otherwise they cannot understand each other. Each protocol mechanism consists of a sending and a receiving part, e.g., encoding and decoding or the sending and receiving part of idle repeat request (IRQ). Single protocol mechanisms support unidirectional flow of application data. The sending partner transfers data packets including application data and control information to the receiving peer. Depending on the particular protocol mechanism the receiving part might return control information to the sender.

Software and hardware implementations of protocol mechanisms in Da CaPo consist of a sending and a receiving module. Figure 3 illustrates the different levels of abstraction between protocol functions and modules. Protocol functions are used in a protocol graph for abstract protocol specification. This specification is the input of the configuration process. Protocol mechanisms assure

that communicating entities use the same protocol and modules embody building blocks for protocol configuration.

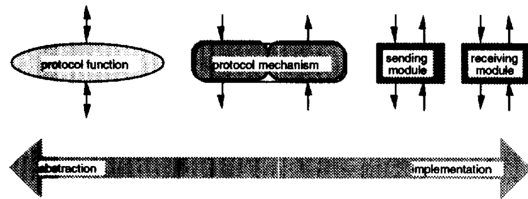

Figure 3: Implementing protocol functions

3.2 Packet handling

The module design aims at the provision of a free and unconstrained combination of modules to protocols [11]. This requires to apply software engineering techniques like encapsulation, information hiding as well as a unified module interface. A unified module interface must be designed according to the tasks of modules in Da CaPo protocols and to the runtime environment. Such an interface must support the communication of modules with their peers and the packet exchange between adjacent modules in the end systems. The services of lower modules in the module graph are used by higher modules to communicate with their peers. They again offer an enhanced service to the next higher modules.

The general task of modules is to handle packets, the payload and/or the header. The modules are passing application data and control information in form of a packet from the sender to the receiving peer and may handle control packets in the opposite direction. In the sending end system packets are passed from the application step by step from module to module up to the T module (the access point of the T service). In the receiving end system the packets are passed from the T module stepwise to the A module (the access point of the C service). We call this direction the *main direction*. Some modules in the receiving system pass control information to the sender in the *back direction*. Main and back direction define a major part of the unified module interface. The four packet handling procedures are (Figure 4):

- `request`: The resource manager delivers a packet in the main direction to the module. The module processes the packet, keeps it internally and directly returns to the resource manager.
- `indication`: The resource manager fetches a processed packet from the module and forwards it in the main direction.
- `request_back`: The resource manager delivers a packet in the back direction to the module. The module processes the packet, keeps it internally and directly returns to the resource manager.
- `indication_back`: The resource manager fetches a processed packet from the module and forwards it in the back direction.

Modules support either the `request` and `indication` procedures (e.g., an encoding module) or they support all four packet handling procedures (e.g., an IRQ module).

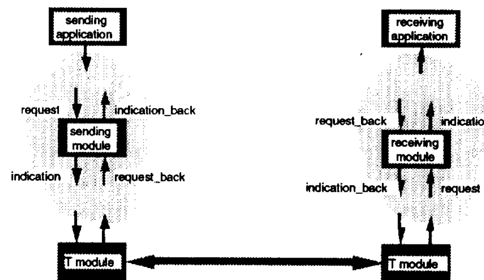

Figure 4: Packet handling procedures

3.3 Module states

The resource manager uses module states to control and to optimize the packet forwarding process. For instance, a sending IRQ module which is waiting for an acknowledgment cannot process additional packets. All modules directly return to the resource manager from a packet handling procedure and indicate their state. We distinguish the following module states:

- DONE: The module is initialized and is waiting for data.
- READY: The module has successfully processed a packet which can be forwarded by the resource manager in the main direction.
- WAIT: The module is waiting for a packet in main direction.
- READY_BACK: The module has successfully processed a packet which can be forwarded by the resource manager in the back direction.
- WAIT_BACK: The module is waiting for a packet in back direction and cannot process packets in the main direction.

Possible state transitions of a reassembly module which only handles packets in the main direction are the following: After the initialization the module is in state DONE. The first data packet provided by a `request` changes the module's state to WAIT. Following packets are reassembled and the module stays in state WAIT until a packet fills the application sized packet. The last `request` changes the module state to READY. The application sized packet can now be handled by higher modules. The

resource manager fetches the packet with `indication` and the module is again in state DONE.

