DiPS/CuPS: a Framework for Runtime Customizable Protocol Stacks

Nico Janssens *Sam Michiels* *Pierre Verbaeten*

Report CW 328, Nov 2001



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

DiPS/CuPS: a Framework for Runtime Customizable Protocol Stacks

Nico Janssens *Sam Michiels* *Pierre Verbaeten*

Report CW 328, Nov 2001

Department of Computer Science, K.U.Leuven

Abstract

In this paper we present a design for runtime Customizable Protocol Stacks, which we call CuPS. CuPS is an add-on to DiPS (Distrinet Protocol Stack framework), a component framework for developing open protocol stacks. The design of the DiPS framework allows one to plug in meta-level extensions. CuPS is such a meta-level extension that permits dynamic, unanticipated customizations such as adding, removing and replacing components. We evaluate and prove the necessity of our customizable open protocol stack approach with the design and implementation of a reliability protocol, which can dynamically be adapted to improve different network characteristics (such as packet loss, jitter and communication delay). The results clearly show that application performance can benefit from runtime customizations at protocol stack level. We strongly believe that a combination of the service level flexibility (on top of a protocol stack) of traditional active network approaches and the CuPS approach can result in the ultimate level of network flexibility.

Keywords : Active Networks, Components, Architecture.

DiPS/CuPS: a Framework for Runtime Customizable Protocol Stacks

Nico Janssens, Sam Michiels and Pierre Verbaeten
K.U.Leuven, Dept. of Computer Science, DistriNet Labs
Celestijnenlaan 200 A
B-3001 Leuven
+32 16 32.76.40

{nicoj,samm,pv}@cs.kuleuven.ac.be

Abstract-- **In this paper we present a design for runtime Customizable Protocol Stacks, which we call CuPS. CuPS is an add-on to DiPS (Distrinet Protocol Stack framework), a component framework for developing open protocol stacks. The design of the DiPS framework allows one to plug in meta-level extensions. CuPS is such a meta-level extension that permits dynamic, unanticipated customizations such as adding, removing and replacing components. We evaluate and prove the necessity of our customizable open protocol stack approach with the design and implementation of a reliability protocol, which can dynamically be adapted to improve different network characteristics (such as packet loss, jitter and communication delay). The results clearly show that application performance can benefit from runtime customizations at protocol stack level. We strongly believe that a combination of the service level flexibility (*on top of* a protocol stack) of traditional active network approaches and the CuPS approach can result in the ultimate level of network flexibility.**

Keywords-- Active/programmable networks, architecture, component technology, performance, resource management/QoS, dynamic adaptability

1. Introduction

Today's networking structures and more specific wireless communication links, behave quite unpredictable due to congestion, physical adaptations and fluctuating packet loss and bandwidth. Such unpredictable behavior influences the communication performance significantly. In fact, due to the closed and fixed structure of most protocols, they cannot be adapted to deal with changing network characteristics in the most optimal way. However, by using open protocol stacks that allow customization, a performance gain can be obtained. Open customizable protocol stacks provide us with the opportunity to adjust the composition of a stack based on the present network status on the one hand, and non-functional requirements (such as QoS) of the applications using the stack on the other hand.

The following scenario illustrates the need for open customizable protocol stacks. Suppose we have a video-conferencing application with strict delay requirements. In addition, this application runs on top of a network characterized by communication links suffering from fluctuating packet loss, and protocol stacks are deployed with traditional reliability support (such as TCP [16]). It is obvious that waiting for retransmission time-outs when packets are lost (as part of the reliability support) increases the communication delay. In order to reduce this delay, the sender could anticipate the probability of resends by sending each packet several times in advance [5]. In the context of the video-conferencing application and its delay requirements, it is very effective to extend the used reliability support with this multiple-sending service in case of low quality communication links with a high packet loss. In contrast, this extension will be overkill when reliable communication links are used. Moreover, within a wireless network the level of packet loss is most often unstable. Therefore, we need open customizable protocol stacks on the routers allowing reconfiguration to counter the impact of such unpredictable behavior, without affecting (e.g. restarting) the applications that use the network.

To cope with the lack of flexibility on today's routers, several active network architectures have been proposed. These architectures achieve flexibility by allowing applications to deploy services (such as our multiple-sending service) on the routers. However, most of these active network platforms, such as CANEs [11][17] and ANTS [18] have restricted themselves to the execution environment of the router, making abstraction of the underlying protocol stack. This virtualization of the available communication protocols in the router's operating system prevents application-level services that are deployed on such a router *to adapt* the internal behavior of these protocols.

As a solution for this limitation, we present in this paper a component framework to build multi-threaded open protocol stacks, DiPS (Distrinet Protocol Stacks), as well as runtime customizability support for these stacks, which is called CuPS. In the remainder of this paper, we will refer to DiPS protocol stacks extended with CuPS support as DiPS/CuPS stacks. DiPS/CuPS protocol stacks can be adapted both by applications that make use of them (such as the services that are deployed on the router of an active

network), as well as by a third party (such as a network administrator who wants to optimize network efficiency). In the latter case, protocol stack adaptations will be completely transparent for services on top of it. By providing these customizable protocol stacks in a NodeOS of an active network engine, the summum of flexibility is reached.

The paper starts with an overview of related research trends that are the basis for the DiPS/CuPS approach. The design and implementation of the DiPS framework and its customizability extension CuPS are described in section 3. The DiPS/CuPS approach is validated in section 4: the paper presents the design and implementation of a DiPS/CuPS reliability protocol and it shows the performance benefits of using such a customizable protocol. The paper ends with current issues and future work in section 5 and the conclusions in section 6.

2. Background

DiPS/CuPS brings together three important research tracks in software architectures: *pipeline architectures, anonymous interaction* and *meta-level architectures*. This section outlines their key contributions in the steps leading to the DiPS/CuPS design.

