Table of Contents

Embedded System Development with Components

Connecting Embedded Devices Using a Component Platform for Adaptable Protocol Stacks 1
S. Michiels, N. Janssens, L. Desmet, T. Mahieu, W. Joosen, P. Verbaeten (K.U.Leuven)

II

Connecting Embedded Devices Using a Component Platform for Adaptable Protocol Stacks

Sam Michiels, Nico Janssens, Lieven Desmet, Tom Mahieu,
Wouter Joosen, and Pierre Verbaeten

K.U.Leuven, Dept. Computer Science,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{sam.michiels,nico.janssens}@cs.kuleuven.ac.be,
WWW home page:
http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/DIPS/

Abstract. Research domains such as sensor networks, ad-hoc networks, and pervasive computing, clearly illustrate that computer networks have become more complex and dynamic. This complexity is mainly introduced by unpredictable and varying network link characteristics, heterogeneous capabilities of attached nodes, and the increasing user expectations regarding reliability and quality of service. In order to deal with this complexity and dynamism of computer networks, the system's protocol stack must be able to adapt itself at run-time. Yet, to handle this complex challenge effectively and efficiently, we claim that it is essential for protocol stacks to be developed with run-time adaptability in mind. This article presents a software architecture tailored to build highly adaptable protocol stacks, along with a component platform that enforces this architecture. Although the presented software architecture focuses on protocol stacks in general, we zoom in on the application of its founding principles in the domain of embedded network devices.

1 Introduction

The use of mobile embedded devices to offer users network connectivity anywhere and anytime increases significantly [1]. In order to achieve seamless interoperability of heterogeneous devices in a highly dynamic network, the protocol stack of each connected device often needs to exhibit a similar degree of dynamism. Connected devices can vary from powerful portable PCs or PDAs to resource limited embedded devices like mobile phones or sensors.

This article presents a software architecture [2] tailored to build highly adaptable protocol stacks, along with a component platform [3] that enforces this architecture. We refer to this combination as DiPS+, the Distrinet Protocol Stack [4]. The key focus in DiPS+ is run-time adaptability to application- and environment-specific requirements or characteristics. The strength of the DiPS+ approach is twofold.

On the one hand, DiPS+ provides for two essential aspects of *run-time adaptability*: it offers support for controlling concurrency behavior of the protocol stack and for swapping components in a transparent manner, while sharing a common component platform core. This considerably facilitates system management, since it allows for modular integration of non-functional extensions that cross-cut the core protocol stack functionality.

On the other hand, DiPS+ proposes a *design method* that imposes the separation of basic protocol stack functionality from additional run-time adaptability support. As will be illustrated further in this article, the employed separation of concerns allows for a programmer to concentrate on a single concern (e.g. the behavior of a DiPS+ protocol stack) without being distracted by other concerns scattered across the same functional code (such as additional adaptability support). This is essential for making adaptable protocol stacks more comprehensible, reusable and flexible.

We believe that the DiPS+ component platform is a convincing case study to illustrate the potential of using fine-grained components and separation of concerns in building highly adaptable network systems. We argue that (1) in order to achieve run-time adaptability, the software must be developed with flexibility in mind, and that (2) modularity and strict separation of concerns are two main characteristics of an adaptable design [4]. Obviously, there are many other specific concerns when developing embedded systems, such as performance control, resource awareness, and real-time constraints. Experience shows that at least the first two of these "embedded system characteristics" benefit from our software architecture as well. We do not claim that the DiPS+ component platform can be used as-is in networked embedded systems; however, we are convinced that its founding principles can be beneficial for this kind of software. Throughout the article, we will clarify the advantages of the DiPS+ ideas for embedded systems.

We have validated DiPS+ successfully, a.o. in the context of concurrency control [5, 6]. The DiPS+ component platform has also been applied in research domains different from component swapping and concurrency control. Discussing all related research tracks would lead us too far and certainly transcends the scope of this article. In summary, DiPS+ offers support for unit testing [7], automatic component composition [8, 9] and framework optimization [4], while current research extends DiPS+ from a local node architecture to a distributed management platform that allows for controlling and adapting multiple connected protocol stacks [10].

The remainder of this article is structured as follows. Section 2 sketches the domain of our case study: it explains the need for providing flexibility in protocol stacks with respect to concurrency control and component hot-swapping. Section 3 presents the DiPS+ component platform, which offers core programming abstractions to improve the development of adaptable protocol stacks. The two sections that follow each describe a specific extension of the DiPS+ platform to control and manage the underlying protocol stack: Section 4 focuses on dynamic load management; Section 5 explains how transparent component

swapping is supported. Section 6 describes the DiPS+ prototype and its validation in various research projects and Master's theses. It also explains how our positive experiences in the domain of networking software can be applied in the broader domain of component-based embedded systems. Section 7 positions our work with respect to related research. Conclusions are presented in Sct. 8.

2 The Specific Case of Protocol Stacks

The development of protocol stacks is often complex and error-prone, especially when additional preconditions (such as the need for run-time adaptability) are imposed. Before elaborating on how advanced separation of concerns contributes to making adaptable protocol stacks more comprehensible, accessible and reusable, we elaborate in this section on the importance of run-time adaptability in the domain of protocol stack software (whether or not in an embedded system). More precisely, we focus on non-functional adaptations (load management) and on functional adaptations (component hot-swapping).

2.1 Load Management

Management of system load in networked systems tries to prevent systems from being overwhelmed by arriving network packets. Load management is highly important for both embedded devices in an ad-hoc network (since they may have limited resources available), and network access devices (which may receive considerable access demand peaks when a large group of users connects in parallel). Since cooperating nodes in an ad-hoc network may be highly heterogeneous with respect to available processing resources, memory, and data transfer capabilities, low-end router nodes easily get overloaded by data transfers induced by more powerful machines. By consequence, adaptive load management is highly relevant for embedded network devices.

Solutions for system load control often depend on run-time circumstances and/or application-specific requirements. In addition, system load should be controlled and managed at run-time to handle changing network circumstances gracefully. These changes can, for instance, be induced by (1) popular services being offered on the network, resulting in increasing network traffic to the server, (2) more clients being added dynamically and/or clients with varying quality of service requirements, or (3) decreasing processing capabilities when the battery of a stand-alone device is getting low. In other words, circumstances may vary at the side of the server, the clients, and the network nodes themselves.

In order to enable (low-end) devices to handle overload situations gracefully, our approach proposes to dynamically balance resource consumption based on application- and environment-specific requirements. This goal is achieved by detecting internal bottlenecks and deploying a solution to the problem in the running protocol stack. A bottleneck occurs when many more packets arrive at a component than can be processed immediately.

Bottlenecks can be processed in many different ways. We concentrate on three approaches: packet classification and prioritization, input rate control, and thread re-allocation from underloaded to overloaded areas. It is important that solutions (e.g. packet classification, input rate control, thread re-allocation) can be performed at any place in the protocol stack and, by consequence, can be based on information not yet available when the packet arrives. Protocol headers, for instance, only release their information when they have been parsed; however, this information may considerably influence further processing of the packet. In addition, the classification strategy used to differentiate between packets may be based on application- and/or environment-specific requirements, in order to take into account changing circumstances at run-time (e.g. ad-hoc network topology, available system resources, network load, etc.). Thread re-allocation focuses on tasks to be executed instead of packets. This allows to customize processing of particular areas in the system by adding or removing threads locally. For example, system performance is improved by increasing parallelism in areas that have become (temporary) I/O bottlenecks.

