

**A DiPS+ Case Study: A Self-healing  
RADIUS Server**

*Sam Michiels  
Lieven Desmet  
Pierre Verbaeten*

*Report CW 378, February 2004*

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

# A DiPS+ Case Study: A Self-healing RADIUS Server

*Sam Michiels*  
*Lieven Desmet*  
*Pierre Verbaeten*

*Report CW 378, February 2004*

Department of Computer Science, K.U.Leuven

## **Abstract**

This report shows performance results of a RADIUS implementation using the DiPS+ software architecture. In addition it compares this implementation with a commercially available RADIUS implementation, and shows that the DiPS+ architecture differentiates between user types and request types. In fact, the DiPS+ prototype prioritizes incoming traffic based on application-specific preferences, and allocates the available processing resources to the highest priority requests.

**Keywords :** Component architecture, self-adaptation, concurrency.

# A DiPS+ Case Study: A Self-healing RADIUS Server

Sam Michiels, Lieven Desmet, Pierre Verbaeten  
DistriNet Research Group, Dept. Computer Science  
Celestijnenlaan 200A, B-3001 Leuven, Belgium

{Sam.Michiels,Lieven.Desmet}@cs.kuleuven.ac.be

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	The Radius protocol . . . . .	2
1.2	The DiPS+ framework . . . . .	2
1.3	Overview . . . . .	3
<b>2</b>	<b>Basic architecture</b>	<b>4</b>
<b>3</b>	<b>Test configuration</b>	<b>4</b>
<b>4</b>	<b>DiPS+ RADIUS implementation</b>	<b>5</b>
<b>5</b>	<b>Performance results</b>	<b>7</b>
5.1	Theorem versus DiPS+ . . . . .	7
5.1.1	Theorem throughput limit . . . . .	7
5.1.2	Theorem stack during overload . . . . .	9
5.1.3	DiPS+ during overload . . . . .	10
5.2	Using concurrency units in DiPS+ . . . . .	11
5.2.1	DiPS+ in combination with user prioritization strategy . . . . .	11
5.2.2	DiPS+ in combination with request type differentiating strategy . . . . .	11
5.2.3	Combined DiPS+ strategy: user and request type differentiation . . . . .	15
<b>6</b>	<b>General conclusion</b>	<b>15</b>

# 1 Introduction

Network servers have to deal with major variations in access load. For example, ADSL<sup>1</sup> login servers encounter peak loads when residential users come home in the evening and connect all together. Similarly, during new year's evening, SMS<sup>2</sup> server are overloaded by millions of greeting messages. Also it is not uncommon to experience more than 100-fold increases in demand when a website becomes popular due to publicity at a popular portal site [8].

Since is not feasible to install a server farm that is big enough to handle the maximal demand (quite cost-inefficient), systems should be able to handle peak loads gracefully. Furthermore, it should be possible to easily incorporate the business policy in handling an overload situation. For example, less important requests can be identified and dropped, in order to ensure the processing of business critical requests.

The main objective of this report is to compare a commercially available Java implementation<sup>3</sup> of the RADIUS authentication and accounting protocol with a DiPS+ prototype. Hereby, we focus on handling overload situation gracefully. The fact that a framework, such as DiPS+ software architecture, is being used will probably result in lower performance. On the other hand, DiPS+ makes it possible for the stack to adapt its performance at runtime, based on the measurement of some traffic indicators.

## 1.1 The Radius protocol

RADIUS (Remote Authentication Dial-in User Service) is a client-server internetworking security system running on top of the UDP transport protocol [6, 5]. It controls authentication (verifying user name and password), accounting (logging relevant user activities), and authorization (access-control) in a multi-user environment.

For example, upon ADSL network login, a dial-up server delegates login information to the RADIUS server of the client's Internet Service Provider (ISP) for authentication and for obtaining network configuration information, such as the local IP address to use on the Internet. As part of accounting services, RADIUS logs user information in a database or a file according to customers' requirements.

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

## 1.2 The DiPS+ framework

DiPS+ [1, 3] is a rapid prototyping infrastructure for building reconfigurable, component-based protocol stacks. The main goal of DiPS is to support reuse and adaptation at design time [7] as well as at run-time [2, 4]. The underlying software architecture of DiPS applications is a combination of a pipe-and-filter composition with a shared data repository.

---

<sup>1</sup>ADSL stands for Asymmetric Digital Subscriber Line and provides broadband Internet access using a telephone line.

<sup>2</sup>SMS (Short Message Service) is a service for sending messages of up to 160 characters to mobile phones that use Global System for Mobile (GSM) communication.

<sup>3</sup>AXL RADIUS Server (formerly know as Theorem Radius Server), <http://www.axlradius.com/>

A DiPS+ protocol stack consists of several protocol layers, each with an up-going, a down-going and forwarding path. A path is a pipe-and-filter composition of loosely coupled functional units. coupled with connectors (figure 1).

