technische universität
dortmund

Masterarbeit

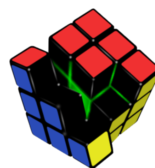
Ebenenübergreifende Bereitstellung von Daten und Lernergebnissen aus der Systemsoftware

Alexander Lochmann

30. September 2014

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Jochen Streicher

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de



Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 30. September 2014

Alexander Lochmann

Zusammenfassung

Das Ziel des Teilprojektes A1 des Sonderforschungsbereichs 876 ist es, ubiquitäre Systemsoftware durch Lernverfahren an die jeweilige Nutzung sowie den Kontext anzupassen [1] Dabei soll durch die Anpassung eine Reduktion des Ressourcenverbrauchs erzielt werden. Am Anfang steht jedoch die Analyse der Daten innerhalb der Systemsoftware. Dafür ist ein Werkzeug von Nöten, das beliebige Daten aus einem Betriebssystem erheben und verarbeiten kann. Die vorliegende Masterarbeit befasst sich mit dem Entwurf und der Implementierung eines solchen Werkzeugs zur ebenenübergreifenden Bereitstellung von Betriebssystemdaten. Zusätzlich sollen über das Werkzeug auch inferierte Daten bereitgestellt werden. In dieser Arbeit wird zu diesem Zweck eine generische Lösung zur Modellierung von Daten innerhalb eines Betriebssystems erarbeitet. Darauf aufbauend wird eine Anfragesprache entwickelt, mit der der Datenfluss innerhalb des Systems abgegriffen und verarbeitet werden kann. Das entwickelte Konzept wird unter dem Betriebssystem *Linux* implementiert. Dieser Prototyp soll der Ausgangspunkt für weitere Forschung bei der Anpassung von ubiquitärer Systemsoftware sein.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Einordnung der Arbeit . . . 1
1.2 Aufbau der Arbeit . . . 2

2 Werkzeuge zur Erhebung von Betriebssystemdaten **3**
2.1 SystemTap . . . 3
2.2 Dtrace . . . 4
2.3 Fay . . . 4
2.4 PiCO QL . . . 5

3 Nutzung von Betriebssystemdaten **7**
3.1 RadioJockey . . . 7
3.2 Paketverzögerung . . . 8
3.3 Erkennung von abnormalem Prozessverhalten in Android . . . 9
3.4 Verbesserungen bei der Prozessorzuteilung in Linux . . . 10

4 Ziele **11**
4.1 Anforderungen . . . 11
4.2 Vorstellung der Fallbeispiele . . . 14
4.2.1 Paketverzögerung . . . 14
4.2.2 Automatische Umschaltung zwischen 2G/3G . . . 14

5 Datenmodell **17**
5.1 Metamodell zur Datenbeschreibung . . . 17
5.2 Fallbeispiele . . . 19
5.2.1 Paketverzögerung . . . 19
5.2.2 Automatische Umschaltung 2G/3g . . . 21
5.3 Fazit . . . 22

6 Anfragesprache **25**
6.1 Anforderungen . . . 26
6.2 Existierende Sprachen . . . 27
6.2.1 Fay . . . 27
6.2.2 TinyDB . . . 28
6.2.3 Aurora . . . 29
6.3 Entwurf der Sprache . . . 30
6.3.1 Datentypen . . . 31

6.3.2 Datenformat 32
6.3.3 Operatoren 32
6.4 Exemplarische Anfragen 37
6.4.1 Paketverzögerung 37
6.4.2 Automatische Umschaltung 2G/3g 38

7 Implementierung 39
7.1 Anforderungen 39
7.2 Kernkomponenten 41
7.2.1 Handhabung des Datenmodells 43
7.2.2 Erzeugung und Verarbeitung von Tupeln 44
7.2.3 Verarbeitung von Anfragen 45
7.3 Ebenenübergreifende Belange 47
7.3.1 Implementierung eines Zeitgebers 47
7.3.2 Kommunikation 49
7.3.3 Synchronisation des Datenmodells 52
7.3.4 Ebenenübergreifende Ausführung von Anfragen 53

8 Evaluation 55
8.1 Ziele 55
8.2 Durchführung 55
8.3 Ergebnisse 57
8.4 Fazit 60

9 Diskussion 63
9.1 Bereitstellung von Lernergebnissen 63
9.2 Datenmodell und Anfragesprache 63
9.3 Implementierung 64
9.3.1 Zugriff auf das Datenmodell 64
9.3.2 Realisierung des Join-Operators 64
9.3.3 Priorisierung von Anfragen 65

10 Zusammenfassung 67

Literaturverzeichnis 71

Abbildungsverzeichnis 73

Tabellenverzeichnis 75

Listingverzeichnis 77

Anhang 79
A.1 Definitionen der Datenstrukturen 79

A.2 SystemTap-Skripte für die Evaluation 80
A.2.1 Aufzeichnung der Prozesserzeugung 80
A.2.2 Aufzeichnung des Netzwerkverkehrs 80

1 Einleitung

Die fortschreitende Miniaturisierung der Elektronik führt zu immer kleineren Geräten bei gleichbleibender oder gar steigender Leistungsfähigkeit. Dadurch wird der Alltag immer weiter mit sogenannten ubiquitären Systemen verwoben. Ganz nach der Vision von Mark Weiser umgeben die Geräte den Menschen [2]. Gerade am Beispiel von *Smartphones* und *Tablets* ist anhand der Absatzzahlen zu sehen, dass sie im Alltag immer präsenter werden [3, 4]. Der Umgang mit den Geräten wird zunehmend gewöhnlicher. Vor allem die Rechenleistung und die üppige Ausstattung mit Peripherie erlaubt den Einsatz in vielen Alltagssituationen. Als ein Beispiel ist die Fülle an Kommunikationstechniken zu nennen, mit denen die Geräte versehen werden. Über die WLAN-Schnittstelle bzw. die Mobilfunkschnittstelle erhalten die Nutzer an unterschiedlichen Orten Zugriff auf das Internet. Nahfunktechnologien wie *Bluetooth* erlauben den Betrieb von Kopfhörern oder Pulsuhren. In Kombination mit dem Positionierungssystem *GPS* halten die *Smartphones* als sogenannte Fitnesstracker auch Einzug in den Freizeitbereich. Zusammenfassend ist zu sagen, dass die Verzahnung mit dem Alltag immer stärker wird. Dementsprechend ist die Verfügbarkeit von Informationen immer bedeutender. D.h. das entsprechende Gerät muss immer einsatzbereit sein. Dem gegenüber steht jedoch die langsame Entwicklung der Batteriekapazität. Laut einer Studie kommt ein *Smartphone* aktuell mit einer Akkuladung ungefähr einen Tag aus [5]. Daher ist abzusehen, dass bei fortschreitender Integration der Elektronik im Alltag, die verfügbare Akkuleistung nicht mehr ausreichen wird, um ein Gerät zumindest einen Tag mit Energie zu versorgen. Deshalb müssen Wege gesucht werden, wie in Zukunft die Verfügbarkeit von Informationen weiterhin gewährleistet wird, bei gleichzeitigem ressourcenschonendem Betrieb von z. B. *Smartphones*.

