INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl Prof. Dr. Heinz-Gerd Hegering

Fortgeschrittenenpraktikum

Implementierung einer MIB für Systemmanagementaufgaben

Rainer Hauck
Nedo Haubelt

30. November 1994

Betreuer: Markus Gutschmidt, Dr. Bernhard Neumair

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Durch die Ablösung von wenigen, homogenen Großrechnern durch Client-Server-Architekturen und Workstationverbunde gewannen heterogene Systemumgebungen immer mehr an Bedeutung. Für die Steuerung und Verwaltung der Vielzahl von Workstations ergibt sich nunmehr ein erhöhter Aufwand.

Um in einer heterogenen Systemumgebung viele Rechner unterschiedlicher Hersteller zu verwalten, werden Techniken, die im Netzmanagement erfolgreich eingesetzt werden, für das Systemmanagement verwendet. Ziel des Systemmanagements ist es den Systembenutzer und den Systembetreiber während des Betriebs von verteilten Systemen und ihrer Endsysteme zu unterstützen.

Zur Überwachung von Endsystemen werden Systemmanagementagenten eingesetzt, die auf dem jeweiligen System laufen und die managementrelevanten Daten des Rechners zur Verfügung stellen. Die Daten werden in einer Management Information Base (MIB) bereitgestellt und können über ein Managementprotokoll gelesen bzw. geschrieben werden. Als Managementprotokoll wird häufig das Simple Network Management Protocol in der Version 1 (SNMPv1) eingesetzt, wobei eine Tendenz zu SNMPv2 zu erkennen ist.

Im Rahmen des Fortgeschrittenenpraktikums wurde ein SNMP-Agent um einen eigenständigen Prozeß (Subagent) erweitert, der die managementrelevanten Daten eines Endsystems bereitstellt. Dazu wurde eine bereits definierte Systemmanagement-MIB implementiert. Die Kommunikation zwischen dem Subagenten, der auf dem Endsystem läuft, und dem SNMPv2-Agenten geschieht dabei über das SNMP Distributed Protocol Interface Version 2.0 (SNMP DPI2.0), einem Protokoll, das in etwa die gleiche Funktionalität besitzt wie das SNMP-Multiplex-Prokoll.

1.2 Das Umfeld des DPI-Subagenten

Aktuelle Implementierungen von Systemmanagementagenten enthalten gleichzeitig das Managementprotokoll sowie die MIB, d.h. beide Aufgaben werden durch genau einen Prozeß realisiert. Wollte man die zugrundeliegende MIB erweitern, so mußte man den Agenten anhalten, die Änderungen vornehmen und den Agenten danach erneut starten. Um die Änderungen im Agenten überhaupt vornehmen zu können, benötigt man zudem den Source Code des Agenten.

Im Rahmen eines weiteren Fortgeschrittenenpraktikums wurde nun ein solcher SNMP–Agent dahingehend erweitert, daß er die Möglichkeit bietet, Teile der MIB, die ursprünglich durch den SNMP-Agenten implementiert wurden, durch andere Prozesse (Subagenten) zu implementieren. Der Subagent kommuniziert dabei mit dem SNMP–Agenten über ein bestimmtes Interprozeßkommunikationsprotokoll, das SNMP Distributed Protocol Interface v2.0. Der Subagent ist somit ein eigener Prozeß, der eine eigene MIB besitzt und diese über das DPI-Protokoll dem SNMP-Agenten zur Verfügung stellt.

Der SNMP-Agent bildet die Schnittstelle zwischen der Managementstation und dem 'eigentlichen' Systemmanagementagenten.

Auf der anderen Seite kommunizert der SNMP-Agent mit der Managementstation über das SNMP-Protokoll. Kommt eine Anfrage von der Managementstation, sucht der Agent die angefragte Variable in seiner MIB. Diese setzt sich dabei aus seiner internen MIB und den MIBs der einzelnen Subagenten zusammen. Damit ein Agent die MIBs seiner Subagenten kennt, müssen sich diese bei dem Agenten anmelden und ihre MIB beim ihm registrieren. Ein Subagent kann sich dabei jederzeit bei einem Agenten an- bzw. abmelden und dabei die gesamte MIB oder nur Teile davon bei diesem registrieren. (Zur weiteren Information siehe [2] und Kap. 2).

Kapitel 2

Die DPI-Schnittstelle aus Sicht des Subagenten

2.1 Überblick

In diesem Kapitel werden nur die für den Subagenten relevanten Teile des SNMP DPI2.0 Protokolls aufgeführt. Die genaue Beschreibung dieses Protokolls (DPI-Architektur und DPI-Paketformat) ist in [2] nachzulesen.

Das DPI-Protokoll wird als Interprozeßkommunikationsschnittstelle zwischen dem SNMP-Agenten und seinen Subagenten, die als eigenständige Prozesse laufen, verwendet. Mit Hilfe dieses Protokollschnittstelle ist es nunmehr möglich die MIB eines Agenten zu erweitern, ohne diesen explizit anzuhalten bzw. Veränderungen an ihm vorzunehmen.

Um Erweiterungen an der MIB im Agenten vorzunehmen, muß sich ein Subagent beim Agenten via DPI-Schnittstelle anmelden und die gesamte MIB bzw. Teilbäume davon bei diesem registrieren. Um eine Verbindung zum Agenten aufzubauen, schickt der Subagent ein DPI OPEN Paket an den Agenten. Dieser bearbeitet die Anfrage des Subagenten und schickt ein entsprechendes DPI RESPONSE Paket zurück.

Nach Erhalt des RESPONSE Paketes muß der Subagent die Teilbäume, die er bereitstellt, beim Agenten registrieren, damit dieser weiß, welche Variablen vom Subagenten bereit gestellt werden. Dazu schickt der Subagent ein DPI REGISTER Paket an den Agenten. Auch diese Anfrage wird vom Agenten durch ein entsprechendes DPI RESPONSE Paket an den Subagenten beantwortet.

Sind dem Agenten die vom Subagenten bereitgestellten Variablen bekannt, so ist damit die Erweiterung der MIB im Agenten abgeschlossen.

2.2 Beschreibung der DPI Architektur

Vom Subagenten werden folgende Requests an den Agenten geschickt:
OPEN, REGISTER, UNREGISTER, CLOSE

Der Subagent erhält auf alle Requests bis auf den CLOSE Request ein RESPONSE Paket vom Agenten. Die RESPONSE Pakete vom Agenten dienen dazu, den Erfolg oder Mißerfolg eines Requests anzuzeigen.

- Der Subagent eröffnet eine Verbindung zum Agenten, indem er ihm einen DPI OPEN Request schickt.
- Durch das Senden eines oder mehrerer DPI REGISTER Pakete an den Agenten, meldet der Subagent einen oder mehrere Teilbäume beim Agenten an.
- Will der Subagent seine Arbeit beenden, so schickt er ein DPI UNREGISTER Paket gefolgt von einem DPI CLOSE Paket an den Agenten. Der Agent antwortet nur auf das UNREGISTER Paket mit einem RESPONSE Paket.
- Der Agent anwortet nicht mit einem RESPONSE Paket auf einen DPI CLOSE Request vom Subagenten. Er beendet die Verbindung zum Subagenten und entfernt alle von ihm registrierten Teilbäume.

Der Subagent kann folgende Requests vom Agenten empfangen und verarbeiten:
GET, GETNEXT, SET, COMMIT, UNDO, UNREGISTER, CLOSE

Die DPI Requests GET, GETNEXT und SET sind direkt von den entsprechenden SNMP Requests, die von der Managementstation beim Agenten eintreffen, abgeleitet [2].

Die DPI Requests COMMIT und UNDO dienen zur korrekten Durchführung eines SET Requests. Mit Hilfe der DPI UNREGISTER und DPI CLOSE Requests kann der Agent seinerseits die Verbindung zu einem Subagenten beenden.

Auf alle Requests, bis auf den DPI CLOSE Request, antwortet der Subagent mit einem RESPONSE Paket.

- Die beim Agenten eintreffenden SNMP Requests werden in korrespondierende DPI Requests umgewandelt.
- Mit Hilfe eines GET bzw. GETNEXT Requests können eine oder mehrere Variablen gelesen werden. Beim Agenten eintreffende SNMP GETBULK Requests werden in mehrere DPI GETNEXT Requests umgewandelt.
- Mit SET Requests werden MIB-Variablen geschrieben. Zur Bestätigung werden DPI COMMIT und UNDO Requests verwendet.

- Möchte der Agent seinerseits eine Verbindung zum Subagenten beenden, so schickt er einen DPI UNREGISTER Request gefolgt von einem DPI CLOSE Request.

Kapitel 3

Implementierung des Subagenten

3.1 Anmeldung beim DPI-Agenten

3.1.1 Eröffnen einer Verbindung

Bevor ein Subagent DPI-Pakete vom Agenten empfangen bzw. an diesen schicken kann, muß er sich beim Agenten anmelden und sich als Subagent identifizieren. Beide Schritte werden durch die Funktion do_connect_and_open() realisiert, die sich im Modul 'dpi_system.c befindet. Die an diese Stelle aufgerufenen Funktionen, die dem tatsächlichen Verbindungsaufbau dienen, befinden sich in den Modulen 'snmp_lDPI.c', 'snmp_mDPI.c' sowie 'snmp_qDPI.c'.

