Konzepte von Betriebssystem−Komponenten

30.01.2003

Quality of Service OS

NEMESIS

Carsten Riedel

sicaried@stud.informatik.uni−erlangen.de

1. Einleitung

1.1 Was ist Dienstgüte ?

Für den Begriff Dienstgüte bzw. Quality of Service(QoS) gibt es keine feste Definition, aber allgemein kann man ihn als Charakteristika eines Dienste bzw. einer Anwendung bezeichnen, die für den Nutzer direkt spürbar sind.
Es gibt verschiedene Parameter, an denen man die Dienstgüte fest machen kann:

- Durchsatz
- Latenzzeit (Zeit für Übermittlung und Bearbeitung)
- Jitter (maximale Schwankung der Latenzzeit)
- Zuverlässigkeit

1.2 Wo trifft man auf den Begriff Dienstgüte ?

Der Begriff QoS tritt überall dort auf, wo es darum geht, zeit− bzw. leistungsorientierte Aufgaben zu erfüllen, die über ein Netzwerk zur Verfügung gestellt werden. Typische Anwendungsfelder, wo man QoS antrifft, sind zum Beispiel:

- Voice over IP (VoIP)
- Videokonferenzen
- Web−basierte Datenbanken
- Differenziertes Accounting von Kunden eines Rechenzentrums

1.3 Entstehung von Nemesis

Im Jahr 1992 wurde von der EU Kommission im Rahmen des ESPRIT Programms das Pegasus Projekt gegründet. In diesen Projekt schlossen sich mehrerer Universitäten und Unternehmen zusammen, mit dem Ziel ein Betriebssystem zu erschaffen, welches QoS−Anwendungen unterstützt.
Die Hauptanforderungen, die sie dabei an ein QoS Betriebssystem stellten, waren:

- es sollte möglich sein sowohl normale als auch Multimedia−Anwendungen gleichzeitig zu betreiben
- es sollte auch möglich sein mehrere nebenläufige Multimedia−Anwendungen zu haben
- derartige Anwendungen sollten dynamische Anforderungen an die Ressourcenverteilung stellen können

Ein weiteres wichtiges Problem, welches man auch im Zusammenhang mit QoS Anwendungen feststellte, war der sogenannte *QoS crosstalk*, der bei Anwendungen auftritt, bei denen mit zeitkritischen Datenströmen über das Netzwerk gearbeitet wird.
Dieser erzeugt unerwünschte Verzögerungen, die auf den Zugriff mehrere konkurrierender Prozesse auf einen gemeinsam geteilten Übertragungskanal zurückzuführen sind. Allgemein hat man festgestellt, dass dieses Phänomen bei allen Ressourcen auftritt, die von zeitkritischen Anwendungen geteilt werden.
Auf der Grundlage dieser Beobachtungen und Anforderungen entstand in den Jahren 1994−1995 an der University of Cambridge das Betriebsystem Nemesis, dessen erste Version von Timothy Roscoe, David Evers und Paul Braham geschrieben wurde. Die erste Version von Nemesis lief auf Digital 3000/400 AXP und 5000/25 (Maxine) Workstations.
Die weitere Entwicklung des Systems wurde dann von mehrere Personen fortgeführt, so dass die heutige Version auf mehreren verschiedenen Plattformen wie beispielsweise Plattformen mit einer x86 Struktur oder auch Alpha Workstations lauffähig ist.

2. Aufbau von Nemesis

2.1 Struktur

Die Struktur von Nemesis ähnelt der eines Mikrokernel Betriebssystem, ein kleiner Kern, der die Verbindung zwischen der darunter liegenden Hardware− und der Anwendungsebene herstellt, und möglichst viel Funktionalität, die ausgelagert ist.