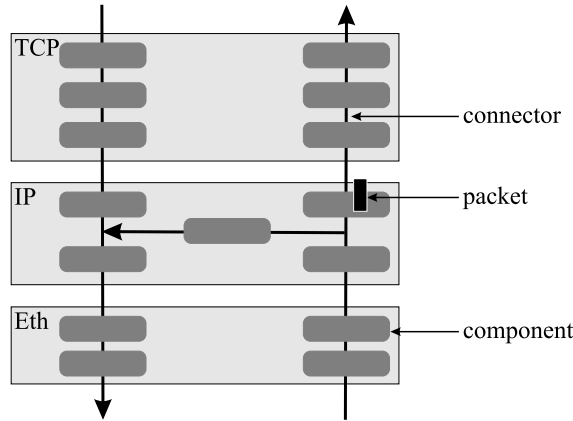


Figure 1: A protocol stack in DiPS

Using DiPS+, network packets are modeled as first class entities, that are transported along the paths. To enable anonymous communication between functional units, a shared data repository is attached to each network packet, as illustrated in figure 2. This anonymous communication is used to label meta information to a packet, which can be used further on along the path. For example within a protocol layer, a `HeaderParser` parses the raw data header of the packet, and adds the extracted header information to the packet by means of meta data. This meta information can be used by proceeding components in the path to customize the processing of the packet.

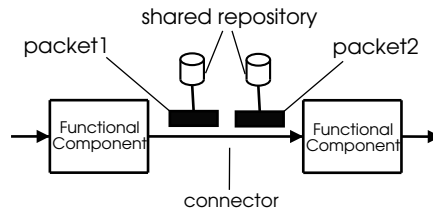


Figure 2: Shared repositories in DiPS

Within DiPS+, state information is usually not kept in the components locally but is attached as meta information to each individual packet that flows through the protocol stack. This approach allows concurrency (parallelism) to be easily added to the component pipeline, in a transparent manner. Moreover, clearly separating concurrency from functional components enables to customize not only the distribution of processing resources (threads) throughout the system in order to optimize the overall throughput, but also the order in which packets are scheduled.

### 1.3 Overview

Section 2 introduces the basic architecture of the case study in which the RADIUS protocol is used, followed by an outline of the used test configuration (section 3). The DiPS+ prototype

implementation of the RADIUS protocol is illustrated in section 4. Performance results of the commercial DiPS+ and Theorem implementations are discussed in section 5, to conclude with a general conclusion in section 6.

## 2 Basic architecture

Figure 3 shows the business architecture in which the RADIUS implementation is to be incorporated. The green box labelled "Theorem Stack" represents the Java implementation of the RADIUS protocol stack. The original Theorem stack can be replaced by a DiPS+ prototype implementation.

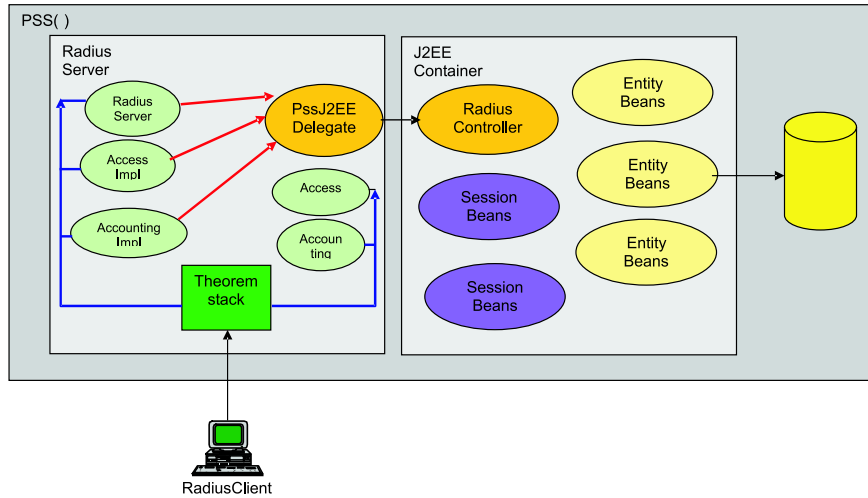


Figure 3: Business architecture for RADIUS implementation

For easy integration with the application logic, a few interfaces were provided by the system architect: `AccessImpl` and `AccountingImpl`. The application logic itself consists of a J2EE application, actually performing authentication and accounting for the incoming requests.

The RADIUS client is located outside the business architecture and sends RADIUS requests over the network. In real life cases, this radius client can be the dial-up server, delegating the login information to the RADIUS server of the ISP for authentication and accounting. In development phase, the RADIUS client can be replaced by a request generator.

## 3 Test configuration

When looking at the processing time of an `AccessRequest` or `AccountingRequest`, one sees that the time consumed by the `RadiusServer` is in the range of 5% of the total processing time. The J2EE application and the database backend itself are responsible for the major part of the the processing time.

In order to compare the 2 stacks, it is found better to test the stacks without the container and database. Therefore, a dedicate stub implementation is foreseen, whereby the configuration and user data are read once at initialisation, from property files. Accounting does not register CDR data at all. This leads to the test configuration, illustrated in figure 4.

