A Run-time Environment for Da CaPo

Martin Vogt[1]
Bernhard Plattner[2]
Thomas Plagemann[3]
Thomas Walter[4]

Abstract

*It is a well known fact that the performance bottleneck of modern high speed networks is located in the end systems. Reasons are the complexity of the applied protocols and the embedding of protocols in the end systems. Da CaPo (**D**yn**a**mic **C**onfigur**a**tion of **P**r**o**tocols) provides an environment which allows the dynamic configuration of protocols with respect to the application requirements and the properties of the offered network services. The goal of the configuration is to reduce the protocol complexity and increase the performance. Modules serve as building blocks for the protocol configuration. Common software engineering principles like encapsulation and information hiding as well as a unified module interface allow the unrestricted configuration of modules to protocols. The runtime environment of Da CaPo links modules to protocols in one UNIX process and realizes an efficient data transport inside the end system because operations reducing performance like data copying or UNIX context switches are minimized in Da CaPo.*

I. Introduction

Modern high speed networks such as optical LAN's or Metropolitan Area Networks using technologies based on fast packet switching or fast circuit switching offer magnitudes more of bandwidth than traditional networks. This plentiful amount of bandwidth has enabled the design and implementation of new distributed applications handling multi-media information. Most of the high-speed networks offer end-to-end connectivity like ATM. In most cases the offered service and quality of service do not directly meet the requirements of the distributed applications. The network-services must be enriched by additional functionality as it is found in transport protocols or facilities for synchronisation and presentation coding. Most end-to-end protocols offering such functionality result in the performance bottleneck in high-speed communications. Reasons are the insufficient processing power of the end-systems, the missing tailoring options in the end-system protocol and the embedding of the communication subsystem in the operating system.

The aim of the Da CaPo (**D**yn**a**mic **C**onfigur**a**tion of **P**r**o**tocols) project at ETH Zurich is to overcome this communication speed bottleneck by configuring end-system protocols. An optimally adapted protocol is configured depending on the offered network-services, the requirements of the application and the processing power available. This protocol represents a light-weight protocol, as no unnecessary protocol functions are included [1].

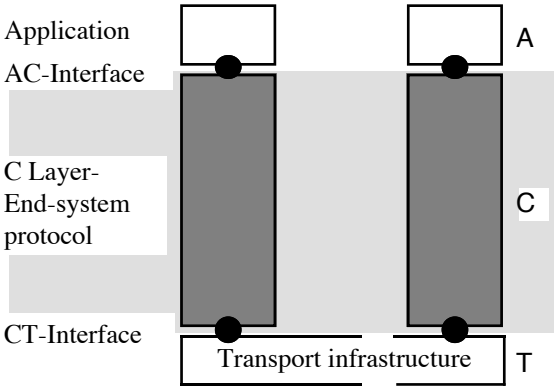

Figure 1: The Da CaPo 3 layer model

The main idea of Da CaPo is to subdivide the communication subsystem into three layers. The lowest layer of the three layer model (Fig. 1), the transport infrastructure (T layer), connects end-systems. The service of this layer (T service) is different compared with the OSI transport service. The T service is generic as it might be the service of a MAC layer of a MAN such as DQDB, an ATM adaptation layer or the service of a transport protocol like TCP or XTP. The T service generally has to be enhanced within the C layer (by a configured protocol) to allow applications (located in the A layer) to cooperate.

Various authors have proposed similar approaches to overcome the performance bottleneck in the end-systems. Haas [2] describes an

1 Mr. Vogt is with the Swiss Federal Institute of Technology Zurich. He may be reached at vogt@komsys.tik.ethz.ch

2 Prof Plattner is with the Swiss Federal Institute of Technology Zurich. He may be reached at plattner@komsys.tik.ethz.ch

3 Mr. Plagemann is with the Swiss Federal Institute of Technology Zurich. He may be reached at plagemann@komsys.tik.ethz.ch