Um eine Verbindung zum Agenten zu eröffnen, wird in do_connect_and_open() die Funktion DPIconnect_to_agent_TCP() aufgerufen. Sie erwartet zwei Argumente:

- einen Hostnamen (Name oder IP-Adresse) der den Rechner spezifiziert, auf dem der Agent läuft. Laufen Agent und Subagent auf dem gleichen Rechner, so kann hier der Name 'loopback' oder 'localhost' verwendet werden.
- einen Community Namen, der verwendet wird, um einen DPI TCP-Port vom Agenten zu erhalten. Wir verwenden den Community Namen 'public'.

Wurde die Verbindung zum Agenten erfolgreich aufgebaut, gibt die Funktion einen Deskriptor zurück, der die Verbindung zum Agenten repräsentiert. Dieser Deskriptor wird hauptsächlich beim Senden und Empfangen von DPI-Paketen verwendet. Im Fehlerfall wird ein negativer Fehlercode zurückgegeben.

Im einem zweiten Schritt identifizieren wir uns beim Agenten. Wir bauen uns hierzu ein DPI OPEN Paket, schicken es an den Agenten und warten danach auf dessen RESPONSE Paket. Der Agent kann dabei unseren OPEN Request verweigern oder akzeptieren.

Ein DPI OPEN Paket wird durch den Aufruf von mkDPIopen() aufgebaut. Diese Funktion erwartet dabei folgende Argumente:

- einen (eindeutigen) Object Identifier, der unseren Subagenten spezifiziert.
- eine optionale Beschreibung des Subagenten.
- einen Timeout (in Sekunden). Der Agent verwendet diesen Wert als Timeout für ein RESPONSE Paket, wenn er eine Anfrage an den Subagenten stellt. Der Agent seinerseits verwaltet seinen eigenen Timeout-Wert. Sollte unser Wert größer sein, so wird der Wert des Agenten verwendet.
- die maximale Anzahl von Variablen, die der SUbagent pro DPI-Paket verarbeiten kann.
- den Zeichensatz, den wir zur String-Repräsentation verwenden.
- die Länge des Paßwortes. (oder Null, falls kein Paßwort verwendet wird)
- einen Zeiger auf das Paßwort (oder NULL, falls kein Paßwort verwendet wird). Es hängt dabei vom Agenten ab, ob der Subagent ein Paßwort angeben muß, wenn er sich bei ihm anmeldet.

Als Rückgabewert liefert mkDPIopen() einen Zeiger auf einen statischen Speicherbereich, in dem das DPI OPEN Paket steht. Im Fehlerfall wird NULL zurückgegeben. Nachdem das OPEN Paket aufgebaut wurde, wird es an den Agenten geschickt. Dies wir durch die Funktion DPIsend_to_agent() realisiert. Sie erwartet folgende Argumente:

- den Deskriptor, der die Verbindung zum Agenten repräsentiert (von DPI_connect_to_agent_TCP())
- einen Zeiger auf das zu verschickende DPI-Paket (von mkDPIopen())
- die Länge des Paketes (die Header-Datei 'snmp_dpi.h' enthält das Makro DPI_PACKET_LEN, mit der die Länge eines Paketes berechnet wird)

Der Rückgabewert der Funktion ist entweder Null (wird durch DPI_RC_OK repräsentiert) oder ein entsprechender Fehlercode, der in 'snmp_dpi.h' definiert ist.

Nachdem wir nun das Paket an den Agenten geschickt haben, warten wir auf die Antwort auf unseren OPEN Request. Das Warten auf ein Paket vom Agenten wird durch die Funktion DPIawait_packet_from_agent() realisiert, welche folgende Argumente erwartet:

- den Deskriptor, der die Verbindung zum Agenten repräsentiert (von DPI_connect_to_agent_TCP())

- einen Timeout in Sekunden (die maximale Zeit, die der Subagent auf ein RESPONSE Paket vom Agenten wartet)
- einen Zeiger auf einen Speicherbereich, in den das ankommende Paket geschrieben wird (falls ein Fehler während des Empfangs auftritt, wird dort NULL hineingeschrieben)
- einen Zeiger auf einen 'LONG INTEGER'; dort wird die Länge des eintreffenden Paketes vermerkt bzw. NULL, falls ein Fehler aufgetreten ist.

Der Rückgabewert der Funktion ist entweder Null (DPI_RC_OK) oder ein entsprechender Fehlercode, der in 'snmp_dpi.h' definiert ist.

Haben wir ein Paket erhalten, müssen wir prüfen ob es sich dabei um ein RESPONSE Paket auf unseren OPEN Request handelt. Zudem müssen wir sicherstellen, daß uns der Agent als gültigen Subagenten akzeptiert hat. Dazu parsen wir das angekommene Paket durch den Aufruf von pDPIpacket(), das einen Zeiger auf ein Paket erwartet, und nehmen danach die entsprechenden Prüfungen vor.

Damit ist der reine Verbindungsaufbau zum Agenten abgeschlossen.

3.1.2 Registrierung der Teilbäume

Nachdem sich der Subagent erfolgreich beim Agenten angemeldet hat, muß er einen oder mehrere Teilbäume der MIB bzw. die gesamte MIB beim Agenten registrieren, damit dieser weiß, welche Variablen vom Subagenten bereitgestellt werden.

Die von dem Subagenten registrierte MIB hat die in Abbildung 1 dargestellte Struktur:

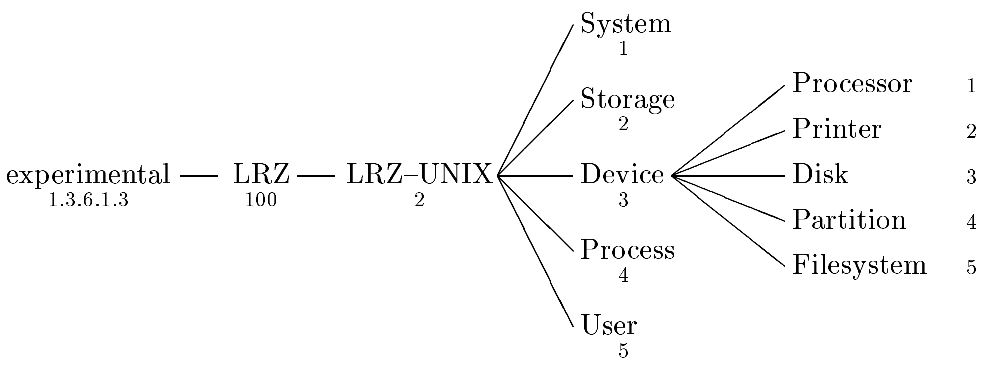

Abbildung 1: LRZ–UNIX MIB

Um sich zu registrieren, erzeugt der Subagent ein DPI REGISTER Paket und schickt es an den Agenten. Der Agent seinerseits antwortet darauf mit einem RESPONSE Paket um den Erfolg bzw. Mißerfolg des REGISTER Requests anzuzeigen.

Um ein DPI Register Paket zu erzeugen, rufen wir die Funktion mkDPIregister() auf, welche folgende Argumente erwartet:

- einen Timeout (in Sekunden) für den zu registrierenden Teilbaum. Wird hier 0 angegegeben, wird der Timeout aus do_connect_and_open() verwendet. Hier können jeweils verschiedene Werte angegeben werden, wenn man für die einzelnen Teilbäume unterschiedliche Verarbeitungszeiten erwartet.
- eine gewünschte Priorität. Mehrere Subagenten können den gleichen Teilbaum mit unterschiedlichen Prioritäten registrieren (dabei ist Prio 0 besser als Prio 1 usw.). Der Subagent mit der höchsten Priorität ist zuständig für den jeweiligen Teilbaum. Wird hier -1 angegeben, so wird die höchste verfügbare Priorität angefordert. Wird hier 0 angegeben, so wird für den Subagenten eine höhere Priorität angefordert, als die bereits höchste vergebene Priorität.
- den MIB-Teilbaum (als OID), den der Subagent verwalten möchte. (die OID muß dabei mit einem '.' enden)
- ein Flag, das anzeigt, ob GETBULK Requests vom Subagenten verarbeitet werden können, oder ob er vom Agenten erwartet, daß ein GETBULK in mehrere GETNEXT Requests verwandelt werden. Der hier beschirebene Subagent kann keine GETBULK-Pakete verarbeiten und läßt deshalb diese Umsetzung vom Agenten vornehmen.

Nachdem das DPI REGISTER Paket erzeugt wurde, wird es an den Agenten geschickt. Danach warten wir auf Antwort. Nach Erhalt des RESPONSE Paketes vom Agenten, bleibt noch zu prüfen ob unsere Registrierung Erfolg hatte oder ob sie fehlschlug (analoge Vorgehensweise wie bei 3.1.1).

Ist die Anmeldung und Registrierung beim Agenten abgeschlossen, ist der Subagent in der Lage, Requests von einer Managementstation bzw. vom Agenten zu verarbeiten.

3.2 Bearbeitung eines eintreffenden Paketes

3.2.1 Allgemeine Überlegungen zur Paketbearbeitung

Die Funktion 'DPIawait_packet_from_agent()' wartet auf das Eintreffen eines Paketes und liefert einen Zeiger auf das empfangene Paket. Dieses wird mit Hilfe der Funktion 'pDPIpacket()' in seine einzelnen Bestandteile zerlegt und in einer Struktur 'snmp_dpi_hdr' gespeichert. Aufgrund des 'packet_type'-Feldes dieser Struktur wird unterschieden, um welche Art von Paket es sich handelt und dementsprechend eine Funktion zur weiteren Bearbeitung aufgerufen (do_get(), do_next(), ...).