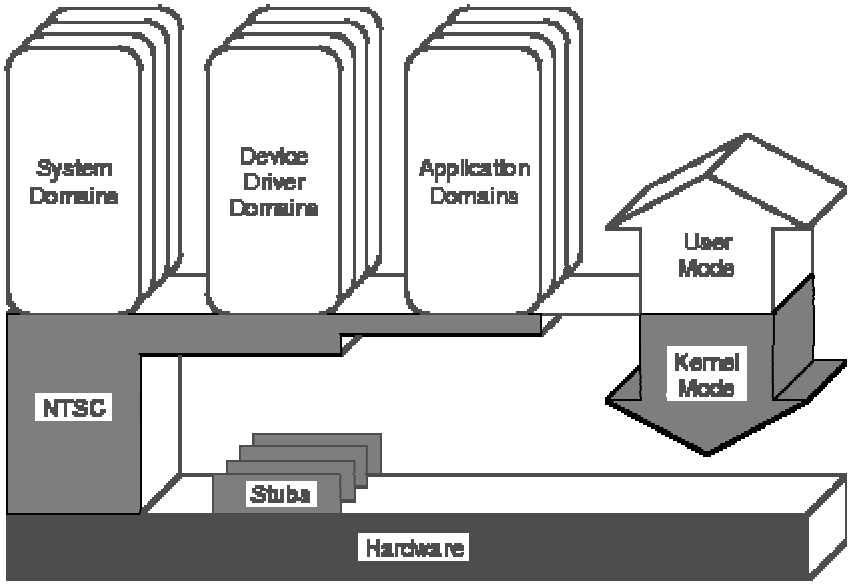

Abb. 1

Betrachtet man Abb. 1, so fällt die vertikale Struktur auf, die Nemesis hat. Diese ist darauf zurückzuführen, dass der Hauptteil der ausgelagerten Funktionalität des Systems nicht durch Server realisiert wird, wie es bei normalen Mikrokernel OS der Fall wäre, sondern dass ein Großteil in den einzelnen Applikationen selbst mit Hilfe von *shared libraries* ausgeführt wird. Nur ein geringer Teil, der fast nur noch Zugriffe auf die geteilte Ressourcen und das Multiplexen betrifft, wird auf einen möglichst niedrigen Level durch System− bzw. Device−Domains erledigt. Durch diese Strategie wird der oben erwähnte *QoS crosstalk* möglichst gering gehalten und es hat zusätzlich den Vorteil, dass die laufenden Programme möglichst viel Kontrolle über ihre Rechenzeit behalten, was bei Multimedia−Anwendungen besonders wünschenswert ist.
Um dies möglichst effizient zu realisieren, hat man in Nemesis das Konzept des gemeinsamen virtuellen Adressraums (*Single Address Space*) gewählt. Darunter versteht man, dass nicht wie üblich jeder Prozess seinen eigenen virtuellen abgekapselten Adressbereich besitzt, sondern dass alle Prozesse den gleiche Adressraum sehen und unter Einschränkung von Zugriffsrechten nutzen können. Dabei ist zu beachten, dass auch jede physikalische Adresse die gleiche virtuelle Adresse besitzt. Durch dieses Konzept wird erreicht, dass nicht jeder Prozess seine eigene Kopie der gesamten benötigten Bibliotheksfunktionen im Speicher haben muss, sondern dass man einfach das gleiche Textsegment unter mehreren Prozessen teilen kann. Bestimmte Funktionen, die von allen Prozessen sehr häufig genutzt werden, sind deshalb auch in einen permanenten gemeinsamen Speicherbereich abgelegt.

2.2 Kernel

Der Kernel von Nemesis ist sehr klein und man könnte ihn auch als Minimalkern bezeichnen, jedoch nennen ihn die Entwickler selbst "verticalized kernel", was auf die Anpassung an die vertikale Betriebssystemstruktur zurück zu führen ist. Er besteht im wesentlichen aus Atropos, den Scheduler für die CPU, und wenigen grundlegenden Funktionen, wie die Behandlung von Prozessor Exceptions, Speicherfehlerbehandlung und ähnlichem.
Neben den grundlegenden Funktionen enthält der Kern noch den sogenannten Nemesis Trusted Supervisior Code (NTSC), den man in zwei Kategorien einteilen kann.
Zum einen gibt es den unprivilegierten NTSC, der von User−Domains aufgerufen werden kann, um mit dem Scheduler zu interagieren oder um einfache Inter−Domain Kommunikation zu ermöglichen.
Zum anderen gibt es noch den privilegierten NTSC, der dazu dient den Prozessor Modus und die Interruptprioritäten zu ändern sowie für die Registrierung von sogenannten first−level Interrupt Stubs zu ermöglichen.
Diese Stubs dienen dazu Hardware Interrupts im Kern in einfacher Weise zu verarbeiten. Der Kern leitet die auftretenden Hardware Interrupts an die first−level Interrupt Stubs weiter, die dann in den meisten Fällen nur die Aufgabe haben den Gerätentreibern direkt einen Event zu schicken, der diesen darüber informiert, dass ein bestimmter Interrupt aufgetreten ist.