4 Mr. Walter is with the Swiss Federal Institute of Technology Zurich. He may be reached at walter@komsys.tik.ethz.ch

architecture consisting of the same three layers and a horizontally oriented protocol for high speed communication (HOPS) built from simple protocol functions. Clark and Tennenhouse [3] investigate "principles to structure a new generation of protocols" and introduce "Integrated Layer Processing" as a protocol engineering principle that allows the implementor to perform all data manipulations in one step (or in several parallel steps) instead of performing them sequentially. O'Malley and Peterson [4] address the issue of constructing a protocol entity from a set of micro-protocols (the equivalent of protocol functions) using the notion of a virtual protocol. The object-oriented ADAPTIVE system [5] allows protocol configuration and reconfiguration guided by classes of protocol mechanisms. F-CSS [6] is a function-based model for a communication subsystem and supports the application-driven configuration of efficient protocol machines tailored to the needs of the application. Protocol mechanisms (realising protocol functions) are used as building blocks for the configuration. F-CSS supports 4 predefined service classes: class I unreliable real time services (e.g. for voice and video), class II reliable real time services (e.g. for process-control applications), class III unreliable non-real time services (e.g. for simple text and graphic transmissions) and class IV reliable non-real time services. Finally, selection of one of several protocols according to the requirements of the application and the service provided by the underlying layer is used in the ISO standard [7].

Protocol configuration aims at the performance aspect. The 'lighter' protocols are the less resources (mainly CPU time) are used. Besides the benefit of performance, protocol configuration systems encourage the implementation of reusable building blocks and may be used as flexible test- and measurement-environments.

In this paper we concentrate on dynamic construction and the execution of protocols as they are realized in Da CaPo. The next section gives an overview on Da CaPo, its foundation and its implementation. Section III of the paper discusses the concepts of the resource manager, the way modules are combined to protocols and the data flow inside a protocol. Section IV describes the current implementation on a single processor machine. Our contribution finishes with some conclusions in section V.

II. An Overview of Da CaPo

The model for dynamic configuration of light-weight protocols [8] represents the foundation of Da CaPo. The realization of this model is characterized by three co-operating active entities to configure modules to well appropriate protocols and execute them:

- the configuration process determines the necessary modules,
- the connection manager negotiates the dynamic protocol and
- the resource manager provides the runtime environment for configured protocols.

A database stores information of general interest. Mainly the configuration process and the resource manager are handling this information.

II.A. The Three Layer Model

Da CaPo is based on a three layer model which splits communication systems into the layers A, C and T (Fig. 1). End systems communicate with another via layer T, the transport infrastructure. The transport infrastructure represents the existing and connected communications infrastructure. We name their services T services.[5] In layer C the end-to-end communication support adds functionality to the T services in such a manner that at the AC-interface a full set of services is provided as needed to support distributed applications (layer A).

Revising layer C services in a top-down manner we say C services are decomposed into a set of protocol functions according to their functionality. Protocol functions are defined by their semantics. Each protocol function encapsulates a typical protocol task like error control, flow control, de/encryption, presentation coding, etc. Protocol functions can be implemented 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 function are characterized by different properties. This may be different throughput figures or different degrees of error correction or detection. For instance the protocol function 'error control' could be implemented by a single parity bit or by the CCITT CRC algorithm. While the computation of the parity bit is very simple and fast, the code detects only single bit errors within a byte. This contrasts to the CCITT CRC which is computationally costly (i.e. exhibits a reduced throughput), but has much better error detection capabilities.

Each C service is composed of a set of protocol functions on top of a T service. Data dependencies between protocol functions define a partial order on the protocol functions and are specified in a protocol graph (Fig. 2). A protocol graph can be considered as a more abstract protocol specification.

If multiple T services may be used, there is one functional decomposition (i.e. one protocol graph) for each T service. To create a protocol entity each protocol function must be instantiated by one of its modules. The configuration process selects the most suitable module to configure the best fitting protocol at connection establishment time.

---

5 Tschudin [9] calls T services "anchored instances".

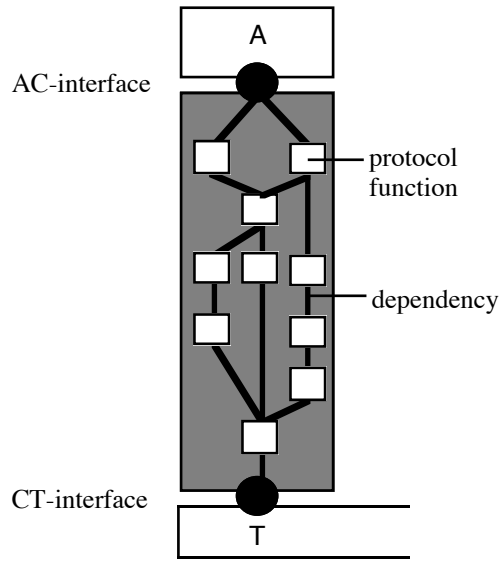

Figure 2: Example of a protocol graph

II.B. The Configuration Process (CP)