1.1 Einordnung der Arbeit

Das Teilprojekt A1 des Sonderforschungsbereiches 876 „Verfügbarkeit von Informationen unter Ressourcenbeschränkung“will genau diesem Problem mittels maschinellen Lernverfahren in ubiquitärer Systemsoftware begegnen [1]. Die Systemsoftware soll derart erweitert werden, dass sie sich autonom durch Lernverfahren an die jeweilige Nutzung sowie den Kontext anpasst. Dadurch soll der Ressourcenverbrauch reduziert werden. Am Beginn steht jedoch die Erhebung und Analyse von Daten aus den ubiquitären Systemen, um zum einen Optimierungspotential aufzudecken und zum anderen Lernverfahren für eine etwaige Anpassung trainieren zu können. Dabei ist es wichtig, dass die Daten systemweit, erhoben werden können; d.h. sowohl im Betriebssystemkern als auch auf der Benutzerebene. Gerade im Hinblick auf das Einsatzgebiet *Smartphone* ist

dies von Bedeutung. Deshalb soll in dieser Arbeit ein Werkzeug zur generischen und ebenenübergreifenden Datenerhebung realisiert werden. Dies umfasst den Entwurf eines Verfahrens zur Modellierung von Betriebssystemdaten sowie einer Anfragesprache zur Erhebung und Verarbeitung von eben jenen. Abschließend soll der Entwurf unter dem Betriebssystem *Linux* implementiert und evaluiert werden.

1.2 Aufbau der Arbeit

Nach der Einleitung wird im Kapitel 2 eine Übersicht die verwandten Arbeiten im Bereich der Werkzeuge zur Erhebung von Betriebssystemdaten gegeben. Daran schließt sich in Kapitel 3 die Vorstellung der verwandten Arbeiten zur Nutzung von Betriebssystemdaten durch maschineller Lernverfahren an. Kapitel 4 widmet sich der Definition der Ziele der vorliegenden Arbeit. In Kapitel 5 erfolgt die Präsentation eines Metamodells zur Erzeugung von Datenmodellen für Betriebssystemdaten. Die Anforderungen für eine datenstromorientierte Anfragesprache sowie die Betrachtung existierender Sprachen geschieht in Kapitel 6. Das Kapitel schließt mit der Vorstellung der Anfragesprache. Danach wird die Implementierung des Datenmodells sowie der Anfragesprache aufgegriffen. Es folgt die Betrachtung verschiedener Problemstellungen bei der Implementierung. In Kapitel 8 wird die Methodik zur Bewertung der Implementierung sowie die Messwerte präsentiert. Im folgenden Kapitel werden die unterschiedlichen Ergebnisse dieser Arbeit diskutiert. Abschließend erfolgt eine kurze Zusammenfassung der wichtigsten Resultate dieser Arbeit.

2 Werkzeuge zur Erhebung von Betriebssystemdaten

In dem folgenden Kapitel soll die Familie der Werkzeuge zur Erhebung von Betriebssystemdaten vorgestellt werden. Die vorgestellten Werkzeuge zeigen verschiedene Wege auf, wie Daten erhoben werden können sowie dem Nutzer der Zugang zu den Daten präsentiert wird. Außerdem werden unterschiedliche Arten der Verarbeitung demonstriert. In den folgenden Abschnitten werden vier Vertreter dieser Familie vorgestellt.

2.1 SystemTap

Das Instrumentierungswerkzeug *SystemTap* dient der Diagnose und Analyse der Linux-Betriebssystemkerns [6, 7]. Es lässt sich beispielsweise die Laufzeit von Funktionsaufrufen oder die Häufigkeit von bestimmten Systemaufrufen ermitteln. Das Hauptelement in *SystemTap* ist ein Marker - genannt *probe-point* -, der bei einem definierten Ereignis auslöst und vom Nutzer spezifiziertes Verhalten ausführt. Ein Marker kann u.a. ein Eintritts- oder Austrittspunkt einer Funktion sein. Mittlerweile unterstützt das Werkzeug verschiedene Arten von Markern. Für den aktuellen Fall sind jedoch nur die Genannten von Interesse. Um über den Eintritt in eine Funktion benachrichtigt zu werden, muss der Programmcode des Kerns an der betreffenden Stelle instrumentiert werden. Hierzu benutzt *SystemTap* den kerneigenen Mechanismus der *kprobes*[1]. Hierbei wird an der gewünschten Stelle die Instruktion durch einen Sprung an eine vordefinierte Stelle ersetzt. Wird die besagte Stelle ausgeführt, erfolgt eine Umleitung an eine vom Nutzer definierte Funktion. Innerhalb des Funktionsrumpfes stehen dem Nutzer die Inhalte aller Register zur Verfügung. Das angesprochene vom Nutzer spezifizierte Verhalten wird in Form eines Skriptes festgehalten und durch *SystemTap* ausgeführt. Das Skript, wie es in Listing 2.1 zu sehen ist, wird in einer *C*-ähnlichen Sprache formuliert.

Listing 2.1: Beispiel für ein *SystemTap*-Skript

```
1 probe kernel.function("SyS_open") {
2     printf("Prozess %d oeffnet eine 
          Datei\n",@cast(task_current(),"task_struct")>pid);
3 }
```

In dem gezeigten Beispiel wird der sogenannte Marker auf die Kernfunktion *SyS_open* gesetzt. Diese entspricht dem Systemaufruf zum Öffnen einer Datei auf der Benutzerebene. Bei jedem Eintritt in die genannte Funktion wird der dargestellte Code ausgeführt.

[1] vgl. Linux-Quellen *Documentation/kprobes.txt*

In diesem Fall erfolgt lediglich eine Ausgabe, welcher Prozess gerade in Begriff ist, eine Datei öffnen. Anhand von Zeile 2 in Listing 2.1 werden zwei weitere Funktionen von *SystemTap* deutlich: Es kann Informationen ausgeben. Diese werden auf die Benutzerebene transportiert und dort ausgegeben oder aber in eine Datei umgeleitet. Ferner erlaubt es den Zugriff auf kerninterne Datenstrukturen, wie in dem vorliegenden Fall die Struktur *task_struct*, um so weitere Informationen abgreifen zu können.

Darüber hinaus sichert *SystemTap* die Ausführung eines Skriptes durch verschiedene Mechanismen ab: Die maximale Anzahl an Schritten innerhalb eines Anweisungsblocks, wie er in dem Listing zu sehen ist, ist begrenzt. Damit soll der Einfluss auf das Laufzeitverhalten des untersuchten Systems beherrscht werden. Ferner finden Überprüfungen beim Zugriff auf Felder statt.

Die Ausführung eines Skriptes erfolgt in mehreren Schritten: Zunächst wird evaluiert, ob alle definierten Marker tatsächlich im Kern existieren. Anschließend erfolgt eine Transformation in *C*-Code. Dieser wird durch *SystemTap* mit weiteren Bibliotheksfunktionen angereichert und abschließend als Kernmodul für den Linux-Kern übersetzt. Als letzter Schritt wird das Modul in den Kern geladen und der Benutzerprozess von *SystemTap* startet über einen Kommunikationskanal zum Modul die Ausführung.