2.1 Pipeline architecture

Building protocol stacks from scratch is not a trivial task. Protocol stack programmers have to deal with a lot of issues at very different levels of abstraction. There are many low level matters to cope with, such as interpreting and constructing packet headers, fragmenting and reassembling packets or interacting with the network interface. As a result of the complex problem domain, the highest priority of most protocol stack programmers is to build a working stack, rather than to focus on reusability and customizability. This often results in massive, unstructured protocol stacks that are very hard to adapt.

As a methodology for creating maintainable and customizable protocol stacks, a pipelined architectural style [4] has been proposed. Examples of pipeline protocol stack architectures are NetScript [1], Scout [14] and Click [13]. Such architecture forces a programmer to define basic protocol stack entities (called components) that process incoming packet-streams. These components are plugged one after the other to create a functional system (such as a protocol stack).

2.2 Anonymous interaction model

Additionally, components have to be completely independent from each other. As a result, they have to communicate by means of an anonymous interaction model. The reason behind this is that such an anonymous interaction model limits mutual dependencies between components. This is in fact a major advantage because it allows individual components to be *reused* in different compositions due to the lack of explicit references to other components. A second advantage is that, thanks to the anonymous interaction model, components can be dynamically or statically *added or removed* from such compositions. Other components do not have to be aware of these changes since there is no need to obtain a reference to the newly introduced entities.

As a result, a pipeline architecture extended with an anonymous interaction model is very valuable for building open customizable protocol stacks. It allows both rapid-prototyping and developing customizable protocol stacks. Initial prototypes only offer the most essential functionality (e.g. by reusing existing components). Incrementally, the prototype can be extended by introducing more specific implementations or by integrating new components that provide more functionality. In addition, by having independent components we can customize a system to specific circumstances by adding or removing specific components.

To illustrate the reusability and flexibility that is gained by choosing for such architecture, we present the basic functionality of a simple router (depicted in figure 1).

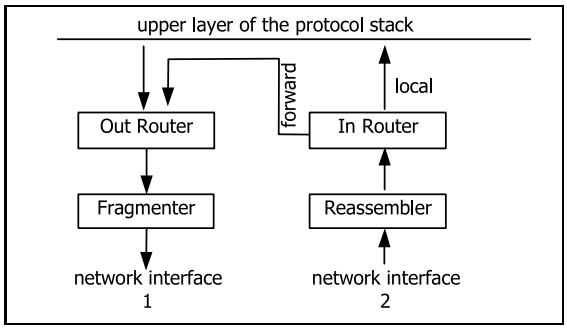

Figure 1: basic router functionality built from fine-grained components

Each of these components has a very specific responsibility. The *Out Router* will determine to which interface outgoing packets should be forwarded. The *In Router* on the other hand, decides whether an incoming packet is for local delivery or whether it should be forwarded. Additionally, the *Fragmenter* and the *Reassembler* component offer fragmentation support.

Suppose extra router context information is available and presume for instance that the router exchanges packets between networks with identical fragmentation characteristics. In this case, the components can be re-arranged into a more optimized forwarding path, as

illustrated in figure 2, where only packets that originate from the local host are fragmented.

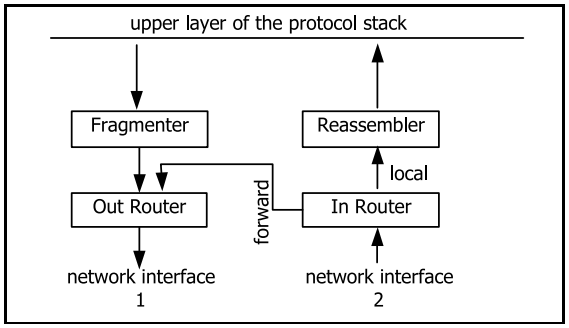

Figure 2: Optimized router-forwarding path

Since arriving packets from network interface 2 already have an acceptable length the *Fragmenter* can be bypassed. Only packets for local delivery will be reassembled. Note that there is no need to adapt the responsibilities of the used components in order to optimize the offered router functionality, since all components are independent of each other.

2.3 Meta-level architecture

When building customizable protocol stacks, both functional aspects (such as the behavior of protocol stacks) as well as non-functional aspects (such as the customization of protocol stacks) must be taken into account, which is not a trivial challenge.

In order to separate the functional from the non-functional aspects of a framework, *meta-level architectures* [7] are of great value. Meta-level architectures define two different levels: a *base level* related to functional aspects, and a *meta-level* handling non-functional aspects. Interactions between both levels are governed by a Meta-Object Protocol (MOP). Note that this separation of concerns results in more flexible and maintainable designs [7].

3. DiPS extended with CuPS: overview

We now present an overview of DiPS [8], a component framework for building multi-threaded, pipelined protocol stacks, and CuPS (see figure 3), a meta-level extension to DiPS that supports non-anticipated runtime customizations.

The objectives of this section are twofold. The first goal is to propose an architecture that allows transparent, non-anticipated customizations of running protocol stacks. Second, the "simplicity" by which DiPS can be extended with such customization support confirms the benefit of using pipeline architectures extended with anonymous interaction support within the field of protocol stacks.

The remainder of this section is structured as follows. Since CuPS is designed as an extension to DiPS, we first elaborate on the DiPS design philosophy in section 3.1, and we zoom in on the DiPS basic building blocks in section 3.2. The customizability support already present in DiPS is described in section 3.3. Section 3.4 describes the CuPS design philosophy and how it supports non-anticipated runtime customizations.

3.1 DiPS design philosophy

DiPS is developed to build all kinds of multi-threaded protocol stacks. Its major goal is to provide support for protocol stack developers and to support the development of flexible protocol stacks, which can be reused (in whole or parts of it), extended or customized. This allows one to build protocol stacks for high-end servers with lots of resources and application-specific requirements. But DiPS also allows static customizations of the protocol stack by removing or adding specific components so that it can be used in a resource-limited and specialized device.