The task of the configuration process (CP) is to select the most suitable modules depending on the application requirements and the available T services [10]. The CP retrieves the functional decomposition of the invoked C service from the database and calculates the properties of possible protocol configurations. The instantiation of each protocol function in a protocol graph with one of the modules creates one possible configuration. The CP starts with the properties of the T service and calculates step by step according to the protocol graph the influence of the selected modules on the offered service. The influence of modules on a service (i.e. module properties) is stored in the database. We use one common syntax to describe the properties of modules, T services, protocols and application requirements. We called this language simply L [8]. All descriptions consist of tuples of attribute types and functions (mostly constants). The application requirements additionally include a weight function which defines the relative importance of an attribute. This extension enables us to deal with the wide range of complex application requirements, mainly introduced by new multimedia applications and to formulate contradictory requirements. Fig. 3 shows the BNF definition of L.

The unified representation enables a direct comparison of application requirements and protocol properties. If a configuration fulfils the application requirements we say the protocol is in compliance with the application requirements. The protocol with the highest compliance degree will be the best protocol which can be configured according to the application requirements.

RequirementsOrProperties ::=
ApplicationRequirements | ModuleProperties | TransportProperties

ApplicationRequirements ::=
"<" ApplicationRequirement ("," ApplicationRequirements |) ">" |

ModuleProperties ::=
"<" ModuleProperty ("," ModuleProperties |) ">"

TransportProperties ::=
"<" TransportProperty ("," TransportProperties |) ">"

ApplicationRequirement ::=
"<" ARType "," (ARValue | "*") "," Weight ">"

ModuleProperty ::=
"<" ARType "," Function ">"

TransportProperty ::=
"<" ARType "," ARValue ">"

ARType ::=
(* an enumeration of application requirements *)

ARValue ::=
(* a denotation of a value *)

Weight ::=
(* an enumeration of functions *)

Function ::=
(* an enumeration of functions *)

Figure 3: BNF definition of L

The necessary information for the CP like properties of modules and T services, functional decompositions and protocol graphs are stored in a local database. In the case of a local configuration, only the information in the local database is considered. In the case of a global configuration the CP considers additionally the properties of the modules and T services of the peer system and selects the globally best protocol. The connection manager is responsible for the information exchange in the case of a global configuration.

Protocol specifications not only describe the functionality, but also the packet structure. The CP calculates the structure of the packet header for the selected protocol configuration. The module properties in the database include the number and length of header fields needed by the module. The CP collects this information from all modules of the protocol according to the module graph from top to bottom and defines the header structure for the current configuration. The CP delivers the description of the configured protocol as a module graph and a header description to the connection manager.

II.C. The Connection Manager (CM)

The connection manager (CM) assures that communicating peers are using the same protocol for a new layer C connection. To achieve this the CM has the task to organize the

- establishment of connections,
- reconfiguration of existing connections,
- exploitment of connections and

- the handling of errors which violate the application requirements.

During the connection establishment phase and the reconfiguration phases the CM negotiates with his peer a common configuration. The CM communicates with his peer via a fixed protocol called management support protocol. If the T service offers a reliable service the management support protocol is empty. For unreliable T services the management support protocol consists of a set of modules offering a reliable service. The CM negotiates the protocol configuration for the application using the management support protocol. Figure 4 illustrates the structure of a layer C connection consisting of two protocols, the management support protocol and the application protocol. The CM supports three different negotiation scenarios, selectable from the application by the application requirements:

- In the case of a local configuration the CM informs the peer CM about the selected protocol and both CM's establish this protocol.
- In the global configuration the CM initiates a local configuration in both end systems, one CM sends a list of the n best results to his peer who compares these results with his local results and selects the best one. Afterwards he informs his peer about the selected protocol and both peers establish this protocol.
- In the combined configuration the communication between both applications starts with a pre-determined protocol. While the applications exchange information via the pre-determined protocol both CM perform a global configuration.

