INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Portierung eines bestehenden Systemmanagementagenten auf IBM AIX

Stefan Schmidt

31. Juli 1996

Betreuer: Alexander Keller
Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Inhaltsverzeichnis

1	Einführung	3
1.1	Motivation	3
1.2	Aufgabenstellung	3
2	Erläuterung zum Systemmanagementagenten	4
2.1	Managementszenario und Subagenten	4
2.2	Beschreibung der Subagenten MIB	5
3	Übernommener Stand	8
4	Portierung	10
4.1	Probleme	10
4.1.1	Umstellung auf DPI Library	10
4.1.2	Systemabhängigkeit	10
4.1.3	Übersichtlichkeit und Wartbarkeit	10
4.2	Systematik	12
4.2.1	Verwendung der DPI Library	12
4.2.2	Ordnung nach Gruppen	12
4.2.3	Ordnung nach Plattformabhängigkeit	13
4.2.4	Portabilität	13
4.2.5	Wiederverwendbarkeit	13
5	Jetziger Stand	15
5.1	Ergebnis der Modularisierung	15
5.2	Aufteilung in Verzeichnisse und Dateien	16
5.3	Zusammenspiel der Quelltexte	17
6	Praktischer Einsatz	20
6.1	Übersetzen	20
6.2	Installation	21
6.2.1	Ausführbares Programm	21
6.2.2	Konfigurationsdatei	21
6.3	Starten	21
6.4	Beispiele	22
6.4.1	Abfragen der Systemlast	22
6.4.2	Anzahl der Prozesse	23
6.4.3	Holen der Prozeßliste	23
6.4.4	Beenden eines Prozesses	23
7	Literaturverzeichnis	24

1 Einführung

1.1 Motivation

In den vergangenen Jahren wurden mehr und mehr Großrechner und Terminals durch Cluster von leistungsfähigen Workstations ersetzt. Der Einsatz von Workstations stellt neue Anforderungen an die Systemadministration: sie erfolgte früher an einem einzigen System, wogegen jetzt bei Workstationclustern viele, möglicherweise heterogene Maschinen gepflegt werden müssen.

Häufig werden hierzu eigene Lösungen entwickelt, z.B. um die Paßwortdatei auf alle Rechner im Cluster zu verteilen, oder um die Prozeßaktivitäten auf den einzelnen Rechnern zu überwachen. Ergebnis ist meist eine Sammlung von maßgeschneiderten Programmen, die sich nur schwer und mit entsprechendem Know How auf einem anderen Workstationcluster wiederverwenden lassen.

Die Idee ist, bewährte Techniken aus dem Netzmanagement auch für das Management von Workstation einzusetzen. Auf den Workstations soll ein Systemmanagementagent im Hintergrund laufen, der dem Administrator die managementrelevanten Daten des Rechners zur Verfügung stellen kann. Über ein Managementprotokoll (hier SNMPv2) können Daten aus der Management Information Base (MIB) abgefragt und ggf. auch geändert werden.

Im Rahmen eines früheren Fortgeschrittenenpraktikums [1] wurde ein Systemmanagementagent für Sun Workstations mit dem Betriebssystem SunOS 4.1.3 implementiert.

1.2 Aufgabenstellung

Der bestehende Systemmanagementagent ist nur auf SunOS 4.1.3 lauffähig, auf anderen Plattformen läßt sich der Quelltext des Agenten nicht ohne größere Änderungen übersetzen. Ziel ist aber, mit dem Systemmanagementagenten eine breitere Palette an Plattformen managementfähig zu machen.

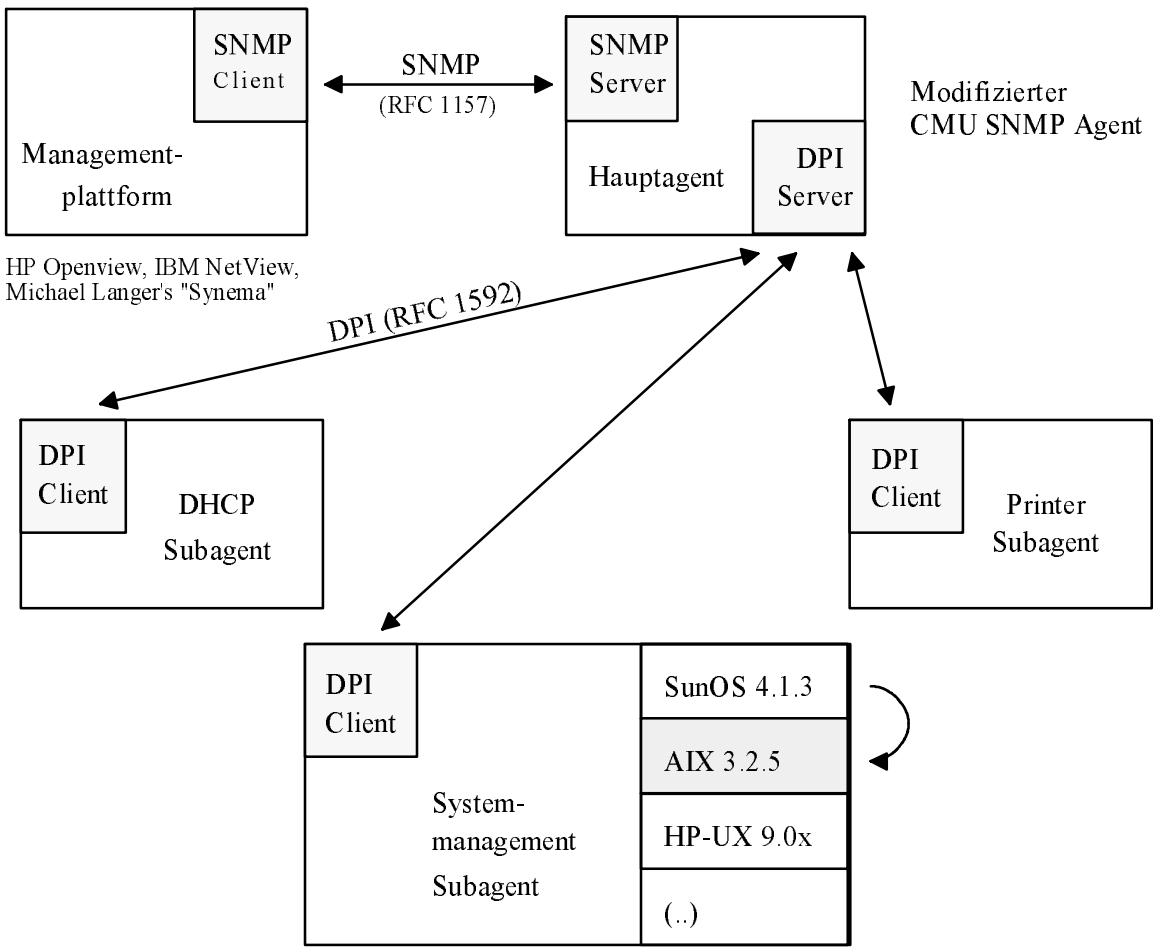
Die Aufgabe dieses Fortgeschrittenenpraktikums bestand in der Portierung des bestehenden Systemmanagementagenten für SunOS 4.1.3 auf das IBM Betriebssystem AIX 3.2.5.