2.2 Dtrace

Das Werkzeug *Dtrace* ist ebenfalls zur dynamischen Instrumentierung eines Betriebssystems gedacht. Es wurde für das Betriebssystem *Solaris* entwickelt [8]. Da die Autoren von *SystemTap* durch *Dtrace* beeinflusst wurden, ähnelt der Funktionsumfang von *Dtrace* dem von *SystemTap* sehr. Daher wird auf diesen nicht näher eingegangen. Allerdings verfügt *Dtrace* über eine Funktionalität, die es von *SystemTap* deutlich abgrenzt: Die Quellen der unterschiedlichen Daten sind nicht in dem Werkzeug selbst enthalten. Vielmehr sind mehrere Quellen in einzelnen Anbietern den sogenannten *Providern* zusammengefasst. Ein einzelner Anbieter stellt eine Menge von Markern für einen bestimmten Bereich zur Verfügung. Dabei ist es unerheblich, ob diese Daten aus dem Kern oder aber von der Benutzerebene stammen - beides ist möglich. Darüber hinaus ist die Menge an Anbietern nicht fest, so dass ein Nutzer z. B. für ein bestimmtes Programm einen neuen Anbieter schreiben kann. Diesen kann ein anderer Nutzer nebst des passend instrumentierten Programms verwenden, um das Programm zu analysieren.

2.3 Fay

Das Instrumentierungswerkzeug *Fay* wurde für das *Microsoft*-Betriebssystem *Windows* entwickelt [9]. Es dient ebenfalls dazu sowohl auf der Kern- als auch auf der Benutzerebene eine dynamische Instrumentierung durchzuführen. Im Gegensatz zu den beiden vorherigen Werkzeugen ist dieses mehr darauf ausgelegt, die Daten, die bei einem *Probe-Point* anfallen, abzugreifen und zu verarbeiten. Es ist weniger dafür gedacht, einen Block von Anweisungen auszuführen, wie es z. B. bei *SystemTap* der Fall ist. Vielmehr liegt das

Hauptaugenmerk auf der ausführlichen Verarbeitung der erhobenen Daten. Die Spezifikation der einzelnen Verarbeitungsschritte geschieht in einer Anfrage. Sie wird vor der Ausführung analysiert und ggf. optimiert. Dabei können Operationen umsortiert oder vereint werden. Eine dedizierte Betrachtung der Anfragesprache erfolgt im Rahmen des Entwurfs einer Anfragesprache in Abschnitt 6.2.1.

Neben der lokalen Instrumentierung verfügt *Fay* über die Möglichkeit, Anfragen auf einen Cluster von Rechnern zu verteilen und die Daten anschließend einzusammeln. Die Besonderheit hierbei ist, dass die Verteilung auf die einzelnen Rechner impliziter Bestandteil der Anfragesprache ist.

2.4 PiCO QL

Das Werkzeug *PiCO QL* gibt dem Nutzer relationalen Zugriff auf die Datenstrukturen im Linux-Kern [10]. Nach außen stellt sich der Betriebssystemkern somit wie eine gewöhnliche Datenbank dar, der man mit der *Structured Query Language*[2] Anfragen stellen kann. Im Gegensatz zu den bisher vorgestellten Werkzeugen findet keine dynamische Instrumentierung statt. D.h. dem Nutzer bietet sich nicht die Möglichkeit, Daten an einer bestimmten Stelle im Kern zur Laufzeit zu erheben. Stattdessen kann er über die relationalen Operatoren Daten miteinander verknüpfen und verarbeiten. Generell steht ihm der Sprachumfang von *SQL* zur Verfügung.

Für den Zugriff auf die kerninternen Datenstrukturen müssen diese auf virtuelle Tabellen abgebildet werden, damit eine relationale Verarbeitung möglich ist. Hierzu wird in der Veröffentlichung eine eigene Sprache vorgestellt, mit der ausgedrückt werden kann, wie die Abbildung von Datenstrukturen auf Tabellen erfolgt. Ferner kann die Repräsentation einzelner Attribute als Spalte sowie der dazugehörige Datentyp dargestellt werden. Aus dieser Beschreibung wird Programmcode generiert, der die Abbildung zur Laufzeit ermöglicht. Der Quellcode für das gesamte Werkzeug wird als Modul übersetzt und in den Linux-Kern geladen. Die Kommunikation mit der Benutzerebene erfolgt dabei über eine virtuelle Datei im *proc*-Dateisystem. Die Verarbeitung übernimmt eine *SQLite*-Datenbank, dessen Programmcode ebenfalls in dem Modul enthalten ist [11].

Um eine Anfrage auszuführen, muss der Nutzer diese in die besagte virtuelle Datei schreiben. Zunächst wird die Syntax der Anfrage überprüft und anschließend durch *SQLite* interpretiert. Nun werden alle Prozessorkerne des Rechners angehalten. Die letztendliche Durchführung der Anfrage geschieht auf einem Kern. Das Resultat wird in einer Ergebnismenge festgehalten, die mittels Lesen aus der virtuellen Datei abgefragt werden kann. Solange das Ergebnis der letzten Anfrage nicht abgeholt wurde, kann keine neue Anfrage gestellt werden.

[2] kurz *SQL*

3 Nutzung von Betriebssystemdaten

Nach der Vorstellung der Werkzeuge zur Erhebung von Betriebssystemdaten soll es in diesem Kapitel um die Nutzung eben jener Daten gehen. Dazu werden im Folgenden vier Arbeiten präsentiert, die Betriebssystemdaten nutzen, um das System zur Laufzeit zu justieren und so eine Verbesserung zu erzielen. Die konkrete Verbesserung hängt von dem in der Veröffentlichung bearbeiteten Anwendungsfall ab.

3.1 RadioJockey

Die Veröffentlichung mit dem Titel *RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage* befasst sich mit der Datenübertragung auf mobilen Endgeräten im 3G-Mobilfunknetz[12]. Konkret geht es um die Vermeidung sogenannte *tail-states*. Dies beschreibt die Phase unmittelbar nach einer Datenübertragung, in der die Verbindung zur Basisstation weiterhin aktiv ist. Der Grund dafür ist, dass für das Senden weiterer Datenpakete, die nicht unmittelbar auf das letzte Paket folgen, eine keine neue Verbindung zur Basisstation aufgebaut werden muss. Stattdessen wird die vorhandene Verbindung weiter genutzt. Allerdings bedeutet dies auch, dass die Übertragung eines einzelnen, kleinen Datenpaketes verhältnismäßig viel Energie verbraucht, da der Mehraufwand durch die *tail-states* in jedem Fall anfällt. Es kommt erschwerend hinzu, dass die Dauer bis zum Trennen der Verbindung vom Netzbetreiber vorgegeben wird und nicht veränderbar ist. Um diesem Problem zu begegnen, wurde die sogenannte *Fast Dormancy* entwickelt. Dies erlaubt dem Endgerät, sofern vom Netz unterstützt, die Verbindung zur Basisstation zu einem von ihm bestimmten Zeitpunkt zu trennen. D.h. es kann nach der Übertragung direkt die Verbindung unterbrechen. Muss jedoch nach der Trennung ein weiteres Paket gesendet werden, so ist wieder eine Verbindung mit der Basisstation zu etablieren. Dies ist mit erheblichem Signalisierungsaufwand und dementsprechenden Energieverbrauch verbunden. Daraus ergibt sich die Fragestellung, wann der richtige Zeitpunkt zum Trennen der Verbindung ist. Genau diesem Problem widmet sich die genannte Veröffentlichung: Es wird aus dem gelernten Systemverhalten eine Vorhersage getroffen, ob die Verbindung zu trennen ist. Dazu werden zunächst alle Systemaufrufe, die im Zusammenhang mit dem Übertragen von Datenpaketen stehen, sowie der Netzwerkverkehr selbst aufgezeichnet. Dann werden beide Aufzeichnungen vereint, so dass der zeitliche Zusammenhang zwischen einem bzw. mehreren Systemaufrufen und übertragenen Datenpaketen ersichtlich wird. Es werden alle Systemaufrufe zwischen zwei Datenpaketen zu einem Segment zusammengefasst. Überschreitet die Zeit zwischen dem letzten Paket dem ersten Paket des darauf folgenden Segments einen bestimmten

Grenzwert, wird das letzte Paket mit der Markierung *EOS*[1] versehen. Alle anderen Segmente erhalten die Markierung *ACTIVE*. Mit der so annotierten Aufzeichnung wird ein Entscheidungsbaum trainiert, der es ermöglicht, eine Abfolge von Systemaufrufen mit *ACTIVE* respektive *EOS* zu klassifizieren. Im laufenden System wird nun anhand der Systemaufrufe und des Entscheidungsbaum eine Vorhersage, über den zukünftigen Verlauf der Verbindung getroffen. Wird *EOS* vorgesagt, erfolgt die unmittelbare Trennung der Verbindung.