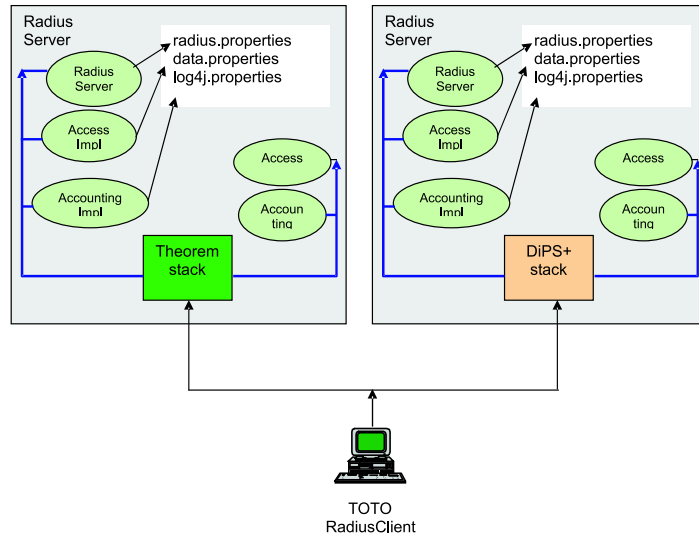


Figure 4: Test configuration

At startup, a user data.properties file is constructed for NUMBER\_OF\_USERS (an item from radius.properties). It has the following syntax:

- Usernames: user000, user034,user299
- Passwords:pwd000,pwd034,pwd999
- FramedIpAddresses: 138.203.0.0, 138.203.0.34, 138.203.2.99

The RADIUS client is replaced by the *Toto performance tool* of SMC, a RADIUS authentication and accounting requests generator. The tool has good performance and has extensive display and logging capabilities. In our case the throughput rate and processing delay of each individual request is measured by the tool, while the generation rate can be controlled.

## 4 DiPS+ RADIUS implementation

Figure 5 shows the high-level DiPS+ design of RADIUS authentication (left side), and accounting (on the right).<sup>4</sup> Following the RADIUS specification, the accounting layer accepts packets via UDP port 1812, the accounting layer is connected via UDP port 1813. The parsing components (`HeaderParser` and `AttributeParser`) interpret the header and the list of attributes of an incoming RADIUS packet. The header and attribute constructors (`HeaderConstructor` and `AttributeConstructor`) create the RADIUS header and the list of attributes, and attach these to an out going RADIUS response packet. The `NASChecker` verifies the originator of the request and drops the packet in case the NAS is unknown. The parsing and constructing components, as well as the NAS checker, are generic and are reused in both the authentication and the accounting layer.

<sup>4</sup>The top and bottom design in Figure 5 only differ in that they insert the two scheduling strategies in different places in the design. The actual strategies are explained later.

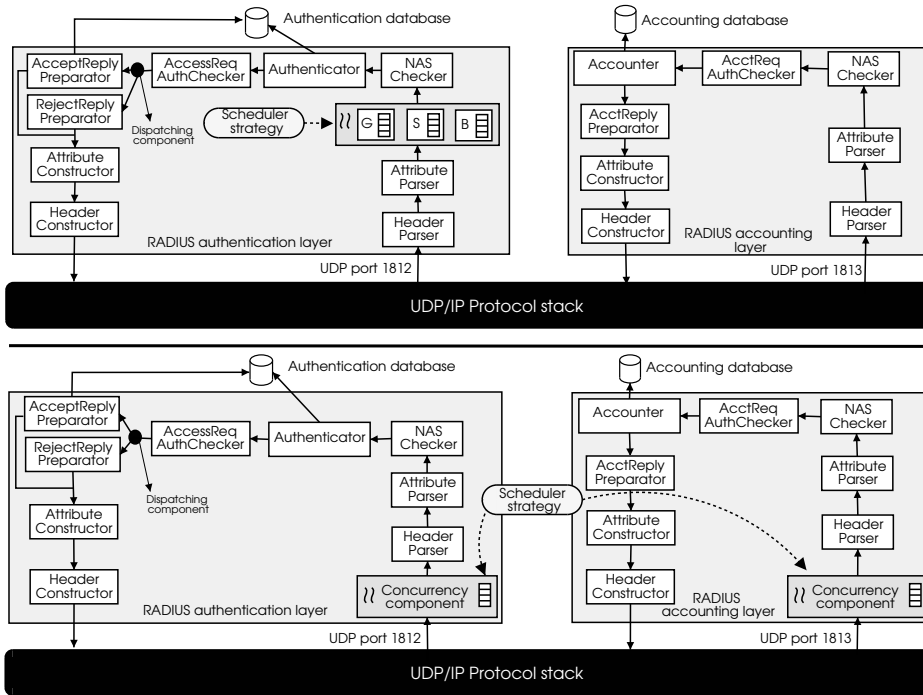


Figure 5: Design of RADIUS authentication and accounting in DiPS+. The top figure illustrates a scheduler strategy that distinguishes gold, silver, and bronze users. The bottom figure, demonstrates a strategy that prioritizes accounting requests over authentication requests.