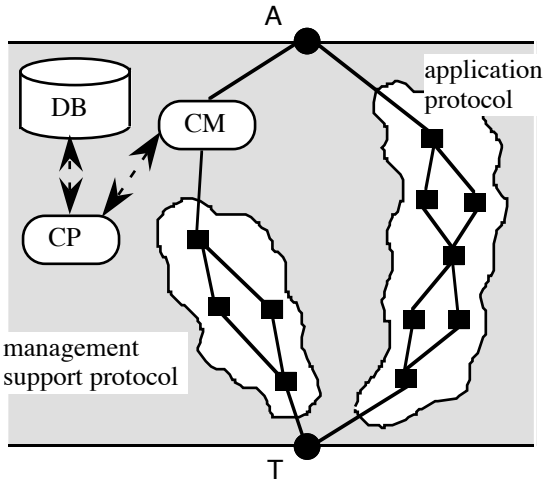

Figure 4: Structure of a Da CaPo protocol

After the successful negotiation of a configuration for the application protocol both CM initiate their resource manager to establish the selected protocol. In the combined configuration the pre-determined application protocol will be substituted by the result of the global configuration. This corresponds to a reconfiguration. Reasons to reconfigure an established connection are changes in the properties of the system or the protocol offending the application requirements. The resource manager detects the changes and requests a reconfiguration from the CM. Established connections can be reconfigured either with loss of data or without loss of data but with additional delay. The application has the possibility to select one of these possibilities by specifying it in the application requirements.

II.D. The Modules

Protocol functions describe the different tasks executed by protocols, e.g. error correction or flow control. A protocol function may be implemented by different protocol mechanisms. A protocol mechanism consists of a part sending data and the corresponding receiving part, e.g. the protocol mechanism CRC will calculate the CRC value on the sender and check it on the receiver. All protocol mechanisms are realised by two modules in hard- or software. As these modules are unidirectional, a protocol configured by modules supports data flow in one direction.

To fulfill their task, many modules have to communicate with their peer entity on the other system in the opposite direction to the data flow. We call the direction defined by the flow of the application data the *main direction* and the opposite direction needed for control information *back direction*. The 'Idle Repeat Request' module receiving data has to confirm each data packet to his peer module at the sender. On the other hand the sending 'Idle Repeat Request' module may process the next packet only after the reception of the confirmation from the receiving module.

All modules are encapsulated to enable the unconstrained configuration. They offer a unified interface [11]. This procedural module interface is used by the resource manager. It consists of the following procedures:

- `request` and `indication`, to pass data to the module and receive data packets from the module after processing.
- `request_back` and `indication_back` to pass and receive control packets, which flow in the opposite direction to the data flow.
- `statistic`, to get statistical data out of the module, e.g. the number of recognised errors or the number of retransmissions.
- `init`, to instantiate and initialise a module, e.g. reserve buffer space, and to pass the structure of packets.
- `exit`, to free all buffer space used by the module and to remove the module.

Modules processing data only in main direction offer only the `request` and `indication` procedures, modules which are needed only in the back direction `request_back` and `indication_back`. Fig. 5 shows a simple module graph with main and back direction.

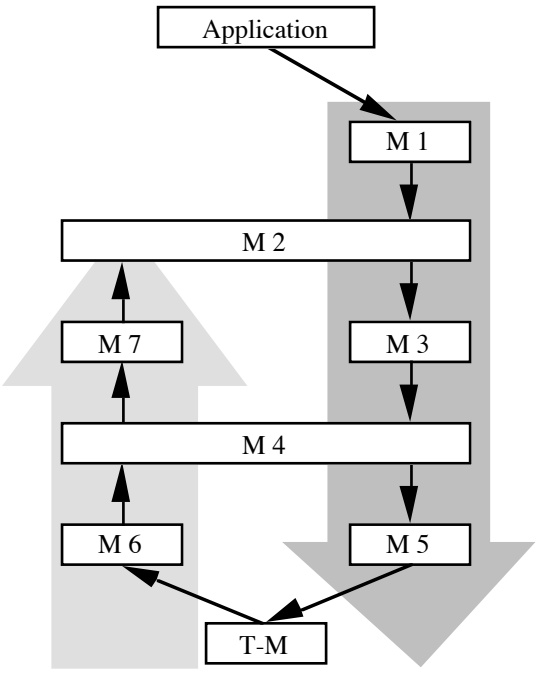

Figure 5: Main- and Back direction

T modules have a special significance. They serve to abstract from the T services. They are linked to the protocols by the same interface as all the other modules. Their processing accesses the T service by its generic interface. With a similar abstraction applications are bound as A modules into the protocol. A modules are part of the application, but offer also the standard module interface and are linked to the protocol.