Das Paket enthält weiterhin einen 'Object Identifier' (OID), der die Variable bestimmt, die in diesem Paket gelesen bzw. gesetzt werden soll. Aufgrund dieses OID ist es nun erforderlich, die entsprechende GET- bzw. SET-Funktion aufzurufen. Hierzu mußte eine Methode gefunden werden, in angemessener Zeit, die aufzurufende Funktion zu bestimmen.

Ein weiteres Problem ergibt sich beim Zugriff auf Tabellen. Die MIB enthält außer einfachen Variablen auch Tabellen, die zum Teil auch ineinander verschachtelt sein können. Auch sind die Indexe mancher Tabellen keine Integerwerte sondern Strings. Das heißt, daß weder die Anzahl noch der Typ der Indexe für alle Variablen gleich sind. Da aber eine einheitliche Aufrufsyntax für alle Funktionen wünschenswert ist, konnten die Indexe nicht einfach als Parameter übergeben werden.

Große Schwierigkeiten bereitet die Behandlung von GETNEXT Requests. Hier muß statt der dem OID des Paketes entsprechenden Variable, die in der MIB lexikographisch folgende Variable zurückgegeben werden. Das bedeutet, daß bei einfachen Variablen die nächste Variable bestimmt werden muß und bei Tabellen der Eintrag mit dem nächstgrößeren Index. Sollte kein größerer Index in der Tabelle existieren, so muß wiederum die nächste Variable gesucht werden. Handelt es sich bei der nächstgrößeren Variable wieder um eine Tabelle, so ist der kleinste Index dieser Tabelle auszuwählen. Bei ineinander verschachtelten Tabellen steigt die Komplexität.

3.2.2 Verwendete Datenstrukturen

Um die Suche nach der aufzurufenden Funktion effizient zu realisieren, wurden zwei Felder definiert, die jeweils einen Eintrag pro Variable enthalten. Die Felder befinden sich im Modul 'device.c'.

Das Feld 'var_table' enthält für jede Variable einen Eintrag, der ihren OID in Form eines 'array of int' beinhaltet. Nicht verwendete Stellen werden mit negativen Zahlen belegt. Der letzten Zahl kommt hierbei eine besondere Bedeutung zu, da sie, ebenfalls negativ dargestellt, die Anzahl der nicht-negativen Stellen und damit die Länge des OID enthält.

6. Stelle des OID   Anzahl benutzter Stellen

{3, 1, 3, 1, 1, 2,-1,-1,-1,-6}   **/* diskAccess */**

1. Stelle des OID   unbenutzte Stellen

Abbildung 2: var_table

Das Feld 'var_desrc_a' enthält für jede Variable Zeiger auf Funktionen zum Lesen bzw. Setzen von Variablen, den Typ der Variablen sowie, im Falle von Tabellenvariablen, einen Zeiger auf eine Funktion, die die Instanz mit dem nächstgrößeren Index bestimmt. Sollte eine der Funktionen für eine bestimmte Variable nicht existieren (z.B. 'read-only'-Variablen enthalten keine SET-Funktion), steht an dieser Stelle NULL.

SET Funktion existiert nicht, da Variable 'read-only' Variablentyp

{(void*)&get_diskAccess, NULL, SNMP_TYPE_Integer32, (void*)&get_next_Disk}

Zeiger auf GET Funktion Zeiger auf GETNEXT Funktion

Abbildung 3: var_descr_a

Die Indizierung der beiden Felder ist identisch, so daß aus der ersten Tabelle der Index bestimmt werden kann und mit Hilfe dieses Indexes aus der zweiten Tabelle die entsprechende Funktion.

3.2.3 GET-Pakete

Die Suche nach der aufzurufenden Funktion gestaltet sich für GET-Pakete folgendermaßen: Der als String vorliegende OID wird mit Hilfe der Funktion 'cut_instance()' in ein 'array of int' zerlegt, um die Vergleichsoperationen einfacher zu gestalten. Dann wird die Funktion 'get_number()' aufgerufen, die den Feldindex dieser Variable aus dem Feld 'var_table' bestimmt. Sollte die Variable nicht existieren, so liefert 'get_number()' einen negativen Wert, aufgrunddessen die Meldung 'noSuchObject' bzw. 'noSuchInstance' erzeugt wird. Mit Hilfe des erhaltenen Feldindexes läßt sich nun aus dem Feld 'var_desrc_a' die aufzurufende Funktion ermitteln.

Um eine einheitliche Aufrufsyntax für alle Funktionen zu erhalten, werden eventuell vorhandene Parameter ebenfalls durch die Funktion 'cut_instance()' im Anschluß an den OID als 'array of int' abgespeichert. Der Beginn der Parameter in diesem Feld ergibt sich aus der Anzahl der für den OID benötigten Stellen, die im Feld 'var_table' enthalten ist. Somit ist es möglich, jeder Funktion nur einen Zeiger auf diese Parameterliste zu übergeben, für deren Interpretation die aufgerufene Funktion selbst verantwortlich ist.

Nachdem die Funktion bestimmt ist, muß sie aufgerufen werden und der Rückgabewert der GET Request in einem RESPONSE-Paket an den Agenten zurückgeschickt werden.

3.2.4 SET-Pakete

Die Suche nach der aufzurufenden Funktion gestaltet sich beim SET Paket genauso wie beim GET-Paket. Allerdings ist hierbei noch eine Unterscheidung in SET-, COMMIT- und UNDO-Pakete zu treffen.

- SET-Paket

  Beim SET-Paket wird nur eine Überprüfung vorgenommen, ob der gewünschte Wert für diese Variable gesetzt werden kann, bzw. ob die für das Setzen der Variable erforderlichen Resourcen verfügbar sind.

- COMMIT-Paket

  Beim COMMIT-Paket wird der Wert der Variablen tatsächlich zugewiesen. Außerdem wird ein Eintrag in einer Liste vorgenommen, der die zu setzende Variable sowie ihren vorherigen Wert enthält. Dies dient dazu, den ursprünglichen Zustand evtl. später wieder herstellen zu können. Diese Liste wird beim Eintreffen des nächsten SET-Paketes wieder gelöscht.

- UNDO-Paket

  Beim UNDO-Paket muß nun die Variable aus der beim COMMIT-Paket erzeugten Liste ausgelesen werden und wieder auf ihren ursprünglichen Wert gesetzt werden. Ist dies aus irgendwelchen Gründen nicht möglich, so wird eine Fehlermeldung erzeugt.

3.2.5 GETNEXT-Pakete

Deutlich schwieriger gestaltet sich die Realisierung der GETNEXT-Funktionalität. Wie oben bereits beschrieben, wird zuerst, mit Hilfe von 'get_number()', der Feldindex der Variablen bestimmt. Sollte die Variable nicht existieren, so liefert 'get_number()' den Index der nächstgrößeren Variablen und kennzeichnet dies durch Darstellung als negative Zahl.

Sollte die Variable existieren, so muß unterschieden werden, ob es sich um eine einfache Variable oder um eine Tabelle handelt. Bei einfachen Variablen werden einfach der Feldindex um eins erhöht und die Parameter auf -1 gesetzt, damit, falls die nächste Variable eine Tabelle ist, das erste Tabellenelement ausgelesen wird. Bei Tabellen verbleiben Index und Parameter unverändert.

Wenn die Variable nicht existiert, so verwendet man den erhaltenen negativen Index, der ja bereits auf die nächstgrößere Variable zeigt und setzt analog die Parameter auf -1.

Nun beginnt eine 'while'-Schleife, die erst verlassen wird, wenn ein Variablenwert gefunden wurde oder wenn das Ende der MIB erreicht ist. Zuerst wird untersucht, ob der Index nun auf eine Tabelle verweist. Sollte dies der Fall sein, so muß eine Funktion aufgerufen werden, die für diese spezielle Tabelle den nächsten Tabelleneintrag ermittelt. Hierzu wird dieser Funktion ein Zeiger auf die Parameterliste übergeben, die dort entsprechend abgeändert wird. Sollte die Tabelle kein größeres Element mehr erhalten, so werden die Parameter auf -1 gesetzt. Dann kann die entsprechende GET-Funktion aufgerufen werden. Sollte diese GET-Funktion kein Ergebnis liefern, dann wird der Index wiederum um eins erhöht und die Schleife beginnt von Neuem. Dies ist insbesondere der Fall, wenn, im Falle einer Tabelle, die Parameter aus den oben beschriebenen Gründen auf -1 gesetzt sind.

3.3 Abmeldung beim DPI-Agenten

Falls ein Subagent seine Arbeit beenden möchte, meldet er sich bei seinem Agenten ab. Der Agent seinerseits kann ebenso die Verbindung zu einem angemeldeten Subagenten beenden, falls ein neuer Subagent einen REGISTER Request mit einer höheren Priorität stellt. Wir können dabei die Verbindung zum Agenten aufrechterhalten und die Kontrolle über unseren Teilbaum zurückerhalten, falls der Subagent mit der höheren Priorität sich wieder abmeldet. Weiterhin kann der Agent die Verbindung zum Subagenten unterbrechen, falls er vom SNMP-Manager dazu aufgefordert wird. Der Verbindungsabbruch wird hier durch das Senden eines DPI UNREGISTER Paketes initiiert.