Die beiden Betriebssysteme unterscheiden sich schon bedingt durch ihre geschichtliche Entwicklung recht stark. SunOS 4 basiert auf 4.3BSD, wogegen AIX 3 weitgehend ein System V mit IBM-eigenen Kernel ist. Diese Unterschiede, besonders bei den Kernels, machen die Portierung nicht einfach; in Abschnitt 3 wird ein Beispiel hierfür gezeigt.

2 Erläuterung zum Systemmanagementagenten

Vor der Besprechung der eigentlichen Portierung wird im folgendenen ein typisches Managementszenario beschrieben, um das Zusammenspiel der einzelnen Komponenten zu erklären.

2.1 Managementszenario und Subagenten



Oben links sitzt der Systemadministrator vor seiner gewohnten Managementplattform, die über das Managementprotokoll SNMP Variablen aus der MIB des Hauptagenten (oben rechts) lesen und ggf. schreiben kann.

Der Hauptagent besitzt neben der SNMP Schnittstelle zur Managementplattform noch über eine weitere Kommunikationsschnittstelle, der DPI (Distributed Protocol Interface) Schnittstelle (siehe auch [5]). Über diese Schnittstelle können sogenannte Subagenten die MIB des Hauptagenten dynamisch erweitern.

Der Subagent meldet sich nach dem Starten beim Hauptagenten über DPI an und teilt ihm mit, für welchen MIB-Baum er zuständig sein wird.

Wenn von der Managementplattform eine Anfrage (GET/SET) über SNMP an den Hauptagenten gelangt, kann dieser anhand der entsprechenden OID feststellen, ob er die Anfrage selber bearbeiten kann, oder ob für die angegebene OID ein bestimmter Subagent zuständig ist.

Der Hauptagent überträgt die Anfrage über die DPI Schnittstelle an den richtigen Subagenten, der dann sein Ergebnis über DPI an den Hauptagenten zurückmeldet. Anschließend kann der Hauptagent die Antwort mit SNMP zur Managementplattform senden.

Der Subagent kann sich auch wieder beim Hauptagenten abmelden und beenden.

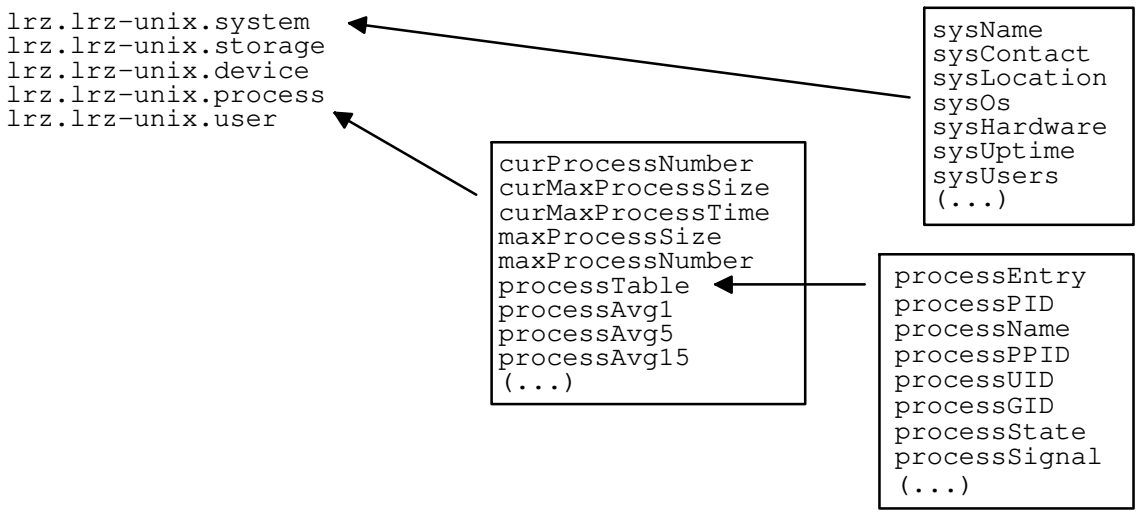
Mit Hilfe von Subagenten kann man also zur Laufzeit des Hauptagenten die MIB erweitern. Ohne den Einsatz von über DPI angebundenen Subagenten müßte man den Hauptagenten direkt modifizieren. Dies geht aber nur, wenn man den Quelltext für den Hauptagenten hat, was allerdings oft nicht der Fall ist, da der Hauptagent vom UNIX Hersteller in Binärform geliefert wird. Hat man den Quelltext, kann man die Erweiterungen im Hauptagenten durchführen, neu übersetzen, den alten Hauptagenten beenden und den neuen Hauptagenten starten.

Traps können vom Subagenten ebenfalls erzeugt werden; diese Möglichkeit wird allerdings von unserem Systemmanagementagenten nicht genutzt.

Neben dem Subagenten für Systemmanagement können noch andere Subagenten existieren; als Beispiel sind in der Skizze der Printer- und der DHCP[1]-Subagent eingezeichnet. Im folgenden geht es ausschließlich um den Subagenten für Systemmanagement.

2.2 Beschreibung der Subagenten MIB

Der Systemmanagementagent registriert die MIB "lrz.lrz-unix" beim Hauptagenten. Die MIB unterteilt sich in fünf Hauptgruppen:



Nähere Informationen zu dieser MIB erhält man in [4]. Folgende Aufstellung soll nur einen kurzen Überblick geben:

---

[1] Dynamic Host Configuration Protocol

Die einzelnen Hauptgruppen sind zuständig für:

- `system`: Informationen zum System, z.B. Hostname, Hardware und Betriebssystem
- `storage`: Informationen zum Speicher, z.B. Größe des Hauptspeichers
- `device`: Informationen zu den einzelnen internen oder externen Komponenten des Systems, z.B. über Prozessoren, Drucker, Festplatten, Bandlaufwerke, etc
- `process`: Information über Prozeßliste
- `user`: Benutzerverwaltung: Löschen, Ändern, Erzeugen von Benutzereinträgen; auch Verwaltung von Gruppen

Insgesamt ist die MIB recht umfangreich und enthält etwa 195 Variablen, 15 Tabellen, 195 GET und 150 SET-Funktionen.

In der obigen Skizze ist die `process` Gruppe noch weiter aufgeschlüsselt, da sie noch mehrmals in weiteren Beispielen verwendet wird.

Variablen aus der `process` Gruppe sind z.B.:

- `curProcessNumber`: Anzahl der Einträge in der Prozeßliste
- `curMaxProcessSize`: Speicherbedarf des momentan größten Prozesses
- `curMaxProcessTime`: Laufzeit des momentan längstlaufenden Prozesses
- `maxProcessSize`: Maximum für die Größe eines Prozesses
- `maxProcessNumber`: Maximum für die Anzahl der Einträge in der Prozeßliste
- `processAvg1`: Systemlast gemittelt über die letzte vergange Minute
- `processAvg5`: Systemlast gemittelt über die letzten 5 vergangen Minuten
- `processAvg15`: Systemlast gemittelt über die letzten 15 vergangenen Minuten
- `processTable`: Tabelle mit den Einträgen der Prozeßliste

`processTable` enthält für jeden Eintrag in der Prozeßliste eine `processEntry` Zeile. Variablen aus einer solchen `processEntry` Zeile sind z.B.:

- `processPID`: Prozeß-ID
- `processName`: Name des laufenden Programms
- `processPPID`: Prozeß-ID des Parent Prozesses
- `processUID`: User ID, unter der der Prozeß läuft
- `processGID`: Group ID, unter der der Prozeß läuft
- `processState`: Prozeßstatus, z.B. RUNNING oder SLEEP
- `processSignal`: nur SET: angegebenes Signal an Prozeß senden

Beispiel für eine `processEntry` Zeile:

```
lrz.lrz-unix.process.processTable.processEntry.processPID.1 = 1
lrz.lrz-unix.process.processTable.processEntry.processName.1 = "init"
lrz.lrz-unix.process.processTable.processEntry.processPPID.1 = 0
lrz.lrz-unix.process.processTable.processEntry.processUID.1 = 0
(...)
```

Gezeigt wird der Prozeß "`init`", der auf jedem UNIX System als allererster Prozeß gestartet wird. Er hat deshalb Prozeß-ID 1 und natürlich UID 0, d.h. er läuft mit root-Berechtigung.