As already mentioned in section 2, DiPS uses a pipeline architecture to create flexible and maintainable protocol stacks. These stacks are built from *fine-grained* components (such as components responsible for header construction, fragmentation or reassembly) as the basic building blocks, rather than from coarse-grained layers as is the case in Scout. This approach is comparable with the modular design of Click. As a result of this design choice, both fine-grained customizations (e.g. replacing the components responsible for the retransmission scheme of TCP) as well as coarse-grained adaptations (such as replacing an IPv4 [15] layer by an IPv6 [2] layer) can be handled.

In addition, DiPS components are equipped with anonymous interaction support. There are only two different ways in which components can interact. The first is direct interaction through a *fixed* external communication interface: components pass information (Packet objects) explicitly through this interface. A second mechanism is implicit interaction through a shared object, which does not introduce explicit dependencies between components. More concretely, DiPS components communicate to each other by attaching information as first-class entities to packets flowing through the system. All other components of the "downstream" (meaning all components these packets will flow through) can retrieve this information independently.

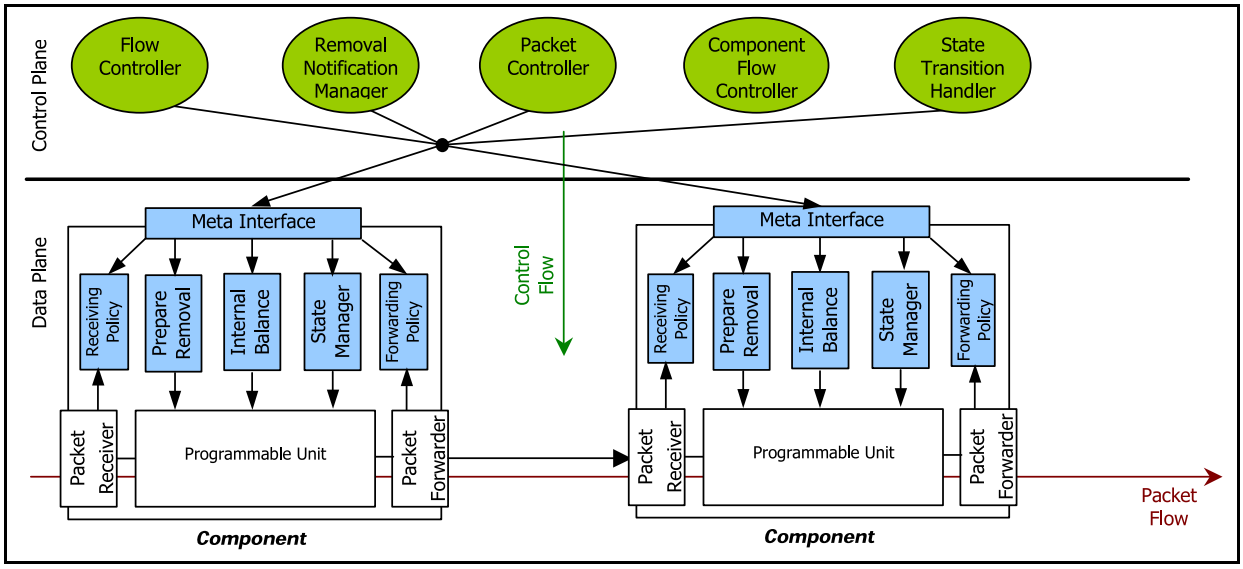

Figure 3: CuPS architecture

An aspect of software systems that influences both performance and resource usage is the concurrency model. In order to allow customization of the used concurrency model of a protocol (stack) without affecting its functional behavior and vice versa, we separated the concurrency model from the functional model in DiPS [9]. This results in two kinds of basic building blocks that are offered by the framework, functional components and concurrency components. Functional components are used to define pure functional responsibilities of the protocol stack, such as header construction, fragmentation, reassembly or encryption. Concurrency components on the other hand, are (only) responsible for the concurrency model of the protocol stack thus defining and controlling threads, critical sections, etc. These abstractions allow a network programmer to focus on the functionality a protocol stack, without taking into account the concurrency model. Afterwards, the concurrency model can be adapted without affecting the functionality of the protocol stack.

This philosophy results in the design of DiPS components as first-class entities, which will be discussed in the next section.

3.2 DiPS component design

Since DiPS makes use of component technology, we zoom in on the DiPS component as building block of protocol stacks in section.

3.2.1 Basic components as service building blocks

In this section we describe the *base-level design* of the basic DiPS components (see figure 4) by elaborating on two main issues: the *assignment of functional and concurrency behavior* and *anonymous communication support*. A component consists of three parts: a *Programmable Unit*, a *Packet Receiver* and a *Packet Forwarder*.

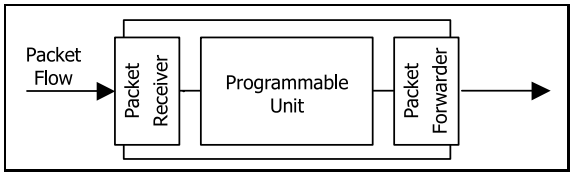

Figure 4: Base-level design of basic DiPS component

The *Programmable Unit* specifies the behavior of each basic component (i.e. the way the component processes incoming packet-flows). We deliberately abstracted away this behavior from the framework support that components offer, such as the anonymous interaction. By separating these concerns, the basic behavior (in the shape of programmable units) defined by a network programmer can be reused in several other kinds of components and frameworks. In order to support the described decoupling between the functional and the concurrency model of a protocol stack, *Programmable Units* are subdivided in both *Functional Units* (such as a fragmentation unit) and *Concurrency Units* (such as a thread-controlling unit).