2.3 Domains

Domains sind Programme in Ausführung unter Nemesis. Sie stellen eine Ansammlung von zusammengehörenden Threads zur Erfüllung einer bestimmten Aufgabe dar.
Viele Funktionen des Betriebsystems in Nemesis werden von den Domains selbst erledigt, wie zum Beispiel das Behandeln von Page−Faults. Tritt ein Page Fault auf, wird er nicht von einen globalen "System Pager" bearbeitet, sondern bei der nächsten Zuteilung der Domain durch einen sogenannten Stretch Driver der Domain abgehandelt, weshalb man die Domains auch als *self−paging* bezeichnet. Diese einzelnen Funktionen können entweder durch den Programmcode in der Domain selbst implementiert sein oder es wird auf die gemeinsame Bibliothek zugegriffen, in der sich beispielsweise komplette Threadpackages befinden, um diese Aufgaben zu erfüllen.
Ein wichtiger Bestandteil, der zu einer Domain gehört, ist der Domain Control Block (DCB), mit Hilfe dessen die Verwaltung der Domains geregelt wird und der noch für Teile der einfachen Kommunikation einzelner Domains genutzt wird. Wird eine neue Domain erzeugt, so geschieht dies über den sogenannten Domain−Manager, der einen solchen DCB mit allen nötigen Anfangsdaten durch privilegierten NTSC erzeugt und diesen in die Datenstruktur des Schedulers linkt.
Der DCB ist ein Per−Domain Datensegment, dass sich in zwei Teile untergliedert. Der eine Teil stellt einen read −only Bereich dar, der nur von privilegierten NTSC−Calls verändert werden kann, in dem die Informationen für das Scheduling der Domain abgelegt sind. Zudem befinden sich hier noch weitere Informationen, die für die Verwaltung der Domain vom System genutzt werden, wie beispielsweise die Zugriffsrechte der Domain auf den Speicher oder auch Teile für die rudimentäre Kommunikation in Nemesis, die nur Leserechte der Domain erfordern.
Für den andere Teil des DCB hat die Domain sowohl Lese− als auch Schreibrechte, sein Verwendungszweck ist einerseits die Sicherung des Prozessorkontextes der einzelnen Threads und andererseits befinden sich hier noch die Anteile der Inter Domain Kommunikation, auf die die Domain mit Lese− und Schreibrechte zugreifen können muss.

2.4 Scheduling

Das Scheduling der CPU unter Nemesis wird in zwei Stufen abgewickelt, einerseits findet zwischen den einzelnen Scheduling Domains (Sdoms) ein Scheduling statt (Inter−Process Scheduling) und andererseits wird innerhalb der Domains selbst nach bestimmten Kriterien die Rechenzeit auf die einzelnen Thread verteilt(Intra−Process Scheduling).
Mit den Begriff Sdoms sind hierbei sowohl einzelne Domains als auch sogenannte Best−Effort Klassen gemeint. Best−Effort Klassen sind Verbände von einer oder mehreren Domains, denen vom Scheduler immer nur die ungenutzte Rechenzeit zur Erfüllung ihrer Ausgaben zugeteilt wird.
Wichtige Aspekte, die dabei sichergestellt werden müssen, besonders beim Inter−Process Scheduling sind:

- Zuteilung des Prozessor unter Berücksichtigung der QoS−Parameter einzelner Domains
- Keine unerlaubte Überschreitungen der zugeteilten Zeit einzelner Domains
- Blockierung bzw. Freigabe einzelner Domains, je nach ihren Anforderungen und eintreffenden Events