Über die MIB kann man nicht nur managementrelevante Daten auslesen, sondern auch steuernd eingreifen. Dies geschieht durch Schreiben auf bestimmte MIB Variablen.
Ein Beispiel hierfür ist die Variable `processSignal` aus der `processEntry` Zeile. Durch Setzen dieser Integer-Variable wird dem Prozeß dieses angegebene Signal gesendet.

Man kann also einen Prozeß beenden, indem man in die `processSignal` Variable des betreffenden Prozesses einen Wert von `9` schreibt. Der Wert `9` steht dabei für das UNIX Signal `SIGKILL`.

Im Abschitt 6 wird hierfür ein Beispiel gezeigt.

3 Übernommener Stand

Der von mir übernommene Quelltext stellt sich wie folgt dar:

```
 1874 Mar 12 12:55 Makefile
15296 Mar 12 12:55 cmp.c
20378 Mar 12 12:55 data.c
15266 Mar 12 12:55 devices.c
16431 Mar 12 12:55 dpi_system.c
16171 Mar 12 12:55 dpi_system.h
   66 Mar 12 12:55 dpi_version.h
 9133 Mar 12 12:55 filesystem.c
 9151 Mar 12 12:55 getnext.c
13733 Mar 12 12:55 printer.c
 4850 Mar 12 12:55 process.c
 2215 Mar 12 12:55 processor.c
23357 Mar 12 12:55 set.c
 4209 Mar 12 12:55 storage.c
 4231 Mar 12 12:55 system.c
 2107 Mar 12 12:55 upage.c
10034 Mar 12 12:55 user.c
```

Dieser Quelltext-Stand läßt sich übersetzen und ist lauffähig unter SunOS 4.1.3.

Das Quelltext-Paket besteht aus einem Makefile, 14 C-Quelltexten und 2 Include-Dateien.

Erkennbar ist eine grobe Aufteilung der Implementierung nach den fünf Hauptgruppen der Systemmanagement-MIB in die Quelltexte `system.c`, `process.c`, `user.c`, `storage.c` und `devices.c`. Teile der `devices` Gruppe sind in die Dateien `printer.c` und `processor.c` ausgelagert.

Allerdings befinden sich alle SET Funktion zusammengewürfelt in der Datei `set.c`, ebenso wie alle GETNEXT Funktionen in der Datei `getnext.c`. Dieses Verfahren ist recht ungewöhnlich, da hierdurch Funktionen, die logisch zusammengehören, über mehrere Dateien verteilt sind. Anstatt dieser Trennung nach SET, GET und GETNEXT wäre eine Trennung nach MIB Variablen besser, d.h. SET, GET und GETNEXT Funktionen einer MIB Variable befinden sich in ein und derselben Datei.

In der Datei `dpi_system.c` ist das Hauptprogramm und die DPI-Schnittstelle enthalten. Das Hauptprogramm meldet den Subagenten beim Hauptagenten an, und wartet dann in einer Endlos-Schleife auf GET, GETNEXT oder SET Anweisungen vom Hauptagenten.

In der Datei `data.c` ist eine Tabelle enthalten, die eine Zuordnung zwischen OIDs und den GET, GETNEXT und SET Funktionen herstellt. Wenn eine Anweisung für eine bestimmte OID vom Hauptagenten beim Subagenten eingeht, kann dieser anhand der Tabelle aus `data.c` die zu dieser OID gehörende GET, GETNEXT oder SET Funktion bestimmen und aufrufen.

Auffällig ist die Verwendung von Include-Dateien: im wesentlichen wird nur eine einzige Include Datei verwendet, nämlich `dpi_system.h`. Die zweite Include Datei

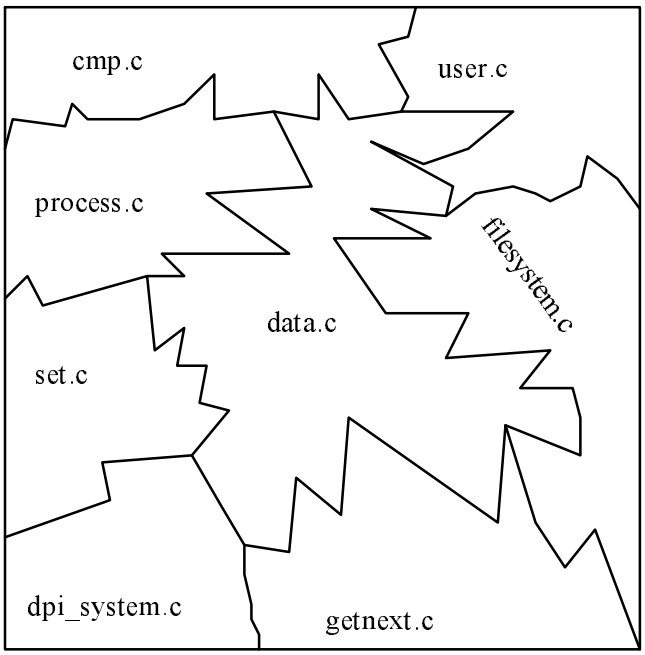
`dpi_version.h` enthält nur eine Versionskennung für die DPI-Schnittstelle, und ist nicht weiter wichtig.

In `dpi_system.h` befinden sich alle Funktionsdeklarationen, Makrodefinitionen und `#include`-Anweisungen. Die einzelnen C-Quelltexte binden dann nur `dpi_system.h` ein.

Diese Vorgehensweise ist zwar bequem ("einfach alles includen, was man vielleicht brauchen könnte"), aber man stellt schnell Nachteile fest: z.B. kann man nicht sofort sagen, ob eine Makrodefinition, die in `dpi_system.h` steht, überhaupt irgendwo gebraucht wird.

Besser und üblicher wäre es gewesen, für jeden C-Quelltext eine Include Datei zu erstellen, die alle Prototypen derjenigen Funktionen enthält, die durch den C-Quelltext zur Verfügung gestellt werden sollen. Makrodefinitionen, die nur zur Implementierung bestimmter Funktionen notwendig sind, gehören in den C-Quelltext dieser Funktion. Viele Funktionen, die nicht nach außen hin sichtbar sein müssen, sollten `static` deklariert sein, und nicht in Include Dateien erscheinen.

Insgesamt ist der Quelltext des Subagenten sehr verstrickt. Folgende Skizze verdeutlicht dies:



4 Portierung

Bei den ersten naiven Versuchen, den Systemmanagementagent zu portieren, stößt man auf viele Probleme, die hier kurz wiedergegeben werden. Ausgehend von den Problemen wird überlegt, wie man sich die Portierung des Systemmanagementagenten erleichtern kann.