Die Bewertung erfolgte sowohl mittels Simulationen als auch auf echten Geräten. Für die Simulation wurden die Systemaufrufe und der Netzwerkverkehr von 14 Anwendungen untersucht. Es wurde gezeigt, dass die Quote der falschen Vorhersagen zwischen 5 und 10% lag. Außerdem lag die theoretische Energieersparnis bei 20-30% bei einem Signalisierungsmehraufwand von ca. 5%. Außerdem wurde der Ansatz auf einem echten Endgerät mit *Windows* evaluiert. Dabei wurde gezeigt, dass bis zu 24 % der Energie eingespart werden kann.

3.2 Paketverzögerung

Eine weitere Veröffentlichung befasst sich mit dem Einsparen von Energie bei der Datenübertragung auf mobilen Endgeräten. In diesem Fall geht es ebenfalls um das 3G-Mobilfunknetz. Sie trägt den Titel *Kernel Level Energy-Efficient 3G Background Traffic Shaper for Android Smartphones* [13]. Anders als die vorige Arbeit geht es hierbei um das kontrollierte Versenden von Datenpaketen. Es sollen die Übertragungsmodi in einem 3G-Netz passend ausgenutzt werden.

In einem *UMTS*-Netz wird zwischen zwei Übertragungsmodi differenziert: *FACH* und *DCH*. Der erstgenannte Modus zeichnet sich durch einen geringeren Energieverbrauch aus. Jedoch bietet er nur eine kleine Datenrate. Die Daten werden über einen geteilten Kanal übertragen, den alle Endgeräte in einer Mobilfunkzelle nutzen. Dem gegenüber der *DCH*-Zustand. Hierbei steht dem Endgerät eine hohe Datenrate zum Preis eines hohen Energieverbrauchs zur Verfügung. Dies wird durch das Aufbauen eines dedizierten Kommunikationskanals für das Endgerät bewerkstelligt. Das Hochschalten vom *FACH*-Zustand in den *DCH*-Zustand erfolgt bei Überschreiten eines übertragenen Datenvolumens. Der konkrete Wert hängt dabei von dem Netz ab. Das Endgerät fällt nach einer Zeit der Inaktivität zurück in den *FACH*-Zustand. Nach weiterer Inaktivität wird die Verbindung fast vollständig abgebaut. Auch diese Parameter werden vom Netz vorgegeben.

Im Rahmen dieser Veröffentlichung wird der ausgehende Datenverkehr derart kontrolliert, dass das Endgerät möglichst lange im Zustand *FACH* verweilt. Eine Umschaltung in den *DCH*-Zustand erfolgt erst, wenn sie unabdingbar ist. Dazu wurde der folgende Algorithmus für den Linux-Kern implementiert und auf einem mobilen Endgerät mit *Android* evaluiert: Es werden im Linux-Kern zwei Listen für ausgehende Datenpakete erstellt. Zu versendende Pakete werden gemäß ihrer auf die Listen verteilt. Alle Pa-

[1] Kurzform für *End of session*

kete mit einer Länge kleiner des oben angesprochenen Übertragungsvolumens, das zu einer Umschaltung führt, kommen in die Liste Q_S. Die übrigen Pakete werden in die Liste Q_L eingehängt. Sofern sich das Gerät im Zustand *DCH* befindet, werden die Pakete beider Listen versendet. Im Zustand *FACH* werden nur Pakete aus der Liste Q_S versendet. Zusätzlich wird in diesem Fall vor dem Senden eines Pakets gewartet, damit das übertragene Volumen pro Zeit nicht zu einem Übergang in den *DCH*-Zustand führt. Somit ist sichergestellt, dass das Endgerät möglichst lange in dem Zustand mit niedrigem Energieverbrauch verweilt. Darüber hinaus wird jedes Paket mit einem Zeitstempel versehen, der sicherstellt, dass Datenpakete trotzdem versendet werden, obwohl das Endgerät aktuell keine aktive Verbindung zu einer Basisstation unterhält. Wird das Verfallsdatum eines Netzwerkpakets erreicht, wird es unmittelbar übertragen. Je nach Zustandsübergang wird in Folge dessen der Inhalt der korrespondierenden Liste ebenfalls versendet.

Der beschriebene Algorithmus wurde auf einem *Android*-Endgerät mit emuliertem und realem Netzwerkverkehr evaluiert. Es wurden verschiedene Ablaufzeiten von Paketen betrachtet. Hierbei wurden je nach Netzwerklast Einsparung von bis 60% erzielt. Zusätzlich wurde der Einfluss der Paketverzögerung auf verschiedene Applikationen untersucht.

3.3 Erkennung von abnormalem Prozessverhalten in Android

Eine weitere Möglichkeit zur Nutzung von Betriebssystemdaten ist die Erkennung von abnormalem Programmverhalten. Dieses Ziel hat sich die Veröffentlichung mit dem Titel *Efficient Anomaly Detection System for Mobile Handsets* als Ziel gesetzt [14]. In dieser Arbeit wird das Verhalten eines Programms einerseits über die Abfolge an Systemaufrufen, die es durchführt, definiert. Außerdem werden pro Systemaufruf die Rücksprungadressen auf dem Stapel betrachtet. Es wird also die Historie der Funktionsaufrufe bis zu dem Systemaufruf selbst berücksichtigt. Ein schadhaftes Programm tendiert im Allgemeinen dazu, eigenen Code auszuführen. Dementsprechend ändert sich die Abfolge an Systemaufrufen nebst der Historie. Somit ändert sich das Programmverhalten. Um eben jenes Verhalten von einem normalen Verhalten zu unterscheiden, muss erst ein Modell trainiert werden, das in der Lage ist, normales Programmverhalten zu erkennen. Daher werden in einer Trainingsphase verschiedene Programme ausgeführt und bei jedem Systemaufruf die Ausführung angehalten, um folgende Daten zu ermitteln: die Nummer des Systemaufrufs und die Historie an Funktionsaufrufe vom Programmstapel. In der anschließenden Detektionsphase wird die Ausführung ebenfalls bei jedem Systemaufruf angehalten, die erforderlichen Daten erhoben und das Modell abgefragt. Dieses liefert eine Bewertung für die aktuelle Situation. Je höher die Bewertung ausfällt, desto wahrscheinlicher ist ein abnormales Verhalten. Abschließend wird die Programmausführung fortgesetzt.