Ein Verbindungsabbruch kommt ebenfalls zustande, wenn der Agent ein DPI CLOSE Paket an den Subagenten schickt. Dies ist der Fall, wenn der Agent einen Fehler erkannt hat oder wenn er vom Manager dazu aufgefordert wurde.

Der Aufruf der Funktion DPIdisconnect_from_agent() schließt die Verbindung zwischen Agent und Subagent. Die Funktion erwartet als Argument den Deskriptor, der die Verbindung zwischen den beiden repräsentiert hat (von DPIconnect_to_agent_TCP()).

Kapitel 4

Beschreibung der Funktionen

4.1 GET-Funktionen

4.1.1 Allgemeines

Um die Werte der MIB-Variablen aus dem System auszulesen, werden zwei Arten von GET-Funktionen verwendet:

- char* get_VARIABLENNAME(char* param_list)
- int* get_VARIABLENNAME (char* param_list)

Je nach Typ der jeweiligen Variable, wird eine der beiden Arten angewandt. Als Rückgabewert wird in beiden Fällen ein Zeiger auf das Ergebnis übergeben. Aus diesem Grund muß in der Funktion sichergestellt werden, daß der Speicherbereich, in dem das Ergebnis steht, nicht überschrieben werden kann. Deshalb wird die Ergebnisvariable als 'static' deklariert, was bedeutet, daß der Speicherbereich der Variablen, über die Laufzeit der Funktion hinaus, reserviert bleibt.

Der Parameter 'param_list' ist ein Zeiger auf ein Feld von Integerwerten, die zur Indizierung von Tabellen verwendet werden. Erfolgt die Indizierung über Strings, so müssen diese Integerwerte zuerst in einen String konvertiert werden. Hierzu dient die Funktion 'make_string()'.

Bei Gruppen von Funktionen, die Variablenwerte auf ähnliche Art und Weise erhalten, wurde eine Funktion der Form 'char* get_NAME(char* param_list, int type) geschrieben, die von den jeweils zugehörigen GET-Funktionen aufgerufen wird. Der Parameter 'type' gibt hierbei die im jeweiligen Fall zu bestimmende Variable an. Mit Hilfe einer 'switch/case'-Anweisung wird dann aufgrund von 'type' der entsprechende Rückgabewert bestimmt.

Sollten hierbei Variablen vom Typ INTEGER auftreten, so wird ihr Typ mit Hilfe eines Casts auf den Typ 'char*' umgewandelt.

Aus Leistungsgründen erscheint es nicht sinnvoll, bei Nachfragen innerhalb eines Paketes, die sich auf dieselbe Variable beziehen, diese Variable mehrmals auszulesen. Ähnlich verhält es sich bei Variablen, deren Wert dadurch bestimmt wird, daß eine größere Struktur ausgelesen wird. Wird innerhalb eines Paketes auf mehrere Variablen zugegriffen, die sich alle aus derselben Struktur ergeben, so sollte diese ebenfalls nur einmal ausgelesen werden. Hierzu wird in den entsprechenden Funktionen eine Kopie der jeweiligen Struktur angelegt, aus der bei folgenden Aufrufen die Werte ausgelesen werden.

Um feststellen zu können, ob die Aufrufe von Anfragen desselben Paketes stammen, wird ein Zähler 'global_counter' eingeführt, der beim Abschicken eines Paketes um eins erhöht wird. Da die derzeitige Version des Agenten keine Pakete mit mehr als einer Variablennachfrage erlaubt, besteht die Möglichkeit, mit Hilfe des Schalters '-cache' dieses Hochzählen auf das Absenden von SET-Paketen zu beschränken. Somit ergibt sich eine deutliche Leisungssteigerung, auch wenn die Nachfrage nach gleichen Variablen nicht in einem Paket stattfindet. Zu beachten ist hierbei, daß insbesondere bei Variablen, die ihren Wert relativ schnell ändern (z.B in der Prozeßtabelle) Probleme auftreten können. Bei Variablen, die ausgelesen werden können, ohne weitere Informationen auslesen zu müssen, wurde auf eine Speicherung verzichtet, da nicht anzunehmen ist, daß innerhalb eines Paketes mehrfach auf dieselbe Variable zugegriffen wird.

Um die Werte für die Variablen zu erhalten, bedient man sich im wesentlichen drei verschiedener Methoden:

- Systemaufrufe

  Die einfachste Art, Informationen über das System zu erhalten, ist die Verwendung von Library-Funktionen, die das gewünschte Ergebnis liefern.

- Auslesen von Kernelvariablen

  Hierzu wird die Funktion 'get_from_kernel()' verwendet. Sie bekommt den Namen der auszulesenden Kernelvariable als Parameter übergeben, öffnet den Kernel, liest die Variable aus und schließt den Kernel wieder. Als weiteren Parameter erwartet die Funktion einen Zeiger, auf einen Speicherbereich, in den das Ergebnis geschrieben werden soll.

- Auslesen aus der Konfigurationsdatei

  Variablenwerte, die nicht vom System bereitgestellt werden, werden aus einer Konfigurationsdatei ausgelesen. Die Funktion 'get_conf()' liest die Werte aus der Datei /etc/lrz-mib.conf aus, wenn nicht die Umgebungsvariable 'LRZCONFFILE' eine andere Datei vorgibt. Die Einträge in dieser Konfigurationsdatei sind in der Form 'VARIABLENNAME=WERT' vorzunehmen. 'VARIABLENNAME' ist hierbei der

Name der Variable, wie er in der MIB festgelegt ist, wobei auf Groß- bzw. Kleinschreibung keine Rücksicht genommen werden muß. Die möglichen Werte, die einer Variable zugewiesen werden können, ergeben sich aus dem Variablentyp. Im Folgenden werden einige Beispiele für Einträge in der Konfigurationsdatei angegeben:

- cputype=SPARC
- clockrate=33
- specint=18.2
- specintyear=SPECint92
- specfp=17.9
- specfpyear=SPECfp92
- printerLocation(np)=LMU-Muenchen...

Im folgenden Abschnitt werden nun die Funktionen genauer erläutert, die von diesem Schema abweichende Methoden anwenden, um die gewünschte Information bereitzustellen.

4.1.2 System Group

Die System Group wird durch das Modul 'system.c' implementiert. Alle Werte der dort implementierten MIB-Variablen, außer 'sysContact' und 'sysLocation', werden durch die entsprechenden Systemaufrufe aus dem System ausgelesen.
Da die Werte für die oben genannten Variablen nicht durch Systemaufrufe zu ermitteln waren, werden sie aus der Konfigurationsdatei gelesen.

Der Wert der Variablen 'sysHardware' wird aus der globalen Varibalen gleichen Namens gelesen. Diese wird beim Start des Subagenten in der Funktion get_devices() belegt.

4.1.3 Storage Group

Die Storage Group wird durch das Modul 'storage.c' implementiert. Innerhalb der 'strorage group' konnten nur die Variablen für RAM ('StoRam') und Virtual Memory ('StoVirtualMemory') implementiert werden, da Informationen zum First Level Cache ('StoFLCache') und Second Level Cache ('StoSLCache') nicht zugänglich waren.

Die Werte der Variablen 'StoUsed' und 'StoState' konnten durch Zugriff auf die entsprechenden Kernelvariablen nur für den Virtual Memory ermittelt werden, da diese Werte für den RAM nicht ermittelbar waren.

4.1.4 Device Group

Die Device Group wird durch die Module 'processor.c', 'devices.c', 'filesystem.c' und 'printer.c' implementiert.

Beim Start des Subagenten werden alle angeschlossenen SCSI Devices (disk, tape, CD-ROM), alle Floppy Laufwerke, alle auf Festplatten definierten Partitionen, alle installierten Drucker sowie alle Druckerwarteschlangen gelesen und in globale Variablen geschrieben. Dies wird durch die Funktionen get_devices(), get_disks(), get_partitions(), get_printer() sowie get_queues() realisiert.

Diese Informationen müssen zur Laufzeit des Subagenten im Speicher gehalten werden, da ein schneller Zugriff auf diese Informationen (z.B während des Browsens der MIB) ansonsten nicht gewährleistet ist und zu einem Timeout führen würde.

Die Funktion get_devices() liefert als Ergebnis die Anzahl aller angeschlossenen SCSI-Devices (disk, tape, CD-ROM) sowie Floppy-Laufwerke. Da unter SunOS 4.1.x kein geeigneter Systemaufruf existiert und auch der Zugriff auf diverse Kernelvariablen nicht das gewünschte Ergebnis lieferte, mußten wir dazu übergehen den Output des betriebssystemeigenen Tools 'devinfo' zu parsen, um an die entsprechenden Informationen zu gelangen. Die Namen der Devices werden in der globalen Variablen 'Devices' abgelegt.
Als Nebeneffekt wird die Hardwarebezeichnung der Maschine in der globalen Variablen 'sysHardware' abgelegt. Hierauf wird mit der Funktion get_sysHardware() aus dem Modul 'system.c' zugegriffen.

Die Funktion get_disks() liefert als Ergebnis die Anzahl aller lokalen Festplatten, Floppylaufwerke und CD-ROM Laufwerke. Die Bezeichnung dieser Devices wird dabei in der globalen Variablen 'Disks' abgelegt.

Die Funktion get_partitions() liefert als Ergebnis die Anzahl aller (möglichen) Partitionen. Dabei ergibt sich die Anzahl der Partitionen aus: ('Anzahl der Festplatten' * 8).

