

The DiPS+ Software Architecture for Self-healing Protocol Stacks

Sam Michiels, Lieven Desmet, Wouter Joosen, Pierre Verbaeten
DistriNet research group, Department of Computer Science,
K.U.Leuven, B-3001 Leuven, Belgium,
sam.michiels@cs.kuleuven.ac.be

Abstract

Research domains such as active networks, ad-hoc networks, ubiquitous computing, pervasive computing, grid computing, and sensor networks, clearly show that computer networks will become more complex and heterogeneous. In many cases, central management and control of the network are far from trivial since both the topology and the connected devices change rapidly in such highly dynamic environments, while load circumstances may vary arbitrarily. The software architecture in a node needs to support flexibility. We have developed an architecture tailored to protocol stack software that allows customizing internal resource management in order to handle overload conditions gracefully. We show that the investment in explicit support for modularity and architectural constraints pays off: the paper elaborates on a case study in which dynamic adaptation of access control behavior leads to significant performance improvements.

1. Introduction

This paper describes the role of software architecture in the life cycle of self-healing protocol stacks. A self-healing protocol stack is able to handle problem conditions (such as overload) gracefully, without manual intervention of an administrator. We propose an architecture tailored to the domain of network software, and show its advantages in an industrial case study. This case study implements the RADIUS authentication and accounting protocol, and customizes it to allow for load management based on various application-specific requirements.

Protocol stack software has specific characteristics and complex requirements, both for functionality and concurrency. In addition, there is a trend towards self-healing networks in which network nodes control and manage themselves. Ad-hoc networks, for instance, lack a fixed network infrastructure and allow for devices to connect and disconnect dynamically, without reconfiguration. For this reason,

central management and control of ad-hoc networks is far from trivial. An ad-hoc network creates a highly dynamic and heterogeneous environment, which requires extensive software support in the devices attached.

The proposed software architecture combines well-known architectural styles in such a manner that designing protocol stacks becomes straightforward, and forces programmers to develop adaptable and testable software [12]. Next to self-management, we have validated the architecture in various research domains, such as testing, optimization, and component hot-swapping [12, 7].

The strength of our approach is that these research tracks are not proposed as individual solutions, but share a well-defined software architecture [17, 3] underneath. In addition, we have developed a component framework [16], which is implemented as a Java tool that allows to compose different configurations and strategies without having to change the source code of the components involved [12], and to build protocol stacks automatically from an architecture description. The resulting prototype is called DiPS+ (DistriNet Protocol Stack). The DiPS+ architecture enables to apply the four proposed solutions not only at software design-time, but throughout the complete software life-cycle, i.e. also at build-, run- and management-time.

This paper is structured as follows. Section 2 presents the DiPS+ software architecture, concerning both the data and the management plane. It also shows how concurrency is modeled separate from functional code. Section 4 explains one of the DiPS+ application domains, load management, which provides the basis for the case study in Section 5. Section 6 compares our approach to related work. The paper is summarized in Section 7.

2. The DiPS+ architecture

2.1. Data and management plane

Taken as a whole, the DiPS+ architecture represents data processing and management as two planes on top of each other. The software architecture of each plane combines

multiple architectural styles such as the pipe-and-filter, the blackboard and, again, the layered style [17]. Loose coupling of components is enforced by the combination of anonymous invocations – via the pipe-and-filter style – and anonymous state sharing – through the blackboard style of communication. Moreover, since extra components can be added transparently, and since state information is usually not kept in the components locally but is attached to each individual packet that flows through the protocol stack, this approach allows to easily add concurrency (parallelism) to the component pipeline in a transparent manner. Moreover, separating concurrency from functional components enables to customize how processing resources (threads) are spread throughout the system in order to optimize the overall throughput.

The data plane in the DiPS+ architecture houses the functional part of the system, i.e. the protocol stack. Without going into detail, this plane identifies components and how they are connected on the one hand, and offers layers as a composition of basic components on the other hand.

On top of the data plane, DiPS+ offers the management plane, which acts as a meta-level to extract information from the data plane and control its behavior. Two main tasks have been identified at this level: message filtering and concurrency control. Message filtering retrieves and collects information from the data plane by intercepting component communication. Concurrency control for its part involves optimally exploiting processing resources, and customizing their scheduling. This enables an application or a system administrator to prioritize the processing of particular tasks or requests in order to optimally apply the available processing resources.

2.2. Inside the data plane

When taking a closer look at the architecture of the data plane, we can identify three main architectural styles – the pipe-and-filter, the blackboard, and the layered style.

2.2.1. Pipe-and-filter style The DiPS+ data plane architecture can be compared with the assembly line in a factory. Such a system consists of a collection of robots interconnected via a conveyor belt, which transports products from one robot to the next one.