Our approach is complementary to existing load management techniques that are, or can be, used to handle overload situations gracefully. The most relevant techniques in this context are quality-of-service (QoS) protocols, load balancing [11], and active networks [12]. Our approach complements these distributed techniques by offering a local platform that is able to detect and (partially) handle overload situations.

2.2 Component Hot-Swapping

Research domains such as ad-hoc networks, sensor networks, 4G wireless networks and pervasive computing, clearly indicate a trend towards more *heterogeneous* mobile computer networks. Network heterogeneity manifests itself in the form of increased diversity in the type of communication technology that devices are equipped with (such as Bluetooth, WiFi, HomeRF and satellite links), as well as in the types of embedded devices connected to the network (differing in memory capacity, processing power and battery autonomy) [13]. In addition, performance characteristics of network nodes and communication links most often change over time, a.o. due to disturbing influences. These *heterogeneous* and *dynamic* performance specifications will affect the inter-operability of connected nodes, and as a result are most likely to compromise the *communication quality* of the network, in particular when a best-effort communication model is employed. For instance, a Bluetooth scatternet (operating at 2Mbps) will probably become a bottleneck when interconnecting a number of 802.11 MANET's (22Mbps throughput).

To fully exploit the potential of such heterogeneous and dynamic networks, it is essential for the protocol stacks of the connected embedded devices to *adapt themselves* at run-time as the environment in which they execute changes (e.g. by installing a compression service to boost the quality of the slow Bluetooth scatternet). To this end, we aim at coping with the increasing user expectations regarding quality of service. By consequence, the underlying protocol stacks

should exhibit a similar degree of dynamism, which illustrates the need for employing *programmable* [14] (i.e. adaptable) network nodes. These programmable networks are strongly motivated by their ability to rapidly change the protocol stack of network nodes without the need for protocol standardization.

In addition, protocol stack reconfigurations should be performed *at run-time* (transparently for end-user applications) to promote permanent connectivity of the embedded devices and thus exploit the full potential of mobile wireless networks. This requires the node architecture to conduct adaptations (recomposition) of the protocol stack functionality without having to shut down and restart active connections. As a result, a running DiPS+ protocol stack can be customized by a third party (such as a network operator or intelligent self-healing network support), without interfering with the execution of applications using the network.

More in detail, we focus on *unanticipated* protocol adaptations, such as feature additions and protocol revisions. Since these adaptations are not anticipated at design-time or deployment-time, component *hot-swapping* is essential to achieve seamless run-time evolution of protocol stacks in mobile embedded devices. In addition, component hot-swapping is justified by the memory constraints inherent in connected *limited* embedded devices, such as intelligent sensors and mobile phones.

Depending on the protocol to be adapted, additional support is required to prevent the replacement of DiPS+ components from jeopardizing the functionality of a running stack, which would compromise the correct functioning of the ad-hoc network. This includes *avoiding packet loss* during a reconfiguration (a.o. essential when changing protocols like TCP that aim to provide full reliability) as well as imposing a *safe state* over the DiPS+ components before conducting the actual reconfiguration. As will be illustrated in Sct. 5, the latter is essential to prevent reconfiguration of a composition from breaking the consistency of the components making up the protocol stack [15, 16].

3 The DiPS+ Component Platform

As stated in the introduction, DiPS+ aims for modular integration of non-functional extensions (such as support for load management and component hot-swapping), which share a common component platform. Strict separation of such non-functional behavior has proven to be an essential feature of adaptable, maintainable and reusable software [17]. To separate non-functional behavior from basic protocol stack functionality, the DiPS+ architecture represents data (packet) processing and protocol stack management as two planes on top of each other, respectively the *data* and the *management plane*.

The data plane in the DiPS+ architecture houses the functional part of the system, i.e. the protocol stack. 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 one or more management planes, which act as meta-levels to extract information from

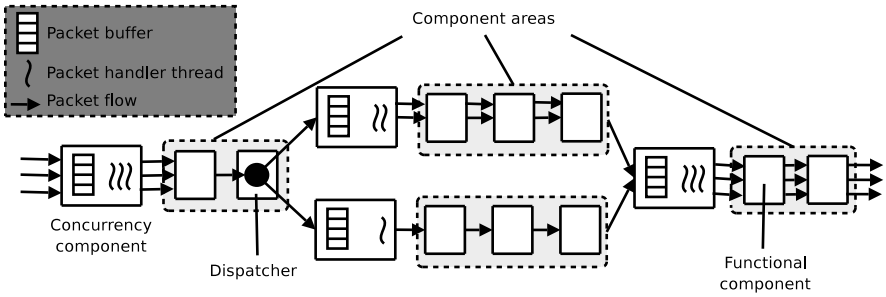

Fig. 1. Example of a DiPS+ component pipeline with a dispatcher that splits the pipeline into two parallel component areas. More processing resources have been assigned to the upper concurrency component.

the data plane and control its behavior. Each management plane is responsible for a specific concern (e.g. load management or component hot-swapping) and is clearly separated from the data plane. In this way, a management plane can be added or removed without affecting components in the data plane.

In the remainder of this section, we elaborate on the architectural styles employed by the data plane and describe how the provided abstractions in the DiPS+ component platform enable run-time adaptability. Afterwards, in Sects. 4 and 5, the modular extendibility of the data plane with support for load management and component hot-swapping will be demonstrated.

3.1 Data Plane: Combination of Architectural Styles

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. By employing these architectural styles, the DiPS+ platform offers a number of framework abstractions (such as components, connectors, and packets) to ease development of adaptable protocol stacks.

Pipe-and-filter Style. The pipe-and-filter style is very convenient for developing network software, which maps naturally to the pipeline style of programming. A protocol stack can be thought of as a downgoing and an upgoing packet flow.

The core abstractions of a typical pipe-and-filter software architecture are connectors (pipes) and components (filters). Connectors provide a means to glue components together into a flow. Each functional component in DiPS+ represents an entity with a well-defined and fine-grained functional task (e.g. constructing or parsing a network header, fragmenting a packet or reassembling its fragments, or encrypting or decrypting packet data). Our architecture distinguishes additional component types for dispatching and concurrency (see Fig. 1). These are not only highly relevant abstractions for protocol stack software, identifying them as explicit entities also facilitates their control. The dispatcher

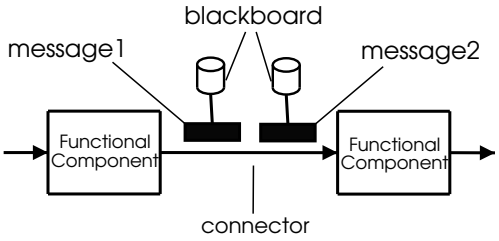

Fig. 2. Anonymous communication via a blackboard architectural style: a blackboard data structure has been coupled to each message to carry meta-information from one component to another.

serves as a demultiplexer, allowing to split a single flow into two or more subflows. Concurrency components and component areas are described in Sct. 3.3.

Blackboard Style. The blackboard interaction style is characterized by an indirect way of passing messages from one component to another, using an in-between data source (blackboard). This style is very convenient in combination with the pipe-and-filter style to increase flexibility and component independence.