II.E. The Database

All information necessary for the configuration of protocols is collected in a database. This includes the functional decomposition of the C services and the protocol graphs, as well as dynamic and static properties of modules and T services. The CP relies on this database to configure a protocol which corresponds to the application requirements. The resource manager updates the database by collecting the statistical data from the modules and changing the description of their properties in the database.

III. Resource Management in Da CaPo

The resource manager (RM) [12] implements the runtime environment for configurable protocols and coordinates the work of the different Da CaPo components. The tasks of the RM are to

- load, link and initialize modules according to the module graph,
- release modules and resources after connection termination,
- guide packets through the protocols,
- monitor the properties of system resources, T services, modules and protocols, and to
- initiate a reconfiguration if necessary.

This chapter shows the concepts used by the RM to solve these tasks. The implementation on a single processor architecture is presented in chapter IV.

III.A. Protocol Instantiation and De-instantiation

The CM asks the RM to build protocols and passes the description of the module graph and the structure of the packets to the RM. The RM fills a data structure, the *protocol context*, which comprises information for all modules of the protocol. This includes the packet structure determined by the CP and information to adapt the module to a particular configuration, e.g. the maximum transfer units of the T service etc. The RM passes this data structure to all modules with the `init` procedure. The protocol is constructed by calling all the `init` procedures of all modules. The calling sequence is determined by the module graph. First, the T module is called, followed by its successors in the module graph.

A module gathers information for the initialisation from the protocol context. It keeps the necessary *instance information* in newly allocated buffer space, as several instances of the same module might be needed in a single protocol graph. As soon as all modules have been initialised, the protocol is established.

Similarly the CM initiates the RM to de-instantiate a protocol. The RM calls the `exit` procedure of all modules according to the module graph 'top-down' one after the other. This causes the modules to free all their resources.

III.B. Packet Forwarding

We now describe the control of the packet forwarding for unidirectional services. The protocol moves data from a sending to a receiving application. Bidirectional services are realised in Da CaPo by two unidirectional protocols.

It is the task of the RM on the sending system to pass the application data through the module graph to the T module. The T service transmits submitted packets to the receiving system. The RM of the receiver now has to hand over the packet step by step to the application. Packets to be processed are passed to modules by calling the `request` procedures. Through the procedure `indication` the RM fetches the processed packet. The application offers the same procedures to submit and

receive data, that means the packet forwarding does not have to distinguish between protocol modules and application. Fig. 6 shows how packets on the sending side are passed from the application to the T module.

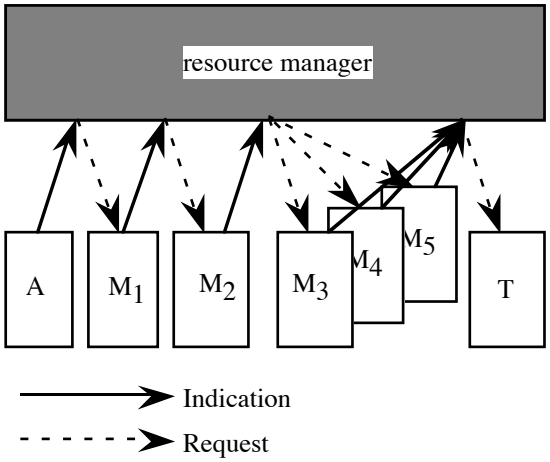

Figure 6: Packet forwarding in a protocol

The example in Fig. 6 illustrates that Module 3, 4 and 5 can be executed in parallel (if the system has several processors or specialised hardware). The RM synchronises the transmission of the packet to the T module.

The data flow for main and back direction cannot be handled separately within a single protocol. Therefore the packet forwarding has to know about available data packets at modules, the application and the T module. Every module returns with the call of `request`, `indication`, `request_back` or `indication_back` whether it has a packet to forward in the main direction or needs to send / receive control information. Control packets have always priority and are passed immediately through the whole graph to avoid deadlocks. At the sending system the T module consumes packets, at a receiving system packets are consumed by the A module. New packets generated by the application or the T service are reported to the RM to be moved through the module graph. Section IV.C shows the implementation of the lift algorithm specific to a UNIX single processor system.