The pipe-and-filter style is very convenient for developing various types of system software such as protocol stacks, file systems, compilers, and web services. A protocol stack can be thought of as a downgoing and an upgoing packet flow. The downward flow of outgoing packets identifies components for header construction, routing, packet fragmenting and serialization. The incoming packet flow involves processing steps such as de-serialization, packet forwarding and reassembly, and

header parsing. File systems for their part identify a similar flow of request parsing, cache lookups, device driver request construction, etc. [18].

The core abstractions of a pipe-and-filter software architecture are components (or filters) and connectors (pipes). Our approach, however, distinguishes additional connection abstractions for dispatching and concurrency. These are not only highly relevant abstractions for protocol stack software, identifying them as separate entities also facilitates their control. Figure 1 shows an example of a simple DiPS+ architecture.

Component. Each DiPS+ component represents a functional entity with a well-defined and fine-grained task. Each component processes messages, i.e. network data packets, for instance by constructing or parsing a network header, fragmenting a packet or reassembling its fragments, or encrypting or decrypting packet data. Components are independent entities, unaware of the other components in the system. As such, DiPS+ components are comparable to the robots or machines in the factory metaphor.

Connector. Connectors serve as architectural pipes, i.e. they provide a means to glue components together into a flow. The connector concept maps to the conveyor belt of the assembly line, as it transports messages from one component to another.

Dispatching. The dispatcher serves as a demultiplexer, allowing to split a single flow into two or more sub-flows. Sub-flows are used to differentiate the treatment of incoming messages. For example, the upgoing packet flow in the network layer of a host with routing capabilities is typically split into one flow for local delivery and one for forwarding packets. In addition, packet dispatching (or demultiplexing) usually takes place at each protocol layer-crossing in the upgoing path. Between the network and transport layer, for instance, the upgoing packet flow of a typical UDP-TCP/IP protocol stack is demultiplexed into a TCP and a UDP sub-flow. Again, there is a comparable concept in the factory metaphor, namely the branches of the assembly line that allow to differentiate, for instance, between products with different sets of optional features to be installed.

2.2.2. Blackboard style Referring to the factory metaphor, we can identify another important architectural style. Each product is typically labeled with meta-information (e.g. a bar code) that specifies how it should be treated along its way through the factory. Robots can interpret and use this information when processing a product. Moreover, they can also attach additional meta-information to inform other robots “down the stream”. Obviously, robots should not remove meta-information, since they have no clue which other robots might need it. This kind of management task is reserved for specific factory administrators who have an overall view of the factory.

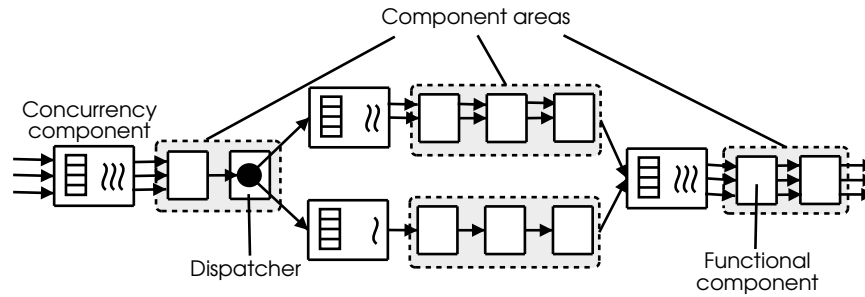


Figure 1. Example of a DiPS+ component pipeline with a dispatcher that splits the pipeline in two parallel component areas. More processing resources have been assigned to the upper concurrency component.

In terms of software architecture, this style of communication corresponds to the blackboard style. It is characterized by indirect message exchange, which results in an anonymous component interaction model.

The blackboard interaction style is very convenient in combination with the pipe-and-filter style. Since each message visits all relevant components following the pipe-and-filter style, messages are the ideal means to serve as central data source. We now zoom in on the blackboard model and explain how it is mapped onto DiPS+.

Message. In order to finish a common task, components forward a message from the source to the sink of the component pipeline. A message is the analogue of a product that is conveyed on a factory's assembly line, and in this paper it is always used to denote a network data packet. Messages are represented as separate entities, since they play a central role in data sharing. Each message can be annotated with meta-information in order to push extra information through the pipeline along with the message, or to influence how a particular message gets processed. Packet-related meta-information can be produced in any functional component when processing incoming packets (e.g. information about addressing, routing, and/or fragmentation). Application-related meta-information is used when application-level parameters (e.g. user priority, required service quality, and/or application type) may impact the internal behavior of the protocol stack. The major benefit of having components attach meta-information to incoming messages is anonymous component interaction: components that consume specific meta-information do not know the producer of these data (and vice versa).