The blackboard model is mapped onto the DiPS+ architecture as follows (see also Fig. 2). In order to finish a common task, DiPS+ components forward an explicit message (packet) object from the source to the sink of the component pipeline. In addition, each message can be annotated with meta-information. Attaching meta-information allows to push extra information through the pipeline along with the message, for instance to specifiy how a particular message should be processed. The message represents the blackboard, which encapsulates both data and meta-information. In this way, components that consume specific meta-information do not have to know the producer of these data (and vice versa). By consequence, components become more independent and reusable since they do not rely on the presence of specific component instances.

Layered Style. Introducing an explicit layer abstraction in a protocol stack architecture 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. Secondly, each layer offers an encapsulation boundary. Every protocol layer encapsulates data received from an upper layer by putting a header in front. Finally, from a protocol stack point of view, layers provide a unit of dispatching.

The general advantage of applying the layered style is that it allows to zoom in and out to an appropriate level of detail. When not interested in the details of every fine-grained component, one can zoom out to a coarse-grained level, i.e. the layer.

3.2 Explicit Communication Ports

The employed architectural styles have resulted in the design of the DiPS+ components. A component in DiPS+ is developed as a core surrounded by explicit component entry and exit ports.

DiPS+ Component. Component activity is split into three sub-tasks: packet acceptance, packet processing, and packet delivery. The DiPS+ framework controls packet acceptance and delivery by means of explicit component entry and exit points (the packet receiver and forwarder).

The design of a DiPS+ component consists of three entities (see also Fig. 3). Packet processing is taken care of by a DiPS+ `Unit` class, which forms the core of a component. The `PacketReceiver` (PR) and `PacketForwarder` (PF) classes act as unit wrappers and uncouple processing units. The DiPS+ `Component` class is a pure framework entity that is transparent to programmers. A component encapsulates and connects a unit together with its packet receiver and forwarder. All components in DiPS+ share a common functional packet interface `incomingPacket(Packet p)`. Some components may offer one or more management interfaces next to their functional interface, as will be described further in Sct. 3.3.

With an eye to enable fine-grained management, the DiPS+ data plane is designed to be *open* for customizations in a well-defined way [18, 19]. DiPS+ components allow for transparent packet interception at the communication ports via their associated `Policy` object (see Fig. 3). The policy delegates each packet to a number of pipelined `ManagementModule` objects, which may be registered by an administration tool at application level. Unlike functional components, management modules encapsulate non-functional behavior (e.g. throughput monitoring, logging, or packet blocking).

Advantages. The combination of the pipe-and-filter and the blackboard architectural style results in two main advantages. First of all, it supports the design of so-called plug-compatible components [20], i.e. components that are unaware of any other component, directly or indirectly. The pipe-and-filter style uncouples adjacent components by means of a connector (represented in DiPS+ by a PF-PR combination). The blackboard style, for its part, allows for anonymous component interaction (represented in DiPS+ by packets and their associated meta-information).

Secondly, the combination enables fine-grained and unit-specific management and control. Both the PR and the PF serve as attachment hooks for the management plane. Such hooks are designed in DiPS+ as separate entities, called policies, which are responsible for the handling of incoming and outgoing packets.

Thanks to this plug-compatible component model and fine-grained management and control of the data plane, extending a protocol stack with load management and/or component hot-swapping becomes much easier and understandable (see Sct. 2).

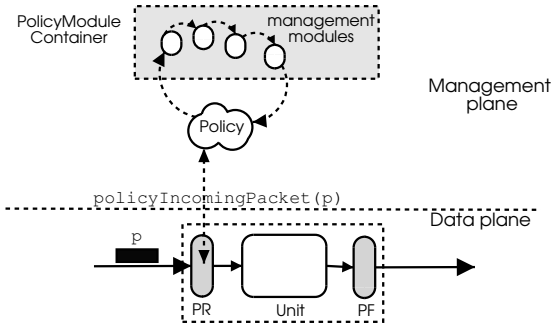

Fig. 3. A DiPS+ component (consisting of a packet receiver, the core unit, and a packet forwarder) with a policy object that intercepts incoming packets (p). The policy delegates incoming packets to a pipeline of management modules.

3.3 Explicit Concurrency Components

Finally, to separate the employed concurrency model of a DiPS+ stack from basic functionality, functional components are complemented with concurrency components. This allows for a developer to concentrate on the concurrency aspect of a DiPS+ stack, without being discarded by other concerns scattered across the same functional stack and vice versa.

A concurrency component allows to increase or decrease the level of parallelism in the component area behind it. In addition, it controls which requests are scheduled and when. Each concurrency component breaks the pipeline into two independent component groups, which will be referred to as component areas (see also Fig. 1).

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 plug-compatible components. As a result, 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.

The design of the concurrency component consists of three major entities: a packet queue, one or more packet handlers, and the scheduler strategy. Its behavior during overload or underload can be customized via its management interface (see Figure 4), which allows to register specific overflow and underflow strategies. In this way, the concurrency component can be controlled without exposing its internal attributes (such as the packet queue). A packet handler is a thread that guides a packet through the component area behind its concurrency

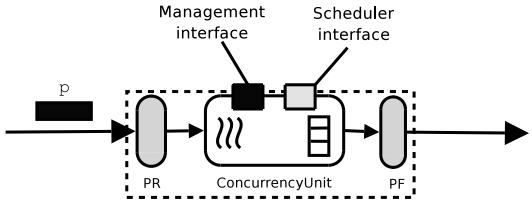

Fig. 4. A DiPS+ concurrency component with its management and scheduler interface.

component. The scheduler strategy of a concurrency component decides which packet will be selected next from the packet queue. The scheduler strategy can be customized via the scheduler interface of a concurrency component (see Fig. 4).

Advantages. Having explicit concurrency components shows three major advantages. First of all, it allows not only to reuse functional 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 in the protocol stack. 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 this way, packet processing can be adapted to both request-specific information (e.g. content type, size, or sender) and the system's state (e.g. available resources) as the packet traverses the component pipeline.

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 I/O bottlenecks, i.e. component areas that are overwhelmed because too many arriving packets require I/O access. 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 are about to release resources that have become scarce.

4 Management Plane for Load Management

As a first validation of the flexibility of the abstractions offered in DiPS+, we illustrate how a DiPS+ composition is extended with load management support in a modular manner.

The need for load management (as described in Sct. 2.1) has resulted in the DMonA (*D*ips+ *Mon*itoring *A*rchitecture) management plane, which controls

and customizes the behavior of the protocol stack. DMonA allows for handling certain overload situations in an application-specific manner via interventions at protocol stack level. These interventions focus on packet classification, controlling the packet arrival rate, and optimally distributing processing threads over the tasks to be executed.

DMonA is a feedback-driven management platform. This means that DMonA (1) extracts information from the underlying protocol stack (via the policy associated with a PR and/or PF), (2) decides whether or not action must be taken (using a monitor policy), and (3) deploys this solution in the protocol stack.

The rest of this section describes how DMonA handles load management, viewed from three complementary perspectives: packet classification, request control, and concurrency control.

4.1 Packet Classification

Packet classification differentiates between packets based on meta-information that is collected in each packet as it traverses the protocol stack. By consequence, the further a packet has traversed the component pipeline, the more meta-information is available for its classification. Packet differentiation can be based, for instance, on parameters such as destination, data size, encapsulated protocol, packet type (connection establishment or data transfer), or on application-specific preferences passed via meta-information.

Packet classification is highly relevant when different categories or types of packets can be recognized, and service quality should be guaranteed for specific categories. During overload, the most important packets can be handled with priority.