Die Funktion get_printers() liefert als Ergebnis die Anzahl aller lokalen Drucker. Es wird dazu auf die Datei /etc/printcap zugegriffen, die alle Informationen zu lokalen bzw. remote Druckern enthält. Die Namen der Drucker werden in der globalen Variablen 'Printers' abgelegt.

Die Funktion get_queues() liefert als Ergebnis die Anzahl aller definierten Druckerwarteschlangen. Unter SunOS 4.1.x (BSD Print-System) ist zu jedem Drucker, egal ob lokal oder remote, eine Queue gleichen Namens definiert. Hierzu wird ebenso wie bei der Funktion get_printers() die Datei /etc/printcap geparst. Die Namen der Queues werden in der globalen Variablen 'Queues' gespeichert. Weiterhin wird in dieser Variablen der zugehörige Druckername vermerkt, und ob es sich bei dem zugehörigen Drucker um einen lokalen oder remote Drucker handelt. Ebenso wird dort das Spool Directory der jeweiligen Queue gespeichert.

4.1.5 Processor Group

Die Processor Group wird im Modul 'processor.c' implementiert und beschränkt sich mit Ausnahme des Aulsesens von System-, User-, Nice und Idle-Time auf das Auslesen der entsprechenden Werte aus dem Konfigurationsfile. Das Auslesen dieser Zeiten erfolgt aus der Kernelvariablen '_cp_time'.

4.1.6 Printer Group

Die Printer Group wird durch das Modul 'printer.c' implementiert. Um alle Variablen korrekt implementieren zu können, war es notwendig an der ursprünglichen MIB ([3])einige Änderungen vorzunehmen. Die neue Modellierung der Printer Group enthält eine 'printer table', eine 'queue table' sowie eine 'job table' innerhalb der 'queue table'.

Innerhalb der 'printer table' konnten die Variablen 'printerAvStatus, 'printerAction' sowie 'printerError' nicht implementiert werden, da diese Informationen mit den uns zur Verfügung stehenden Möglichkeiten nicht ermittelbar waren.

Die Variablen 'printerName' bzw. 'printerLocation' werden aus der globalen Variable 'Printers' bzw. aus der Konfigurationsdatei gelesen.

Die Variablen 'printerOpStatus', 'printerUsStatus' und 'printerAvStatus' wurden zunächst unter Zuhilfenahme des UNIX-Kommandos '/usr/etc/lpc' implementiert. In der Testphase hat sich aber herausgestellt, daß diese Art der Implementierung zu Performanceverlust bzw. zu Timeouts zur Laufzeit des Subagenten führt. Aus diesem Grunde mußten wir eine andere Möglichkeit suchen, um an die gewünschten Informationen zu gelangen.

Jedem Drucker ist eine Queue zugeordnet, die über ein entsprechendes Spool Directory verfügt, indem bestimmte Konfigurationsdateien verwaltet werden. Eine davon ist die Datei 'lock', die unter anderem dazu verwendet wird, den Status des Druckers zu ermitteln. Zur Bestimmung von 'printerOpstatus' bzw. 'printerAvStatus' betrachten wir das 'execute' Bit des Users oder der Gruppe für diese Datei. Ist das 'execute' Bit des Users oder der Gruppe nicht gesetzt, so ist der 'printerOpstatus' 'ENABLED'. Ist das 'execute' Bit des Users nicht gesetzt, so ist der 'printerAdStatus' 'DISABLED'. Die Variable 'printerUsStatus' wird weiterhin unter Zuhilfenahme des 'lpc'-Kommandos implementiert.

Innerhalb der 'queue table' konnten alle Variablen implementiert werden. Bei 'queueStatus' wird dabei genauso vorgegangen wie bei der Implementierung der Variable 'printerOpstatus' bzw. 'printerAvStatus'. Die Variable 'queuePrinterLocation' zeigt an, ob es sich um einen lokalen oder remote Drucker handelt. Die Variablen 'queuePrinterIndex' bzw. 'queuePrinterName' liefern den Index des Druckers in der 'printer table' bzw. den Namen des Druckers (in der Form: 'HOSTNAME:PRINTERNAME'), dem die Queue zugeordnet ist, falls es sich um einen remote Drucker handelt oder einen leeren String, falls der Drucker lokal ist.

Die 'job table' enthält Informationen zu Druckaufträgen, die in einer Queue gespoolt werden. Hierbei konnten nur die Variablen 'queueJobID', 'queueJobOwner' sowie 'queueJobSize' implementiert werden, da nur diese vom BSD Print System unterstützt werden.

4.1.7 Disk Group

Die Disk Group wird durch das Modul 'devices.c' implementiert. Bis auf die Variablen 'diskRs' (disk reads per second), 'diskWs' (disk writes per second) und 'diskPu' (percentage disk usage), deren Werte nicht ermittelbar waren, konnten alle Variablen implementiert werden.

Besonderes Augenmerk liegt auf der Variablen 'diskCapacity'. Dabei muß unterschieden werden ob es sich bei der Disk um ein CD-ROM, ein Floppy Laufwerk oder eine Festplatte handelt. Bei CD-ROMs wird zuerst geprüft, ob es derzeit gemountet ist. Wenn ja, kann die Kapazität des 'Mount Points' mit dem entsprechenden Systemaufruf ermittelt werden. Ansonsten wird als Kapazität -1 zurückgegeben, um anzuzeigen, daß diese Information derzeit nicht verfügbar ist. Bei Floppy Laufwerken wird die Kapazität mit 1,44 MB angegeben, und bei Festplatten werden zuerst die Geometriedaten der Festplatte (Zylinder, Köpfe und Sektoren) bestimmt, aus denen dann die Kapazität errechnet wird.

4.1.8 Partition Group

Die Partition Group wird durch das Modul 'devices.c' implementiert. Innerhalb dieser Gruppe konnten alle Variablen implementiert werden.

Als 'partitionLabel' wird dabei der zugehörige 'Mount Point' aus der Datei /etc/mtab zurückgegeben oder 'No Mount Point', falls die Partititon derzeit nicht gemountet ist. Die 'partitionID' ist der Name der Partition (z.B. /dev/sd1a oder /dev/sd0h). Als 'partitionSize' wird die Größe der jeweiligen Partition zurückgegeben, oder -1, falls diese Partition nicht existiert. Bei 'partitionFSIndex' bzw. 'partitionDiskIndex' werden jeweils Zeiger auf die Einträge in der 'Predefined Filesystem Table' bzw. 'Disk Table' zurückgegeben, zu denen die Partitionen gehören.

4.1.9 Filesystem Group

Diese Gruppe mußte vollständig neu modelliert werden, da im Zuge der Implemtierungsphase erkannt wurde, daß diese Gruppe in der ursprünglichen Modellierung nicht vollständig bzw. nicht eindeutig implementiert werden konnte.

Während der Implementierung dieser Gruppe ergab sich folgende Problematik: Die eingangs modellierte 'filesystem table' sollte sowohl alle Filesysteme aus der Datei /etc/fstab (vordefinierte Filesysteme) als auch aus der Datei /etc/mtab (tatsächlich gemountete Filesysteme) enthalten. Die Einträge innerhalb der Tabelle müssen dabei auf diesselbe Weise indiziert werden.

Verwendet man einen fortlaufenden Index zur Indizierung der Elemente, so stößt man auf folgendes Problem:
Die Datei /etc/mtab enthält Einträge aller derzeit gemounteten Filesysteme. Nun kann es vorkommen, daß ein solches Filesystem vom 'Automounter' gemountet wurde, nachdem auf dieses zugegriffen wird. Der 'Automounter' unmounted das entsprechende Filesystem wieder, wenn auf dieses Filesystem bestimmte Zeit lang nicht zugegriffen wird. Somit wird auch der entsprechende Eintrag aus der Datei /etc/mtab entfernt und alle dahinterliegenden Einträge werden um einen Platz nach vorne geschoben. Dieses Vorgehen ist dabei für den einzelnen Benutzer völlig transparent. Greift nun ein Benutzer über den entsprechenden Index auf ein Filesystem zu, das innerhalb dieser Tabelle seine Position (und somit den Index) geändert hat, so erhält man die falschen Informationen zu diesem Eintrag. Somit ist die Indizierung dieser Tabelle über einen fortlaufenden Index nicht möglich.

Verwendet man nun den Filesystemnamen als Index, so stößt man dabei auf folgendes Problem:
Die Datei /etc/fstab enthält Einträge zu vordefinierten Filesystemen bzw. Partitionen, die über das Shell-Kommando 'mount' jederzeit gemountet werden können. Nun kann es vorkommen, daß zu einem Block Device (bzw. zu einem Filesystemnamen) mehrere verschiedene Einträge existieren. Indiziert man diese Einträge über den Filesystemnamen, so kann man grundsätzlich nur den ersten Eintrag erreichen. Somit eignet sich die Indizierung über den Filesystemnamen nicht für diese Tabelle.

Teilt man nun die 'filesytem table' in eine 'predefined filesystem table' und eine 'mounted filesystem table' und indiziert man die erste über den fortlaufenden Index und die zweite über den Filesystemnamen, so kann man die oben beschriebenen Probleme einfach umgehen. Die Informationen bzw. Einträge zur 'predefined filesystem table' werden aus der Datei /etc/fstab gelesen, die Einträge und Informationen für die 'mounted filesystem table' werden der Datei /etc/mtab entnommen bzw. durch die entsprechenden Systemaufrufe gewonnen. Durch die Ummodellierung war es nun möglich alle Variablen vollständig zu implementieren.