Two examples will clarify the benefit and power of attaching application-specific preferences as meta-information to a message. First of all, it allows for application-, or even user-based prioritization of messages. The attached priority may be used for selecting a message to process from a message queue. To prioritize incoming messages, a special "prioritizer" component could

be introduced in the upcoming component pipe of the protocol stack. Another, more challenging case is to use application-specific communication preferences to influence how network packets are transferred over the physical network. Imagine a vehicle with network capabilities installed, and with multiple wireless communication channels available. Each of these channels may use a specific wireless communication protocol with specific characteristics concerning communication cost, speed, or reliability. All these characteristics may be highly relevant to the user. In such case, data transfer may be influenced by attaching user-based meta-information to outgoing packets. Based on this meta-information, the most appropriate wireless communication device can be selected at the bottom of the protocol stack. Without meta-information, it would be very difficult to make such application-specific decision at this point.

Anonymous communication. Anonymous component interaction enables composing DiPS+ components without affecting their source code. This independency allows for reuse and adaptation of individual units in different protocol stack configurations.

The idea behind the blackboard architectural style is that a collection of independent entities (producers and consumers of information) interacts with a common data structure (the so-called blackboard), which represents the current state.

The blackboard model is usually represented by three major parts [17]. First of all, the *knowledge sources* represent application-specific knowledge and computation. They can interact with the blackboard, but never directly with each other. Secondly, the *blackboard data structure* itself encapsulates state data and uncouples knowledge sources from each other. Knowledge sources make changes to state information in the blackboard, thus incrementally producing a solution to the problem at stake. The third part of the model is the *controller*, driven entirely by the state of the blackboard. Control may be located in the knowledge

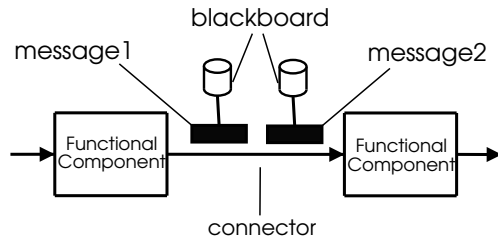


Figure 2. Anonymous communication via a blackboard architectural style: a blackboard data structure has been coupled to each message to carry meta-information from one component to another.

sources, the blackboard, a separate entity or a combination of these.

The blackboard model is mapped onto the DiPS+ architecture as follows. The knowledge sources correspond to the DiPS+ components. Each component can manipulate or attach information to an incoming message, and this information can be anonymously interpreted by any other component in the stack. Obviously, components have to agree on the type of information that is published via the blackboard. Yet, they are unaware of exactly which component has produced the information. Secondly, the blackboard data structure encapsulates the meta-information that is attached to the message traversing the protocol stack from one DiPS+ component to another (see also Figure 2). In this way, each message has a separate blackboard, which traverses the protocol stack along with its message. Finally, there is no explicit control of the blackboard. Each DiPS+ component may adapt its behavior based on the blackboard state, and decide whether and when meta-information is retrieved from an incoming message. Similarly, each component may decide to add or change meta-information.

2.2.3. Layered style Introducing a separate layer abstraction is highly relevant for several reasons. First and foremost, it is very natural to have a design entity that directly represents a key element of a protocol stack. Often, a protocol layer is implemented, integrated or reused in its entirety instead of as a group of independent components. This does not obsolete the relevance of fine-grained components, rather it augments the level of support towards protocol (stack) programmers. Similarly, a factory's assembly line is not structured as a straight connection of processing entities. Rather, these entities are grouped into coarse-grained logical sub-systems that serve a higher-level goal. Such sub-systems may be, in the case of a car factory, metal engineering (pressing or welding), interior processing or body-work painting. The hierarchy of fine-grained entities and more coarse-grained sub-systems allows to zoom in and out to

the appropriate level of detail.

Secondly, each layer offers an encapsulation boundary. Every protocol layer encapsulates data received from an upper layer by putting a header in front of them. Each layer corresponds to one header that contains the information to be shared with the peer layer at the remote host. For incoming packets, each layer parses the protocol header and attaches relevant information to the packet as meta-information.

Finally, from a protocol stack point of view, layers provide a unit of dispatching. Layers can be stacked in many ways. The simplest way – one layer on top of another – does not require layer dispatching. Yet, in cases of multiple layers on top of one (N on 1), one layer on top of multiple layers (1 on N), or a combination of multiple layers on top of multiple other layers (N on N), dispatching is definitely required. Once more, this confirms the relevance of offering layers as a separate abstraction.

2.3. Inside the management plane

DiPS+ addresses two important concerns for management and control. Packet interception, the first concern, allows to interpret the packet flow through the component pipe, for example by counting how many packets arrive per time unit, or by measuring the average packet size. The second concern is concurrency management, which aims at balancing processing resources – threads – throughout the system in order to maximize global throughput. Processing resources are spread throughout the DiPS+ architecture by introducing concurrency components in specific places. Concurrency components sub-divide the system into independent component groups, which will be referred to as component areas (as illustrated in Figure 1). Concurrency components and their advantages are explained thoroughly in the next section, the rest of this section deals with packet interception.