Packet classification can easily be added to a protocol stack thanks to three abstractions offered in the DiPS+ component platform: meta-information, dispatchers, and concurrency components. Meta-information is used by applications or components to annotate packets. These annotations influence how dispatchers and concurrency components process packets. A dispatcher is associated with a specific classification strategy, which is used to demultiplex the component pipeline in parallel sub-pipelines based on meta-information. A concurrency component for its part encapsulates a packet buffer and a specific scheduler strategy, which decides what packet to process next from the buffer. Either, the dispatcher can delegate packets to different concurrency components, one for each category; in this case, the packet scheduler selects packets from multiple queues. Or, the dispatcher can delegate packets to one ordered buffer that puts high priority packets first; in this case the packet scheduler is associated with one packet buffer and fetches packets in priority order.

Given the flexibility of DiPS+, DMonA support can be limited in order to allow for system administrators to install specific classification strategies (in the dispatchers) and scheduler strategies (in the concurrency components). Packet classification has been validated in the context of an industrial case study that

customized the RADIUS authentication protocol so as to differentiate between gold, silver, and bronze types of users [5, 6].[1]

4.2 Controlling Arrival Rate

From a request control perspective, system load is managed by limiting or shaping the arrival rate of new requests to a sustainable level. Such traffic control may, for instance, selectively drop low-priority packets to preserve processing resources for the most important requests. This is crucial when too much requests arrive to be handled by the available processing resources.

Request control is highly relevant to protect the system from packet bursts and to allow for it to handle them gracefully by removing incoming packets early in the processing pipeline (e.g. in the protocol stack of the system). In addition, by prioritizing packets based on packet- and application-specific knowledge, the least important packets are removed first.

Traffic control has been effectively employed in networks, for example, to provide applications with quality-of-service guarantees by individually controlling network traffic flows (also known as traffic shaping) [21]. Typically, a leaky bucket algorithm [22] is used to adjust the rate at which incoming packets are forwarded. In addition, a variety of performance metrics have been studied in the context of overload management, including throughput and response-time targets [23–25], CPU utilization [26–28] and differentiated service metrics based on a given performance target [29, 30]. Welsh [23] proposes the 90^{th} percentile response-time as a realistic and intuitive measure of client-perceived system performance. It is defined as follows: if the 90^{th} percentile response-time is t, then 90% of the requests experience a response-time equal to or shorter than t.

When applying DMonA in the context of traffic control, we need to provide information collectors (i.e. sensors) at the entry of a monitored component area, a monitor policy that decides on the actions to be taken, and a component area to be controlled. As a concrete example, we use the 90^{th} percentile approach of Welsh [23]. First of all, a response-time sensor measures the response-times for packets passing through a component area. Such sensors are installed at each concurrency component's packet forwarder and determine how long it takes between a request leaving the concurrency component and the release of the associated thread. A DMonA information collector collects the response-times of all packets that have passed through a component area. Secondly, the 90^{th} percentile algorithm itself is offered as a monitor policy, which processes the collected information at regular times. In this case the algorithm checks whether 90% of the packets experience a response-time equal to or shorter than some pre-defined threshold t. Thirdly, the leaky bucket controls the admission rate of packets entering the monitored area. The leaky bucket is installed as a management

[1] This research has been carried out in order of Alcatel Bell as 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).

module, associated with the packet receiver policy of the concurrency component in front of the area under control. This packet receiver is the perfect place for such control, since it represents the entry of a component area.

4.3 Concurrency Control

While packet classification and request control focus on packets, concurrency control focuses on the tasks to be executed in the protocol stack. From a concurrency perspective, load management distributes the available processing power (i.e. threads) across the system's component areas (tasks) such that the overall system performance is optimized [31]. This means that the DMonA 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 should solve these bottlenecks by migrating processing resources associated with the concurrency component in front of a component area, from underloaded to overloaded component areas.

Concurrency control is an effective technique for load management, since it allows to control how processing threads are applied at any time (e.g. to handle the highest priority tasks first), and compensates for blocking invocations inside the protocol stack.

Because our approach allows for concurrency components to be added at arbitrary places in the protocol stack, bottleneck areas can easily be detected by measuring the throughput of each area. In addition, concurrency components allow for handling bottlenecks intelligently by increasing or decreasing the number of associated packet handler threads in certain component areas, which can be highly effective for parallel areas with blocking components. Moreover, as already described in Section 4.1, concurrency components support packet classification via their specific scheduling strategy.

More specifically, DMonA monitors the packet stream by installing throughput sensors, i.e. management modules that count the number of passed packets. Figure 5 shows how sensors are plugged in at both the packet receiver and forwarder of a concurrency component. The DMonA monitor collects on a regular basis the information stored in both sensors and resets them to start the next collecting phase. One possible monitor policy adjusts thread scheduling based on the concurrency component's progress [4], comparable to the feedback-driven approach proposed by Steere [32]. Based on this status information, the DMonA monitor decides *when and how* to adapt local concurrency behavior to improve performance. Proposed monitor decisions can be deployed in two ways. On the one hand, a concurrency component can be linked with or unlinked from a packet handler thread. This is done via the concurrency unit's scheduler interface (see Figure 5). On the other hand, the buffer overflow and underflow strategies of a concurrency component can be replaced by calling its management interface.

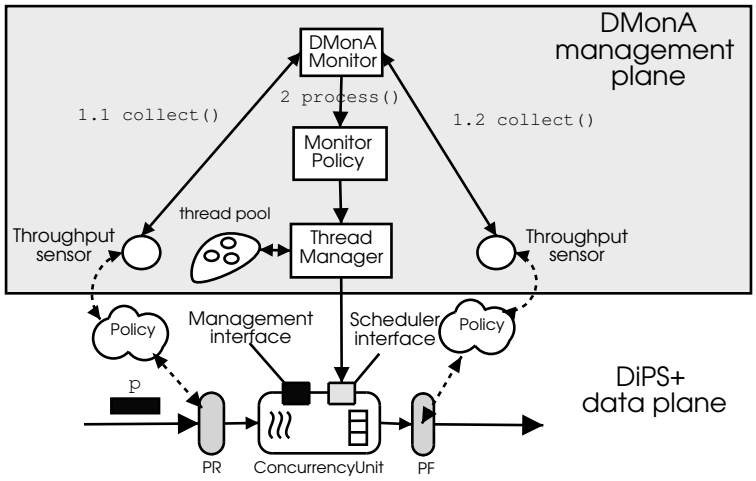

Fig. 5. Illustration of DMonA attached to DiPS+ via two policies, one associated with the packet receiver and one with the packet forwarder. Processing resources are retrieved from a pool of free resources and allocated to a concurrency unit via its scheduler interface.

5 Management Plane for Transparent Component Hot-Swapping

The need for run-time adaptable protocol stacks (as described in Section 2.2) has resulted in the development of the **CuPS** (*Cu*stomizable *P*rotocol *S*tack [2]) platform, a modular extension to the DiPS+ framework responsible for conducting seamless reconfigurations of a running protocol stack (illustrated in Figure 6). Since we aim for unanticipated adaptations, protocol stack reconfigurations imply *changing a stack composition*, rather than being limited to parameter tuning.

The algorithm employed by the CuPS platform to orchestrate a reconfiguration of a protocol stack composition at run-time, involves three stages:

Installation of new component area. The adaptation process starts with the installation of the new functional components, resulting in the co-existence of the old component area (still in use) and the new version (not yet activated).