Next to the assignment of responsibilities, DiPS components contain support for anonymous interaction

between one another, in order to reduce dependencies. This support is offered by the single predefined incoming and outgoing point each basic component is equipped with, respectively the *Packet Receiver* and the *Packet Forwarder* (see fig. 4). The *Packet Receiver* defines the external communication interface of a component for the incoming packet streams. In other words, it specifies how packets can enter a component. The *Packet Forwarder* defines a way to deliver a packet to the *Packet Receiver* from the next component in the flow. In contradiction to NetScript, where both one-way and two-way out ports are provided, DiPS interaction is uni-directional.

3.2.2 Composed Components as a grouping mechanism

Traditionally, protocol stacks are regarded as consisting of a number of layers stacked on top of each other. Even though the DiPS framework works with fine grained components as basic building blocks, the presence of a layer abstraction as a grouping mechanism gives programmers free choice about how they like to view a protocol stack (i.e. both fine and coarse-grained). This layer abstraction indicates the need for *composed components* as a grouping mechanism. Composed components can have several incoming and outgoing points. As an example, a layer abstraction is a special composed component, with an incoming and an outgoing point for both the down- and up-going path.

3.2.3 Connectors

Finally, *connectors* as first-class entities are used for adaptation purposes, in order to connect components from one manufacturer with components from another one. Since both manufacturers can use a different external communication interface (i.e. incoming points different from the ones other components are equipped with), an appropriate connector is needed for translation.

3.3 DiPS built-in runtime customization philosophy

While the employed pipeline architecture allows a lot of flexibility in defining exactly the right configuration to meet the requirements of a specific protocol stack, it cannot cope with configuration customizations at runtime. An example where this occurs is a protocol stack which routes urgent packets over a fast, but costly network, while normal packets are routed normally [6]. A protocol stack with basic pipeline architecture cannot support scenarios like this one. To deal with this kind of *built-in* protocol stack dynamism, a *dispatching component* is offered by DiPS that behaves as a switch for the packet flow passing by [10]. A concrete example of such a dispatching component is the *In Router*, as illustrated in figures 1 and 2. Note that such dispatching points only offer support for *anticipated* runtime adaptations of the protocol stack. As will be discussed in the next section, CuPS has been specifically designed to support *non-anticipated* runtime customizations.

3.4 CuPS extension: customizability support

In the previous sections, we have described the DiPS methodology to build open flexible protocol stacks. What we're still missing though is support for dynamic non-anticipated (re)customizations of DiPS protocols while they are being used. CuPS will offer this support. CuPS can be described as an extension to DiPS for runtime customizability, and can be added when needed. The main goal of CuPS is to provide support for the replacement of components, without tampering the functionality of a running stack (i.e. without losing packets). In addition, CuPS should enable seamless customization of protocol stacks, transparent for applications that are using them. The remainder of this section elaborates on the CuPS design.

CuPS-support can be divided in two planes, support at the control plane (CP) and at the data plane (DP) (as illustrated in figure 3). The data plane is responsible for the packet flow, while the control plane conducts the adaptations of the data plane. Section 3.4.1 elaborates on these customizations by means of a high-level customization scenario. Section 3.4.2 describes the support offered at the data plane to deal with interventions from the control plane.

3.4.1 Control plane support: a scenario

Let's start with a simple setup (as is illustrated in figure 5) that reflects the internal composition of a protocol stack. We would like to customize the behavior of that protocol stack by replacing components X and Y with a new component Z, not yet present in the component flow. Components X and Y will be referred to as *replaceable components*.

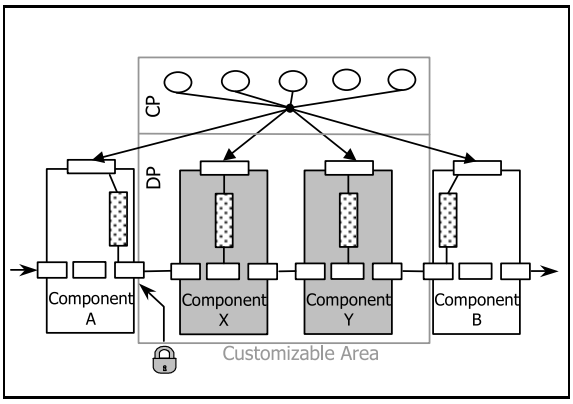

Figure 5: simple setup of runtime customizable protocol stack

In order to realize a seamless transition from the original composition to the new one, it is very important for the replaceable components to be in an *empty status*. This means that they cannot be processing any packets internally, because those packets will get lost when such components are removed from the pipe.

Flow control. The switching scenario starts by blocking the packet streams that are passing through the replaceable components. This will be the responsibility of the *Flow Controller* (see figure 3). Note that, driven by the need for an empty status of the replaceable components, packets are blocked *before* they enter the customizable area, as illustrated in figure 5.

Preparation for removal. After the packet streams are blocked, the replaceable components are acquainted by the *Removal Notification Manager* (figure 3) about their pending removal, which offers them the opportunity to prepare their elimination. Typically, this implies release of resources (useful for concurrency components) and/or the notification of the removal to its peer processes.

Packet bookkeeping. Although the packet flows are stopped before entering the customizable area, there are still no hard guarantees about the components being in the required empty status. Indeed, packets may have entered the replaceable components just before the flows were stopped. An *external bookkeeping* mechanism is required to count all packets entering the customizable area, as well as all packets leaving that area. When the balance of incoming and outgoing packets is zero, the replaceable components have processed all packets, and as a result they are in the required empty status. Unfortunately, counting packets that enter and leave the customizable area will not be sufficient, since this excludes the option of new packets being created or existing packets being removed inside a component. For example, a fragmentation component breaking up each packet into several fragments will never have an equal balance between the amount of packets that enter and leave the component. As a solution, each replaceable component must keep track of the amount of created and removed packets by means of an *internal bookkeeping*. Consequently, the replaceable components are in the required empty status when the amount of incoming, outgoing, created and removed packets is balanced. More in detail, this balanced state is achieved when the number of entered and created packets equals the number of packets that are removed or have left the customizable area. This verification will be the responsibility of the *Packet Controller* (see figure 3).