In order to enable fine-grained management, the system must be open to administrators or administrative tools in a well-defined way. Exposing specific implementation issues in such a way that important strategies can be customized to system and application preferences is a well-known technique called open implementation [9]. In addition, control must be customizable to particular circumstances at runtime, since a protocol stack often cannot be brought down to install extra management control features.

In view of that, DiPS+ offers a management meta-level on top of the basic data plane of components and connectors. Packets are transparently intercepted and delegated to this higher level, where they can be controlled, interpreted, manipulated, etc.

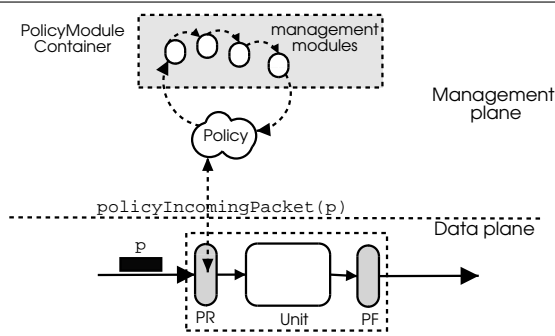


Figure 3. A DiPS+ component (consisting of a packet receiver, the core unit, and a packet forwarder) with a policy object that intercepts incoming packets (p). The policy delegates incoming packets to a pipeline of management modules.

To this end, architectural connectors are represented in the DiPS+ design by an explicit packet receiver and forwarder. Offering explicit packet receivers and forwarders does not only uncouple units from each other, but also provides attachment hooks for the management layer. Such hooks are designed as separate entities (called policies), which are responsible for the handling of incoming and outgoing packets. Unlike functional components, policies encapsulate non-functional behavior (e.g. throughput monitoring, logging, or pattern recognition).

Policies. A typical solution used for open systems that must be customizable at run-time is a meta-object protocol (MOP) [8]. Basically, a MOP specifies a generic interface to a higher (meta-)level, used to add message processing that typically does not belong to the purely functional behavior of the system (e.g. concurrency control, monitoring, fault handling, etc.).

The DiPS+ design allows for transparent packet interception at the packet receivers and forwarders via the associated `Policy` object. The policy describes the interface of the management meta-level. Via this interface, a packet object is intercepted at the packet receiver (or forwarder) and delegated to the meta-level.

A policy describes the strategy to handle each incoming packet. The strategy delegates each packet to a number of `ManagementModule` objects, which may be registered by an administration tool at application level. Each individual management module describes the processing of a packet, for instance, increasing a generic packet counter, interpreting specific information from the packet, or even attaching to the packet extra meta-information based on information accumulated over time.

Generic management modules can be reused from a repository. Yet, the policy design also allows for application-

specific modules to be registered. To this end, multiple management modules are connected into a pipeline. Figure 3 illustrates a packet receiver policy and shows how incoming packets are bypassed to a pipeline of management modules at the meta-level. The last module delivers the packet back to the packet receiver (or forwarder).

Customizing policies. It should be clear that our primary goal is to provide software support for injecting policies into a protocol stack, and not to develop management policies. Currently, we use simple policies to validate our software architecture, but customization is possible by injecting policies from third parties. Application-specific management modules can be added to a so-called `PolicyModuleContainer`. This allows for applications to customize in a controlled manner how packets are handled at the management plane.

3. Modeling concurrency

3.1. Concurrency components

The concurrency component in DiPS+ is used to store packets that cannot be processed immediately. It selects and handles packets in an autonomous and asynchronous manner. In this way, a concurrency component breaks the pipeline in two independent component areas.

Concurrency components are responsible for two tasks: injecting active behavior into the system and request scheduling. The former means that a concurrency component is associated with one or more packet handler threads. Packet handlers associated with a concurrency component `C` guide packets through the component area following `C`. On a mono-processor, these threads are interleaved by the scheduler of the operating system, thus resulting in virtual parallelism. Yet on multi-processors, each concurrency component may be associated with a different processor, which enables true parallelism.

Request scheduling, the second task of a concurrency component, controls when and which packets are selected for processing. The selection criterion may be based on packet-specific information such as the size of the packet, its source or destination, or its type (e.g. connection setup versus data transfer). This allows to differentiate between incoming packets, based on packet-specific information.

When mapped onto the assembly line metaphor, the concurrency component corresponds to the combination of a storage space and one or more product conveyors. The storage space buffers incoming products that cannot be processed immediately. The conveyors correspond to processing resources.

3.2. Advantages

Having explicit concurrency components shows three major advantages. First of all, it allows not only to reuse components whether or not concurrency is present, but also to reconfigure and customize the system where concurrency needs to be added. In this way, the system's structure can be fine-tuned to specific circumstances and requirements, for instance by adding concurrency components only if needed.