Activation of new component area. Next, the newly installed functional components become activated. This is achieved through stopping and disconnecting the old component area by redirecting packets towards the new version. At this point in the adaptation procedure, the new component area is plugged into the stack composition and will process transmitted packets.

[2] This research has been carried out with financial support of the Fund for Scientific Research Flanders, Belgium – F.W.O. RACING #G.0323.01

Removal of old component area. Finally, the old component area is removed. Since it has been stopped during the activation stage, it can safely be removed.

In the remainder of this section, we elaborate on the *activation*[3] *phase* to illustrate how an existing DiPS+ composition is extended with CuPS support in a *modular* manner.

5.1 Self-contained Components in a Best-effort Environment

A first category of reconfigurations encloses the deployment of component areas strictly composed of functional components that are *self-contained*, i.e. components not depending on cooperation with other components to implement a service. Two examples of such self-contained protocol stack components are a filter component to relieve a congested node and a logging component. In addition, this class of reconfigurations assumes for packet loss or packet scrambling not to compromise the correct functioning of the network. Since performance (throughput) is an important characteristic of a protocol stack, most network protocols (such as IP) offer best-effort services and as such comply with this requirement.

When both conditions are fulfilled, activating such a component area boils down to adapting the current composition. No additional support is needed to control the state (activity) of the DiPS+ component area that is subject to activation. By consequence, the activation phase is limited to removing the connectors binding the old component area into the protocol stack, and plugging in the new area. With this, packets that are processed by the old component area during the activation stage (depending on the employed concurrency model) will get lost. Note that due to the use of plug-compatible components, the actual recomposition of DIPS+ component areas has been reduced to a trivial problem of adding and removing connectors.

5.2 Self-contained Components Demanding Safe Deployment

Depending on the properties of the network service that is subject to adaptation, packet loss during protocol reconfiguration could compromise the correct functioning of the protocol. As an example, we refer to the adaptation of a running TCP stack. When packets are lost during the activation process, TCP will consider these errors as packet loss due to congestion and hence will reduce its congestion window [33]. This will cause a substantial degradation of performance in terms of throughput, even though sufficient bandwidth might be available.

As such, this family of protocol stack reconfigurations covers seamless adaptation of self-contained components, enforcing a *safe state* to be imposed on the component area under change. Since no other components depend on self-contained components to complete a service, such a safe state for reconfiguration

[3] Henceforth the terms *reconfiguration* and *adaptation* are used as an alternative for *activation*.

is obtained when the component area is made *passive*. This implies that the functional components (1) are currently not processing any packets and (2) have no pending packets to be accepted and processed. In this way, packet loss caused by packets being processed while a component is swapped can be prevented.

1) Packet Blocking. By consequence, CuPS support is needed to block packet flows before passing through a DiPS+ component area facing a reconfiguration. This is achieved by holding up all outgoing packets of adjacent packet forwarders directed to the component area that is subject to adaptation. When the reconfiguration is completed, the execution of these blocked packets will be resumed. To extend the targeted DiPS+ components with such blocking support in a modular and transparent manner, their packet forwarders are equipped with special `Policy` objects for intercepting packets (conducted by the CuPS platform).

The employed separation between the functionality of DiPS+ components (offered by a programmer) on the one hand and additional CuPS support to deactivate other components on the other hand, has a number of advantages:

First of all, minimal interference with the rest of the system can be guaranteed. Interrupting interactions in a composition can be restricted to those locations where an actual reconfiguration is needed. Instead of stopping the concurrency components (as proposed in [34]), only the adjacent DiPS+ components that initiate interactions (by forwarding packets) on the component that is subject to adaptation need to be blocked (as illustrated in Fig. 6). With this, conducting a safe reconfiguration does not depend on the employed concurrency model implemented by the number of concurrency components and their location (controlled by the DMonA platform). This implies that CuPS and DMonA can operate simultaneously, but independently from each other, sharing the same DiPS+ protocol stack.

Secondly, due to separating support to block outgoing packets from the functional behavior of a DiPS+ component, changing the way of holding up packets at the packet forwarder will not interfere with existing component functionality and vice versa. As an example, we demonstrate the possibility to choose between two different blocking strategies. To obtain a safe reconfiguration, one could decide to *block the execution thread* in which the outgoing packets are initiated, using a `ThreadBlockingPolicy`. An alternative could be to *queue outgoing packets* without interrupting the execution thread by selecting the `PacketQueueingPolicy`. Such a change can be achieved transparently by only adapting the packet forwarders of the DiPS+ components that are involved.

Finally, the impact of a blocking operation on a DiPS+ component can be made more fine-grained. Instead of stopping all interactions initiated by a component (e.g. by interrupting the execution thread of that component), only the packet forwarders initiating interactions that engage components that need to become passive should get blocked (illustrated by `Component A` in Fig. 6). In this way, packets that are sent out using other packet forwarders can still be initiated.

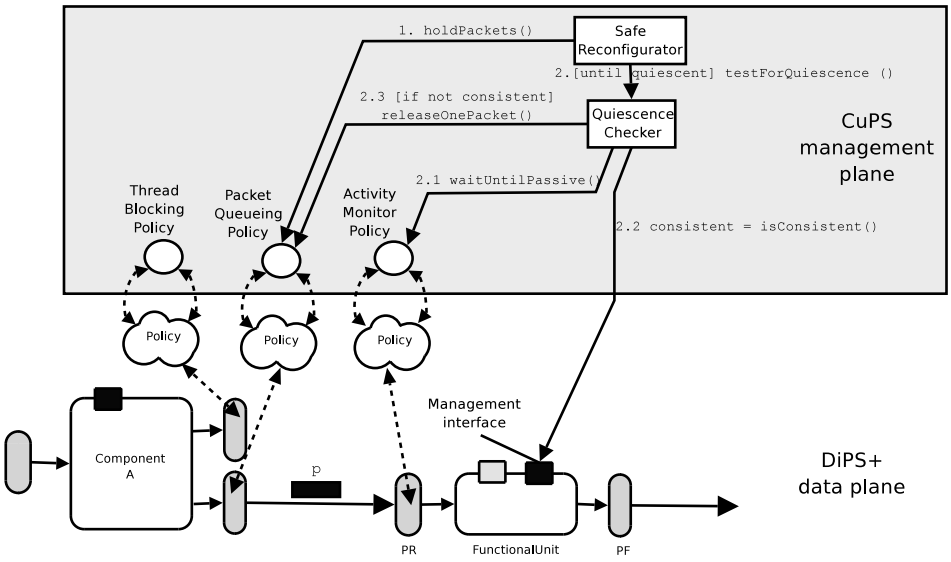

Fig. 6. Illustration of CuPS attached to DiPS+.

2) Activity Monitoring. In addition to holding up packets to be accepted and processed by the component area subject to adaptation, safe adaptation also requires this component area to be inactive (i.e. currently not processing any packets).

Due to the reactive behavior of a functional component, monitoring code to check whether such a DiPS+ component is *active* or *idle* can (automatically) be added by simply extending the policy employed by the packet receiver of this component. In case of concurrent interactions, activity inside a DiPS+ component can be monitored by means of a counter situated at its packet receivers, which is incremented on invocation and decremented upon return [35] (illustrated by means of the `ActivityMonitorPolicy` in Fig. 6). When only sequential interactions are used, a counter can be replaced by a boolean flag. This reduces the monitoring overhead for each interaction.