3.4 Verbesserungen bei der Prozessorzuteilung in Linux

Die nun betrachtete Veröffentlichung hat als Ziel die Reduktion der Ausführungszeit von Prozessen mittels maschineller Lernverfahren [15]. Zu diesem Zweck wird einem Prozess zusätzliche Prozessorzeit zugewiesen. Die Menge der zusätzlichen Zeit wird durch einen Entscheidungsbaum vorhergesagt. Das Trainieren des Entscheidungsbaums erfolgt auf zwei Wegen: Eine initiale Bewertung eines Programms geschieht aufgrund der Metainformationen, die in der ausführbaren Datei eines Programms enthalten sind - den sogenannten *ELF*-Information[2]. Wird das Programm häufiger ausgeführt, erfolgt eine zusätzliche Bewertung anhand der Laufzeiten vergangener Ausführungen. Die Vorhersage über die zusätzliche Prozessorzeit wird beim Start des Programms vorgenommen. Die Information wird über einen Systemaufruf an das Subsystem, das für die Prozessorzuteilung zuständig ist, weitergegeben. Dafür musste der Linux-Kern um einen zusätzlichen Systemaufruf erweitert werden. Außerdem wurde die Prozessorablaufsteuerung modifiziert, um die zusätzliche Prozessorzeit zu berücksichtigen.

[2]Abkürzung für *Executable and Linkable Format*

4 Ziele

Nach einer ersten thematischen Einordnung dieser Arbeit soll in diesem Kapitel eine tiefgreifendere Betrachtung der Motivation erfolgen. Daraus werden in Abschnitt 4.1 konkrete Anforderungen abgeleitet. Den Abschluss bildet die Vorstellung der Fallbeispiele in Abschnitt 4.2.

4.1 Anforderungen

Im Rahmen der vorliegenden Arbeit soll ein Fundament zur Verwirklichung der Vision des Teilprojektes A1 aus dem Sonderforschungsbereich „Verfügbarkeit von Informationen durch Analyse unter Ressourcenbeschränkung“geschaffen werden. Im Kontext des Teilprojekts A1 sollen ubiquitäre Systeme derart modifiziert werden, dass sie sich selbständig an die jeweilige Nutzung und den Kontext anpassen und sich so verbessern. Eine Verbesserung kann sich in verschiedenen Dingen manifestieren, wie z. B. der Reduktion der Energieaufnahme oder durch eine geringere Antwortzeit - je nach Nutzung und Kontext. Bevor eine Verbesserung erzielt werden kann, müssen zunächst Betriebssystemdaten erhoben und analysiert werden. Anschließend wird daraus mit einem maschinellen Lernverfahren ein initiales Modell trainiert. Entscheidend ist hierbei die ressourcenschonende Erhebung respektive Analyse, da ubiquitäre Systeme naturgemäß über eine geringere Rechenleistung sowie über eine beschränkte Energieversorgung verfügen. Im laufenden Betrieb werden nun ebenfalls Betriebssystemdaten erhoben und erlauben mithilfe des Modells Vorhersagen zu tätigen, wie systeminterne Parameter zu setzen sind, um sich an die aktuelle Nutzung bzw. den Kontext anzupassen. Dieser Zyklus wird fortwährend durchlaufen, so dass eine stetige Systemanpassung erfolgt. Der Ausgangspunkt in beiden Fällen ist die Erhebung und Bereitstellung von Betriebssystemdaten - und der inferierten Daten. Damit eine ressourcenschonende Analyse möglich ist, müssen die Verarbeitungsschritte analysiert und ggf. Optimierungen durchgeführt werden. Dies ist jedoch nur mit einem formalen Modell zur Repräsentation der Daten sowie einer dedizierten Anfragesprache möglich.

Keines der Werkzeuge zur Erhebung von Betriebssystemdaten, wie sie in Kapitel 2 ausführlich vorgestellt wurden, bieten nicht die geforderte Funktionalität. Sie bieten zwar ein analysierbares Datenmodell, dennoch bieten sie keinerlei höhere Abstraktion von der konkreten Implementierung oder eine Möglichkeit zur Erweiterung. Abgesehen von dem Werkzeug *Fay*, das in Kapitel 6 noch weiter behandelt wird, bieten die übrigen Werkzeuge keine dedizierte Anfragesprache, um formell zu beschreiben, welche Daten wie zu verarbeiten sind. Sie sind lediglich primitive Werkzeuge zum Abgreifen von Daten aus einem laufenden System. Allerdings stellen sie Funktionalität zur Verarbeitung der

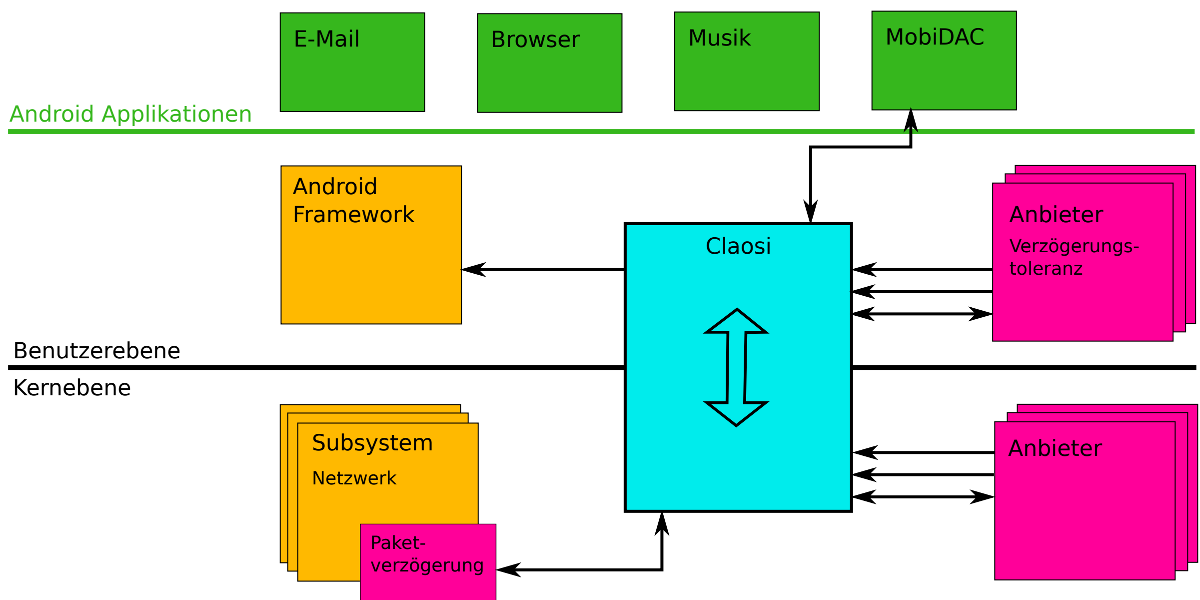


Abbildung 4.1: Struktur des *Claosi* für das Betriebssystem *Android*

gewonnen Daten in Form von Aggregation oder Filterung bereit. Dennoch sind sie als Lösung für die oben genannten Ziele unbrauchbar.

Ferner wurden in Kapitel 3 Arbeiten vorgestellt, die bereits versuchen ein System hinsichtlich unterschiedlicher Kriterien zu optimieren. Sie setzen ebenfalls auf den oben erklären Zyklus aus erheben, vorhersagen und steuern. Jedoch handelt es sich immer um Ad-hoc Implementierungen, die für den jeweiligen Anwendungsfall zugeschnitten sind. Für die Erhebung der Daten wurden meist nur die vorhandenen Schnittstellen des Linux-Kerns benutzt, oder aber ein Subsysteme genauso modifiziert, dass nur die erforderlichen Daten erhoben werden konnten. Zusätzlich wurden zu meist invasive Veränderungen am Betriebssystemkern vorgenommen, um für den jeweiligen Anwendungsfall auf das Betriebssystem einzuwirken. Diese Lösungen sind zu speziell und deshalb nicht nützlich bei der Umsetzung der vormals beschriebenen Ziele. Somit besteht der Bedarf, eine generische Lösung zur Erhebung und Verarbeitung von Daten zu entwerfen und zu implementieren.