Secondly, it allows for fine-grained and distributed control of scheduling requests. Each concurrency component may incorporate a customized scheduling strategy, using all meta-information attached to the request by upstream components. This information may not yet be available at the beginning of the component pipeline. In case of protocol stacks, for instance, an incoming network packet only reveals its encapsulated information by parsing protocol headers. The more headers are parsed, the more information is available, for instance to prioritize between multiple source addresses.

Such fine-grained control is much more difficult to achieve with traditional thread allocation models, such as the thread-per-request or the thread pool model. Indeed, these models associate a request with a thread that guides it throughout the system without further control.

A third advantage of having concurrency components spread throughout the system, is that it allows to prioritize not only between incoming packets, but also between component areas. On the one hand, this considerably facilitates finding and solving internal throughput bottlenecks, i.e. component areas where many more packets arrive than can be processed. On the other hand, concurrency components may help prioritize particular component areas based on application-specific requirements. DiPS+ concurrency components allow, for instance, to associate additional threads with those component areas that incorporate crucial sub-tasks. The component pipeline as depicted in Figure 1 identifies two parallel component areas. This setup (i.e. two parallel tracks) may benefit most from spreading concurrency components since it demultiplexes the packet flow and, by consequence, allows to differentiate between the possible sub-flows. One area may, for instance, encrypt the data in each incoming packet, while the other area only encrypts the header information, which is clearly less processing-intensive.

3.3. Architectural support

Concurrency components exploit the benefits of both the pipe-and-filter and the blackboard architectural style. The pipe-and-filter style divides the system into independent components. Because of this independence, concurrency

components can be added anywhere in the pipeline, without affecting the functional components within. The DiPS+ dispatcher allows to split a component pipeline into parallel sub-pipes. In this way, each sub-pipe can be processed differently by putting a concurrency component in front of it. Thanks to the blackboard style of data sharing associated with each individual message, component tasks are typically packet-based, i.e. each component handles incoming packets by interpreting or adding meta-information. This allows to increase parallelism since most components have no local state that is shared by multiple threads in parallel.

4. Load management

One of the application domains of DiPS+ is load management, i.e. intelligently processing incoming (over)load. Load management can be viewed from two perspectives. From a concurrency point of view, load management distributes the available processing power (threads) across the system such that the overall system performance is optimized [13]. By consequence, the management plane should be able to detect performance bottlenecks, i.e. component areas where packets arrive faster than they can be processed. In addition, the management plane must solve these bottlenecks by migrating processing resources, which are associated with the concurrency component in front of a component area, from underloaded to overloaded component areas.

At the same time, load management can be viewed from a request control perspective. Request control deals with overload by limiting or shaping the arrival rate of new requests to a sustainable level. To this end, request control classifies incoming packets in order of priority based on several parameters, such as their destination, data size, encapsulated protocol, packet type (connection establishment or data transfer), or application-specific preferences passed via meta-information. Low priority packets may be handled separately, postponed until all higher priority packets have been processed, or simply dropped to release system resources for higher priority packets.

Concurrency and request control are compatible and synergistic though. Obviously, thread re-allocation is only relevant if enough thread resources are available to handle incoming packets. For extremely high loads, however, even optimal resource allocation will not prevent the system from being overwhelmed. Instead of balancing resources, such cases require controlling and reducing the input to prevent the system from collapsing under the high load pressure.

In [13] we describe DMonA (the DiPS+ Monitoring Architecture) and present a prototype that allows to install various concurrency control strategies at run-time. This paper, and the next section in particular, shows an industrial case study that focuses on request control. Changing strategies at run-time is not taken into account in this setup.