4.1 Probleme

4.1.1 Umstellung auf DPI Library

Der vorhandene Quelltext enthält in der Datei `dpi_system.c` die DPI-Schnittstelle zum Hauptagenten. Mittlerweile existierte jedoch am Lehrstuhl diese DPI-Schnittstelle als herausgelöste C-Library, die man nur noch zum übrigen Progamm hinzulinken muß, um einen DPI-fähigen Subagenten zu erhalten.

Eine Besonderheit dieser Library ist, daß sie bereits das Hauptprogramm `main()` für den Subagenten enthält. Der Entwickler eines Subagenten muß jetzt im wesentlichen nur noch eine Sprungtabelle mit OIDs und Zeigern auf die GET, SET und GETNEXT-Funktionen bereitstellen und kann sich auf die Implementierung dieser Funktionen konzentrieren.

Desweiteren muß eine Initialisierungsfunktion `init_subagent()` und eine Routine zum Versenden von Traps, `trap_verschicken()`, zur Verfügung gestellt werden.

Die Funktionsweise dieser Library verdeutlicht man sich am besten mit dem einfachen Beispiel-Subagenten in `/proj/sysagent/Subagent/src/Beispiel/`.

4.1.2 Systemabhängigkeit

Ein weiteres Problem ist die sehr starke Systemabhängigkeit, die praktisch dazu führt, daß sehr viele Teile des Systemmanagementagenten neu geschrieben werden müssen, wenn man den Subagenten auf ein anderes Betriebssystem portieren will.

Dazu zählen fast alle Funktionen, die direkt mit dem Kernel etwas zu tun haben, wie z.B. das Auslesen der Prozeßliste. Aber auch nicht kernel-nahe Funktionen müssen u.U. neu implementiert werden: SunOS verwendet ein BSD Drucksystem, wogegen man heute bei vielen Unixen nur noch das System V Drucksystem vorfindet.

Als Beispiel einer kernelabhängigen MIB Variable dient `lrz.lrz-unix.process.curProcessNumber`, welche die Anzahl der momentan in der Prozeßliste befindlichen Einträge enthält. Die GET Funktion für diese MIB Variable wird durch `get_curProcessNumber()` implementiert.

Der folgende Kasten zeigt eine mögliche Implementierung für SunOS 4.1.3:

```
const int *get_curProcessNumber( int *not_used )
{
    static int count;
    kvm_t *kd;

    kd = kvm_open( NULL, NULL, NULL, O_RDONLY, NULL );
    count = 0; while( kvm_nextproc( kd ) ) ++count;
    kvm_close( kd );

    return &count;
}
```

Unter SunOS 4.1.3 erlangt man mit `kvm_open()` Zugriff zum Speicherbereich des Kernels. Anschließend muß man die Prozeßliste von vorne bis hinten durchgehen und die Einträge mitzählen, bis schließlich `kvm_nextproc()` das Ergebnis NULL, d.h. Ende der Liste, zurückliefert.

Eine mögliche Implementierung unter IBM AIX 3.2.5 sieht hingegen so aus:

```
const int *get_curProcessNumber( int *not_used )
{
    static int count;

    count = getproc( procinfo, NPROCS,
                     sizeof( struct procinfo ) );

    return &count;

}
```

Unter IBM AIX 3.2.5 gibt es den recht wertvollen Systemaufruf `getproc()`, der selbständig die gesamte Prozeßliste aus dem Kernelbereich in einen bereitgestellten Speicherbereich kopiert. Als Rückgabewert liefert `getproc()` die Anzahl der kopierten Prozeßlisteneinträge zurück; dies ist genau der Wert für die MIB Variable `lrz.lrz-unix.process.curProcessNumber`.

Die Funktion `get_curProcessNumber()` ist nur ein sehr einfaches Beispiel von Routinen, die komplett neu implementiert werden müssen.

4.1.3 Übersichtlichkeit und Wartbarkeit

Der Quelltext des Subagenten besteht aus recht wenigen, dafür aber umfangreichen Dateien.

Ein Fortsetzen an diesem Punkt würde wohl dazu führen, daß z.B. `set.c` auf das etwa Doppelte anwächst, da so gut wie jede darin enthaltene Funktion neu implementiert werden muß. Andererseits gibt es doch wieder einige Funktionen, die auf IBM AIX unverändert übernommen werden könnten.

Fragwürdig wäre auch auch die Erweiterbarkeit und Wartbarkeit des geschaffenen Codes, da der Quelltext mit sehr vielen `#if`, `#else` und `#endif` übersäht wäre. Häufige und

möglicherweise geschachtelte bedingte Kompilierung des Quelltextes mit Hilfe von `#if/#else/#endif`-Konstrukten kann den Quelltext sehr unleserlich machen.

4.2 Systematik

Um die kurz angerissenen Probleme in den Griff zu bekommen, wurde überlegt, wie man den Quelltext umstrukturieren und modularisieren kann, damit die Portierung auf IBM AIX 3.2.5 und evtl. noch folgende Portierungen einfacher werden.

Die folgenden Punkte beschreiben die Kriterien für die durchgeführte Modularisierung des Subagenten-Quelltexts:

4.2.1 Verwendung der DPI Library

Als erstes ist es unbedingt notwendig, die Routinen für die DPI-Schnittstelle durch die vorhandene DPI Library zu ersetzen. Diese DPI Library basiert auf dem IBM Code für DPI 2.0, und läßt sich ohne große Schwierigkeiten auf jedem Unix übersetzen, da sie als Schnittstelle zu TCP/IP einfach nur das BSD socket Interface verwendet, welches überall verfügbar sein sollte.

Durch das Herauslösen der DPI Schnittstelle aus dem Quelltext des Subagenten wird dieser einfacher und man muß jetzt nur noch die Zuordnungstabelle OIDs <--> GET, GETNEXT, SET Funktionen bereitstellen und diese Funktionen implementieren.

An der vorhandenen DPI Library wurde eine wichtige Modifikation vorgenommen:

Beim Start des Subagenten muß eine Initialisierungsroutine aufgerufen werden, die z.B. einliest, welche Filesysteme und Drucker vorhanden sind. Da aber der Quelltext des Subagenten nur noch aus der Sprungtabelle und den GET, GETNEXT und SET Funktionen besteht, muß der Aufruf der Initialisierungsroutine aus der `main()` Funktion der DPI-Library erfolgen.

Diese Möglichkeit war in der DPI-Library nicht vorgesehen und mußte erst nachträglich eingebaut werden. Bei Programmstart ruft nun die `main()` Funktion die Routine `init()` auf, die vom Quelltext des Subagenten bereitgestellt werden muß. Wenn man keine besondere Initialisierung des Subagenten benötigt, kann man als `init()` Routine einfach nur eine leere Funktion verwenden.

Im Quelltext des Beispiel-Subagenten ist die Verwendung der Initialisierungsroutine recht gut ersichtlich (`/proj/sysagent/Subagent/src/Beispiel/`).