2.4.1 Inter−Process Scheduling

Für das Scheduling zwischen den einzelnen Sdoms ist der Kernel−Scheduler Atropos zuständig.
Um diese Aufgaben durchzuführen, wird er mit einen Tupel {s, p, x, l} von QoS−Parametern für jede einzelne Sdom versorgt. Dabei gibt s die erwünschte Zeitscheibengröße und p die Periode an, innerhalb der die gewünschte Rechenzeit zugeteilt werden soll. Zusammen bestimmen diese beiden Größe die Bandbreite an Rechenzeit, die der einzelnen Domain zugeteilt werden soll. Der Parameter x ist eine Variable vom Typ Boolean und gibt an, ob die Sdom wünscht überschüssige Rechenzeit zu nutzen, die niemanden zugeteilt ist. Der letzte Parameter l steht für latency hint und hilft dem Scheduler dabei Domains angemessen zu behandeln, die durch Blockieren in ihrer eigentlichen Perioden keine Rechenzeit zugewiesen bekommen haben.
Intern arbeitet Atropos nach den Earliest Deadline First (EDF) Algorithmus, der auf den Prinzip basiert, dass derjenige Prozess zuerst bedient wird, der noch keine Rechenzeit erhalten hat und dessen Periode p am weitesten verstrichen ist. Für jede Sdom besitzt daher der Scheduler eine Deadline d, die angibt wie viel Zeit der Periode einer Domain schon verstrichen ist, und eine Variable r (remaining), die angibt wie viel Rechenzeit einer Domain, in ihrer Periode noch zu steht. Mit Hilfe dieser beiden Parameter verteilt nun der Scheduler entsprechend die Rechenzeit.
Da während des Betriebs des Systems die Anzahl der laufenden Programme sich ändern kann, müssen dementsprechend auch die Daten für den Scheduler modifiziert werden. Dies wird separat durch den QoS Manager realisiert, der auch darauf achtet, dass stets weniger als 100% Rechenzeit an Sdoms zugeteilt wird.

2.4.2 Intra−Process Scheduling

Die interne Scheduling−Strategie einer Domain ist auf die einzelnen Aufgaben der Anwendung angepasst, sie kann wie alle anderen Funktionalität entweder durch ein vorgefertigtes Bibliothek−Paket oder durch eigens dafür implementierten Code realisiert werden.

2.5 Virtual Prozessor Interface

Eng Verknüpft mit dem Kernel−Scheduler ist das virtuelle Prozessor Interface, es stellt ähnlich wie bei anderen Betriebssystemen die Kommunikationsschnittstelle zwischen Kern und Domains dar, die eine Hardware−unabhängige Nutzung des Prozessor ermöglicht.
Im Gegensatz zu anderen Betriebssystemen wird den Anwendungen nicht eine exklusive Nutzung der CPU vorgetäuscht, sondern das Interface versorgt die einzelnen Anwendungen mit wichtigen Informationen, die diese in ihre internen Scheduling−Entscheidung einbeziehen.
Um dies zu realisieren, nutzt es die DCBs der Domains, in denen sich alle wichtigen Daten zur Verwaltung der Domains befinden.

2.5.1 Prozessorzuteilung

Die Zuteilung des Prozessor und damit die (Re−)Aktivierung der Domain unter Nemesis ist in der Regel nicht ein Wiederherstellen (Resume) des Zustands, bevor der Domain die CPU entzogen wurde, sondern die Domain wird über ihren activation handler aufgerufen (Upcall). Bei der Aktivierung werden über einen Stapel im DCB Informationen wie Grund und Art des Aufrufes, aktuelle Systemzeit, Zeitpunkt, zu dem die Domain die CPU abgeben musste, an die Domain weitergegeben. Diese verwendet der activation handler dann dafür, unter Einbeziehen zusätzliche Events, die eventuell eingetreten sind, spezifische Entscheidungen zu treffen, wie mit dem Programmablauf fortzufahren ist.
Es gibt nur eine Ausnahme, bei der eine Wiederherstellung der Aktivierung der Domain vorgezogen wird, und diese tritt dann auf, wenn ein Thread der Domain, während der Ausführung eines kritischen Bereiches unterbrochen wurde.
Ob nun ein Upcall oder ein Resume der Domain durchgeführt, stellt das Virtual Prozessor Interface über das sogenannte Aktivierungsbit im DCB fest.