III.C. Monitoring of System Behaviour

Da CaPo guarantees either to maintain the application requirements or to abort the connection if this is impossible. This may be achieved only by the RM monitoring all relevant communication parameters regularly. We distinguish four classes of values:

- Values describing the state of the end system. These are identical for all Da CaPo applications.
- Values collected directly by the RM.
- Values delivered by modules.
- Derived values.

Some values are determined directly by the monitor, e.g. throughput or total system load, but most of them are collected via the statistic interface of the modules. The fourth class comprises values which may not be observed directly and have to be derived from other values. For instance the residual error rate may be estimated from the number of errors found and the well known properties of the error correction algorithm.

IV. Implementation of the Resource Manager

We have developped the first prototype of Da CaPo on a Sun 10/20 running UNIX (SunOS 4.1.3). We pay special attention to the integration into the operating system. So we can avoid influences with adverse effects on throughput which show up on process switches, buffer management or data copying [1]. Some simplification compared to the previous chapter results from the implementation on a single processor system, as there is no need for synchronisation of the modules.

IV.A. Mapping of Da CaPo Processes to UNIX Processes

Each Da CaPo application including all configured protocols is mapped onto a single UNIX process. This contrasts to other authors which propose a process per packet [4]. Da CaPo itself needs no process switches. This is achieved by linking the whole code of Da CaPo including all available modules to every application. Except for information needed by different Da CaPo processes on the same system (stored in the database) different Da CaPo applications on the same end system are independent.

Da CaPo itself distinguishes the three states WORK, LIFT and MONITOR. It is the responsibility of the resource manager to initiate the transitions between these states. The state WORK comprises all tasks which are not related to data flow and monitoring. This means that application, CM, CP and parts of the RM itself execute in this state. It is the basic state of an application, from which transitions to the other states may occur. The RM changes the state to LIFT as soon as a packet is available. This is the state with the highest priority. The process executes the lift algorithm until no module has a packet ready. Afterwards the state machine switches back to the basic state. The state MONITOR serves to monitor the properties of protocols and to collect statistical data. This transition is periodically triggered by a timer.

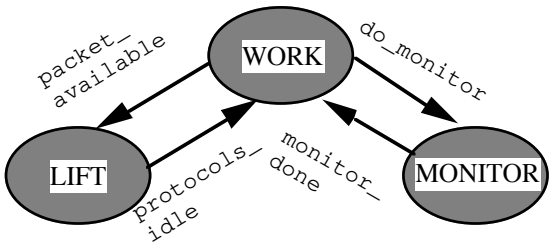

Figure 7: Da CaPo states

The transition to the state LIFT may be triggered by a procedure call, a signal or a timer. It is always handled by the procedure `'packet_available'`. This procedure guarantees that no additional transition takes place and all packets are treated at return. If an application is ready to send a packet, it directly calls this procedure. When a packet arrives from the network, the T module sends a signal. The signal handler then calls the procedure `'packet_available'`. Modules, which require data transmission at specific times, e.g. to sustain a given throughput or to request a retransmission, trigger the transition as soon as the corresponding timer expires.

IV.B. Timer and Signal Management

The implementation of protocols in one UNIX process requires a dedicated timer handling. UNIX supports only one timer per process which is not sufficient when working with a protocol. The resource manager is responsible for the timer management. He keeps an ordered list of installed timers. Modules install and start timers with a procedure call including the context information of the module and its instance identification. The resource manager uses internally the systemcall `ualarm`. If the timer expires the resource manager reads the first instance identification in the list and is able to call the appropriate module. A similar mechanism is necessary to propagate signals to the modules. But as there is no possibility to discriminate the receiving module of a signal, a broadcast mechanism is used. If a module catches a signal, it first checks whether it has to take any action.

IV.C. Lift Algorithm

The module graph in the data direction has a structure with arbitrary branches between the endpoints A module and T module. As all the modules have to be executed sequentially on a single processor system, the RM first orders the graph and thereby creates a sequential list of modules. The first module in the list is the one which puts new packets into the protocol. On the sending system this is the A module. Only this module delivers through its indication interface new data packets. Via the request interface these packets are passed on to the next module in the list. This module executes the necessary work. Afterwards the packet is again available at its indication interface. At the other end of the list the packet is passed to a T module. A packet reaching this module has been completely processed and is handed over to the T service. If another packet is available at the A module, this packet is processed, otherwise the algorithm terminates. As with a lift data packets are fetched at one end, processed at a stop at each floor and unloaded at the other end. The head module on a receiving system is a T module, which receives the packets from the network, as tail serves the A module.