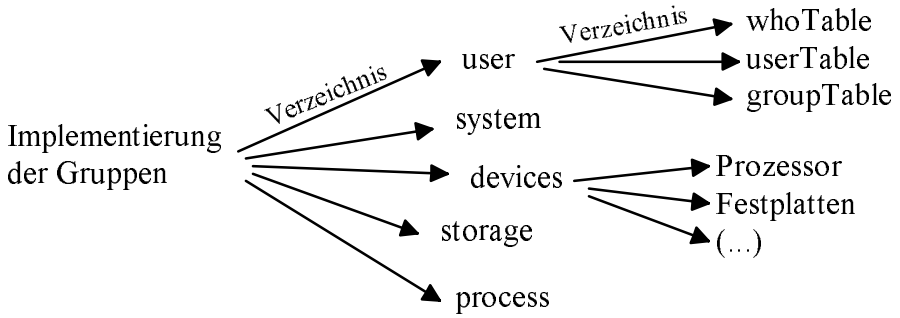
4.2.2 Ordnung nach Gruppen

Ferner ist eine strenge Unterteilung des Quelltextes gemäß den Zugehörigkeiten zu den fünf Hauptgruppen wünschenswert. Im übernommen Quelltext ist dies ansatzweise schon der Fall, allerdings wurden dort alle SET Funktionen in einen einzigen Quelltext geworfen.

Denkbar wären auch noch weitere Unterteilungen nach Untergruppen oder einzelnen MIB Variablen. Eine Unterteilung in Untergruppen ist bei der `device` Gruppe sinnvoll, da man jedes Gerät in einer separaten Datei behandeln kann.

Ziel ist es, alle Funktionen, die logisch zusammengehörende MIB Variablen betreffen, in jeweils einem Modul zusammenzuschließen. Damit kann man die entsprechenden Quelltexte recht klein und übersichtlich halten. Bei Bedarf wäre eine weitere Unterteilung denkbar.

Wenn man die Gruppen in verschiedene Unterverzeichnisse schiebt, kann sich folgendes Bild ergeben:



4.2.3 Ordnung nach Plattformabhängigkeit

Im Subagenten gibt es Quelltext, der für alle Plattformen gleich ist, und somit bei jeder Portierung immer wieder unverändert übernommen werden kann.

Es wäre geschickt, genau diese plattformunabhängigen Teile vom restlichen Code zu trennen und in separate Verzeichnisse zu stellen. Bei folgenden Portierungen kann man dann diese Teile wieder unverändert übernehmen, und muß nur den plattformabhängigen Teil des Subagenten überarbeiten.

Bei den plattformabhängigen Quelltexten kann man noch nach den entsprechenden Plattformen unterteilen. Auf diese Weise kann eine Sammlung von `#if`, `#else` und `#endifs` im Quelltext vermeiden, denn zu viele Anweisungen zur bedingten Kompilierung machen den Quelltext sehr schnell unleserlich.

4.2.4 Portabilität

Ein weiteres Augenmerk bei der Modularisierung wurde auf Portabilität gelegt. Grundsätzlich sollen weitere Portierungen so einfach wie nur möglich sein und nur die plattformabhängigen Teile betreffen.

Inwiefern sich dies tatsächlich realisieren läßt, steht nicht ganz fest, da es durchaus sein kann, daß Programmteile, die als plattformunabhängig eingeordnet wurden, auf irgendeiner Zielplattform nicht ohne weiteres funktionieren.

4.2.5 Wiederverwendbarkeit

Wichtig ist auch die Wiederverwendbarkeit des Quelltextes. Es sollte möglich sein, den Code z.B. bei einer Portierung für CORBA verwenden zu können.

Es gibt zwei Möglichkeiten der Wiederverwendbarkeit des Quelltextes:

- Alle GET, GETNEXT und SET Funktionen des Subagenten zu einer Library zusammenfassen: wenn man von einem anderen Programm auf diese Funktionen

zugreifen will, braucht man nur die Library des Subagenten hinzulinken. Viele dieser Funktionen erwarten allerdings einen Parameter beim Aufruf, den andere Programme, die möglicherweise nichts von MIBs und OIDs wissen, nicht übergeben können.

- Alle Quelltexte der GET, GETNEXT und SET Funktionen so klein und einfach wie nur möglich halten, damit man diese Funktionen leicht modifiziert in ein anderes Programm einfach mit "copy&paste" übernehmen kann. Dies bedeutet zwangsläufig sehr viel Handarbeit.

Beide Ansätze haben gewisse Nachteile. Dafür eine Lösung zu finden, ist aber nicht Thema dieses Fortgeschrittenpraktikums.

Um die Wiederverwendbarkeit zu fördern, sollte der Quelltext natürlich striktes ANSI-C sein.

5 Jetziger Stand

Ausgehend von oben aufgeführten Kriterien wurde der Quelltext des Subagenten vollständig neu strukturiert, um einen modularen Aufbau zu erhalten.

5.1 Ergebnis der Modularisierung

Das Ergebnis der Modularisierung wird gezeigt am einfachen Beispiel der MIB Variable `lrz.lrz-unix.process.loadAvg1`, `...loadAvg5` und `...loadAvg15`. Diese MIB Variablen enthalten jeweils die gemittelte Systemlast über die letzte, die letzten 5 und die letzten 15 vergangenen Minuten.

Die GET Funktionen für diese MIB Variablen werden jeweils durch die Funktionen `getloadAvg1()`, `getloadAvg5()` und `getloadAvg15()` implementiert (im folgenden als `getloadAvg{1,5,15}()` bezeichnet).

Jede dieser drei Funktionen verwendet die zugrundeliegende Routine `getloadAvg()`:

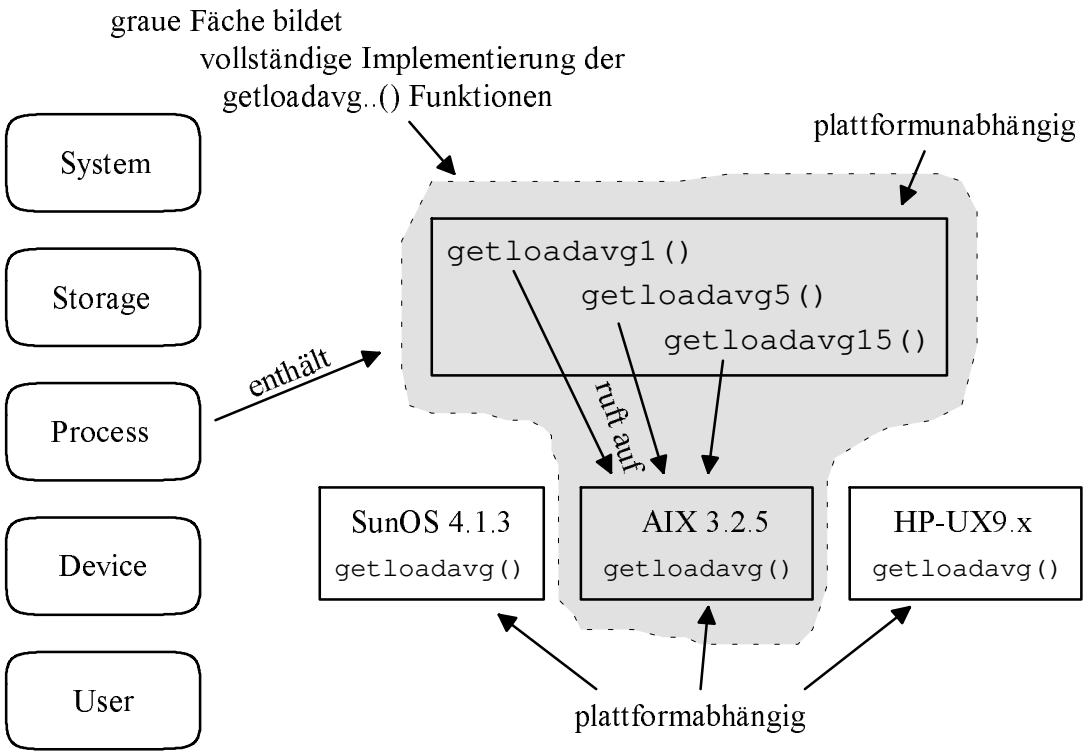
```
const *int getloadavg1() { return getloadavg(0); }
const *int getloadavg5() { return getloadavg(1); }
const *int getloadavg15() { return getloadavg(2); }
```

Die Basisfunktion `getloadavg()` liefert, je nach dem, mit welchem Argument sie aufgerufen wurde, das eigentlich Ergebnis zurück.