2.5.2 Events

Den einzigen Kommunikationsmechanismus den Nemesis direkt unterstützt, ist das Senden von Ereignissen.
Jede Domain besitzt in ihrem DCB eine Tabelle für Verbindungsendpunkten für Ereigniskanäle (Event Channels), über die Verbindungen zu anderen Domains ermöglicht werden. Einer dieser Punkte ist bereits von der Erzeugung der Domain an mit dem *Binder* verbunden, über den die Domain alle anderen Verbindungen zu den gewünschten Domains aufbauen kann. Eine Event wird unter Nemesis als Integerwert repräsentiert, der durch einen send() −Aufruf atomar verändert werden kann.
Abb. 2 zeigt wie ein Prozess A über den Kernel ein Ereignis n an den Prozess B sendet. Dabei wird mit dem sys_send_event(n) Aufruf, der Wert des Events n, in der Ereignisstabelle des Prozesses A, in die Ereignistabelle des Prozesse B kopiert und zusätzlich der Event in einen Ereignispuffer eingetragen.
Die Informationen an welchen Prozess das Ereignis n geschickt werden sollen, entnimmt der Kernel aus einer Tabelle des DCBs von Prozess A.

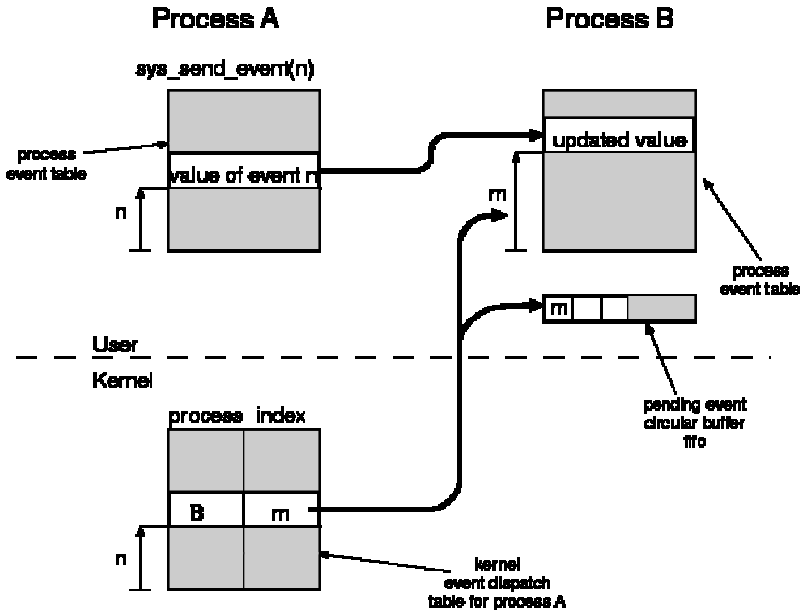

Abb. 2

Über diesen einfachen Ereignismechanismus werden höhere Inter Domain Kommunikation unter Nemesis realisiert.

2.6 I/O Channels (RBuf)

I/O Channels bzw. RBufs sind ein Beispiel dafür, wie unter Nemesis höhere Inter Domain Communication (IDC) auf dem Konzept von Event Channels und Shared Memory aufbaut.
I/O Channels sind für den Transfer von größeren Datenmengen zwischen zwei Domain gedacht, dafür werden drei gemeinsame Speicherbereiche benötigt, ein größerer Bereich, der Datenbereich (*Data Area*), und je ein kleinerer Bereich, der Kontrollbereich (*Control Area*), in der Sender− bzw. Empfängerdomain, der als Puffer für die io_recs genutzt wird.
Die io_recs sind Paare von Adressbasis und Länge der einzelnen Pakete, die im Datenbereich abgelegt sind, und liefern so der Domain alle relevanten Information für den Zugriff und die Bearbeitung dieser.
Die Koordinierung, ob und wo neue io_recs im Kontrollbereich liegen, wird über Events gesteuert, die Anfang und Ende des beschriebenen Pufferbereichs angeben, d.h. jeder Kontrollbereich benötigt noch zwei Event Channel zur Synchronisation.
Die Kommunikation verläuft nun nach dem Master−Slave−Prinzip, d.h. eine Domain übernimmt die Rolle des Master und hat dadurch die Kontrolle über den Datenbereich, sie legt Größe und Position der einzelnen Datenpakete im Speicherbereich fest, und übermittelt diese per io_recs zur Slave−Domain.
Die Slave−Domain kann nun diese Pakete übernehmen und Pakete gleicher Größe und Position zurücksenden.
I/O Channels können in drei verschiedenen Modi laufen:

i. Rx (receive) mode:
   Der Master schickt "leere" Pakete an den Slave, welcher diese dann mit Daten füllt und zurücksendet, d.h. der Master gibt nur Bereiche im Datenbereich an, in die ihm der Slave Daten schreiben soll.
ii. Tx (transmit) mode:
   Der Master schickt io_rec mit Datenbereich, die Informationen für den Slave enthalten, und der Slave schickt "leere" Paket zurück bzw. gibt diese wieder für den Master frei.
iii. Dx (duplex) mode:
   Sowohl Master als auch Slave schicken io_recs, die Datenbereich mit Inhalt beschreiben.