5.3 Safe Deployment of Tightly-coupled Components

The last category of reconfigurations encloses the activation of component areas containing tightly coupled components, i.e. components depending on cooperation with other components (locally, or in a distributed fashion) to implement a service. This cooperation is formalized by means of a *transaction*, consisting of a sequence of one or more asynchronous *interactions*. Referring to a fragmentation service, a transaction to fragment and reassemble a packet encapsulates a number of interactions, each representing the transfer of one fragment (packet) from a `fragmenter` to a `reassembler`. This cooperation implies that, from a

reconfiguration point of view, the cooperating components are only *consistent* after termination of a transaction (i.e. when all fragments are received by the `reassembler` and the original packet has been restored). As a consequence, when imposing a safe state for reconfiguration of a tightly coupled component, the dependencies formalized by the transaction should be taken into account.

Kramer and Magee [34] have stated that achieving safe software reconfigurations requires the software modules that are subject to adaptation (in this context the `reassembler`) to be both *consistent* and *frozen* (passive). When software modules are consistent, they do not include results of partially completed services (or transactions). By forcing software modules to be frozen (passive), state changes caused by new transactions are impossible. Kramer and Magee describe this required consistent and frozen state as the *quiescence* of a component.

As stated in the previous section, forcing a component area to be frozen has been accomplished (in a modular manner) by separating the functional behavior of a module from potential support to block its outgoing interactions. Since there is no knowledge about the state of the tightly coupled component at the moment packets are blocked, reconfiguration may lead to inconsistency (caused by replacing the component when protocol transactions are only partially completed). When referring to the fragmentation service, replacing the `reassembler` when it has not yet received all fragments (and thus could not reassemble the original packet) will break the consistency between fragmenting and reassembling component (and in that way, the correct functioning of the fragmentation service). By consequence, additional support is required to drive a component area into a consistent state. This has been achieved by extending DiPS+ packet forwarders with special policies allowing "controlled" packet blocking support. After blocking the packet forwarders that are directing packets to the component that will be replaced, it should be possible for the CuPS platform (which conducts the actual reconfiguration) to check whether *safe reconfiguration* of the component is achievable. When this is not the case, blocked interactions are resumed one by one until the required safe state for reconfiguration is attained.

Checking whether safe reconfiguration of a component is achievable requires *verification of its execution state*. For that purpose, we have extended tightly-coupled DiPS+ components (that are eligible for reconfiguration) with monitoring code to reflect their current execution state. More in detail, verifying the internal state of a DiPS+ component is achieved by checking its internal `Unit` through introspection via the `ManagementInterface`. CuPS will only check this state in the face of an actual reconfiguration when the targeted component is idle.

```
<layer name="ipv4layer">
<!-- down going path -->
  <component class="dips.protocol.ip.ipv4.IPv4OutRouter"
       name="outrouter"/>
  <component class="dips.protocol.ip.ipv4.IPv4HeaderConstructor"
       name="hdrcons"/>
  <upperentrypoint name="outrouter"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
       name="cd1"     from="outrouter"     to="hdrcons"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
       name="cd2"     from="hdrcons"       to="downdelivery"/>
<!-- Up going path -->
  <component class="dips.protocol.ip.ipv4.IPv4HeaderParser"
       name="hdrpars"/>
  <component class="dips.repository.concurrencyunit.ActiveUnit"
       name="active_comp"/>
  <dispatcher class="dips.protocol.ip.ipv4.LocalForwardDispatcher"
       name="inrouter"/>
  <dispatcher output="local"  name="inrouter" to="updelivery"/>
  <dispatcher output="forward" name="inrouter" to="outrouter"/>
  <lowerentrypoint name="hdrpars"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
       name="cu1"     from="hdrpars" to="active_comp"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
       name="cu2"     from="active_comp" to="inrouter"/>
</layer>
```

Listing 1.1. An example of the layer property description for the DiPS+ IPv4 layer.

6 DiPS+ Prototype and Validation

6.1 Prototype

To validate the DiPS+ component platform and its potential for supporting run-time adaptability, we have developed a proof-of-concept prototype in Java, running on standard PC hardware.[4]

The DiPS+ prototype allows for building a protocol stack from an architecture specification. The DiPS+ architecture is represented in XML [36]. This representation specifies the core architecture entities, like components and protocol layers, along with how these entities are interconnected. To this end, descriptions for component connectors and layer glues, dispatchers and concurrency components are provided as well.

By way of example of a DiPS+ description, Listing 1.1 zooms in on the IP layer of a protocol stack. It lists all essential elements layers can be composed

[4] The protocol stack in Java is integrated in the Linux OS using a virtual Ethernet device (via the ethertap module in the Linux kernel).

of: components (lines 3, 3–6, and 13), connectors (lines 8–11, and 22–24), a dispatcher (lines 17–20), a concurrency component (line 15), and the upper and lower entry point (lines 7 and 21). Each of these items is represented as such in the architecture description, which makes the listing self-explaining.

Having an architecture description separated from the implementation has major advantages. First of all, the internals of the DiPS+ platform are transparent for the protocol stack developer, resulting in a black-box framework. Developing a DiPS+ protocol stack boils down to designing the appropriate components and providing in a correct composition description. A stack builder tool is used to automatically transform the architecture descriptions into a running protocol stack. By consequence, a developer can configure different compositions without having to write extra code, or to change or recompile the source code of individual components.

A second advantage is that the use of an architectural description allows for specific (optimizing or test) builders to be applied to the same architecture description. Testing a protocol layer in isolation, for instance, reuses the architecture description, but creates the layer in a different context (i.e. a test case instead of a protocol stack).

Finally, an architecture description allows for optimization, in the sense that an optimizer can analyze the architecture and change it in order to become more efficient. When, for instance, a network router is known to be connected to two networks with the same maximum segment size, it can be reconfigured to omit reassembly and refragmentation of forwarded packets, since they are fragmented in sufficiently small pieces already. Only packets for local delivery must be reassembled in this case.

6.2 Validation

We have successfully validated the DiPS+ approach, a.o. in an industrial case study that compared a DiPS+ and a commercial Java implementation of the RADIUS authentication and authorization protocol [5, 6].[5] Performance results clearly show the advantage of using application-specific scheduling strategies during overload. Moreover, the DiPS+ RADIUS server is able to gracefully cope with varying (over)load conditions. DiPS+ did not only facilitate the development of the RADIUS protocol, it also allowed to experiment with different scheduling strategies without having to change any functional code.

In addition, the DiPS+ framework has been validated in the context of on-demand composition of a protocol stack, based on application-specific requirements.[6] We have built a prototype in DiPS+ that allows an application to express

[5] 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).

[6] This research has been carried out in the context of the ITEA/PEPiTA project and supported by the Flemish Institute for the advancement of scientific-technological research in the industry (IWT PEPiTA #990219).

high-level service requirements (e.g. reliability of data transfer, encryption, local or networked transfer, etc.) that must be supported by the underlying protocol stack. Based on these requirements, a combination of protocol layers is suggested by a stack composition tool [8, 9], and a protocol stack is built by the DiPS+ builder [37]. This illustrates the flexibility of the DiPS+ platform.