Die drei Funktionen `getloadAvg{1,5,15}()` sind offensichtlich für alle Plattformen gleich; nur das zugrundeliegende `getloadAvg()` muß von Betriebssystem zu Betriebssystem angepaßt werden.

Dieses Beispiel mag trivial erscheinen, aber es zeigt die Vorgehensweise, um möglichst viel plattformunabhängigen Code und dafür möglichst wenig plattformabhängigen Code zu erhalten.

Die folgende Skizze verdeutlicht das Zusammenspiel der `getloadAvg`-Funktionen nochmals:



5.2 Aufteilung in Verzeichnisse und Dateien

Der Quelltext des Subagenten ist grundsätzlich in einen plattformabhängigen und einen plattformunabhängigen Quelltextbaum unterteilt.

Von dem plattformabhängigen Quelltextbaum kann es mehrere Ausführungen geben, je nach dem, welche Portierungen schon durchgeführt worden sind. Vorbereitet sind folgende Verzeichnisse:

```
src_generic/
src_rs6000-ibm-aix3.25/
src_hppa1.1-hp-hpux9.01/
src_sparc-sun-sunos4.1.4/
src_sparc-sun-solaris2.5/
```

Das `src_generic` Verzeichnis enthält alle plattformunabhängigen Quelltexte, wie z.B. die Sprungtabelle mit der OID <--> GET, GETNEXT, SET Funktionszuordung und die `getloadAvg{1,5,15}()` Funktionen.

Die Namen der plattformabhängigen Verzeichnisse bilden sich aus der entsprechenden Bezeichnung der Zielplattform nach dem Schema <Prozessor>-<Hersteller>-<Betriebssystem>. Dies ist genau die Ausgabe, die das Programm `config.guess` von GNU liefert. Dadurch wird es möglich, daß die Makefiles ohne zusätzliche Anpassung selbst erkennen können, auf welcher Plattform gerade kompiliert wird. Anhand der Plattformbezeichnung weiß Make dann, in welchen Verzeichnissen die passenden Quelltexte zu finden sind.

In diesem plattformabhängigen Teil kann man z.B. die Funktion `getloadAvg()` finden.

In den Quelltextverzeichnissen befindet sich jeweils eine Verzeichnisstruktur, die der Aufspaltung nach den fünf Hauptgruppen entspricht. Das src_generic Verzeichnis enthält also folgende Verzeichnisstruktur:

```
group_system/
group_storage/
group_device/
group_device/group_disk/
group_device/group_filesystem/
group_device/group_printer/
group_device/group_processor/
group_device/group_partition/
group_process/
group_user/
```

Ggf. kann die Aufteilung in Verzeichnisse noch weiter gehen als hier angegeben.

Wenn man sich das `src_generic` und das `src_rs6000-ibm-aix3.2.5` "überlagert" vorstellt, erhält man die vollständige Implementierung des Subagenten für IBM AIX 3.2.5. Wenn man `src_generic` und `src_sparc-sun-sunos4.1.3` "überlagert", erhält man die Implementierung für SunOS 4.1.3.

Für das zu erzeugende ausführbare Programm `sysagent` gibt es auch verschiedene Unterverzeichnisse, die sich ebenfalls an dieselbe Namenskonvention wie die plattformabhängigen Quelltextverzeichnisse halten:

```
bin_hppa1.1-hp-hpux9.01/
bin_rs6000-ibm-aix3.2.5/
bin_sparc-sun-sunos4.1.4/
bin_sparc-sun-solaris2.5/
```

Wenn der Systemmanagementagent auf einer Plattform übersetzt wird, dann wird das ausführbare Programm im entsprechenden `bin_` Verzeichnis angelegt.

5.3 Zusammenspiel der Quelltexte

Da sich durch die Einführung der DPI Library und der übrigen Umstrukturierung einige Änderungen im Ablauf des Subagenten ergeben haben, wird im folgenden das Zusammenspiel der Quelltexte kurz dargestellt.

Die Basis des Subagenten bildet die DPI Library, die über DPI mit dem Hauptagenten in Verbindung steht.

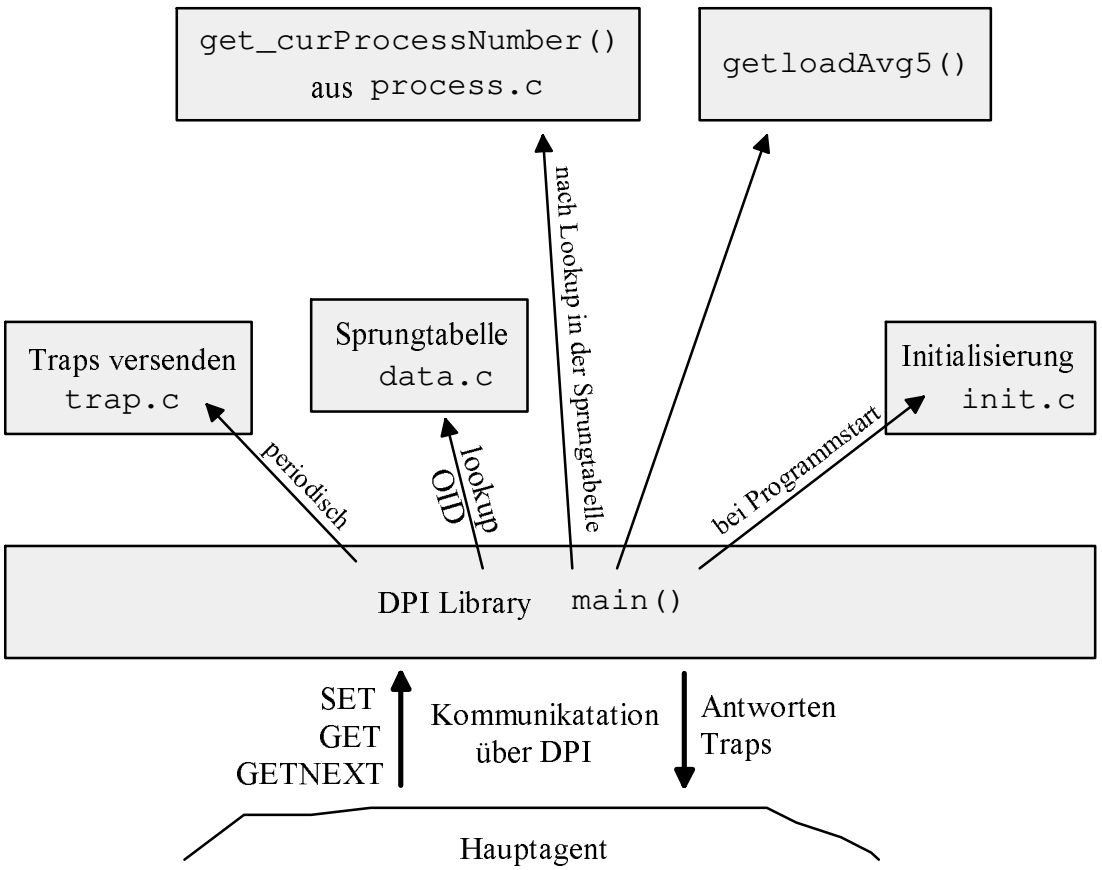
Aus der Sicht des Entwicklers eines Subagenten interessiert die Seite hin zum Hauptagenten nicht weiter, da alle anfallenden Aufgaben diesbezüglich selbständig und transparent von der DPI Library erledigt werden.