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

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

## 5 Performance results

This chapter summarizes the most relevant performance results of both the Theorem and DiPS+ implementation of the RADIUS protocol. First, the Theorem implementation is compared to a standard DiPS+ implementation (without active units). Secondly, the DiPS+ advantage is highlighted by adding active units with both a user and request type differentiation policy. Finally, this chapter shows how these two policies can easily be combined.

### 5.1 Theorem versus DiPS+

This section first determines the Theorem performance limit, secondly its behavior in case of overload is demonstrated, and finally the chapter shows DiPS+ behavior in case of overload. The latter result also shows that the overhead of DiPS+ is marginal.

These tests compare the Theorem implementation of RADIUS with the DiPS+ version. We increase the load in three steps:

- Step one (first 150 seconds): pc8: 150 requests/sec; pc9: 150 requests/sec; pc10: 75 requests/sec
- Step two (150-300 seconds): pc8/9/10: 150 requests/sec
- Step three (300-450 seconds): pc8: 200 requests/sec; pc9: 150 requests/sec; pc10: 150 requests/sec

We first show the highest possible load for the Theorem stack. Subsequently, we show how Theorem and DiPS+ behave during overload.

#### 5.1.1 Theorem throughput limit

The two performance graphs below illustrate the throughput limit of the Theorem stack. PC8 generates a constant load of 200 authentication requests per second. PC10 increases the load from 10 to 200 authentication requests per second, with increments of 10 requests per second (figure 6).

We can conclude that the Theorem RADIUS server starts to drop packets from a total load higher than 350 requests per second.

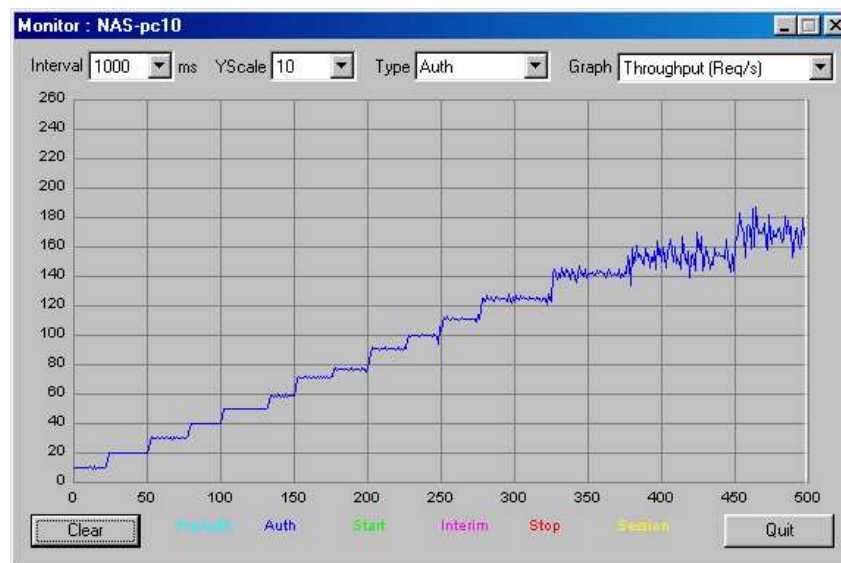
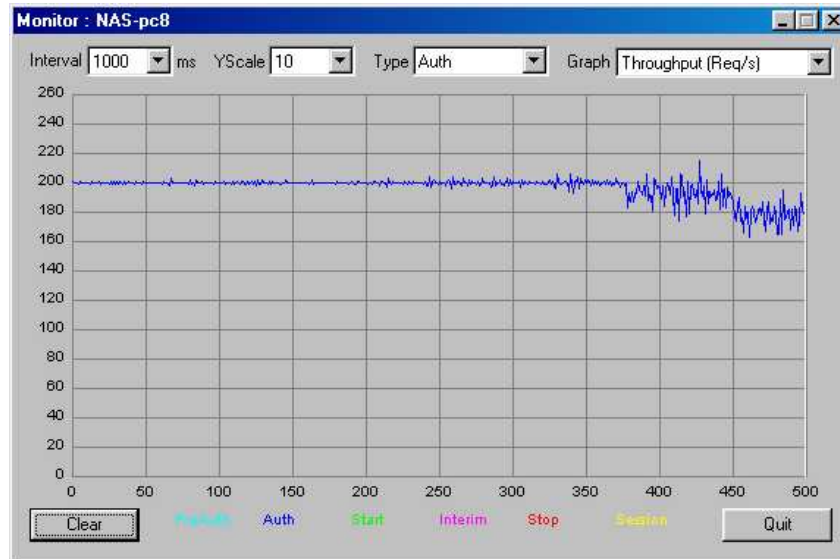


Figure 6: PC8 generates a constant load of 200 requests per second, while PC10 increases the load from 10 to 200 requests per second. The RADIUS server can handle the load (i.e. the load generated by pc8 and pc10) until 350 requests per second. From then on, throughput for pc8 starts to drop.