Die 'backup table' die sich ursprünglich innerhalb der 'filesystem table' befand, wird nun als Tabelle in die 'predefined filesystem table' aufgenommen. Zur Implementierung der beiden Variablen 'fsBackupLevel' und 'fsBackupDate' wird auf die Datei /etc/dumpdates zugegriffen, die Informationen über gelaufene Backups enthält.

4.1.10 Process Group

Die Implementierung der Process Group erfolgt im Modul 'process.c'. Im Folgenden werden einige Funktionen dieser Gruppe näher beschrieben.

- get_Process()

  Die Funktion get_Process() liefert sämtliche Informationen, die aus der Stuktur 'proc' entnommen werden können. Sie liest diese Struktur für den jeweiligen Prozeß aus und gibt das Element der Struktur, das über den Parameter 'type' bestimmt wird zurück.

- get_curMaxProcessSize()

  Diese Funktion durchläuft einmal die gesamte Prozeßtabelle und merkt sich den größten Wert für die Summe von Text-, Daten- und Stacksegment.

- get_curMaxProcessTime()

  Hier wird ebenfalls einmal die Prozeßtabelle durchlaufen. Allerdings finden sich die Informationen über die vom Prozeß in Anspruch genommene Zeit nicht in der 'proc'-Struktur sondern in der 'user'-Struktur. Deshalb muß diese für jeden Prozeß ausgelesen werden und dann der Maximalwert der verbrauchten Zeit bestimmt werden. Zu beachten ist hierbei, daß für Prozesse, die sich im 'Zombie'-Status befinden, keine 'user'-Struktur existiert. Daher werden diese Prozesse hier nicht berücksichtigt.

- get_processName()

  Unter dem Namen des Prozesses verstehen wir das Kommando, das benutzt wurde, um den Prozeß zu starten. Man kann es ebenfalls aus der 'user'-Struktur auslesen. Da der 'Swapper'- sowie der 'Pager'-Prozeß keinen Namenseintrag in dieser Struktur besitzen, werden sie anhand ihrer Prozeß-ID identifiziert und, analog zum SPS-Programm, als 'Unix Swapper' bzw. 'Unix Pager' bezeichnet. Bei 'Zombie'-Prozessen wird statt einem Namen '** EXIT **' ausgegeben.

4.1.11 User Group

Die Implementierung der User Group erstreckt sich auf die beiden Module 'user.c' und 'upage.c'. Hier ergab sich das Problem, daß es aufgrund der MIB-Definition möglich sein sollte, in der 'user table' bzw. in der 'group table' Tabellenzeilen zu erzeugen bzw. zu entfernen. Jede Tabellenzeile enthält ein Statusfeld, in dem angegeben ist, ob die Tabellenzeile dem System bekannt ist oder nicht. Dieses Feld wird auch benutzt, um Zeilen zu löschen bzw. dem System bekannt zu machen (siehe SET-Funktionen).

Die aktiven (dem System bekannten) Zeilen der 'user table' befinden sich in der lokalen Passwortdatei /etc/passwd. Von hier können sie mit Hilfe der Funktion 'local_getpwnam()' ausgelesen werden. Eventuell vorhandene, nicht aktive Zeilen müssen im Speicher aufbewahrt werden und dürfen nicht in der Passwortdatei erscheinen. Hierzu wurde eine dynamisch erzeugte, verkettete Liste von Passworteinträgen geschaffen, deren Startpunkt in einer globalen Variablen gespeichert bleibt. Beim Auslesen von Variablenwerten muß also sowohl die Passwortdatei als auch die im Speicher befindliche Liste nach dem entsprechenden Eintrag durchsucht werden. Bei der 'group table' verhält es sich analog.

Ein weiteres Problem ergab sich beim Auslesen der Quotas für die verschiedenen User. Es muß zuerst überprüft werden, ob der angegebene User existiert, dann muß nachgesehen werden, ob das gewünschte Filesystem vorhanden ist, bevor die Quotas bestimmt werden können. Da bei der Inaktivierung eines Users die Quotas der zu diesem Zeitpunkt gemounteten Filesysteme gespeichert werden, kann es zu Inkonsistenzen kommen, wenn später diese Filesysteme nicht mehr gemountet sind bzw. andere Filesysteme gemountet werden. Bei den Quotas ist weiterhin zu beachten, daß diese zum Teil in Blöcken, zum Teil aber auch in Kbyte angegeben werden. Deshalb ist an manchen Stellen eine Multiplikation bzw. Division mit zwei erforderlich.

Die 'who table' ergibt sich durch Auslesen der Datei /etc/utmp. Hierbei ist zu beachten, daß die Datei eventuell noch Einträge bereits ausgeloggter User enthalten kann. Einen ausgeloggten User erkennt man daran, daß statt einem Namen ein leerer String eingetragen ist, gleichgültig ob Werte für die anderen Variablen noch in der Datei vorhanden sind.

4.2 GETNEXT-Funktionen

Die Getnext-Funktionen befinden sich im Modul 'getnext.c'. Sie sind alle von der Art 'void get_next_TABELLENNAME(int *param_list)'. Zweck dieser Funktionen ist es, den nächstgrößeren Indexwert einer Tabelle zu ermitteln. Als Paramter erhalten diese Funktionen wiederum einen Zeiger auf ein Feld von Indexen, die von der Funktion entsprechend verändert werden können.

Viele Tabellen werden über eine fortlaufende Nummer indiziert. Hier ist es möglich, den Index einfach um eins zu erhöhen und dann festzustellen, ob das Ende der Tabelle mit diesem Indexwert überschritten würde. Hierzu wird einfach eine Variable dieser Tabelle mit einer GET-Funktion und dem erhaltenen Indexwert abgefragt. Sollte der Wert nicht existieren, so ist das Ende der Tabelle erreicht und es wird der Index auf -1 gesetzt.

Bei Tabellen, die keine fortlaufende Nummer bzw. einen String zur Indizierung verwenden, gestaltet sich dies natürlich wesentlich schwieriger. Hier wird ebenfalls zur Leistungssteigerung eine Speicherung des zuletzt bestimmten Wertes vorgenommen. Man kann davon ausgehen, daß Tabellen vom Anwender zeilenweise ausgelesen werden. Ist dies der Fall, so muß mehrfach hintereinander für den gleichen Index der folgende Eintrag bestimmt werden. Hier bietet es sich natürlich an, diesen Wert zu speichern und im Falle einer identischen Nachfrage, den gespeicherten Wert zurückzugeben. Diese Speicherung erfolgt wiederum nur innerhalb eines Paketes, außer sie wird mit dem Schalter '-cache' auch paketübergreifend eingeschaltet (siehe GET-Funktionen).

Im Folgenden werden nun die wichtigsten GETNEXT-Funktionen beschrieben:

- get_next_MountedFilesystem()

Die 'mounted filesystem table' wird über den Namen des Filesystems indiziert. Um den nächsten Eintrag zu erhalten, ist es notwendig, die Datei /etc/mtab zu durchsuchen, und den lexikographisch folgenden Eintrag auszuwählen. Dann muß mit Hilfe der Funktion 'make_new_param_list()' dieser String wieder in das zur Indizierung verwendete Format eines Feldes von Integerwerten verwandelt werden.

- get_next_Process()

  Die Prozeß-Tabelle wird über die Prozeß-ID der Prozesse indiziert. Da diese nicht in fortlaufender Reihenfolge vergeben werden, und die Prozeß-Tabelle auch nicht geordnet vorliegt, ist es notwendig, die gesamte Tabelle zu durchlaufen und den Prozeß mit der nächstgrößeren PID zu bestimmen. Da die Prozeß-Tabelle sehr groß sein kann, ist dies sehr zeitaufwendig. Um eine gewisse Beschleunigung zu erreichen, wird die Suche abgebrochen, wenn ein Prozeß gefunden wird, der eine um eins höhere PID besitzt als der gegebene.

- get_next_Group() und get_next_User()

  Bei diesen Funktionen wird die Datei /etc/group bzw. /etc/passwd nach dem Eintrag durchsucht, der dem durch den Indexwert spezifizierten Eintrag folgt.

Noch umständlicher gestaltet sich die Suche, wenn Tabellen von Tabellen indiziert werden sollen. Hier gilt es im einzelnen folgende Möglichkeiten zu beachten:

- Der Index der äußeren Tabelle zeigt nicht auf einen existierenden Eintrag

  In diesem Fall muß der nächstgrößere Eintrag der äußeren Tabelle bestimmt werden und dann, falls ein solcher Eintrag existiert, der erste Eintrag der zugehörigen inneren Tabelle zurückgegeben werden.

- Der Index der äußeren Tabelle zeigt auf einen existierenden Eintrag

  Dann muß bestimmt werden, ob ein größerer Eintrag in der inneren Tabelle existiert. Ist ein derartiger Eintrag vorhanden, so kann dieser als Ergebnis übergeben werden. Sollte dies nicht der Fall sein, so muß wiederum die äußere Tabelle um eine Zeile weitergeschaltet werden und dann wieder der erste Eintrag der inneren Tabelle ausgegeben werden.