Auf der Seite des Subagenten sind grundsätzlich die drei Quelltexte `trap.c`, `data.c` und `init.c` von Bedeutung.

In der Datei `init.c` befindet sich die Initialisierungsroutine `init_subagent()` des Subagenten. Diese Routine wird beim Starten des Subagenten von der `main()` Routine der

DPI Library aufgerufen, und bietet die Möglichkeit, erste Initialisierungen durchzuführen, bevor noch irgendwelche GET, GETNEXT oder SET Anforderungen ankommen.

Vorzugsweise führt man in dieser Routine alle Initialisierungen durch, die recht aufwendig sind und verhältnismäßig lange dauern. Auch können schon die Inhalte einiger Variablen bestimmt und im Speicher gehalten werden, um anschließende Anfragen ohne Verzögerungen beantworten zu können.



Nach Abarbeitung dieser Initialisierungsroutine und der Anmeldung des Subagenten beim Hauptagenten, geht `main()` Routine der DPI Library in eine Endlosschleife, in der auf eingehende Anfordungen vom Hauptagenten gewartet wird und periodisch geprüft wird, ob es Traps zum versenden gibt.

Zum Prüfen, ob Traps versendet werden müssen, wird regelmäßig von `main()` aus die Routine `trap_versenden()` aufgerufen, die in der Datei `trap.c` enthalten ist. Diese Routine stellt ein Hintertürchen dar, durch das es möglich wird, Traps vom Subagenten an den Hauptagenten zu senden. Der vorliegende Systemmanagementagent versendet keine Traps, deshalb enthält diese Routine hier nur einen leeren Rumpf.

Die Datei `data.c` enthält im wesentlichen eine Tabelle, mit der eine Zuordnung zwischen OIDs und den zugehörenden GET, GETNEXT und SET Funktionen hergestellt wird.

Eine Zeile aus dieser Sprungtabelle hat den Aufbau: OID, GET-Funktion, SET-Funktion, Rückgabetyp, GETNEXT-Funktion.

Wenn eine Anfordung vom Hauptagenten eintrifft, kann `main()` der DPI Library in dieser Tabelle nach der entsprechenden OID suchen und erhält dadurch die Zeiger auf die GET, GETNEXT und SET Funktionen.

Wenn z.B. eine GET Anfrage für OID `.1.3.6.1.3.100.2.4.3.0` (MIB Variable `lrz.lrz-unix.process.curProccessNumber`) bei der DPI Library eintrifft, dann liefert der Tabellen-Lookup den Zeiger auf die Funktion `get_curProcessNumber()`, die dann aufgerufen wird.

6 Praktischer Einsatz

In diesem Abschnitt wird der praktische Einsatz des Subagenten für Systemmanagement beschrieben, von der Übersetzung über die Installation bis hin zu einfachen, ausgewählten Beispielen.

6.1 Übersetzen

Der Übersetzungvorgang wird vollautomatisch von den Makefiles durchgeführt.
Wenn man erstmalig den Übersetzungsvorgang startet, geht man am besten so vor:

In das Verzeichnis mit den Quelltexten gehen:

```
schmidts@ibmhegering1:~ -> cd /proj/sysagent/Subagent/src
```

Der Quelltextbaum muß sich nicht unbedingt in `/proj/sysagent/Subagent/src` befinden. Man kann ihn auch an eine andere Stelle kopieren und dann dort kompilieren.

Make aufrufen:

```
schmidts@ibmhegering1:/proj/sysagent/Subagent/src -> make clean
schmidts@ibmhegering1:/proj/sysagent/Subagent/src -> make depend
schmidts@ibmhegering1:/proj/sysagent/Subagent/src -> make all
```

"`make clean`" löscht evtl. vorhandene Objektdateien. Dies ist wichtig, wenn man vorher für eine andere Plattform kompiliert hat, und jetzt einen sauberen Ausgangszustand braucht.

"`make depend`" speichert für alle Quelltexte die Abhängigkeiten von Include-Dateien. Nach Änderung einer Include-Datei weiß Make dann automatisch, welche C-Quelltexte von der geänderten Include-Datei abhängen, und kompiliert nur diese neu.

Schließlich startet "`make all`" die Übersetzung des ganzen Paketes. Erzeugt wird dabei die DPI-Library, der Beispiel-Subagent und der Systemmanagement-Subagent.

Nach erfolgreicher Kompilation befindet sich die DPI-Library im Verzeichnis `/proj/sysagent/Subagent/src/Interface/lib`:

```
schmidts@ibmhegering1:/proj/sysagent/Subagent/src/Interface/lib -> ls -l
total 900
-rw-rw-r--   1 schmidts sysmgmt   122764 Oct 31 1995  libdpi_SunOS.a
-rw-rw-r--   1 schmidts sysmgmt   183217 Jun 27 10:41 libdpi_aix.a
-rw-rw-r--   1 schmidts sysmgmt   115342 Oct 27 1995  libdpi_hpux.a
```

In `/proj/sysagent/Subagent/src/System/bin_rs6000-ibm-aix3.2.5` befindet sich der Systemmanagementagent:

```
schmidts@ibmhegering1:/proj/sysagent/Subagent/src/System/bin_rs6000-ibm-
  aix3.2.5 -> ls -la
total 4116
drwxrwsr-x   2 schmidts sysmgmt      512 Jun 27 10:45 ./
drwxrwsr-x  12 hainzing sysmgmt      512 Jun 27 10:40 ../
-rwxr-xr-x   1 schmidts sysmgmt  2094876 Jun 27 10:45 sysagent*
```

6.2 Installation

Nachdem der Übersetzungsvorgang erfolgreich abgeschlossen wurde, ist die Installation des Subagenten sehr einfach.

6.2.1 Ausführbares Programm

Man kopiert am besten von Hand den ausführbaren Systemmanagementagenten in das Zielverzeichnis. In diesem Beispiel wird `/usr/local/etc/` verwendet:

```
schmidts@ibmhegering1:/proj/sysagent/Subagent/src/System/bin_rs6000-ibm-
  aix3.2.5 -> cp sysagent /usr/local/etc/.
```

Man kann auch ein beliebig anderes Verzeichnis wählen.