### 5.1.2 Theorem stack during overload

The server is pushed to its limit (0 - 150 seconds), i.e. 350 requests per second. When PC10 tries to increase its load to 150 requests per second (150 - 300 sec), throughput drops for both PC8 and PC9, while throughput of PC10 only increases up to 110 requests per second. When PC10 further increases its generated load to 200 requests per second (300 - 450), its throughput increases from 110 to 140 requests per second, while throughput of PC8 and PC9 further drops to approximately 100 requests per second.

We can conclude that during overload the Theorem server drops packets *randomly* from the three clients.

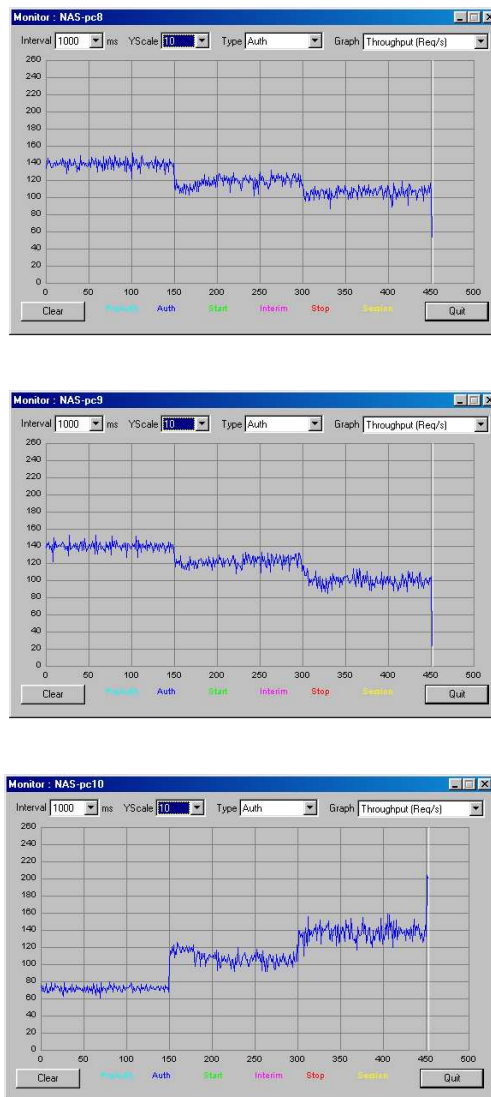


Figure 7: Theorem stack: when PC10 increases its load, performance of PC8 and PC9 is affected and the requested load at PC10 of 200 requests per second is never reached

### 5.1.3 DiPS+ during overload

The same overload test (as presented in the previous section) is applied to the DiPS+ implementation. The server is again pushed to its limit (0 - 100 seconds). When PC10 increases its load to 150 requests per second (101 - 175 sec), throughput drops for both PC8 and PC9.

First of all, we can conclude that the performance limit of the DiPS+ implementation is slightly less than 350 requests per second, which is approximately equal to the Theorem limit. This overhead is partly caused by the highly modular component design with explicit packet receivers and forwarders in between each couple of components. Secondly, we conclude that the standard DiPS+ implementation suffers from random packet loss over the three client PCs, similar to the Theorem implementation.

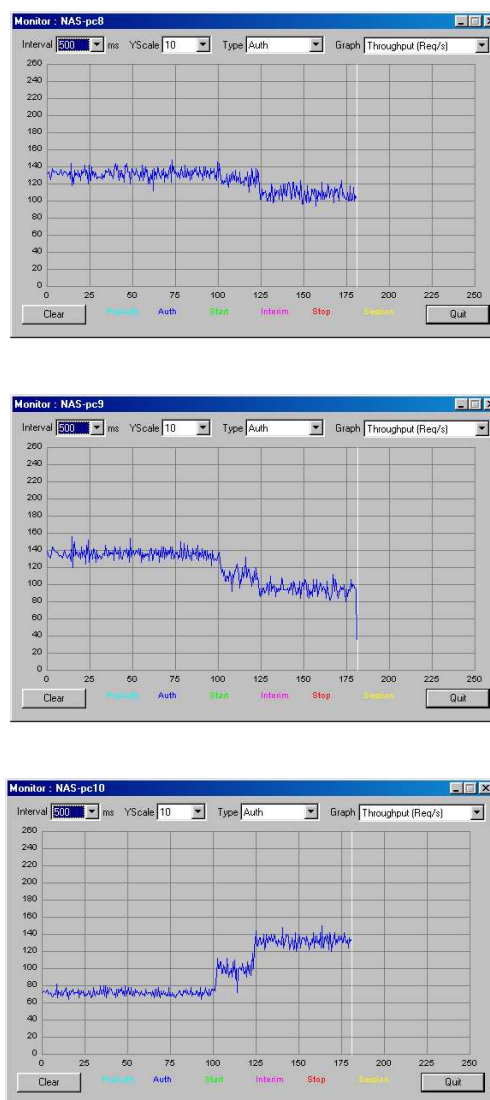


Figure 8: DiPS+ performance results

## 5.2 Using concurrency units in DiPS+

This section demonstrates that DiPS+ allows to control an overload situation intelligently by adding active concurrency units into the component pipeline(s). With each active unit a custom scheduling strategy is associated. First we apply a user based strategy that prioritizes gold users over silver and bronze users. Secondly we apply a request-type based strategy that only allows new authentication requests if no accounting requests of active sessions are currently pending.