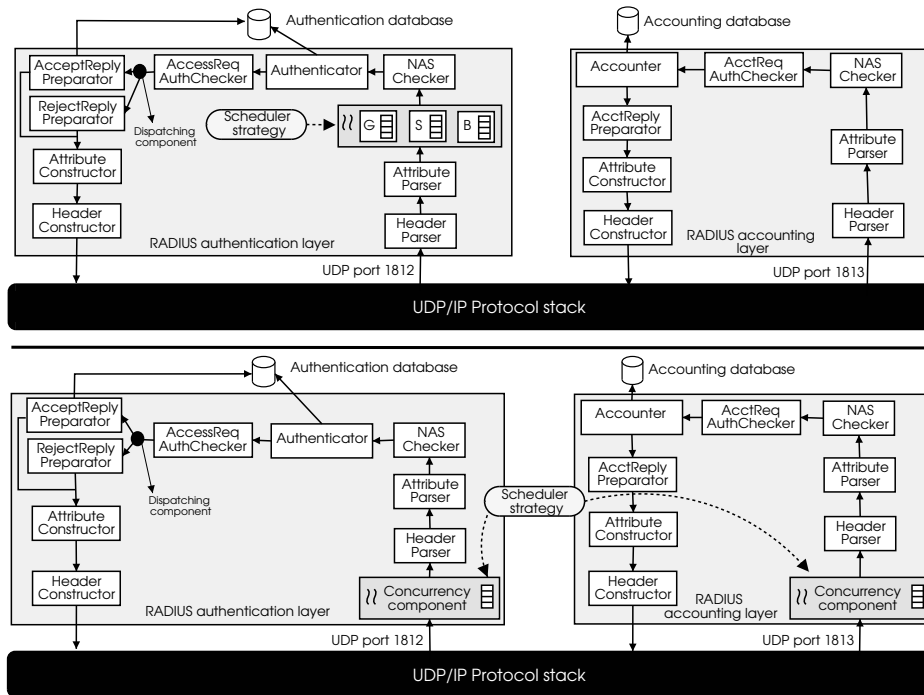


Figure 4. Design of RADIUS authentication and accounting in DiPS+. The top figure illustrates a scheduler strategy that distinguishes between gold, silver, and bronze users. The bottom figure demonstrates a strategy that prioritizes accounting requests over authentication requests.

5. Case study

An authentication, authorization and accounting (AAA) server can easily become a bottleneck when, for instance, thousands of users contact it simultaneously to get Internet access, which confirms the need for adaptability as provided by DiPS+. By consequence, an AAA server should be flexible enough to handle such overload situations gracefully. Indeed, simply dropping packets randomly does not ensure that the overload can be controlled in a pre-defined manner.

To validate the opportunities concerning adaptability, we have designed and implemented the RADIUS authentication and accounting protocol according to the DiPS+ style, and introduced concurrency components to allow for request control based on various application-specific requirements.¹ In this way, low-priority packets can be queued or dropped in favor of high-priority packets. Thanks to its modularized design and the flexibility offered by DiPS+ concurrency components, various priority schedul-

ing strategies can be inserted in multiple places in the RADIUS protocol.

We briefly discuss the RADIUS protocol as well as its design in DiPS+ and describe two examples of application-specific packet scheduling strategies. One is based on user information, the other on the type of RADIUS request (authentication or accounting).

5.1. The RADIUS protocol

RADIUS (Remote Authentication Dial-in User Service) is a client-server internetworking security system running on top of the UDP transport protocol. It controls authentication (verifying user name and password), accounting (logging relevant user activities), and authorization (access-control) in a multi-user environment. Upon ADSL² network login, a dial-up server delegates login information to the RADIUS server of the client's Internet Service Provider (ISP) for authentication and for obtaining network configuration information, such as the local IP address to use on the Internet. As part of accounting services, RADIUS logs user information in a database or a file according to cus-

¹ This research has been carried out in order of Alcatel Bell, and has been part of the SCAN (Service Centric Access Networks) research project, supported by the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN #010319).

² ADSL stands for Asymmetric Digital Subscriber Line and provides broadband Internet access using a telephone line.

tomers' requirements. A typical RADIUS session consists of a sequence of authentication and accounting requests: (1) the user is authenticated by sending an access request; (2) the session is initiated using an accounting-start request; (3) during an active session accounting-interim requests can be sent optionally to log status information; (4) finally, the session is terminated using an accounting-stop request.

5.2. RADIUS design for DiPS+

Figure 4 shows the high-level DiPS+ design of RADIUS authentication (left side), and accounting (on the right).³ Following the RADIUS specification, the authentication layer accepts packets via UDP port 1812, the accounting layer is connected via UDP port 1813. The parsing components (`HeaderParser` and `AttributeParser`) interpret the header and the list of attributes of an incoming RADIUS packet. The header and attribute constructors (`HeaderConstructor` and `AttributeConstructor`) create the RADIUS header and the list of attributes, and attach these to an outgoing RADIUS response packet. The `NASChecker` verifies the originator of the request and drops the packet in case the NAS is unknown. The parsing and constructing components, as well as the NAS checker, are generic and are reused in both the authentication and the accounting layer.

The authentication layer consists of four extra components specifically for authentication, and a dispatching component to distinguish between processing accepted and rejected packets. The `Authenticator` component looks up the user's password in an external DiPS+ layer resource (e.g. a database or a file). The `AccessRequestAuthenticatorChecker` compares this password with the encrypted password that is encapsulated in the authentication request. Finally, a positive or negative authentication response is prepared by the `AcceptReplyPreparator` or the `RejectReplyPreparator`. The former contacts the authentication database to look up the configuration information (e.g. the IP address) to respond to the client.

The accounting layer consists of three accounting-specific components. The `AcctRequestAuthenticatorChecker` verifies the integrity of an incoming accounting request. The `Accounter` logs the information in the accounting request to an external DiPS+ layer resource (e.g. a database or a file). The `AcctReplyPreparator` creates a response packet and attaches the necessary meta-information.

³ The top and bottom design in Figure 4 only differ in that they insert the two scheduling strategies in different places in the design. The actual strategies are explained later.