6.2.2 Konfigurationsdatei

Die Konfigurationsdatei für den Subagenten steht normalerweise in `/etc/lrz-mib.conf`. Wenn man eine andere Datei als Konfigurationsdatei angeben will, so muß man vor dem Start des Subagenten die Environmentvariable LRZCONFFILE entsprechend gesetzt haben.

```
schmidts@ibmhegering1:~ -> cat /etc/lrz-mib.conf
-- Dieses File ist Bestandteil der Implementierung der
-- LRZ-Unix-MIB.
-- Es enthaelt die Werte einiger MIB-Variablen, die nicht
-- auf andere Weise zu ermitteln waren.

--Konfigurationsfile fuer ibmhegering1

clockrate=33
specint=18.2
specintyear=92
specfp=17.9
specfpyear=92
printerLocation(np)=LMU-Muenchen
sysLocation=Leopoldstr. so-und-so, 12345 Muenchen
sysContact=Stefan
```

Die Konfigurationsdatei enthält Werte von MIB Variablen, die sich nicht selbständig vom Subagenten ermitteln lassen. Einige Variablen (z.B. `sysContact`) können durch entsprechende SET Befehle auch erst später geändert werden. Die Bedeutung der einzelnen Variablen ist in [1] dokumentiert.

Wenn man die Datei von Hand anpaßt, ist auf die Einhaltung des Zeilenformats zu beachten:

```
<Variable>=<Wert>
```

Vor und nach dem =-Zeichen steht kein Leerzeichen, Tabulator, etc.! Auf richtige Groß- und Kleinschreibung ist zu achten.

Als Ausgangspunkt kann man die Konfigurationsdatei `lrz-mib.conf` aus dem Verzeichnis `/proj/sysagent/Subagent/src/System/src_generic/` verwenden.

6.3 Starten

Bitte vor dem Starten des Subagenten ggf. noch den Hauptagenten starten.

Da der Subagent Systemaufrufe durchführen muß, die nur unter root Berechtigung zugelassen sind, muß man vor dem Start des Subagenten ein "su" machen.

```
schmidts@ibmhegering1:~ -> su
root's Password:

root@ibmhegering1:~ -> /usr/local/etc/sysagent &
(...)
root@ibmhegering1:~ -> exit
```

Besser ist es jedoch, den Hauptagenten und alle gewünschten Subagenten schon bei Systemstart zu aktivieren. Bei System V basierten Unix Umgebungen kann man hierzu entsprechende Eintragungen in der Datei `/etc/inittab` oder in anderen Startup-Dateien (`/etc/rc*`) vornehmen (s. entsprechende Manpages).

Vom Starten der Agenten über sog. "set user id root" Shellscripte ist dringend abzuraten, da das Einrichten solcher Scripten ein großes Sicherheitsloch bedeutet.

Um ein Schreiben auf MIB Variablen zu erlauben, muß der Systemmanagementagent mit der Option `-write` gestartet werden. Wenn diese Option nicht angegeben wird, werden alle SET Anweisungen schon in der DPI-Library abgefangen und nicht ausgeführt.

Weitere Optionen für den Subagenten sind in [1] dokumentiert.

6.4 Beispiele

Im folgenden werden einfache Beispiele beschrieben, die den ein Umgang mit dem Subagenten zeigen. Es werden nur die Programme `snmpwalk`, `snmpget` und `snmpset` vorausgesetzt.

Um größere Aktionen, wie z.B. das Hinzufügen eines neuen Benutzers durchzuführen, verwendet man am besten graphische Managementplattformen, wie z.B. "Synema" (Scotty/Tk/Tcl muß richtig installiert sein) oder eine entsprechende OpenView Umgebung.

Für die gezeigten Beispiele reichen jedoch `snmpwalk`, `snmpget` und `snmpset` aus.

6.4.1 Abfragen der Systemlast

Die Systemlast kann über die bereits vorgestellten Variablen

```
lrz.lrz-unix.process.loadAvg1
lrz.lrz-unix.process.loadAvg5
lrz.lrz-unix.process.loadAvg15
```

abgefragt werden:

```
~ -> snmpget -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.7.0
lrz.lrz-unix.process.loadAvg1.0 = 461

~ -> snmpget -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.8.0
lrz.lrz-unix.process.loadAvg5.0 = 213

~ -> snmpget -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.9.0
lrz.lrz-unix.process.loadAvg15.0 = 107
```

Diese Werte bedeuten eine gemittelte Systemlast von 4.61, 2.13 bzw. 1.07 über die letzte 1, 5 bzw. 15 Minuten.

6.4.2 Anzahl der Prozesse

Über die MIB Variable `lrz.lrz-unix.process.curProcessNumber` kann die Anzahl der Prozesse abgefragt werden:

```
~ -> snmpget -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.3.0
lrz.lrz-unix.process.curProcessNumber.0 = 119
```

Es befinden sich also momentan 119 Einträge in der Prozeßliste.

6.4.3 Holen der Prozeßliste

Die vollständige Prozeßliste kann mit einem einfachen `snmpwalk` geholt werden:

```
~ -> snmpwalk -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.6.1
lrz.lrz-unix.process.processTable.processEntry.processPID.1 = 1
lrz.lrz-unix.process.processTable.processEntry.processPID.1330 = 1330
lrz.lrz-unix.process.processTable.processEntry.processPID.1824 = 1824
lrz.lrz-unix.process.processTable.processEntry.processPID.2302 = 2302
lrz.lrz-unix.process.processTable.processEntry.processPID.2816 = 2816
(...)
lrz.lrz-unix.process.processTable.processEntry.processName.1 = "init"
lrz.lrz-unix.process.processTable.processEntry.processName.1330 = "portmap"
lrz.lrz-unix.process.processTable.processEntry.processName.1824 = "srcmstr"
lrz.lrz-unix.process.processTable.processEntry.processName.2302 = "syncd"
lrz.lrz-unix.process.processTable.processEntry.processName.2816 = "errdemon"
(...)
lrz.lrz-unix.process.processTable.processEntry.processPPID.1 = 0
lrz.lrz-unix.process.processTable.processEntry.processPPID.1330 = 1824
lrz.lrz-unix.process.processTable.processEntry.processPPID.1824 = 1
lrz.lrz-unix.process.processTable.processEntry.processPPID.2302 = 1
lrz.lrz-unix.process.processTable.processEntry.processPPID.2816 = 1
(...)
```

Die Bedeutung der einzelnen Einträge kann man in der Dokumentation der MIB [4] nachlesen.

6.4.3 Beenden eines Prozesses

Um einen Prozeß zu beenden, kann man ihm z.B. über die MIB Variable `lrz.lrz-unix.process.processTable.processEntry.processSignal` Unix Signal `SIGKILL(9)` senden.

In diesem Beispiel wird der Prozeß mit der Nummer `40212` beendet:

```
-> snmpset -v 1 ibmhegering1 public .1.3.6.1.3.100.2.4.6.1.12.40212 i 9
lrz.lrz-unix.process.processTable.processEntry.processSignal.40212 = 9
```

Über ein nachfolgendes `snmpget` kann man sich dann noch überzeugen, daß der Prozeß nicht mehr in der Prozeßliste ist.

7 Literaturverzeichnis

[1] Rainer Hauck und Nedo Haubelt, Implementierung einer MIB für Systemmanagementaufgaben, Fortgeschrittenenpraktikum, 1995
[2] IBM AIX 3.2.5 Manual Pages
[3] SunOS 4.1.3 Manual Pages
[4] Uwe Krieger, Konzeption einer Managementinformationsbasis für das Management von UNIX-Endsystemen, Diplomarbeit, 1994
[5] Erwin Hainzinger, Erweiterung eines SNMPv2-Agenten um die Schnittstelle zur Interprozeßkommunikation, Fortgeschrittenenpraktikum, 1994
[6] IBM AIX 3.2.5 Info Explorer