### 5.2.1 DiPS+ in combination with user prioritization strategy

We insert into the DiPS+ RADIUS layer an application-specific scheduling strategy, which prioritizes gold (pc8) users from silver (pc9) and bronze users (pc10), as illustrated in upper part of figure 5. The test is sub-divided in 4 steps:

	period	pc 8	pc 9	pc 10
1	1-100 sec	150 req/s	150 req/s	25 req/s
2	101-200 sec	150 req/s	150 req/s	75 req/s
3	201-300 sec	150 req/s	150 req/s	150 req/s
4	301-400 sec	200 req/s	150 req/s	75 req/s

First of all, figure 9 shows tht gold users are not affected by the overload situation. When more gold requests arrive, silver (and bronze) requests are dropped. Because of overload, no bronze requests are processed. Second conclusion is that throughput is lower than the DiPS+ performance limit, since all requests are partly processed until user-information is parsed. Obviously, this results in performance overhead when it turns out that an incoming packet must be dropped. However, by using active units, DiPS+ allows to control the impact of the overload, which compensates the non-optimal throughput.

### 5.2.2 DiPS+ in combination with request type differentiating strategy

We compare behavior of the Theorem, the standard, and the active concurrency unit based DiPS+ implementation as shown in the lower part of figure 5.

In all test cases we use the following parameters:

- pc8/9/10: 30 requests/sec
- for each active session: 1 accounting interim request per second
- each session takes 10 seconds

Each client sends an interim accounting request per second, for each active RADIUS session. In addition, each client sends 30 authentication requests per second, and each session takes 10 seconds. At this rate, 300 sessions are active in parallel (i.e. 30 authentications/sec x 10 sec). As the graphs show, authentication requests are accepted and processed at 30 requests per second, although interim accounting requests are being dropped (only 200 requests/sec instead of 300 (i.e. 30 authentications/sec x 10 sec x 1 accounting interim/sec)), It would be better to block new authentication to the level of which the server can process all accounting requests.

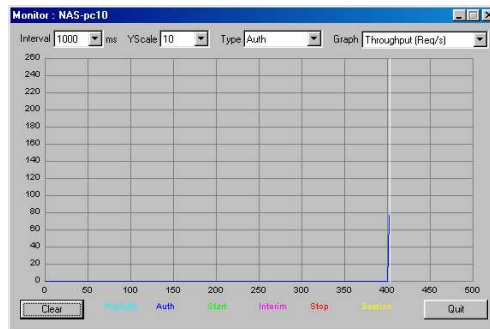
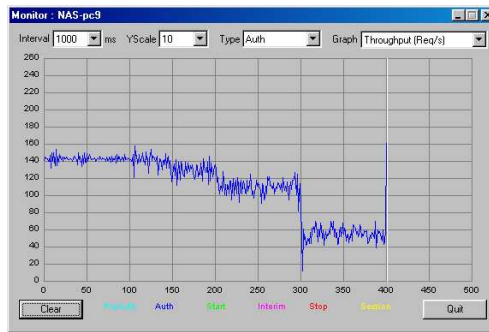
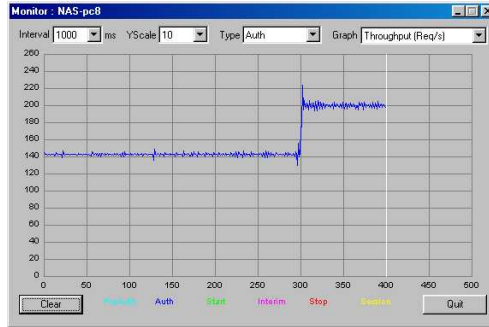
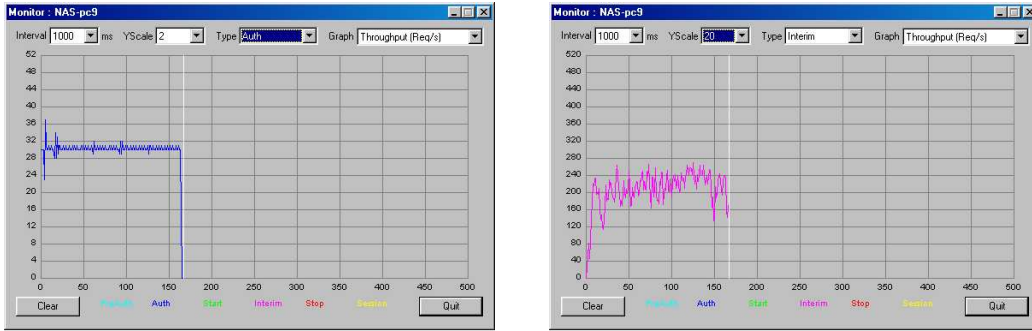