Sate transfer. At this point in our scenario, there are hard guarantees concerning the empty status of the replaceable components. Unfortunately, this empty status will not guarantee a seamless transition. Sometimes it is necessary to transfer the state of a replaceable component to its new counterpart. As an example, consider the replacement of the acknowledgement strategy used by a deployed reliability protocol (such as replacing ACK [16] by NAK [3] support). In that case, a seamless transition can only be realized when component related state information (such as the sequence number of the expected packet) is transferred from the old to the new components. The *State Transition Handler* (see figure 3) of the control plane will take care of such state transition. Note that this *State Transition Handler* is only responsible for *transferring* a state representation between several components. How this representation is generated and interpreted will be the responsibility of the involved components.

Component replacement. Finally, our component pipe is prepared to go through a seamless and transparent transformation. The only step remaining is the removal of the replaceable components and the integration of new ones. This will be done by the *Component Flow Factory* (see figure 3). Note that it is also possible to remove the replaceable component without integrating new ones. In that case, the customizable area will be bypassed. Afterwards, the blocked packet flows are released by the *Flow Controller*, which results in packets passing through the recently inserted components.

It is obvious that there is quite a lot of interaction between the control plane and the data plane (such as blocking the packet flows, notification of the replaceable components about getting removed, checking whether the replaceable components are processing packets inside, state transitions, etc.) In the next section, the support of components located at the data plane to handle these interactions is discussed.

3.4.2 Data plane support: Meta-level extensions for DiPS components

To deal with the customization related interactions originating from the control plane as described in the previous section, DiPS components are equipped with a *generic meta-interface* (see figure 3) in addition to the external communication interface. Such a meta-interface provides a means to expose a number of critical control plane related issues (such as checking for packets being processed inside) in a principled manner.

The most important advantage of this approach is *separation of concerns*: network programmers (who are responsible for defining the functionality of a stack) must be able to keep on using the existing external communication interface (defined by the Packet Receiver) when they want to define the base-level behavior of components (as stated in section 3.2.1). In other words, this must be possible without taking into account additional non-functional requirements (such as interactions originating from the control plane) for which support can be added afterwards. As a result of a separated communication- and meta-interface, the "packet flow" passing through the base level of our components is divided from the "control flow" through the meta-level (see figure 3).

How the described control plane related interactions are handled by a component depends on its programmable unit internal interface and is therefore not re-usable. For that reason, CuPS introduces so-called *meta-modules,* which can be plugged in the meta-level of a component to provide the information required by, and handle interactions originating from the control plane or to adapt its behavior or internal state.

In the remainder of this section, we describe the CuPS meta-modules replaceable components are equipped with in order to complete the customization scenario described in the previous section. These meta-modules are also illustrated in figure 3.

Flow control. The first important meta-modules are those responsible for *controlling the packet flow* that passes through the data plane. This support is added by means of the *Receiving Policy* and the *Forwarding Policy* as illustrated in figure 3. Due to their packet flow controlling objectives, these meta-modules are used by the *Packet Receiver* and *Packet Forwarder*. Consequently, packets entering and leaving replaceable components are handled by resp. the *Receiving Policy* and *Forwarding Policy*.

The goals of these modules are twofold. First, they are equipped with the external bookkeeping support used by the *Flow Controller* located at the control plane. As a result, all packets passing by are counted. Note that only the components enclosing the customizable area (component A and B in figure 5) need to be equipped with such bookkeeping meta-modules. These are the points where counting incoming and outgoing packets is required to check for an empty status of the customizable area.

The second goal of the flow control modules is blocking the packet streams that pass by, which is the initial step of the customization scenario. This blocking must take place at a point before entering the customizable area. As a result, the *Forwarding Policy* of the component located in front of the customizable area (component A in figure 5) is supplied with such blocking support.

Prepare for removal. Each replaceable component is equipped with a *Prepare Removal* meta-module that defines how a component prepares its elimination. Note that replaceable components can be equipped both with default (e.g. when no preparation is required) as with component specific *Prepare Removal* meta-modules (e.g. to release resources)

Internal packet bookkeeping. The *Internal Balance* meta-module keeps track of the amount of created and removed packets. As a result, packets cannot be created or removed any longer inside the programmable unit of a component by use of default constructors and destructors. Instead, a packet factory (responsible for creating and destroying packets) is connected with the Internal Balance meta-module.

State transfer. The *State Manager* is responsible for converting component specific state-related information into a generic first-class representation and vice versa. The control plane will transfer this first-class state-representation between several components, more specifically between the *State Managers* of these components.

The state-representation has to be *generic* in order to house all kinds of component specific state information. Due to this generic representation, anonymous interactions between components are preserved.

Since component state information is very specific, it is important to define a *universal terminology* to express the content of the generic state representation in order to avoid misunderstanding. As an example, suppose an ACK sending component (part of a customizable reliability service) will be replaced by its NAK sending counterpart. As part of this replacement, the state of the ACK sending component (which in this case only contains the expected packet number) has to be transferred to the NAK component. This transition will only succeed when the state manager meta-modules of both the ACK and the NAK sending component use the same terminology for defining the involved expected sequence number. Otherwise, the state manager meta-module of the NAK sender will not understand the content of the generic state representation it is supposed to process.

Note that the component specific state managers are essential for a successful state-transfer. At this point in our research, we have validated the state transition support by rather simple cases, such as the described expected sequence number transfer. More complex state managers, that allow transferring an entire state machine, are subject to future research.