3.4 Protocol context and module statistics

The packet handling procedures are used to forward packets inside the module graph, but they are not adequate to perform all the information exchange between modules and resource manager. In order to work, the modules need information referring to the current protocol configuration. We call this information the *protocol context*. The main contents of a protocol context is a description of the packet header structure determined by the configuration process. Each module retrieves the position of its header fields from the protocol context. Additionally, the protocol context includes information to adapt the modules to the particular configuration, e.g., a segmenting module requires the maximum transfer unit of the T service.

The resource manager has to monitor the properties of protocols. This includes collecting internal statistics of modules like the average number of retransmissions in a sending IRQ module or the number of detected errors in a receiving CRC module. Statistics and protocol context are exchanged between resource manager and modules by the following procedures:

- `statistics`: The resource manager retrieves internal statistics of a module.
- `init`: The resource manager initializes all modules of a protocol in the protocol establishment phase and provides them with the protocol context. The init procedure additionally allocates the module context (Section 4.1).
- `exit`: The resource manager calls the module's `exit` procedure in the release phase of a protocol and frees all its resources.

3.4 T modules and A modules

T modules and A modules are special types of modules because they terminate the module graph. T modules realize access points to T services and A modules realize access points to layer C services. Only modules of these types have to handle data streams of different protocols. The A module might have to combine two unidirectional protocols to a bi-directional C service and the T module might have to de-multiplex different protocols. Both module types only need one pair of packet handling procedures (`request` and `indication`) because they 'consume' or 'produce' packets.

For instance, in a distributed video application a camera produces video frames. The `indication` procedure of the A modules can be realized by a device driver reading video frames from the video hardware. The `request` procedure of the receiving peer writes video frames through the device driver onto the frame buffer of the display hardware. The `init` procedure prepares the particular device drivers and the video hardware (if necessary).

The tasks of T modules depend on the T services and can be classified into two groups: those which address application processes and those which do not. We introduce the tasks of this groups with two examples of well known T services: TCP and the MAC layer of Ethernet [12].

A T module for TCP opens a TCP connection to the peer in the `init` procedure and uses internally the socket interface. The `request` procedure provides a packet for the T module which in turn processes, i.e., sends it to its peer. Incoming packets are accepted by the T module and are forwarded as soon as the resource manager calls the `indication` procedure. The `exit` procedure closes the TCP connection.

A T module for connection-less services of the Ethernet MAC layer has to multiplex outgoing packets and to demultiplex incoming packets. The `init` procedure is used to inform the T module about new protocols using the T service. For each new protocol the T module inserts an entry in a de-multiplexing table[2]. The `exit` procedure is used to remove the table entry or the filter. The `request` procedure copies the packet in an Ethernet frame to the Ethernet controller which sends the packet. Incoming Ethernet packets are stored in different queues according to the service using protocols and can be picked up by the resource manager with the `indication` procedure.

4. Implementation aspects

The entire module design is object-oriented but for pragmatic reasons we used ANSI C in a UNIX environment for the implementation of Da CaPo. The goal of the module implementation is to minimize performance reducing operations and to efficiently integrate them into the run-time environment.

4.1 Module instances

The run-time environment of Da CaPo integrates modules to layer C protocols in one UNIX process, called *Da CaPo process*. Multiple instances of one module can be used in one process. The different module instances must be distinguished. A simple example is the use of a go-back-n module in the management support protocol and in the application protocol. Different instances of one module are distinguished by an instance identifier pointing to their context information. The `init` procedure is used to create and fill a new structure for the context information and to inform the module about its instance identifier,

---

2 Another possibility is to define a filter for each protocol [13].

i.e., it creates a new module instance. Each module instance implements a finite state machine and the current state of the finite state machine is stored in the context information of the module.

All procedures of the module interface require a pointer to the context information structure to identify the particular instance. The resource manager maintains the relations between module instances, context information and packets.