5.3. User differentiation

A first strategy for intelligent request control is based on the idea that RADIUS authentication requests can be prioritized according to the associated user. An ISP, for instance, may identify gold, silver and bronze types of users. One of the advantages of being a gold user is that RADIUS authentication requests have priority over silver and bronze users. This guarantees fast gold authentication responses, even during peak loads, while silver and bronze users are queued or dropped.

As the top of Figure 4 illustrates, user differentiation is accomplished by inserting a concurrency component in the pipeline, after the RADIUS header and attributes have been parsed. At this point, all required user information is available to classify gold, silver, and bronze users. Based on this classification, the concurrency component stores incoming packets in three separate packet queues. The scheduling strategy associated with the packet handler of the concurrency unit then simply selects packets from the gold queue, until it is empty. As long as no gold packets are available and the silver queue is not empty, silver packets are processed. Finally, bronze packets are only processed when no pending gold or silver packets are available.

5.4. Request type differentiation

A second strategy is based on the idea that the RADIUS server should maximize the number of active RADIUS sessions. This means that the server should not process any authentication requests as long as accounting requests are pending. Indeed, accepting authentication requests would imply that extra accounting (start/interim/stop) requests are initiated, which only increases the number of pending accounting requests. By blocking authentication requests temporarily, the server controls the load to be processed, and thereby protects itself from going down under the high load. Moreover, established sessions are not affected by such an overload situation.

As the bottom of Figure 4 illustrates, request type differentiation can easily be added in the DiPS+ design by introducing two concurrency components as the entry points for the authentication and the accounting layer. The associated scheduler strategy selects authentication packets only when no accounting packets are available.

5.5. Results

Presenting in detail the performance results of the RADIUS implementation clearly transcends the scope of this paper. These results, as well as a comparison between DiPS+ and a commercial RADIUS implementation in Java, have been extensively described in a technical report [14].

In summary, the overhead of using DiPS+ instead of the commercial Java version is minimal (less than 5%),⁴ and the results clearly show the advantage of using application-specific scheduling strategies during overload. On the one hand, gold users are always processed at the maximum throughput while, during overload, the non-DiPS+ version randomly spreads the throughput degradation over all user types. On the other hand, by dropping authentication requests when the throughput limit is reached, the DiPS+ version is able to maximize the number of active sessions at the highest possible throughput. Moreover, the throughput in this case is substantially higher than for the non-DiPS+ version, since existing sessions are not affected by the overload of incoming authentications.

6. Related Work

6.1. Protocol stack frameworks

Although multiple software design frameworks for protocol stack development have been described in the literature [5, 2, 1, 4], we compare the DiPS+ approach to three software architectures, which are tailored to protocol stacks and/or concurrency control: Scout [15], Click modular router [11], and SEDA [18].

6.1.1. Scout The Scout operating system [15] uses a layered software architecture that is similar to layers in the DiPS+ data plane, yet does not offer fine-grained entities such as components for functionality, dispatching, or concurrency. Its primary target are multimedia network protocol stacks. Unlike conventional systems, which are typically structured around computation-centric abstractions (e.g., processes, jobs, tasks), Scout proposes a communication-oriented operating system design.

Scout is designed around a communication-oriented abstraction called the path, which represents an I/O channel throughout a multi-layered system and essentially extends a network connection into the operating system. A path in Scout is comparable to a DiPS+ packet in combination with its meta-information. DiPS+ components can share specific attributes with components down the packet flow, possibly across layers. As such, a DiPS+ packet leaves a trace through the protocol stack, similar to a path in Scout.

6.1.2. Click modular router The Click modular router [11] is based on a design analogous to DiPS+. Although one can recognize a pipe-and-filter architectural

style, Click pays much less attention to software architecture than DiPS+. A Click router is built from fine-grained packet processing modules with private state, called elements. Each element has input and output ports, comparable to the DiPS+ packet receiver and forwarder. Elements can communicate either directly using function calls or indirectly using a packet queue that uncouples adjacent elements. Click supports two packet transfer mechanisms: push and pull.

DiPS+ offers a uniform push packet transfer mechanism and allows for active behavior inside the component graph by means of concurrency components. A concurrency component does not only uncouple component areas in the pipeline, it also offers active behavior as a separate first-class entity. This allows for internal concurrency to be managed and adapted to dynamic circumstances in the system or its environment.

6.1.3. Staged Event-Driven Architecture SEDA [18] offers an architecture for supporting massively concurrent systems, primarily web servers. SEDA decomposes an application into a flow of independent modules, called stages. Its modularized design allows for fine-grained development, maintenance and control of the system. Stages can communicate with each other by sending events. Every stage consists of a queue of incoming requests (i.e. events), an event handler that processes incoming requests, and a thread pool to support parallelism. A SEDA stage can be managed by a controller that affects scheduling and thread allocation.