3.4.3 Customizable areas

Although we are convinced that DiPS in combination with CuPS can have a major impact on performance (cfr. the test results in section 4), it is clear that extending DiPS with CuPS brings some overhead. Flexibility, especially in combination with a meta-level architecture, is often an all-or-nothing approach, i.e. if a system needs to be flexible one have to take into account the overall overhead. The DiPS/CuPS design allows the protocol stack developer to inject the flexibility support where it is needed by defining so-called customizable areas. Only components located within such a customizable area are equipped with specific meta-modules to allow customizations. The other components (i.e. pure DiPS components) of the protocol stack remain untouched and, by consequence, do not suffer from any overhead due to runtime customizability support.

3.5 Implementation

DiPS as well as CuPS are both fully implemented in Java. Java software engineering benefits and support for dynamic class loading makes it an ideal fast prototyping language. We have implemented DiPS versions of a.o. TCP, UDP, IPv4, IPv6 and Ethernet. Next to those standard protocols we have implemented a specific reliability protocol that enables to tune reliability aspects (such as an acknowledgment strategy with positive, negative or selective ACKs). DiPS/CuPS can be used as a user level Linux process in combination with a slightly modified Linux 2.2.16 kernel (only to allow direct communication with the Linux ethernet interface). We have also provided a special library that intercepts socket calls and delegates them to DiPS/CuPS. This way, many Linux applications can transparently use DiPS sockets. In addition, support for generic addressing has been developed [12] in order to cope with protocol stack customizations that have an impact on the used addresses, such as replacing IPv4 by IPv6.

4. Case Study: a runtime customizable reliability protocol

We validate the use of DiPS/CuPS by describing the design and implementation of a customizable reliability protocol (CRP) (see figure 6). Such a reliability protocol is developed to support application-specific QoS requirements, such as communication delay (delay between sending and receiving a packet) or jitter (standard deviation of the arrival rate of packets). These requirements are typically very important for multimedia applications.

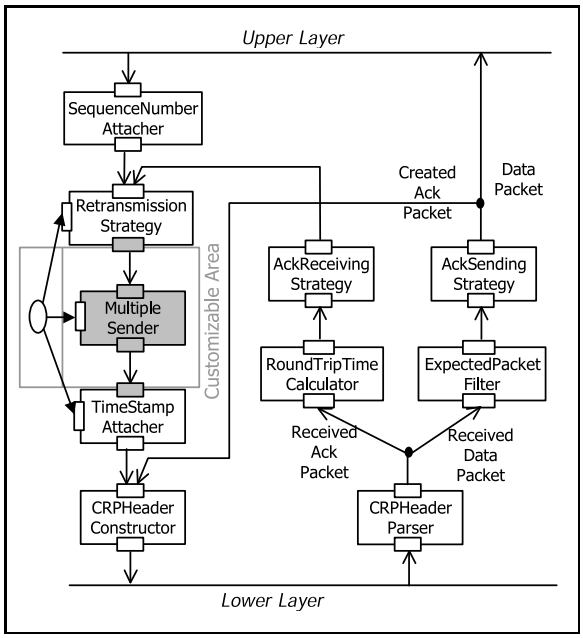

Figure 6: Design of Customizable Reliability Protocol (CRP)

On the level of protocol stacks, jitter and delay are heavily influenced by the reliability support that is used (such as the retransmission or the acknowledgment strategy). For unstable (wireless) networks with varying packet-loss characteristics a flexible reliability protocol such as CRP is a must. Note that TCP for instance also provides reliability support. However, this support is static and is not configurable to an application's service requirements.

The objectives of this case study are twofold. The first goal is to give a concrete example of how a DiPS/CuPS protocol looks like. Second, we present performance results, which clearly show the performance improvements when using a customizable protocol stack.

4.1 CRP design

4.1.1 Basic reliability support

The basic functionality provided by most reliability protocols is quite simple: in order to recover from packet loss, the sending part will resend the data until an acknowledgement from the receiver has arrived. This

resending behavior is supported by the *RetransmissionStrategy* component, located at the down going path of CRP. When a packet arrives at this component, an internal clock is started that will time-out if no acknowledgement packet has been received within a certain time interval. In this case the packet will be retransmitted. Note that CRP can as well be equipped with other retransmission strategies, which allow choosing between packet loss recovering and delay/jitter requirements [5].

The *ExpectedPacketFilter* distinguishes regular packets from retransmissions. This component checks the sequence number of each arriving packet to notice whether it is a new packet or a duplicate. Only the new packets are allowed to pass, the duplicates will be discarded.

For each of these new and well-arrived packets a confirmation has to be sent. This is the responsibility of the *AckSendingStrategy*, located at the up going path: for each packet that is received, an acknowledgement packet is constructed. The dispatching component located after the *AckSendingStrategy* will forward both kind of packets depending on their type: acknowledgements are transmitted back to the sender of the packet, data packets are delivered to the upper layer in the protocol stack.

When an acknowledgement packet is received, it is analyzed by the *AckReceivingStrategy* component. As a result of this analysis, the acknowledged packets are removed from the retransmission queue in the *RetransmissionStrategy* component.

To allow the distinction between new packets and retransmissions by the *ExpectedPacketFilter*, the *SequenceNumberer* component attaches a sequence number to each packet that passes by.

In order to obtain efficient retransmissions, accurate time-out values (depending on the round-trip time (RTT)) are very essential. These time-outs indicate when a transmitted packet can be presumed being lost and in addition needs to be retransmitted. In order to cope with network characteristics such as varying communication delays, CRP is equipped with dynamic timer support. This support is provided by both the *TimeStampAttacher* and the *RoundTripTimeCalculator*. These components respectively deal with the attachment of a timestamp to outgoing packets and the calculation of the RTT for each received acknowledgement (based on that time stamp). These components frequently estimate the actual round-trip times and update the retransmission time-out in the *RetransmissionStrategy*.