Multiple Master's theses have explored and validated the DiPS+ component platform from various perspectives. First of all, DiPS+ has been used to design and implement particular protocols (e.g. SIP [38], IPv6 [39], a TCP booster [40], dynamic routing protocols in [41, 42], an IPSec based VPN solution [43] and a stateful firewall in [44]). Secondly, DiPS+ has been applied in various domains to explore its applicability. The work in [45], for instance, describes how network management techniques can be used in combination with DiPS+ protocol stacks. Finally, more research related theses have explored fundamental extensions to the DiPS+ component framework and architecture (e.g. self-adaptability [46], and concurrency control [47]).

Two main conclusions may be drawn from our experiences in guiding Master students during their thesis. First of all, the DiPS+ framework and architecture can quickly be assimilated, even by students with limited experience in software architectures and a mainly object-oriented design background. Nevertheless, creating a high-level modularized DiPS+ design of a network protocol was not always trivial, and sometimes required assistance of a DiPS+ team member to put the student on the right track. In our view, the students' lack of design experience and the often poor documentation of protocol specifications lie at the basis of the complicated modularization process. The main advantage, compared to an object-oriented design, is that packet flows are clearly defined and well-identifiable, which makes a DiPS+ design much more understandable. Once the high-level design becomes clear, development of individual components is straightforward. Secondly, the theses show that DiPS+ allows for a highly incremental software development process. Stated differently, the first running prototype can usually be delivered quickly after implementation has started (i.e. after a few weeks). From then on, the prototype can easily be customized and extended towards the stated requirements.

Although the DiPS+ component framework has been proposed in the context of protocol stacks, we are convinced of its applicability in other operating system domains. The first research results in the context of USB device drivers [48, 49] and file systems [31, 46] are very promising. These systems reflect a layered architecture, which perfectly matches the DiPS+ architecture.

7 Related Work

7.1 Protocol Stack Frameworks

Although multiple software design frameworks for protocol stack development have been described in the literature [50–53], we compare the DiPS+ approach to three software architectures, which are tailored to protocol stacks and/or concurrency control: SEDA [23], Click modular router [54], and Scout [55].

SEDA [23] offers an event-based architecture for supporting massively concurrent web servers. A stage in SEDA can be compared with a DiPS+ component area along with its preceding concurrency component. Yet, 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. In addition, SEDA does not provide developers with an architecture specification, which makes it difficult for developers to understand the data and control flow through the set of independent stages.

The Click modular router [54] is based on a design very analogous to DiPS+. Although one can recognize a pipe-and-filter architectural style, Click pays much less attention to software architecture than DiPS+. 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 exlicit concurrency components.

The Scout operating system [55] uses a layered software architecture, yet does not offer fine-grained entities such as components for functionality, dispatching, or concurrency. 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.

7.2 Concurrency and Separation of Concerns

A critical element in our research is the separation of concurrency from functional code. Kiczales [56] defines non-functional concerns as *aspects* that crosscut functional code. An aspect 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), aspects tend to disappear at run-time, which makes it very difficult (if not impossible) to adapt aspects dynamically. Apertos [17] introduces concurrent objects that separate mechanisms of synchronization, scheduling and interrupt mask handling from the functional code. This makes software more understandable, and reduces the risk of errors.

8 Conclusion

Our contribution represents a successful case study, DiPS+, on the development of component-based software for protocol stacks that are adaptable at run-time. The employed architectural styles and the resulting component abstractions (1) increase the flexibility and adaptability of protocol stack software and (2) facilitate the development process of software that is complex and error-prone by nature, especially when additional concerns (such as the need for run-time adaptability) are imposed.

The combination of the pipe-and-filter and the blackboard architectural style has resulted in the design of plug-compatible DiPS+ components. By consequence,

DiPS+ components are unaware of other components they are connected to, directly or indirectly. This is a major advantage in terms of flexibility, as it allows for individual components to be reused in different compositions. In addition, by employing these architectural styles (together with the layered style), the DiPS+ platform offers a number of framework abstractions (such as components, connectors, and packets) to ease the development of adaptable protocol stacks. Finally, separate component types for functionality, concurrency and packet dispatching allow for a developer to concentrate on a single concern (e.g. concurrency) without being distracted by other concerns that are scattered across the same functional code.

As stated in the introduction, a second objective of the DiPS+ component platform is to allow for modular integration of non-functional extensions that cross-cut the core protocol stack functionality. In this article, we have illustrated (by means of DMonA and CuPS) that the use of explicit communication ports is essential to transparently extend a DiPS+ protocol stack with support for controlling the packet flow. More precisely, they serve as hooks for connecting the data and the management plane.

We have discussed our experiences with using the DiPS+ component platform in real-life situations. Although a seamless transformation of the DiPS+ platform towards embedded systems is not yet feasible, we argue that the principles behind DiPS+ (i.e. a combination of component-based development and separation of concerns) are crucial for component-based embedded network systems. In our opinion, this combination does not only facilitate the implementation of component hot-swapping and concurrency control (as we have demonstrated), but also seems very useful for other concerns such as on-demand and safe software composition [8, 9, 57], transparent data flow inspection [23], performance optimization [54], isolated and incremental unit testing [7], and safe updates of distributed embedded systems [10].

In our opinion, such concerns as data flow monitoring, component hot-swapping, unit testing, performance optimization, and safe composition are crucial for embedded software and will become even more so with the ongoing trend towards mobile and ad-hoc network connectivity of (highly heterogeneous) embedded devices. We hope that this case study can convince embedded system developers of the need and the power of a well-defined software architecture and component platform.

References

1. Hubaux, J.P., Gross, T., Boudec, J.Y.L., Vetterli, M.: Towards self-organized mobile ad hoc networks: the Terminodes project. IEEE Communications Magazine **31** (2001) 118–124
2. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice-Hall (1996)
3. Schneider, J.G., Nierstrasz, O.: Components, scripts and glue. In L. Barroca, J.H., Hall, P., eds.: Software Architectures – Advances and Applications. Springer-Verlag (1999) 13–25

4. Michiels, S.: Component Framework Technology for Adaptable and Manageable Protocol Stacks. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2003)
5. Michiels, S., Desmet, L., Joosen, W., Verbaeten, P.: The DiPS+ software architecture for self-healing protocol stacks. In: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4), Oslo, Norway, IEEE/IFIP, IEEE (2004)
6. Michiels, S., Desmet, L., Verbaeten, P.: A DiPS+ Case Study: A Self-healing RADIUS Server. Report CW-378, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium (2004)
7. Michiels, S., Walravens, D., Janssens, N., Verbaeten, P.: DiPS: Filling the Gap between System Software and Testing. In: Proceedings of Workshop on Testing in XP (WiTXP2002), Alghero, Italy (2002)
8. Şora, I., Verbaeten, P., Berbers, Y.: A description language for composable components. In: Proceedings of 6th International Conference on Fundamental Approaches to Software Engineering (FASE 2003). Volume 2621., Warsaw, Poland, Springer-Verlag, Lecture Notes in Computer Science (2003) 22–36
9. Şora, I., Cretu, V., Verbaeten, P., Berbers, Y.: Automating decisions in component composition based on propagation of requirements. In: Proceedings of 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004), Barcelona, Spain (2004)
10. Janssens, N., Steegmans, E., Holvoet, T., Verbaeten, P.: An Agent Design Method Promoting Separation Between Computation and Coordination. In: Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004), ACM Press (2004) 456–461
11. Joosen, W.: Load Balancing in Distributed and Parallel Systems. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (1996)
12. Wetherall, D., Legedza, U., Guttag, J.: Introducing new internet services: Why and how. IEEE Network, Special Issue on Active and Programmable Networks **12** (1998)
13. Solis, I., Obraczka, K.: Flip: a flexible interconnection protocol for heterogeneous internetworking. Mob. Netw. Appl. **9** (2004) 347–361
14. Campbell, A.T., De Meer, H.G., Kounavis, M.E., Miki, K., Vicente, J.B., Villela, D.: A survey of programmable networks. SIGCOMM Comput. Commun. Rev. **29** (1999) 7–23
15. Janssens, N., Michiels, S., Mahieu, T., Verbaeten, P.: Towards Transparent Hot-Swapping Support for Producer-Consumer Components. In: Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003), Warsaw, Poland (2003)
16. Janssens, N., Michiels, S., Holvoet, T., Verbaeten, P.: A Modular Approach Enforcing Safe Reconfiguration of Producer-Consumer Applications. In: Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004), Chicago Illinois, USA (2004)
17. Itoh, J., Yokote, Y., Tokoro, M.: Scone: using concurrent objects for low-level operating system programming. In: Proceedings of the tenth annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), Austin, TX, USA, ACM Press, New York, NY, USA (1995) 385–398
18. Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.C.: Open implementation design guidelines. In: Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston, MA, USA, ACM Press, New York, NY, USA (1997) 481–490

19. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA (1991)
20. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1998)
21. Breslau, L., Jamin, S., Schenker, S.: Comments on the performance of measurement-based admission control algorithms. In: Proceedings of IEEE INFOCOM 2000. (2000) 1233–1242
22. Tanenbaum, A.S.: Computer Networks. Prentice Hall (1996)
23. Welsh, M.F.: An Architecture for Highly Concurrent, Well-Conditioned Internet Services. PhD thesis, University of California at Berkeley, Berkeley, CA, USA (2002)
24. Chen, H., Mohapatra, P.: Session-based overload control in QoS-aware web servers. In: Proceedings of IEEE INFOCOM 2002, New York, NY, USA (2002)
25. Chen, X., Mohapatra, P., Chen, H.: An admission control scheme for predictable server response time for web accesses. In: Proceedings of the tenth international conference on World Wide Web, ACM Press, New York, NY, USA (2001) 545–554
26. Abdelzaher, T.F., Lu, C.: Modeling and performance control of internet servers. Invited paper at 39th IEEE Conference on Decision and Control (2000)
27. Cherkasova, L., Phaal, P.: Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, HP labs (1998)
28. Diao, Y., Gandhi, N., Hellerstein, J.L., Parekh, S., Tilbury, D.: Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In: Proceedings of Network Operations and Management Symposium, Florence, Italy (2002)
29. Kanodia, V., Knightly, E.: Multi-class latency-bounded web services. In: Proceedings of 8th IEEE/IFIP International Workshop on Quality of Service (IWQoS 2000), Pittsburgh, PA, USA (2000)
30. Lu, C., Abdelzaher, T., Stankovic, J., Son, S.: A feedback control approach for guaranteeing relative delays in web servers. In: Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS), Taipei, Taiwan (2001)
31. Michiels, S., Desmet, L., Janssens, N., Mahieu, T., Verbaeten, P.: Self-adapting concurrency: The DMonA architecture. In Garlan, D., Kramer, J., Wolf, A., eds.: Proceedings of the First Workshop on Self-Healing Systems (WOSS'02), Charleston, SC, USA, ACM SIGSOFT, ACM press (2002) 43–48
32. Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., Walpole, J.: A feedback-driven proportion allocator for real-rate scheduling. In: Proceedings of the third USENIX Symposium on Operating Systems Design and Implementation (OSDI'99), New Orleans, LA, USA, USENIX Association, Berkeley, CA, USA (1999) 145–158
33. Hoebeke, J., Leeuwen, T.V., Peters, L., Cooreman, K., Moerman, I., Dhoedt, B., Demeester, P.: Development of a TCP protocol booster over a wireless link. In: Proceedings of the 9th Symposium on Communications and Vehicular Technology in the Benelux (SCVT 2002), Louvain la Neuve (2002)
34. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306
35. McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., Marlet, R.: Specialization tools and techniques for systematic optimization of system software. ACM Transactions on Computer Systems **19** (2001) 217–251

36. Harold, E.R., Means, W.S.: XML in a Nutshell. Second edn. O'Reilly & Associates, Inc. (2002)
37. Michiels, S., Mahieu, T., Matthijs, F., Verbaeten, P.: Dynamic Protocol Stack Composition: Protocol Independent Addressing. In: Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001), Budapest, Hungary, SERVITEC (2001)
38. Vandewoestyne, B.: Internet Telephony with the DiPS Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2003)
39. Janssen, G.: Implementation of IPv6 in DiPS. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2002)
40. Larsen, T.: Implementation of a TCP booster in DiPS. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2004)
41. Buggenhout, B.V.: Study and Implementation of a QoS router. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2001)
42. Elen, B.: A flexible framework for routing protocols in DiPS. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2004)
43. Vandebroek, K.: Development of an IPSec based VPN solution with the DiPS component framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2004)
44. Cornelis, I., Weerdt, D.D.: Development of a stateful firewall with the DiPS component framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2004)
45. Bjerke, S.E.: Support for Network Management in the DiPS Component Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2002)
46. Desmet, L.: Adaptive System Software with the DiPS Component Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2002)
47. Michiels, D.: Concurrency Control in the DiPS framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2003)
48. Coster, W.D., Krock, M.D.: CoFraDeD: a Component Framework for Device Drivers. Technical report, internal use only, PIMC/K.U.Leuven, Dept. of Computer Science, Leuven, Belgium (2001)
49. Michiels, S., Kenens, P., Matthijs, F., Walravens, D., Berbers, Y., Verbaeten, P.: Component Framework Support for developing Device Drivers. In Rozic, N., Begusic, D., Vrdoljak, M., eds.: International Conference on Software, Telecommunications and Computer Networks (SoftCOM). Volume 1., Split, Croatia, FESB (2000) 117–126
50. Hutchinson, N.C., Peterson, L.L.: The x-kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering **17** (1991) 64–76
51. Bhatti, N.T.: A System for Constructing Configurable High-level Protocols. PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA (1996)
52. Ballesteros, F.J., Kon, F., Campbell, R.: Off++: The Network in a Box. In: Proceedings of ECOOP Workshop on Object Orientation in Operating Systems (ECOOP-WOOOS 2000), Sophia Antipolis and Cannes, France (2000)
53. Hüni, H., Johnson, R.E., Engel, R.: A framework for network protocol software. In: Proceedings of the tenth annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), Austin, TX, USA, ACM Press, New York, NY, USA (1995) 358–369

54. Kohler, E.: The Click Modular Router. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA (2001)
55. Montz, A.B., Mosberger, D., O'Malley, S.W., Peterson, L.L.: Scout: A communications-oriented operating system. In: Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island, WA, USA, IEEE Computer Society Press (1995) 58–61
56. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Akşit, M., Matsuoka, S., eds.: Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241 of LNCS. Springer-Verlag, Jyväskylä, Finland (1997) 220–242
57. Desmet, L., Piessens, F., Joosen, W., Verbaeten, P.: Improving software reliability in data-centered software systems by enforcing composition time constraints. In: Proceedings of Third Workshop on Architecting Dependable Systems (WADS2004), Edinburgh, Scotland (2004) 32–36