Abb. 3 zeigt die schematische Struktur eines IO Channels.

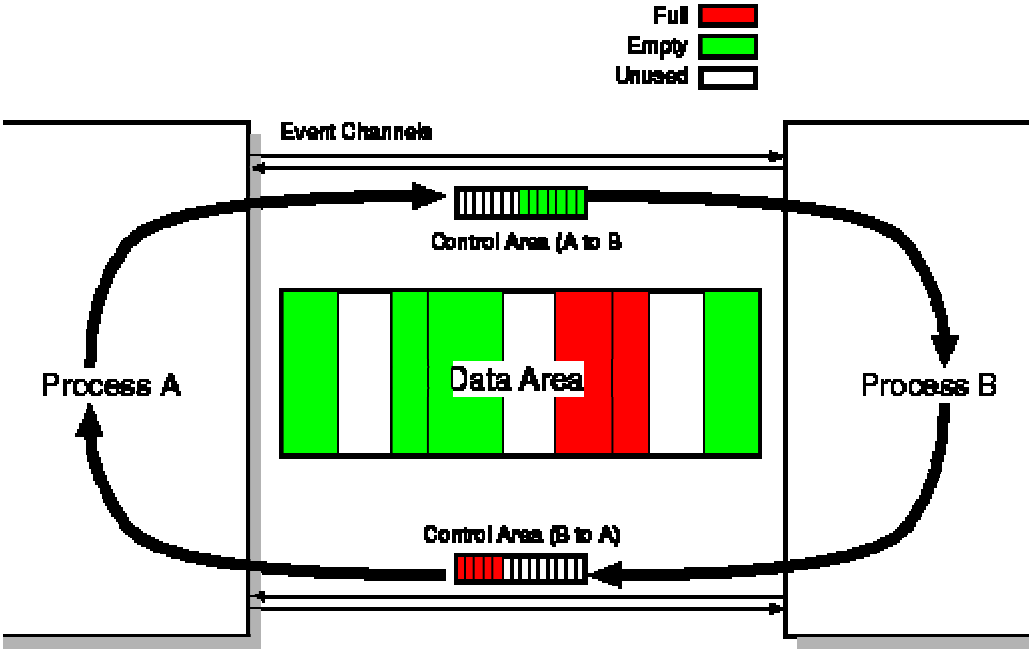

Abb. 3

2.7 Gerätetreiber

Bei der Implementierung von Gerätetreibern unter Nemesis ging man einen etwas ungewöhnlichen Weg, was auch darauf zurückzuführen ist, dass die Grundidee beim Entwurf von Nemesis war, alle verfügbaren Ressourcen auf einer möglichst niedrigen Ebene unter den Anwendungen zuteilen. Damit ein Gerätetreiber überhaupt QoS−Garantien gewähren kann, muss er wissen, von welcher Domain die Daten stammen, die er bearbeitet. Deshalb sind alle Gerätetreiber unter Nemesis verbindungsorientiert. Auch hier wird nur ein kleiner Teil der E/A−Operationen vom Treiber selbst ausgeführt, der Hauptteil wird von den Client unter Verwendung der shared libraries durchgeführt.