Stages in SEDA can be compared with a DiPS+ component area along with its preceding concurrency component. Stages as well as concurrency components allow for specific areas in the system to be individually conditioned to load. However, the SEDA controller and associated stage are tightly coupled, whereas DiPS+ clearly separates a concurrency component from the functional code. As such, SEDA does not provide a clean separation between the functional and the management level, as the DiPS+ architecture does. This strong integration of management into SEDA does not allow changing management strategies easily, let alone removing management altogether by simply unplugging a management level as in DiPS+.

6.2. Concurrency and separation of concerns

A critical element in our research is the separation of concurrency from functional code. Kiczales [10] defines non-functional concerns as *aspects* that cross-cut functional code. An aspect (for example concurrency) is written in a specific aspect language and is woven into the functional code by a so-called aspect weaver at pre-processing time. Although this approach clearly separates all aspects from the functional code (at design-time), all aspects disappear at

⁴ The overhead induced by the DiPS+ architecture (e.g. indirect component interaction via explicit entry/exit points) depends a.o. on the processing-time spent inside a component, which can vary considerably. By analyzing the current set of DiPS+ components, we can conclude that overhead varies between 5 and 15% [12].

run-time, which makes it very difficult (if not impossible) to adapt aspects dynamically.

Apertos [6] introduces concurrent objects that separate mechanisms of synchronization, scheduling and interrupt mask handling from the functional code (such as a device driver). Therefore, programmers can concentrate on writing functional code without having to write auxiliary code for synchronization. This makes software more understandable, and reduces the risk of errors.

To allow for both design- and run-time adaptability of the concurrency model, we present separated concurrency components that are adaptable at run-time. Similar work is done in SEDA [18].

7. Conclusion

Only few (research) implementations of protocol stacks define and apply an explicit software architecture. One reasonable explanation for this lack of architecture support is that a software architecture may involve extra overhead and, by consequence, reduce system performance (cfr. explicit entry/exit points, layers, dispatchers, etc.).

Although software architectures related to DiPS+ clearly show advantages in both developing and adapting protocol stacks, they often lack a clear separation of concurrency and/or management aspects from the functional code. This makes it difficult to customize the behavior of a protocol stack to application-specific requirements.

Our prototype implements a well-defined software architecture and offers explicit support for concurrency and run-time adaptation. The RADIUS case study has shown that, thanks to its modularized design and the flexibility offered by concurrency components, the DiPS+ architecture allows to experiment with various request control strategies in different places in the RADIUS protocol, and without having to change any functional source code. Moreover, the overhead of the DiPS+ architecture could be compensated by deploying application-specific strategies inside the protocol stack.

To the best of our knowledge, there are no similar reports on experimental systems that illustrate the benefits of software flexibility while enabling efficient behavior.

References

- [1] F. J. Ballesteros, F. Kon, and R. Campbell. Off++: The Network in a Box. In *Proceedings of ECOOP Workshop on Object Orientation in Operating Systems (ECOOP-WOOS 2000)*, Sophia Antipolis and Cannes, France, June 2000.
- [2] N. T. Bhatti. *A System for Constructing Configurable High-level Protocols*. PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, Dec. 1996.
- [3] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. J. Wiley & Sons, New York, NY, USA, 1996.
- [4] H. Hüni, R. E. Johnson, and R. Engel. A framework for network protocol software. In *Proceedings of OOPSLA'95*, pages 358–369, Austin, TX, USA, Oct. 1995.
- [5] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
- [6] J. Itoh, Y. Yokote, and M. Tokoro. Scone: using concurrent objects for low-level operating system programming. In *Proceedings of OOPSLA'95*, pages 385–398, Austin, TX, USA, Oct. 1995.
- [7] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Transparent Hot-Swapping Support for Producer-Consumer Components. In *Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, Apr. 2003.
- [8] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
- [9] G. Kiczales, J. Lamping, C. V. Lopes, e.a.. Open implementation design guidelines. In *Proceedings of ICSE'97*, pages 481–490, Boston, MA, USA, 1997.
- [10] G. Kiczales, J. Lamping, e.a.. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of ECOOP'97*, pages 220–242. Jyväskylä, Finland, June 1997.
- [11] E. Kohler. *The Click Modular Router*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, Feb. 2001.
- [12] S. Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, Nov. 2003.
- [13] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. Self-adapting concurrency: The DMonA architecture. In *Proceedings of the First Workshop on Self-Healing Systems (WOSS'02)*, pages 43–48, Charleston, SC, USA, 2002.
- [14] S. Michiels, L. Desmet, and P. Verbaeten. A DiPS+ Case Study: A Self-healing RADIUS Server. Report CW-378, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium, Feb. 2004.
- [15] A. B. Montz, D. Mosberger, S. W. O'Malley, and L. L. Peterson. Scout: A communications-oriented operating system. In *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61, Orcas Island, WA, USA, May 1995.
- [16] J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In J. H. L. Barroca and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. 1999.
- [17] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.
- [18] M. F. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, Aug. 2002.