Eine generische Lösung soll im Rahmen dieser Arbeit entworfen werden - das sogenannte *Claosi*[1] Werkzeug. Ihre Struktur ist in Abb. 4.1 verdeutlicht. Den Kern bildet eine zentrale Instanz, die sowohl im Betriebssystemkern als auch auf der Benutzerebene existiert. Sie bietet sogenannten *Providern* die Möglichkeit, sich zu registrieren und sowohl Daten bereitzustellen als auch Daten anderer Anbieter abzufragen. Ein Anbieter kann entweder für sich alleine existieren oder ein Bestandteil eines Subsystems sein, wie in der Abbildung zu erkennen ist. Die Aufgabe besteht darin, zwischen den einzelnen Anbietern zu vermitteln. Sie hat Sorge zu tragen, dass die geforderten Daten von Anbieter A zu Anbieter B zugestellt werden. Es spielt hierbei keine Rolle, auf welche Ebene die beiden Anbieter angesiedelt sind. Im Zweifel müssen die Daten von einer auf die andere

[1] Cross-layer availability of operating system information

Ebene transportiert werden. Ein mögliches Einsatzgebiet ist das mobile Betriebssystem *Android*. Daher soll, wie in Abb. 4.1 dargestellt, das Zusammenspiel mit dem *Android Framework* und mit *Android* Applikationen möglich sein.

Bei dem Entwurf sowie der Implementierung sollen die folgenden vier Bereiche abgedeckt werden: Zunächst ist eine Modellierung der Daten erforderlich. Dazu muss ein Metamodell erstellt werden, anhand dessen konkrete Datenmodelle instanziiert werden können. Das Metamodell soll verschiedene Datenquellen berücksichtigen. Es soll außerdem ersichtlich sein, welchen Rückgabetypen die Quellen besitzen. Daher muss das Metamodell zum einen die Menge der intrinsischen Datentypen umfassen, zum anderen muss die Definition von eigenen, komplexen Datentypen ermöglicht werden. Zudem sollen Elemente eines Datenmodells zu semantischen Gruppen zusammengefasst werden können. Ein Vorteil der systematischen Modellierung von Daten besteht in der zukünftigen Annotation einzelner Quellen mit dem dazugehörigen Energieverbrauch. Hiermit ist eine energiegewahre Erhebung der Daten realisierbar.

Darauf aufbauend soll eine datenstrom-getriebene Anfragesprache entworfen und partiell Implementiert werden. Sie stellt eine formale Beschreibung der Erzeugung sowie der Verarbeitung der Daten dar. Sie soll den Fluss der Daten vom Ursprung über die Verarbeitung bis hin zur Zustellung an den Anfragenden abbilden. Mit ihrer Hilfe erhalten die in einem Datenmodell beschriebenen Daten eine Semantik. Ferner ermöglicht die formelle Beschreibung einer Anfrage eine maschinelle Interpretation. Auf dieser Basis können Analysen ausgeführt werden, um Optimierungen, wie das Vereinen von Anfragen oder Umsortieren einzelner Bestandteile, überhaupt zu ermöglichen. Zusätzlich kann hiermit der Schutz sensibler Daten realisiert werden. Als ein Beispiel ist das Erzeugen von Aliassen beim Zugriff auf eine bestimmte Quelle zu nennen.

Nachdem zuerst zwei formale Anforderungen dargestellt wurden, soll der Blick nun auf zwei technische Gesichtspunkte gerichtet werden: Es muss ein systemweites, konsistentes Datenmodell geben, da die Bereitstellung der Daten systemweit erfolgen soll. Ferner muss das Werkzeug Sorge tragen, dass die geforderten Daten an die entsprechende Stelle transportiert werden, ggf. ist hierfür ein ebenenübergreifender Transport nötig. Beide Punkte erfordern eine Kommunikation zwischen der Komponente im Betriebssystemkern und der Komponente auf der Benutzerebene. Darüber hinaus muss es eine Programmierschittstelle geben, damit, wie in Abb. 4.1 dargestellt, weitere Anbieter nachgeladen werden können. Diese soll ebenfalls die Möglichkeit erhalten, Anfragen zu registrieren.

Die genaue Betrachtung sowie Umsetzung der vier Gesichtspunkte ist Bestandteil dieser Arbeit. Dabei widmet sich Kapitel 5 der Modellierung der Daten. Die konkrete Modellierung wird anhand von Beispielen verdeutlicht. Die Vorstellung der Fallbeispiele erfolgt im folgenden Abschnitt. In Kapitel 6 geht auf die Anforderungen einer Anfragesprache nebst dessen Realisierung ein. Die zuletzt dargelegten technischen Aspekte sind Inhalt von Kapitel 7.

4.2 Vorstellung der Fallbeispiele

In den folgenden Kapiteln werden Lösungen für die im vorigen Abschnitt aufgezeigten Problemstellungen vorgestellt. Zum besseren Verständnis sollen anhand von zwei Fallbeispielen die entwickelten Lösungen verdeutlicht werden. Sie sollen aufzeigen, wie Vielseitig die gewählte Lösung ist. Dazu werden in den betreffenden Kapiteln die dort vorgestellten Konzepte auf die Fallbeispiele angewandt. Dabei ist ein Fallbeispiel so gewählt, dass eine gewisse Überdeckung mit einem Szenario aus der in Kapitel 3 präsentierten verwandten Arbeiten besteht. Dies soll zeigen, dass mit der entworfenen Lösung der Anwendungsfall aus dieser Arbeit ohne weiteres umsetzbar ist. Wie später noch dargestellt wird, vermag diese Lösung weitaus mehr zu leisten. Allerdings werden die Fallbespiele nicht implementiert. Vielmehr dienen sie als Beleg für die erarbeiteten Konzepte. Lediglich Teile der Fallbeispiele aus Abschnitt 4.2.1 und Abschnitt 4.2.2 wurden für die Evaluation implementiert. Genaueres dazu wird in Kapitel 8 erläutert. Im Folgenden werden nun die zwei Fallbeispiele vorgestellt.

4.2.1 Paketverzögerung

Das erste Fallbeispiel - genannt „Paketverzögerung“- hat die gleiche Intention wie die in Abschnitt 3.2 vorgestellte Arbeit. In 3G-Mobilfunknetzen soll die Übertragung von einzelnen Datenpaketen vermieden werden. Die Effizienz bei der Übertragung eines einzelnen Datenpakets ist vergleichsweise schlecht, da pro Übertragung ein großer Signalisierungsaufwand erforderlich ist. Daher sollen mehrere Netzwerkpakte zusammengefasst und als Bündel gesendet werden. In der besagten Arbeit werden die Pakete je nach Größe verzögert. Diese Vorgehensweise wird hier nicht verwendet. Stattdessen sollen alle Netzwerkpakete, die von dem System versendet werden, um eine bestimmte Zeit verzögert werden. Die systemweite Verzögerung ist das Minimum über die aktuelle, maximale Verzögerung aller Applikationen. Die applikations-spezifische Verzögerung wird durch ein gelerntes Modell vorgesagt, das beim Einhängen eines Netzwerkpaketes in die geräte-spezifische Warteschlange abgefragt wird. Dazu muss für das zusendende Paket der dazugehörige Prozess ermittelt werden. Ferner muss zu jedem Prozess die korrespondierende Applikation ermittelt werden, da diese Vorgehensweise auch auf dem mobilen Betriebssystem *Android* eingesetzt werden soll. Zur Erzeugung des erwähnte Modells muss vorab der ein- und ausgehende Netzwerkverkehr mitgeschnitten und zu jedem Pakete der Prozess bzw. die Applikation bestimmt werden. Mit diesen Daten wird durch ein maschinelles Lernverfahren ein Modell trainiert.