Dieses Prinzip soll am Beispiel der Funktion 'get_next_UserQuota()' erläutert werden. Die Indizierung erfolgt über zwei Strings: den Usernamen, der einen User (die äußere Tabelle) bestimmt, sowie den Filesystemnamen, der die Filesysteme (die innere Tabelle) identifiziert, für die Quotas angezeigt werden sollen.

Als erstes wird überprüft, ob der angegebene Username existiert. Sollte dies nicht der Fall sein, so wird mit Hilfe der Funktion 'get_next_User()' der Username bestimmt, der dem gegebenen folgt. Da auf jeden Fall das erste Filesystem dieses Users ausgegeben werden soll, wird der zweite Index auf -1 gesetzt. Sollte kein User mehr existieren, dessen Name dem gegebenen lexikographisch folgt, so wird die Funktion verlassen und die auf die User-Tabelle folgende Variable abgefragt.

Sollten wir uns immer noch in der Funktion befinden, so ist die Situation die folgende: Der Index der 'user table' zeigt auf einen existierenden Eintrag, der Index der 'quota table' eventuell nicht. Nun muß also der nächste Eintrag der 'quota table', gefunden werden. Hierzu wird eine Schleife verwendet, die erst verlassen wird, wenn ein gültiger Index gefunden wurde bzw. die äußere Tabelle keine größeren Einträge mehr enthält. Es wird also das Filesystem gesucht und zurückgegeben, das dem gegebenen folgt. Sollte das Ende der Tabelle erreicht werden, so wird wieder der nächstgrößere User-Eintrag bestimmt und der Index der 'quota table' auf -1 gesetzt. Somit wird im nächsten Schleifendurchlauf mit Sicherheit das erste Element der 'quota table' gefunden.

Die Vorgehensweise ist analog zu derer bei den Funktionen 'get_next_GroupUser()' sowie 'get_next_FilesystemBackup()'.

4.3 SET-Funktionen

Sämtliche Funktionen, die zum Setzen von Variablenwerten bzw. zum Erzeugen von Tabellenzeilen verwendet werden, befinden sich im Modul 'set.c'.

4.3.1 Allgemeines

Die Funktionen zum Setzen von Variablen sind von der Art 'set_Variablenname(int *param_list, char *value, ... , int set_or_commit)'. Der Parameter 'param_list' identifiziert wiederum die Tabellenzeile, auf die sich die SET Request bezieht, während der Parameter 'value' den zu setzenden Wert enthält. Da dieser vom Typ 'char *' ist, muß evtl. noch eine Umsetzung auf einen anderen Typ vorgenommen werden. Mit Hilfe des Parameters 'set_or_commit' kann unterschieden werden, ob es sich um eine COMMIT Request handelt, bei der der Variablenwert tatsächlich gesetzt werden soll oder um eine SET Request, bei der überprüft wird, ob ein Setzen der Variablen möglich wäre.

Das Setzen der Variablenwerte erfolgt durch einen Eintrag in der Konfigurationsdatei mit Hilfe der Funktion 'set_conf()' oder durch einen Systemaufruf, dem der entsprechende Wert übergeben wird.

Rückgabewert der Set-Funktionen ist eine Fehlermeldung, die angibt, ob bzw. warum die SET Request nicht ausgeführt werden konnte.

4.3.2 Setzen von Variablenwerten

- set_queueAction()

  Mit Hilfe dieser Variablen kann man Printer Queues an- bzw. ausschalten. Zu beachten ist hierbei, daß dies über das Setzen bzw. Löschen des 'execute_by_group'-Bits des 'Lock'-Files im entsprechenden Spool-Directory erfolgt.

- set_queueJobAction()

  Mit Hilfe dieser Variablen wird ein einzelner Druckauftrag aus einer Warteschlange gelöscht. Hierbei mußte auf das Shellkommando 'lprm' zurückgegriffen werden, da keine andere brauchbare Möglichkeit gefunden wurde, um diese Funktionalität zu erreichen.

- set_mountedFsQuota()

  Diese Variable dient dem Ein- bzw. Ausschalten der Quotas auf den einzelnen Filesystemen. Auch hier wurden die Shellkommandos 'quotaon' bzw. 'quotaoff' verwendet, um das gewünschte Ziel zu erreichen. Die andere Möglichkeit wäre gewesen, die Library-Funktion 'quotactl' zu verwenden. Nach dem Aufruf dieser Funktion wäre es notwendig gewesen, zusätzlich noch die Datei /etc/mtab zu editieren. D.h., bei dem einzelnen Filesystem hätte im Feld 'Mount Options' vermerkt werden müssen, ob die Quotas derzeit ein- oder ausgeschaltet sind. Dies war aber aus Gründen der Performance nicht realisierbar.

- set_predefinedFsAction()

  Diese Variable wird dazu verwendet, um ein 'predefined' Filesystem, d.h., ein Filesystem, das in der der Datei /etc/fstab steht, zu mounten bzw. eine Konsistenzprüfung des Filesystems zu veranlassen. Für die Implementierung wurden die Shellkommandos 'mount' bzw. 'fsck' verwendet. Dafür sprachen folgende Gründe:
  Die Datei /etc/fstab kann Einträge für Filesysteme enthalten, die sich auf remote Rechnern befinden. Um nun diese Filesysteme mounten zu können, muß neben dem Filesystemnamen auch der Hostname in Form einer IP-Adresse angegeben werden. Dies war nicht realisierbar. Zur Initiierung eines Filesystemchecks wurde nur das Shellkommando 'fsck' gefunden und verwendet.

- set_mountedFsAction()

  Mit Hilfe dieser Variable kann ein bereits gemountetes Filesystem ummounted werden.

4.3.3 Erzeugen und Löschen von Tabellenzeilen

In Tabellen, die Variablen enthalten, deren Status 'read-create' ist, ist es möglich, neue Tabellenzeilen einzufügen bzw. Zeilen zu löschen ([1]). Hierzu ist eine Variable vom Typ 'row-status' erforderlich, die den Status der Zeile beschreibt bzw. zum Setzen eines neuen Status verwendet werden kann.

Tabellenzeilen können drei Zustände einnehmen:

- active

  Ist eine Tabellenzeile 'active', so steht sie dem System zur Verfügung, z.B. ein aktiver User ist in der Passwortdatei eingetragen.

- notinservice

  Diese Tabellenzeilen existieren zwar, stehen dem System aber nicht zur Verfügung. Das bedeutet, daß ihre Werte zwar gespeichert sein müssen und ausgelesen werden können, der Eintrag, z.B. in der Passwortdatei, aber nicht existiert. Zeilen mit Status 'notinservice' können jederzeit in den aktiven Status versetzt werden.

- notready

  Ist der Status einer Zeile 'notready', so sind nicht alle Variablen dieser Zeile derart mit Werten belegt, daß diese Zeile in einen aktiven Zustand versetzt werden könnte. Ein User kann z.B. nicht in die Passwortdatei eingetragen werden, wenn für ihn kein Home Directory bestimmt wurde. Wird einer Variablen einer Zeile, die sich in diesem Zustand befindet, ein Wert zugewiesen, so muß jeweils überprüft werden, ob der Status der Zeile auf 'notinservice' verändert werden muß.

Möchte man den Status einer Zeile ändern, so überschreibt man die Statusvariable der Zeile mit einem der folgenden Werte:

- active

  Die Zeile wird in den aktiven Zustand versetzt, d.h. sie wird dem System bekannt gemacht.

- notinservice

  Die Zeile wird inaktiviert, d.h. sie ist dem System nach Ausführen der Funktion nicht mehr bekannt, muß allerdings gespeichert werden, um ausgelesen werden zu können bzw. um später wieder aktiviert zu werden.

- destroy

  Hiermit wird eine Zeile gelöscht, ohne daß sie wiederhergestellt werden kann.

- createandgo bzw. createandwait

  Diese Werte werden verwendet, um neue Tabellenzeilen anzulegen. Man setzt die Statusvariable einer noch nicht existierenden Zeile auf einen der beiden Werte, worauf die Zeile angelegt wird und die übrigen Variablen, sofern vorgesehen, mit Default-Werten belegt werden. Der Unterschied der beiden Möglichkeiten besteht darin, daß bei 'createandgo' die Zeile sofort in einen aktiven Zustand versetzt wird, während bei 'createandwait' die Zeile 'notinservice' gesetzt wird. Sollten nicht für alle Variablen Werte zur Verfügung stehen, so wird die 'createandgo'-Aufforderung zurückgewiesen, während bei 'createandwait' die Zeile in den Status 'notready' versetzt wird.

Im Rahmen dieser MIB-Implementierung mußte diese Funktionalität für die Tabellen 'user table' und 'group table' zur Verfügung gestellt werden. Da die Implementierung beider Tabellen recht ähnlich ist, wird hier nur am Beispiel der 'user table' erläutert, wie derartige Tabellen realisiert wurden.

Die Funktion, die für das Setzten des Statusfeldes der 'user table' verantwortlich ist, nennt sich 'set_userStatus()'. Hier wird zuerst überprüft, ob die gewünschte Statusänderung überhaupt möglich ist. Dies kann z.B. nicht der Fall sein, wenn man eine Zeile, die sich im Status 'notready' befindet, in den Status 'active' versetzten möchte. Ist die Änderung möglich, so ist, je nach gewünschter Änderung, eine ganze Reihe von Maßnahmen zu treffen:

- notinservice

  Wenn eine Tabellenzeile aus dem aktiven Zustand in den Zustand 'notinservice' gebracht werden soll, so wird der Eintrag des Users aus der Datei /etc/passwd mit Hilfe der Funktion 'change_passwd()' gelöscht. Bevor dies geschieht, werden die Daten dieses Benutzers in einer dynamisch erzeugten Struktur gespeichert, die in eine Liste derartiger Strukturen eingehängt wird. Der Anfang dieser Liste ist in der globalen Variablen 'user_row_p' gespeichert.