The *CRPHeaderContructor* and *CRPHeaderParser* are responsible for the creation and interpretation of the reliability header that contains packet specific information such as the sequence number, the timestamp and the acknowledgment number.

4.1.2 Customization: support for multiple sending

As stated in the previous section, packet loss is countered by resending packets after a retransmission timeout has expired. This time interval has an impact on the experienced delay and jitter. If the interval is too long, lost packets will be detected late which results in high communication delay and jitter. But if the interval is too short, the network is overloaded with unnecessarily retransmitted packets. This redundant network load could cause congestion, which could have an impact the delay and jitter of other communication flows [5].

In order to improve the service quality of such links (and more specifically delay and jitter) without having to sacrifice packet loss recovery, we refer to the multiple-sending example as described in the introduction of this paper. Packet loss is anticipated by sending each packet several times in advance instead of waiting for time-outs [5]. This support is provided in CRP by the *MultipleSender* component (see figure 6).

The introduction of the *MultipleSender* in CRP is only worthwhile when communication links suffer from a considerable packet loss. In case of reliable links, sending a number of duplicates of each packet results in unnecessary bandwidth consumption. As a result, it is obvious that the *MultipleSender* should only be deployed when necessary.

To cope with such unpredictable packet loss (which is characteristic for wireless communication links), CRP is extended with a customizable area to allow the introduction of the *MultipleSender* when necessary (see figure 6). As a result, CRP can be customized at runtime to cope with each network situation in the most optimal way.

4.2 Performance Tests

In this section we study an initial performance comparison of CRP in terms of delay, jitter and network load. These tests will indicate the performance gain that can be realized by deploying the MultipleSender to tackle network situations suffering from high packet loss in the most optimal way.

4.2.1 Experimental setup

CRP has been tested on a simulated network environment, in order to imitate all kinds of network situations. This simulated network environment was running under Linux 2.2.16 using the Sun 1.3.1 Virtual Machine.

CRP was tested both without multiple sending support, as well as equipped with two different *MultipleSenders*, duplicating each packet one and two extra times. Each of these CRP configurations were tested on simulated

network situations, suffering from packet-losses in a range between 0 to 25%. For each test setup (defined by the combination of the CRP configuration and the level of packet loss), the sender of our simulation environment has produced 2000 packets of 1 Kbytes at the rate of 10Hz.

4.2.2 Results and analysis

Before elaborating on the analysis of the test results, we want to emphasize the irrelevance of the absolute values of these results, due to the use of a simulated network environment. Rather, our objective is to compare delay, jitter and network load for each test setup.

Delay. The delay improvements realized by deploying multiple sending support are depicted in figure 7. From these test results, we draw two conclusions.

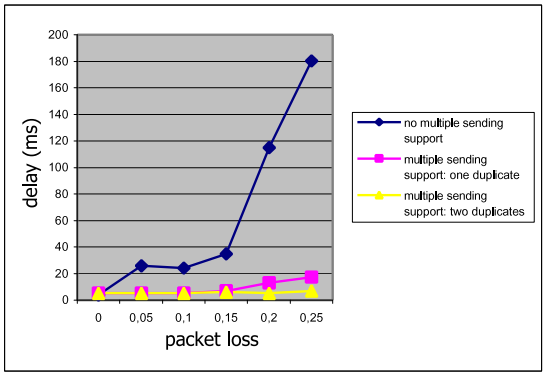

Figure 7: delay improvements gained by several CRP configurations

First, it is obvious that the deployment of multiple sending support results in stable and predictable delay characteristics. This can be deducted from the constant behavior of the curves that illustrate the deployment of multiple sending support (both in the case of sending one and two duplicates). On the other hand, the delay curve of CRP without multiple sending support is far more steep.

A second conclusion concerns the level of delay improvement that can be realized by using multiple sending support. This improvement is rather small in case of little packet-loss on the communication channel (i.e. less then 15%). In case of high packet-loss, significant delay improvements (up to 45 times better) can be realized by use of multiple sending support. Note that the more duplicates that are sent, the less additional delay improvement is gained.

Jitter. The jitter improvements realized by deploying multiple sending support are illustrated in figure 8. This graph clearly indicates that jitter will be improved (up to 4 times better) by sending one or two duplicates for each packet. Next, we conclude that these jitter improvements (in contradiction to the gained delay improvements) are more or less linear for the whole range of simulated packet-losses.

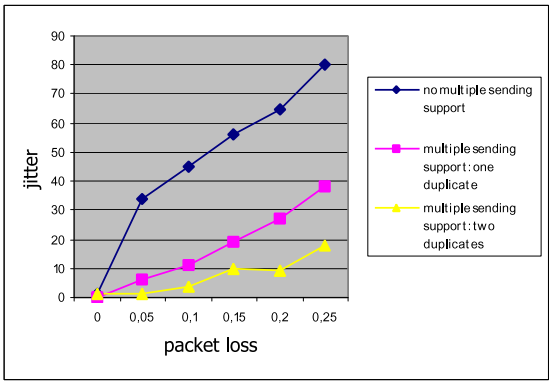

Figure 8: jitter improvements gained by several CRP configurations

Network load. Finally, we investigate the network load caused by multiple sending support. The results are depicted in figure 9. It is obvious that CRP extended with multiple sending support results in a high network load. When the level of packet-loss increases, the network load will decrease because more packets are getting lost. Using CRP without multiple sending extension on the other hand, the behavior of network load acts the opposite. Indeed, when the level of packet-loss increases, the network load will increase due to resends of packets that are presumed being lost. Note that, due to their differing nature, the network load of CRP with and without multiple sending support might converge to the same point (see network load for packet loss of 25%).