4.2.2 Automatische Umschaltung zwischen 2G/3G

Im vorangegangen Abschnitt wie auch in Abschnitt 3.2 wurde bereits deutlich, dass die Übertragung von einer geringen Datenmenge über das 3G-Mobilfunknetz teuer hinsichtlich des Energieverbrauchs ist. Das Fallbeispiel „Automatische Umschaltung zwischen 2G/3G“widmet sich ebenfalls dieser Problematik - allerdings von einer anderen Seite.

Für die regelmäßige Übertragung von kleinen Datenmengen sowie von kurzzeitig größeren Datenmengen empfiehlt es sich, das 2G-Mobilfunknetz zu verwenden, da dieses pro Übertragung günstiger ist. Bei deutlich größeren Datenmengen ist es hingegen besser, auf das schnellere 3G-Mobilfunknetz zu wechseln. Das Einsatzgebiet dieses Fallbeispiels sind ebenfalls mobile Geräte mit dem Betriebssystem *Android*. Die angesprochene Unterscheidung ist der Kern des Fallbeispiels: Je nach erwartetem Netzwerkverkehr soll das Mobilfunknetz gewechselt werden. Dazu wird aus dem ein- und ausgehenden Netzwerkverkehr pro Applikation ein Modell trainiert, dass den zu erwartenden Verkehr der jeweiligen Applikation für die nächste Zeiteinheit voraussagt. Liegt die Summe der Datenmenge aller Applikationen über einem Schwellwert, wird in das 3G-Mobilfunknetz gewechselt. Unterschreitet die Summe einen bestimmten Schwellwert, erfolgt ein Wechsel zurück ins 2G-Netz. Zusätzlich wird der Zustand des Bildschirms beobachtet. Ist der Bildschirm für eine längere Zeit ausgeschaltet, erfolgt ein Wechsel in das 2G-Netz.

5 Datenmodell

In Kapitel 4 sind die Anforderungen an *Claosi* herausgearbeitet worden. Dazu gehört unter anderem, der Entwurf einer Anfragesprache, mit der Anweisungen gegeben werden können. Für den Entwurf sowie die spätere Implementierung einer solchen Anfragesprache ist es erforderlich, dass eine formale Beschreibung der Daten existiert, auf denen später Operationen ausgeführt werden sollen. Eine solche Beschreibung wird als Datenmodell bezeichnet. Ferner ist noch ein Metamodell vonnöten, das die Syntax definiert, nach der eine Instanz des Datenmodells erstellt werden kann. Ebendiese Punkte sollen in diesem Kapitel vorgestellt werden.

In Abschnitt 5.1 soll zunächst das Metamodell vorgestellt werden, sowie eine kurze Abgrenzung zu existierenden Arbeiten gemacht werden. Anschließend werden anhand der Fallbeispiele aus Abschnitt 4.2 in Abschnitt 5.2 drei Instanzen eines Datenmodells präsentiert.

5.1 Metamodell zur Datenbeschreibung

Zur Beschreibung ist, wie bereits erwähnt, ein Metamodell erforderlich. Dieses legt die Syntax fest, nach der Instanzen erzeugt werden dürfen. Für *Claosi* wird das in Listing 5.1 vorgestellte Metamodell verwendet. Es basiert auf dem Modell von Jochen Streicher [16].

Listing 5.1: Metamodell zur Beschreibung eines Datenmodells

```
1 model ::= namespace
2 namespace ::= namespace | source | event | object | complex
3 type ::= str | int | float | byte | array | complex
4 identifier ::= str | int | float | byte
5 complex ::= type+
6 source ::= SOURCE name : type
7 event ::= EVENT name : type
8 object[identifier]::= (namespace | source | object)*
```

In dem Modell sind folgende Schlüsselwörter und Datentypen vorgesehen:

- ***model*:** Das Schlüsselwort *model* repräsentiert die Wurzel eines jeden Datenmodells. Es darf genau einmal in einem Datenmodell vorkommen.

- ***namespace*:** In einem Namensraum oder auch *namespace* werden alle Entitäten zusammengefasst, die zu einer logischen Gruppe gehören. Namensräume dürfen beliebig verschachtelt werden. Ferner werden alle weiteren Elemente eines Datenmodells unterhalb eines Namensraumes angesiedelt.

- ***type*:** Unter dem Begriff *type* werden alle elementaren und komplexen Datentypen eines Datenmodells subsummiert. Als elementare Datentypen werden *str*, *int*, und *byte*. Der Datentyp *array* bezeichnet lediglich eine mehrelementige Menge und bedingt daher die Angabe eines konkreten Typs. Zusätzlich können komplexe Datentypen (oder auch *complex*) definiert werden, die aus mindestens einem elementaren oder komplexen Datentyp bestehen.

- ***complex*:** Es ist möglich, innerhalb eines Datenmodells eigene, komplexe Datentypen zu definieren. Diese müssen mindestens aus einem Element bestehen. Ein komplexer Datentyp kann beispielsweise als Rückgabetyp für ein Ereignis verwendet werden.

- ***source*:** Eine der drei Typen von Datenquellen innerhalb eines Datenmodells ist eine sogenannte *source* - eine einfache Datenquelle. Sie hat einen über die Zeit veränderlichen Wert. Ferner kann sie zu jedem Zeitpunkt und in beliebigem Intervall abgefragt werden. Somit liegt es beim Aufrufer, wie oft Daten erzeugt werden. Der Rückgabewert kann sowohl ein elementarer als auch ein komplexer Datentyp sein.

- ***event*:** Im Gegensatz zu einer einfachen Datenquelle wird bei einem Ereignis durch den dahinter liegenden Provider festgelegt, wann Daten erzeugt werden. Ein Ereignis liefert zusätzlich Kontextinformationen. Sie werden über den Rückgabetyp im Datenmodell festgehalten. Eine konkrete Instanz des Rückgabetyps steht nur zum Zeitpunkt des Eintreten des Ereignisses zur Verfügung.

- ***object*:** Ein Objekt bildet Entitäten mit einer definierten Lebenszeit ab. Das heißt, es gibt ein Ereignis im System, dass eine Instanz erzeugt bzw. diese zerstört. Ein Objekt besitzt einen sogenannten *identifier*, über den eine Instanz eindeutig identifizierbar ist. Hierbei muss es sich um einen elementaren Datentyp handeln. Objekte können beliebig verschachtelt werden und können auch Container für ein Ereignis oder eine einfache Quelle sein. Letzteres ist aber nicht zwingend erforderlich. Ein Objekt kann auch ohne Kindknoten existieren. Wird ein Objekt in einer Anfrage direkt referenziert, so wird nur der *Identifier* der jeweiligen Instanz in den Datenstrom geschrieben.