Some modules create several packets from a single packet passed to them, e.g. the segmentation module. A module implementing such a function sets its status accordingly with the return of a partial packet. As soon as such a partial packet reaches the end point of the list, the RM fetches the next packet from this module (which is not at the beginning of the list) and guides it to the end. The counterparts are modules, which combine several smaller packets to a bigger one (reassembly). These modules set the status of `indication` to indicate that they are not yet ready to deliver a packet, but need more input to do so. In this case, the algorithm restarts at the head of the list and delivers as many packets as necessary to build the bigger one. If both these kinds of modules are included within the same graph, the algorithm oscillates between them before it eventually reaches one of the endpoints.

As modules may generate packets based on some timer, the algorithm is even more complex. These modules install a timer with the RM. Thus the RM is able to detect a timer going off, which is a hint that a packet might be available at this module. In this case the lift algorithm starts once at this module and not at one of the endpoints. If the procedure indication delivers a packet, the lift algorithm will move it on to its destination.

Each module may request (by its status value) a single pass of the lift algorithm through the control graph. As soon as this request is posted, no further action takes place within the data graph, but a complete pass through the control graph is executed. The procedures `request_back` and `indication_back` guarantee that this pass will not block. If the module list starts with a T module, data and control path are blocked until a packet becomes available from the network.

IV.D. Buffer Management

Modules producing packets like sending A modules and receiving T modules must allocate buffer space for new packets if no preallocated and unused buffer space is available. Succeeding modules access the packet via a pointer to avoid unnecessary copy operations. The packet is split into the header part and the data part. The header part has a fixed structure and size calculated by the configuration manager. This allows a higher degree of parallel modules. One module which modifies the data part can run in parallel with multiple modules

which modify their header fields. Some modules just insert a fixed value into their header field, such as the address of the peer machine. Da CaPo takes advantage of this property and uses a template. The modules enter the constant value during initialisation, afterwards they are safely removed from the list of modules.

To optimize the allocation and deallocation of buffer space for packets we implemented a specialized buffer management. The allocation operation of the buffer management initiates only an UNIX `m_alloc` if there is no preallocated or unused buffer available. The deallocate operation decrements a reference counter for the buffer. The value zero of this reference counter indicates an unused buffer. As usual, modules which allocate buffer space are responsible to deallocate it afterwards. Modules which keep packets internally like the sending Idle Repeat Request module have to increment the reference counter and decrement the counter if they do not need the packet any more (the sending IRR module received an acknowledgement).

This simple buffer management is not sufficient in cases where a packet is segmented. A segmenting module delivers pointers which reference parts of the original buffer. The original buffer can only be deallocated if all these segments are unused, while a single segment never will be deallocated. The buffer management takes these dependencies into account with a reference from the segments to the original packet. The creation of a segment causes an increment of the reference counter of the original buffer, while 'deallocation' of a segment decrements it.

V. Conclusions

The architecture of Da CaPo aims at the dynamic configuration and reconfiguration of protocols. The preceding sections give an overview on the main parts of Da CaPo and their interworking. We introduced a unified module interface which allows free combinations of modules to protocols. Even standard protocols such as the ISO CLNP may be configured [15]. The module design allows the simple and fast integration into the runtime environment, which minimizes performance reducing operations.

All major parts of Da CaPo are specified and implemented. The integration of all parts has been finished by end of May. We intend to verify the efficiency of the Da CaPo approach with this prototype. The verification includes the examination of the unified module interface and the overhead introduced by the resource manager. Furthermore we intend to study the mechanisms and the performance of protocol reconfiguration. In this context the problem of monitoring and guaranteeing service quality which is up to now only insufficiently solved [13] is of major interest. The statistic interface of the modules allows a simple integration of new solutions.

A major advantage of the free configuration of modules is the possibility to compare the performance of single protocol mechanisms in the same environment. On the other hand we intend to use this possibility for a teaching system. Students should be provided with a graphical interface to select modules and configure them in a module graph [14].