4.2 Buffer and timer management

We implemented a specialized buffer management to optimize the allocation and de-allocation of buffer space for packets.[3] The UNIX system call `malloc` is only applied if no pre-allocated or unused buffer space is available. Modules which allocate buffer space are responsible to de-allocate the buffer space. Modules which store packets like the sending IRQ module lock the packet and unlock it if they do not further use it (the sending IRQ module received an acknowledgment). Furthermore, the buffer management supports the segmentation of a packet in multiple small packets without any copy operation.

The implementation of layer C protocols in one UNIX process requires a special timer handling. UNIX supports only one timer per process. The timer management in Da CaPo allows multiple timers in one process. All timer requests are stored with the instance identification of the corresponding module in an ordered list according to the expiration time. If a timer expires, the resource manager invokes the appropriate module instance.

4.3 States of Da CaPo processes

A Da CaPo process is in one of the three states WORK, LIFT or MONITOR. The resource manager initiates the transitions between these states.

The basic state WORK comprises all tasks not related to data flow and monitoring. This means that application, connection manager, configuration process and parts of the resource manager itself execute in this state.

The resource manager changes the state to LIFT as soon as a packet is available at a T module or an A module. This state transition is triggered by a signal. A timer enforced 'production' of a packet, e.g., retransmission of a packet in an IRQ module after the time-out, is the second possibility for this state transition. Both possibilities are handled by the procedure `packet_available` which guarantees that no additional transition takes place and all packets are treated at return in state LIFT. The process executes the lift algorithm until all modules are in state DONE or WAIT. The lift algorithm forwards packets through the module graph to its destination, i.e., the T module or the A module.[4] Afterwards the resource manager switches the Da CaPo process to the state WORK.

The state MONITOR serves to monitor the properties of protocols and to collect statistical data. This transition is periodically triggered by a timer.

5. Related work

Various authors have proposed similar approaches to simplify protocol implementation by modular design or to overcome the performance bottleneck in end systems. Hutchinson and Peterson introduce the x-kernel [9] which provides an explicit architecture for constructing and composing network protocols out of protocol objects. Protocols implemented on the x-kernel are static because the relations between the protocol objects are defined at the time a kernel is configured. O'Malley and Peterson [15] address the issue of constructing a protocol entity from a set of micro-protocols (the equivalent to protocol functions) and virtual protocol. Virtual protocols guide messages through the protocol graph according to the 'routing information' in the message header. This allows a flexible selection of protocols in one fixed protocol graph (implemented on the x-kernel). UNIX System V Streams [10] enable the linear composition of software modules to network protocols at runtime. The goal of Streams is to simplify protocol implementation, without regard to application requirements [14]. Haas [1] describes an architecture consisting of three layers and a horizontally oriented protocol for high speed communication (HOPS) built on simple protocol functions. Haas proposes a "protocol with a menu", whereby users can request some combinations of functions that are needed to achieve some level of performance. The object-oriented system ADAPTIVE [16] focuses on a transport system architecture and allows protocol configuration and reconfiguration constrained by one protocol graph. The function-based model for a communication subsystem F-CSS [17] is the most similar approach to Da CaPo. F-CSS supports the application-driven configuration of protocol machines tailored to the needs of an application. F-CSS is designed to run on dedicated hardware (transputer network).

6. Conclusions

We have introduced design and implementation guidelines for modules for the unconstrained protocol configuration in Da CaPo. Naturally, such a general approach includes overhead in comparison to highly specialized monolithic

[3] The functionality of the buffer management is similar to the buffer management in the x-Kernel [9].

[4] This algorithm is described in detail in [13].

protocols. But the diversity of current and future application requirements (in particular multi-media applications) and network services (and their QoS) makes it hard to implement for every combination a special light-weight protocol.

All major parts of Da CaPo are implemented under SunOS 4.1.3. First protocol configurations for video transfer (including a JPEG compression HW module) are running on SPARC Station 10/20 over ethernet. Our current work is to verify the overhead introduced by the unified interface and the performance of the run-time environment. Obviously, both aspects depend on packet size and granularity of modules. Furthermore, we intend to study the mechanisms and the performance of protocol reconfiguration. In this context, the problem of monitoring and guaranteeing QoS parameters, which is up to now only insufficiently solved [18], is of major interest.