In Abb. 4 wird die Grundstruktur eines Gerätetreibers dargestellt, wie unschwer zu erkennen ist, besteht er aus zwei Bestandteilen, die meistens in derselben Domain impelmentiert sind:

- Den Device Data−Path Module
- Den Device Control−Path Module

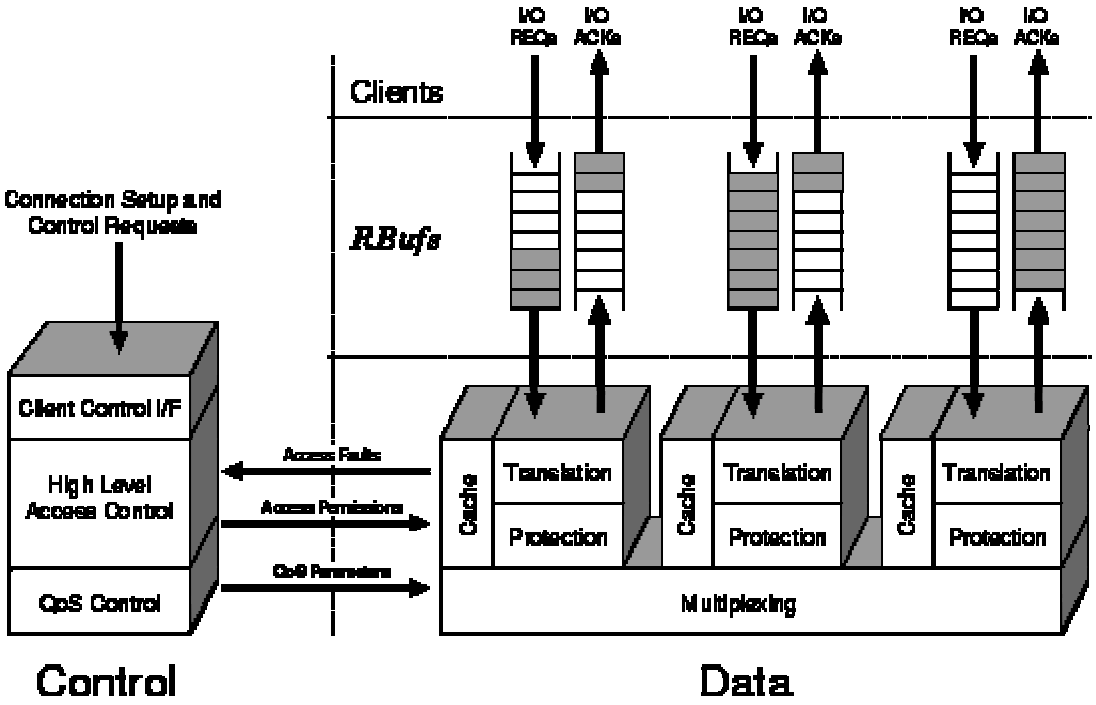

Abb. 4

2.7.1 Device Data−Path Module (DDM)

Der DDM hat die Aufgabe einen sicheren Zugang zur Hardware für die Clients unter Einbezug von QoS−Garantien zu ermöglichen, dabei ist zu beachten, dass nur ein Minimum an nötiger Funktionalität implementiert ist, um dies zu gewährleisten.
Die drei Hauptaufgaben des DDM sind:

- Die Übersetzung der Adressrepräsentation in ein passendes Format (Translation)
- Schutz vor unberechtigten Zugriff des Client (Protection)
- Multiplexen der Resource unter den verschiedenen Clients unter Beachtung der QoS−Anforderungen

Man sieht, dass das DDM nur für den direkten Zugriff auf die Resource zuständig ist und sich weder mit der Einrichtung einer Verbindung noch mit der Festlegung der QoS−Parameter auseinander setzen muss, dies verhindert unnötigen *QoS crosstalk.*

2.7.2 Device Control−Path Module (DCM)

Das DCM ist dafür zuständig Verbindungen für die einzelnen Anwendungen zum dem Gerät aufzubauen, was auch die Festlegung der QoS−Parameter, der benötigten Adressrepräsentation und den Schutz vor unberechtigten Zugriff betrifft. Dies wird dadurch realisiert, dass das DCM alle benötigten Daten bereitstellt und dann über die Multiplexerschicht des DDM die Verbindung für die einzelnen Anwendungen einrichtet.
Falls Änderungen in den Anforderungen einer Anwendungen auftreten, läuft dies auch über den DCM und der DDM kann währenddessen seinen Betrieb ungestört fortsetzen. im Pegasus Projekt

2.8. Literatur

(1) Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Braham, David Evers, Robin Fairbairns, Eoin Hyden : „The Design and Implementation of an Operating System to Support Multimedia Applications“ http://www.cl.cam.ac.uk/Research/SRG/netos/old−projects/nemesis/documentation.html

(2) „The Nemesis System Documentation“ http://nemesis.sourceforge.net