VI. Acknowledgements

At this place it should be mentioned that a major part of the work for designing and implementing parts of Da CaPo was performed as student work and diploma thesis. Particularly we thank Chukuwma Ebo, Andreas Gotti and Marcel Dasen , Roman Graf Alireza Oloumi, Hasan, Peter Imhof and Thomas Ward for their contributions.

VII. References

[1] 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; Volume 38, Number 11, Nov. 1990, pp. 2025-2039.

[2] Haas, Z.: "A Communication Architecture for High-speed Networking", in: Proceedings of IEEE INFOCOMM '90, Ninth Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE Computer Society Press, Los Almos, California; Volume 2, Jun. 1990, pp. 433-441.

[3] Clark, D.D., Tennenhouse, D.L.: "Architectural Considerations for a New Generation of Protocols", in: ACM SIGCOMM Computer Communication Review; Volume 20, Number 4, Sep. 1990, pp. 200-208.

[4] O'Malley, S.W., Peterson, L.L.: "A Highly Layered Architecture for High-Speed Networks", in: Protocols for High-Speed Networks, II, Majory J. Johnston (Editor), Elsvier Science Publishers B.V. (North-Holland), 1991, pp. 141-156.

[5] Box, D. F., Schmidt D. C., Suda, T.: "ADAPTIVE An Object-Oriented Framework for Flexible and Adaptive Communication Protocols", Proceedings hpn92, 4th IFIP conference on high performance networking, Dezember 1992

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

[7] ISO 8072, CCITT X.214 "Transport Service Definition".

[8] Plagemann, T., Plattner, B., Vogt, M., Walter, T., "A Model for Dynamic Configuration of Light-Weight Protocols", IEEE Third Workshop on Future Trends of Distributed Computing Systems, 1992

[9] Tschudin, C.: "Flexible Protocol Stacks", in: SIGCOMM '91 Conference, Communications Architectures & Protocols, Zürich, Switzerland, Computer Communications Review; Volume 21, Number 4, Sep. 1991, pp. 197-204.

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

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

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

[13] 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.

[14] Ebo, C. "An X Window based Management Tool for Da CaPo" (in German), Students Work at Laboratory of Computer Engineering and Networks, Swiss Federal Institute of Technology Zurich, Feb. 1993

[15] Hasan: "Implementation of ISO CLNP in Da CaPo" (in German), Diploma Thesis at Laboratory of Computer Engineering and Networks, Swiss Federal Institute of Technology Zürich, Feb., 1993

Author Information

Mr. Vogt joined the Computer Engineering and Networks Laboratory in February 1990, where he works towards his Ph. D. and is responsible for the administration of the computer systems. He received his Diploma Degree in computer sience from the Swiss Federal Institute of Technology Zurich in 1988. Thereafter he was employed for two years with a counsulting company. He is mainly interested in protocol architectures, multimedia communication, high speed protocols and embedding of protocols into the operating system.

Dr. Plattner, born 1950 in Bern, Switzerland, is a Professor of Computer Engineering at ETH Zurich, where he leads a communication systems research group. He received a diploma in electrical engineering from ETH in 1975 and a Ph.D. in Computer Science in 1983. His research currently focuses on applications of communication systems, higher layer protocols and high-speed networking. He is also interested in real-time computing, process execution monitoring and debugging. He has been active in the design and the implementation of the Swiss National Network for Research and Education (SWITCH).

Dr. Plattner is a co-author of a successful book on message handling and X.400, of which the first and second edition were published 1989 and 1990, respectively. He has given numerous seminars on data communications and computer networking.

Dr. Plattner is a member of the RARE Technical Committee, which oversees the technical development of academic networking in Europe.

Mr. Plagemann received his Diploma Degree in computer science from the University of Erlangen-Nürnberg, Germany in 1990. He joined the Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, in 1990, where he is currently working towards his Ph. D. His interests include user interfaces for communication services, multimedia communication, high-speed protocols, and protocol architectures for high speed networks.

Mr. Walter joined the Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, in 1986, where he is currently working towards his Doctor degree. He received his Diploma Degree in 1987 from the University of Hamburg, Germany. His major interests are in distributed systems, formal description techniques and conformance testing, and protocol architectures for high speed networks. Mr. Walter is member of Gesellschaft fuer Informatik (FRG) and ACM.