Figure 9: DiPS+ results for user differentiation: gold users (PC8), silver users (PC9) and bronze users (PC10). Gold results are not affected by the load of silver and bronze. Contrary, when more bronze requests are generated, silver throughput is affected. The main reason is that even though bronze requests are rejected after all, they require some processing overhead at the server. When gold throughput increases (from 150 to 200), silver throughput drops from 110 to 50 requests per second. Since the total throughput is limited to 280 requests per second, all bronze requests are dropped. In turn, overload of bronze requests does not affect gold throughput.

## Theorem results

Figure 10 represents the throughput of authentication and accounting requests with the Theorem stack. We can conclude that Theorem does not differentiate between requests types and accepts new authentication requests, even though accounting interim requests are being dropped by the server.



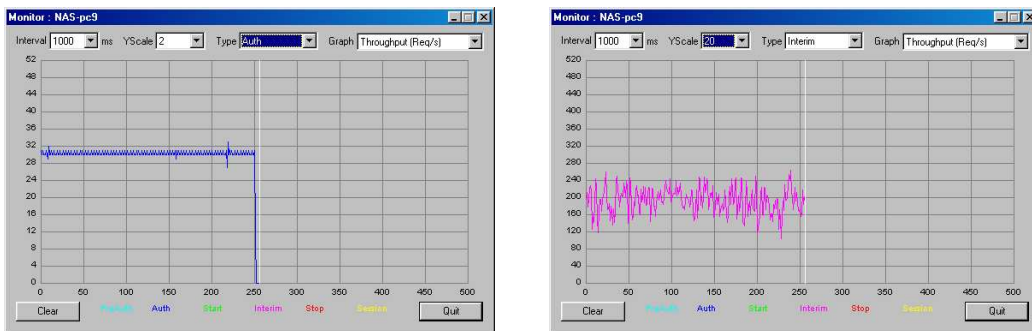
(a) Number of authentication requests per second

(b) Number of processed accounting interim requests per second

Figure 10: Theorem performance results for request type differentiation

## Standard DiPS+ results (without scheduling strategy)

From figure 11, we can conclude that standard DiPS+ behavior is similar to Theorem: new authentication requests are accepted while accounting interim requests of ongoing sessions are being dropped. The number of parallel sessions as again around 300 (i.e. 30 authentications/sec x 10 sec).



(a) Number of authentication requests per second

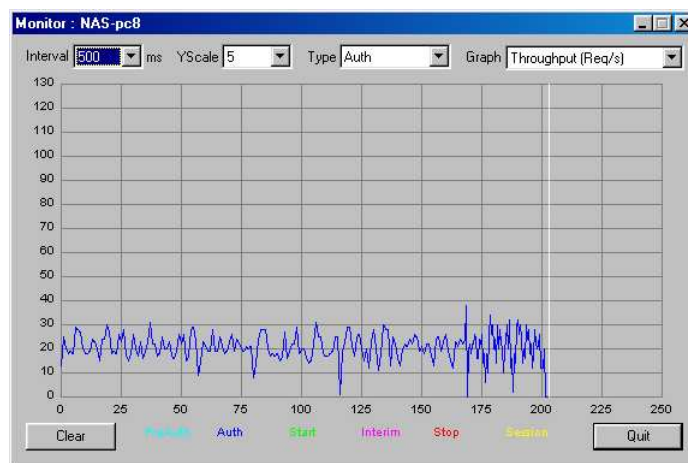
(b) Number of processed accounting interim requests per second

Figure 11: Standard DiPS+ performance results for request type differentiation

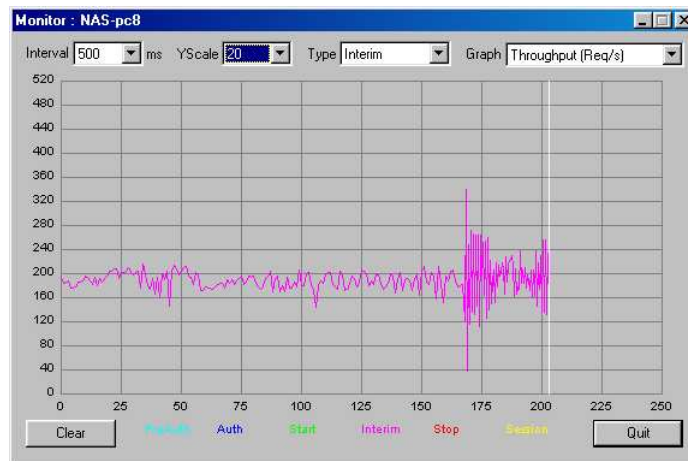
### DiPS+ with scheduling strategy

In this case, incoming authentication requests are blocked to prevent the server from dropping accounting requests from active sessions. The server processes 20 requests per second, and accepts more when it is possible. As the second graph illustrates, the server processes all 200 accounting interim requests. The number of sessions in parallel is now 200 (i.e. 20 authentications/sec x 10 sec), which is the upper limit for the server.