A major advantage of the free configuration of modules is the possibility to compare the performance of single protocol mechanisms in the same environment. Of particularly interest is the extension of the prototype by a user-interface that students should be provided with a graphical interface to design protocol graphs, select modules and to test interactively their configuration.

Acknowledgments

A major part of the implementation was performed as students works and diploma thesis. Thanks are due to Chukuwma Ebo, Andreas Gotti and Marcel Dasen, Roman Graf, Alireza Oloumi, Hasan, Peter Imhof and Thomas Ward.

References

[1] Haas, Z.: "A Communication Architecture for High-speed Networking", in: Proc. of IEEE INFOCOMM '90, 9th Annual Joint Conf. of the IEEE Computer and Communications Societies, Los Almos, California; Vol. 2, June 1990, pp. 433-441.

[2] Doeringer, W.A., Dykeman, D., Kaiserswerth, M., Meister, B.W., Rudin, H., Williamson, R.: "A Survey of Light-Weight Transport Protocols for High-Speed Networks", in: IEEE Transactions on Communications; Vol. 38, No.11, Nov.1990, pp. 2025-2039.

[3] Plagemann, T., Plattner, B., Vogt, M., Walter, T., "A Model for Dynamic Configuration of Light-Weight Protocols", in: Proc. IEEE Third Workshop on Future Trends of Distributed Computing Systems, Taipei, Taiwan, April 1992, pp. 100-106

[4] Tschudin, C.: "Flexible Protocol Stacks", in: SIGCOMM '91 Conf., Communications Architectures & Protocols, Zurich, Switzerland, Computer Communications Review; Vol. 21, No. 4, Sept.1991, pp. 197-204.

[5] Oloumi, A.: "Configuration of Light-Weight Protocols Algorithm", Diploma Thesis at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, Sept. 1992.

[6] Hasan: "Implementation of ISO CLNP in Da CaPo"(in German), Students Work at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, March 1993

[7] Imhof, P.: "Connection Management in Da CaPo" (in German), Diploma Thesis at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, March 1993

[8] Gotti, A., Dasen, M.: "Resource Management in Da CaPo" (in German), Students Work at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, March 1993

[9] Hutchinson,N. C., Peterson, L. L.: "The x-Kernel: An architecture for implementing network protocols" in: IEEE Transactions on Software Engineering, Jan. 1991, pp. 64-76

[10] Harris, D.: "Streams", Kochanan, S.G., Wood, P.H. (Editors): "UNIX Networking", pp. 133-170

[11] Ward, T.: "Modules for a File-Transfer-Service in Da CaPo" (in German), Diploma Thesis at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, March 1993

[12] Graf, R. : "T-Modules for Da CaPo".(in German), Students Work at Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, March 1993

[13] Vogt, M., Plagemann, T., Plattner, B., Walter, T.: "A Run-time Environment for Da CaPo", in: Proc. of INET'93, Int. Networking Conf. Internet Society, San Francisco, August 1993

[14] Olander, D.J., McGrath, G.J., Israel, R.K.: "A Framework for Networking in System V", in: Proc. USENIX 1986 Summer Conf., Atlanta, June 1986, pp. 38-45

[15] O'Malley, S.W., Peterson, L.L.: "A Dynamic Network Architecture", in: ACM Transactions on Computer Systems, Vol. 10, No. 2, May 1992, pp. 110-143

[16] Box, D. F. ,Schmidt, D. C., Tatsuya, S.: "ADAPTIVE An Object-Oriented Framework for Flexible and Adaptive Communication Protocols", in: Proc. hpn92, 4th IFIP Conf. on high performance networking, Dec. 1992.

[17] Zitterbart, M., Stiller, B., Tantaway, A.M.: "Application-Driven Flexible Protocol Configuration", in: Proc. Communication in Distributed Systems (in German), KIVS'93, Springer Verlag, Jan. 1993, pp. 384-398

[18] Kurose, J. : "Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks", in: Computer Communication Review, Vol. 23, No. 1, Jan., 1993