- active

  Soll eine Zeile in den Status 'active' versetzt werden, so muß sie sich vorher im Status 'notinservice' befinden. Ist dies der Fall, so wird die Zeile aus der Liste der inaktiven Zeilen ausgelesen, dort gelöscht und in die Passwortdatei eingetragen. Außerdem müssen die File-Quotas für diesen User gesetzt werden.

- createandwait

  Soll eine neue Tabellenzeile mit dem Befehl 'createandwait' erzeugt werden, so wird Speicherplatz für diesen Eintrag reserviert und die Variablen, für die Defaultwerte vorgesehen sind, werden mit diesen Werten belegt. Die File-Quotas des neuen Users werden auf null gesetzt. Der Status der Zeile wird auf 'notready' gesetzt, da z.B. noch keine User-ID des neuen Users festgelegt wurde.

Kapitel 5

Einfügen neuer MIB-Variablen

Um eine neue MIB-Varibable in den Agenten einzufügen, sind folgende Schritte durchzuführen:

- Sofern es sich um eine lesbare Variable handelt, muß eine Funktion erstellt werden, die den Wert der Variablen ausliest (GET-Funktion). Die Funktion bekommt als Parameter einen Zeiger auf ein Feld von Integerwerten, deren Interpratation als Tabellenindex der Funktion selbst überlassen bleibt. Der auszulesende Wert der Variable muß in einen Speicherbereich geschrieben werden, der auch nach Beendigung der Funktion nicht überschrieben werden kann. Insbesondere bietet sich die Möglichkeit an, die Variable, die den Rückgabewert enthält als 'static' zu deklarieren. Als Return-Wert wird dann ein Zeiger auf diesen Speicherbereich übergeben. Ist die Variable aus irgendeinem Grund nicht auszulesen, so ist ein NULL-Zeiger zurückzugeben.

- Bei Variablen, deren Wert auch gesetzt werden kann, muß zusätzlich eine Funktion geschrieben werden, die diese Aufgabe übernimmt (SET-Funktion). Diese Funktion bekommt wiederum einen Zeiger auf ein Feld von Integern zur Indizierung und darüberhinaus einen Zeiger auf den Wert, der der Variablen zugewiesen werden soll. Es handelt sich um einen Zeiger vom Typ 'char*'. Sollte es sich um eine Variable anderen Typs handeln, so ist eine Umsetzung vorzunehmen. Der dritte Parameter dient zur Unterscheidung, ob es sich bei dem Request um eine SET- oder COMMIT-Aufforderung handelt. Bei der SET-Aufforderung muß lediglich überprüft werden, ob ein Setzen der Variablen möglich ist, während bei der COMMIT-Request der Wert tatsächlich geändert wird. Als Returnwert wird eine eventuell nötige Fehlermeldung zurückgegeben. (Fehlermeldungen : siehe [4]). Wurde die Funktion fehlerfrei ausgeführt, so ist als Returnwert 'SNMPerror_noError' zurückzugeben.

- Im Falle einer Tabellenvariablen, muß schließlich eine Funktion existieren, die den nächstgrößeren Indexwert der Tabelle bestimmt (GETNEXT-Funktion). Diese Funktion liefert keinen Returnwert, sondern ändert direkt die als Parameter übergebene

Liste von Indexwerten. Sollte bereits der letzte Eintrag der Tabelle erreicht sein, so schreibt die Funktion an die erste Position des Indexfeldes den Wert -1. Diese Funktion wird aufgerufen, um bei GETNEXT Requests die Tabelle in der richtigen Reihenfolge durchlaufen zu können.

- Nun muß in der Datei 'data.c' ein Eintrag für die neue Variable angelegt werden. Hierzu wird an der entsprechenden Stelle im Feld 'var_descr_a' ein Eintrag eingefügt. Dieser besteht aus

  - einem Zeiger auf die GET-Funktion
  - einem Zeiger auf die SET-Funktion (oder NULL, falls die Variable 'read-only' ist)
  - dem Typ der Variablen (mögliche Typen, siehe [4])
  - einem Zeiger auf die GETNEXT-Funktion (oder NULL, falls es sich um eine einfache Variable handelt)

- Ein weiterer Eintrag muß in der selben Datei im Feld 'var_table' erfolgen. Hier wird der 'OID' der Variablen festgelegt. An dieselbe Indexposition wie im Feld 'var_descr_a' wird der 'OID' als Feld von Integerwerten eingetragen. Nicht benutzte Werte bekommen hierbei negative Zahlen zugewiesen. Der letzte Wert enthält die Anzahl der gültigen Werte. Auch sie wird negativ eingetragen.

- Als Abschluß ist nun noch in der Datei 'dpi_system.h' der Wert der Konstante 'MAX_VAR_NUMBER', der die Anzahl der implementierten Variablen enthält, um eins zu erhöhen.

Kapitel 6

Starten des Subagenten

6.1 Installation

Beim Start des Subagenten gilt es folgendes zu beachten:

- Der Subagent muß auf dem gleichen Rechner laufen wie der Agent, da der Subagent, in seiner derzeitigen Version, bei der Anmeldung an den Agenten davon ausgeht, daß dieser auf der gleichen Maschine läuft.(Start des Agenten siehe [2])
  Will man den Subagenten auf einem anderen Rechner starten, so muß in der Funktion 'do_connect_and_open()' im Modul 'dpi_system.c' folgende Änderung vorgenommen werden:
  Der Funktion 'DPIconnect_to_agent_TCP() muß anstelle von 'localhost' der Name des Rechners (Hostname oder IP-Adresse), auf dem der Agent läuft, übergeben werden.
- Agent und Subagent müssen jeweils unter der Kennung 'root' laufen. Nur so ist für den Subagenten gewährleistet, daß er korrekte Werte für die MIB-Variablen zurückliefert, die aus dem Betriebssystemkernel ausgelesen werden.
- Die Konfigurationsdatei, aus der der Subagent Teile seiner Informationen liest, muß sich im Verzeichnis '/etc' unter dem Namen 'lrz-mib.conf' befinden oder es muß die Umgebungsvariable LRZCONFFILE gesetzt sein und auf die entsprechende Konfigurationsdatei zeigen.

Es existiert derzeit eine lauffähige Version des Subagenten. Werden zukünftig Änderungen am Subagenten bzw. an einem der zugehörigen Module vorgenommen, so genügt es, durch den Aufruf von 'make' eine neue Version des Subagenten zu erstellen.

Um eine neue Version des Subagenten zu erstellen, müssen folgende Dateien vorhanden sein:

- snmp_dpi.h
- snmp_lDPI.c, snmp_lDPI.h
- snmp_mDPI.c, snmp_mDPI.h
- snmp_qDPI.c, snmp_qDPI.h
  (Diese Module enthalten die vordefinierten Funktionen der DPI API)
- dpi_system.c, dpi_system.h, dpi_version.h
- system.c, storage.c, devices.c, processor.c, printer.c, filesystem.c, process.c, upage.c, user.c,
- getnext.c, set.c
- data.c, cmp.c
- Makefile

Alle Module, bis auf 'upage.c' wurden mit dem 'gcc'-Compiler unter SunOS 4.1.2 übersetzt. Da der 'gcc'-Compiler für SunOS 4.1.3 konfiguriert war, mußte das Modul 'upage.c' mit dem 'cc'-Compiler übersetzt werden, da hier alle zur Übersetzung notwendigen Bibliotheken installiert waren.

Beachte: Die Datei 'upage.c' muß von Hand mit dem 'cc'-Compiler unter SunOS 4.1.2 übersetzt werden, da zwar das zugehörige Object File durch das 'Makefile' hinzugebunden aber nicht explizit erstellt wird (Aufruf: 'cc -c upage.c').

6.2 Aufrufsyntax

Der Subagent, der die LRZ-UNIX-MIB implementiert, kann mit folgenden Optionen gestartet werden:

dpi_system [**-d**] [**-write**] [**-cache**]

-d Startet den Subagenten im Debug-Modus; damit kann verfolgt werden, welche Art von Request für welche Variablen eintreffen, bzw. welche Werte an den Agenten zurückgeschickt werden.

-write SET-Requests werden zugelassen; SET-Variablen können nur geschrieben werden, wenn diese Option verwendet wird.

-cache Das Caching bei ankommenden Paketen wird eingeschaltet.

Literaturverzeichnis

[1] William Stallings, SNMP, SNMPv2 and CMIP - The Practical Guide to Network-Management Standards, Addison–Wesley Co., Inc., Reading, MA, 1993

[2] Erwin Hainzinger, Erweiterung eines SNMPv2–Agenten um eine Schnittstelle zur Interprozeßkommunikation, Fortgeschrittenenpraktikum, 1994

[3] Uwe Krieger, Konzeption einer Managementinformationsbasis für das Management von UNIX–Endsystemen, Diplomarbeit, 1994

[4] Bert Wijnen, DPI Version 2.0 API – Programmers Reference, IBM International Operations, 1994