We can conclude that the request based scheduling strategy allows to control the load of the RADIUS server and prevents it from being overwhelmed.



(a) Number of authentication requests per second



(b) Number of processed accounting interim requests per second

Figure 12: DiPS+ (with scheduling) performance results for request type differentiation

### 5.2.3 Combined DiPS+ strategy: user and request type differentiation

The test is sub-divided in 3 steps:

	period	pc 8	pc 9	pc 10
1	1-100 sec	30 req/s	30 req/s	30 req/s
2	101-200 sec	30 req/s	60 req/s	30 req/s
3	201-300 sec	60 req/s	30 req/s	30 req/s

Finally, we combine the two presented scheduling strategies (one based on user type, the other based on request type) in figure 13. As expected, authentication requests are only processed as long as the server can handle all accounting requests from active sessions. In addition, gold authentications have priority over silver and bronze requests (phase 3). During phase two, when additional silver requests arrive, the server only allows new authentication requests when a session ends (resulting in ping-pong graph).

We can conclude that DiPS+ allows to install and combine various scheduling strategies into the component pipeline without affecting the functional components.

## 6 General conclusion

We have demonstrated that the overhead of DiPS+ is minimal compared to the Theorem implementation. Overhead is introduced by two means: the modularized DiPS+ design with explicit packet receivers and forwarders, and the intelligent processing of incoming requests by the active concurrency unit strategies. On top of that we have shown that by using DiPS+ concurrency units along with their scheduling strategy are well-suited to control overload situations. This report presents two scheduling strategies, one based on user-type, the other based on request type, and the combination of the two. Yet, many other application-specific strategies can be implemented and integrated in DiPS+ by introducing active concurrency units into the component pipeline.

## References

- [1] K.U.Leuven DistriNet Research Group. DiPS home page. <http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/DIPS/>.
- [2] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002.
- [3] S. Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, November 2003.
- [4] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. Self-adapting concurrency: The DMonA architecture. In D. Garlan, J. Kramer, and A. Wolf, editors, *Proceedings of the First Workshop on Self-Healing Systems (WOSS'02)*, pages 43–48, Charleston, SC, USA, 2002. ACM SIGSOFT, ACM press.

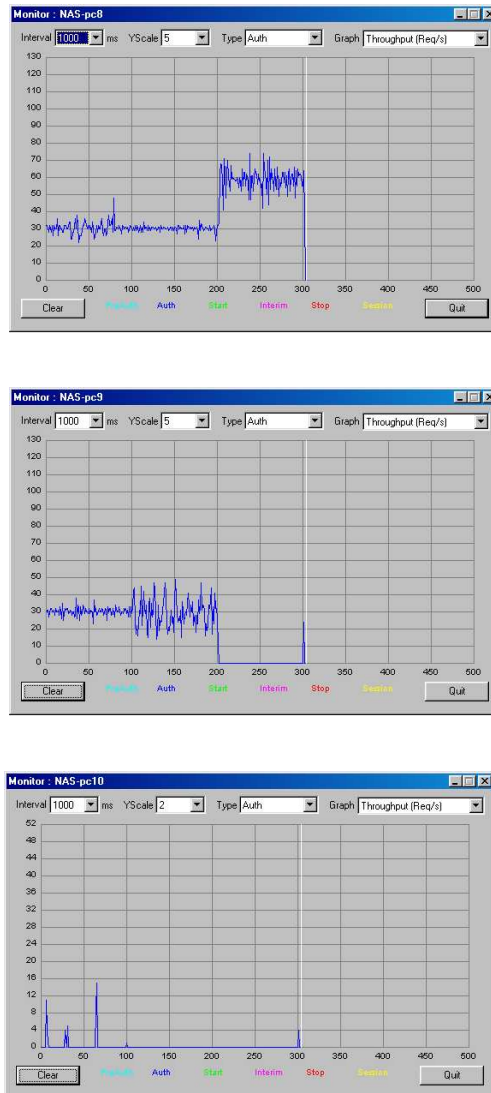


Figure 13: DiPS+ performance results for user and request type differentiation

- [5] C. Rigney. Radius accounting. <http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2866.html>, 2000.
- [6] C. Rigney, A. Rubens, W. Simpson, and S. Willens. Remote authentication dial in user service specification (radius). <http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2138.html>, 1997.
- [7] I. Şora, P. Verbaeten, and Y. Berbers. Using Component Composition for Self-customizable Systems. In *Proceedings of Workshop On Component-Based Software Engineering: Composing Systems from Components*, pages 23–26. IEEE, 2002.
- [8] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. *Seismological Research Letters*, 71(4), July/August 2000.