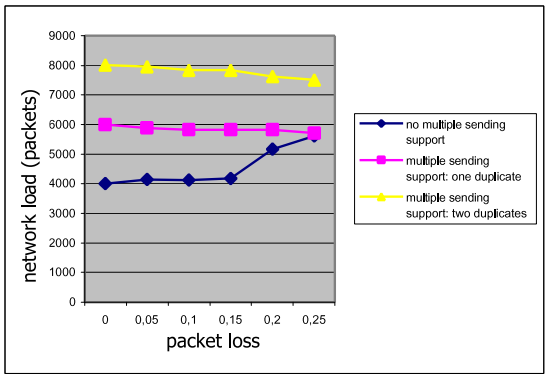

Figure 9: network load caused by several CRP configurations

5. Issues and future work

As illustrated in section 4, CuPS has been initially validated by the development of CRP, our customizable reliability protocol. At this moment, the customization support offered by CRP is limited to the deployment and removal of multiple sending support at runtime. The validation of CuPS based on more complex customizable protocols will raise new issues. One of these issues is support for transferring more complex component-related states between CuPS components, e.g. in order to transfer an entire state machine. Another issue is the validation of composition customizations. When a component is injected in a protocol stack, it is important to check whether or not this customization interferes with the correct working of the protocol stack.

6. Conclusions

This paper presents a new approach for developing protocol stacks, called DiPS, and an extension for non-anticipated runtime customizability, called CuPS. This approach is based on three software design principles: pipeline architectures, anonymous interaction and meta-level architectures.

The combination of those basic principles leads to an open software system, which allows to be tuned to static and/or dynamic circumstances. The highly modularized approach and the anonymous interaction model of DiPS support the reuse of basic building blocks. The possibility to plug in meta-levels on top of DiPS allows it to be extended with non-functional support, such as runtime customizability support provided by CuPS.

The paper describes the design and implementation of DiPS/CuPS as well as an implementation of a DiPS/CuPS reliability protocol. This protocol proves that the DiPS/CuPS approach is very useful for applications with strict service requirements that run on top of networks with fluctuating packet loss. This is not an imaginary combination when we take into account that wireless communication devices with strict communication delay requirements are becoming more and more popular. Test results show the benefits of and the need for flexible protocols that can be dynamically tuned to different circumstances.

By offering such flexibility within the protocol stack of an active router, the DiPS/CuPS approach is a good candidate for the "next generation" of active network architectures. It is crucial for active routers to allow protocol stack level customizations for instance to deal with fluctuating network characteristics such as packet loss. It is very difficult, if not impossible, to provide such support at service level, as is done in many active network architectures today.

7. Acknowledgements

This paper presents research sponsored by a GOA-project: AgCo2 (Agents for Coordination and Control), in which research is done round this topic.

8. References

[1] S. da Silva, D. Florissi, and Y. Yemini. *Composing Active Services in NetScript.* DARPA Active Networks Workshop, Tucson, AZ, March 9-10, 1998.

[2] S. Deering, and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification.* RFC 2460, IETF, December 1998.

[3] R. Fox. *TCP big window and NAK options.* RFC 1106, IETF, June 1989.

[4] D. Garlan and M. Shaw. An *Introduction to Software Architecture*. CMU-CS-94-166. Carnegie Mellon University. 1994.

[5] N. Janssens. *Optimisation of a distributed robot controller.* Master's thesis, K.U.Leuven, Department of Computer Science, Leuven, Belgium, June 2000. (only available in Dutch)

[6] P. Kenens, S. Michiels, O. Debels, S. Van Baelen, W. Joosen, F. Matthijs and P. Verbaeten. *The SmartMove communication architecture.* Technical report, SmartMove internal use only. K.U.Leuven, August, 1999.

[7] G. Kiczales, *Towards a New Model of Abstraction in Software Engineering*. In Proceedings of the International Workshop on New Models in Software Architecture; Reflection and Meta-Level Architecture. 1992.

[8] F. Matthijs. *Component framework technology for protocol stacks.* PhD thesis, K.U.Leuven, Department of Computer Science, Leuven, Belgium, December 1999. Available at www.cs.kuleuven.ac.be/~samm/netwg/dips/th_all.ps.gz.

[9] F. Matthijs. *Concurrency in the DiPS architecture.* In Proceedings of Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 2000.

[10] F. Matthijs, W. Joosen, B. Robben, and P. Verbaeten. *Obtaining Flexible System Software Architectures Using Reflection Points.* In Proceedings of Workshop on Object-Oriented Software Architectures, Technical report 13/98, University of Karlskrona/Ronneby, Sweden, 1998.

[11] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert and E. Zegura. *Bowman and CANEs: Implementation of an Active Network.* Invited paper at 37th Annual Allerton Conference, Monticello, IL, Sept 1999.

[12] S. Michiels, T. Mahieu, F. Matthijs, and P. Verbaeten. *Dynamic Protocol Stack Composition: Protocol independent Addressing.* In Proceedings of 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001). June 2001.

[13] R. Morris, E. Kohler, J. Jannotti, and F. Kaashoek. *The Click modular router.* In Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99), pages 217-231, Kiawah Island, South Carolina, December 1999.

[14] D. Mosberger and L. Peterson. *Making Paths Explicit in the Scout Operating System.* In Proceedings of OSDI '96, pages 153-168, October 1996.

[15] J. Postel, *Internet Protocol*, RFC 791, IETF, September 1981.

[16] J. Postel. *Transmission Control Protocol.* RFC 793, IETF, September 1981.

[17] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele and E. Zegura. *Active Reliable Multicast on CANEs: A Case Study.* In Proceedings of IEEE OpenArch 2001, Anchorage, Alaska, April 2001.

[18] D. Wetherall, J. Guttag, and D.L. Tennenhouse. *ANTS: A toolkit for building and dynamically deploying network protocols.* In Proceedings of IEEE OpenArch 1998, San Fransico, CA, April 1998.