Mittels des oben beschriebenen Metamodells lassen sich Datenmodelle für verschiedene Szenarien erzeugen, die Vorteile gegenüber den Datenmodellen bieten, wie sie z. B. bei Werkzeugen wie SystemTap oder Fay verwendet werden [6, 9]. Wie bereits in Kapitel 4 erläutert wurde, liegt den erhobenen Daten ein implizites Datenmodell zu Grunde. Einerseits gibt der Kontext, aus dem die Daten stammen, Auskunft über deren Semantik sowie deren Struktur. Der Kontext im Zusammenhang mit Kerneldaten kann beispielsweise das Subsystem sein, wo die Daten erhoben werden. Andererseits gibt die verwendete Programmiersprache sowie der Zweck der Variablen den Rahmen für die Struktur der Daten vor.
Die Kernfunktion *netif_receive_skb(struct sk_buff *skb)*[1] beispielsweise erhält als Pa-

[1]Siehe Linux-Quellen *net/core/dev.c* sowie [17]

rameter einen Zeiger auf ein *struct sk_buff*[2], was die Datenstruktur des Linux-Kerns für ein empfangenes Netzwerkpaket bildet. Der Zugriff auf die Headerinformationen eines Netzwerkpaketes ist über entsprechende Attribute der Datenstruktur bereits möglich. Allerdings enthält die Struktur noch weitere Daten, die vom Betriebssystem benötigt werden, in diesem Szenario jedoch überflüssig sind. Weiterführende Kontextinformationen wie z. B. die ID des Prozesses, der dieses Paket zugestellt bekommt, sind hingegen nicht ohne Weiteres ersichtlich. Ferner ist dies nur mit entsprechendem Wissen über den Linux-Kern möglich. In Abschnitt 5.2 werden daher unterschiedliche Datenmodelle vorgestellt, in denen durch passende Verknüpfungen der einzelnen Entitäten z. B. der Bezug zwischen einem empfangenen Netzwerkpaket und dem dazugehörigen Prozess hergestellt werden kann.

Des Weiteren erlaubt das hier vorgestellte Metamodell die Abstraktion von der darunterliegenden Implementierung. In dem bereits betrachteten Szenario des Empfangen von Netzwerkpaketen ist erforderlich, den Aufruf der passenden Kernfunktionen abzufangen. Die Funktion *netif_receive_skb(struct sk_buff *skb)* wurde bereits vorgestellt. Jedoch muss im Liunxkern der Aufruf zwei weiterer Funktionen abgefangen werden. Hierzu gehören *netif_rx(struct sk_buff *skb)* und *napi_gro_receive(struct sk_buff *skb)*. Für die Erfassung von eigehenden Paketen ist diese Information jedoch unerheblich. Über das Metamodell lässt sich das beispielsweise zu einem *event* namens *onRX* zusammenfassen. Genaueres hierzu wird ebenfalls im Abschnitt 5.2 vorgestellt.

5.2 Fallbeispiele

Nachdem im Abschnitt 5.1 bereits die Syntax, nach der ein Datenmodell erzeugt werden kann, vorgestellt worden ist, sollen in diesem Abschnitt konkrete Datenmodelle anhand der Fallbeispiele aus Abschnitt 4.2 vorgestellt werden.

5.2.1 Paketverzögerung

Für das Fallbeispiel „Paketverzögerung“erfolgt zunächst die Aufzeichnung von eingehendem und ausgehendem Netzwerkverkehr sowie der Zuordnung zu den jeweiligen Prozessen respektive Applikationen im Vordergrund. Zu diesem Zweck sind drei Namensräume erforderlich: *net*, *process* und *ui*.

Im Namensraum *net* befindet sich das Objekt *device*, das die Gruppe der Netzwerkgeräte repräsentiert. Sie werden - wie in Unix üblich - über eine Zeichenkette identifiziert. In einem Computer mit zwei Netzwerkkarte beispielsweise, gäbe es zwei Instanzen von *device*: einmal *eth0* und *eth1*. Ein *device* verfügt über drei Ereignisse: *onEnqueue*, *onRX* und *onTX*. Die beiden letztgenannten werden beim Senden respektive Empfangen von Netzwerkpakete ausgelöst. Das Ereignis *onEnqueue* wird beim Einhängen eines zu sendenden Pakets in die Warteschlange des betreffenden Netzwerkgerätes ausgelöst. Als Kontextinformation wird eine Instanz des komplexen Datentyps *type packetType* beigefügt - vgl. Listing 5.2, Zeile 10-19. In ihm werden die Typen des *MAC*-, Netzwerk-

[2]siehe Linux-Quellen *include/linux/skbuff.h*

und Transportprotokolls in Form eines Bytes festgehalten. Außerdem befindet sich der Header der jeweiligen Netzwerkschicht als Byte-Array in dem Datentyp. Die Gesamtlänge des Netzwerkpakets wird im Feld *dataLength* vermerkt. Abschließend enthält jede Instanz ein Referenz auf das Socket, dem es zugestellt wird - siehe *socket*. Zusätzlich besitzt jede Instanz eines *device* zwei Quellen, die die Summe alle bisher empfangenen bzw. gesendeten Bytes zurückliefern - vgl. Listing 5.2, Zeile 4-5. Als drittes und letztes Element enthält der Namensraum *net* das Objekt *socket*. Dies bildet den Kommunikationsendpunkt des Betriebssystems für eine Verbindung ab. Es bietet über zwei Quellen - *type* und *flags* - Zugriff auf die Attribute des Sockets. Abschließend enthält der Namensraum *net* eine einfache Datenquelle *delayTolerance* - vgl. Listing 5.2, Zeile 24. Sie liefert in Form eines Integers die maximale Verzögerung pro Paket, die das Betriebssystem derzeit verwenden soll. Die Zeitangabe erfolgt in Sekunden.

Der Namensraum *process* dient der Abbildung von Prozessen im Datenmodel. Für das vorliegende Szenario enthält er lediglich ein Element: Das Objekt *process* repräsentiert ein Prozess im Betriebssystem. Die eindeutige ID des Objektes ist daher vom Typ *int* und entspricht der Prozess-ID. Pro Instanz von *process* stehen zwei *sources* zur Verfügung. Sie liefern die Gesamtzeit, die ein Prozess im Benutzermodus (*utime*) bzw. Systemmodus (*stime*) gerechnet hat, in Mikrosekunden zurück. Ferner besitzt ein Prozess eine *source* zur Abfrage aller durch eine bestimmte Instanz geöffneten Sockets. Sie liefert eine Liste bestehend aus den IDs des Objekts *net.socket* aller durch eine bestimmte Instanz von *process* geöffneten Sockets zurück.

Abschließend ist noch auf den Namensraum *ui* einzugehen. Dieser repräsentiert die graphische Schnittstelle. Zu diesem Zweck enthält der Namensraum das Objekt *app*. Es repräsentiert eine Anwendung, die aus einer Gruppe von Prozessen besteht. Identifiziert wird eine Instanz von *app* eindeutig über eine Zeichenkette mit dem Name der Applikation. Zur Abfrage der Menge aller Prozesse einer Applikation bietet das Objekt *app* eine *source processes* an. Hierüber können alle IDs der zugehörigen Prozesse ausgelesen werden. Bei dem Rückgabetyp handelt es sich um eine Referenz auf eine Instanz von *process.process*.

Listing 5.2: Datenmodell für das Fallbeispiel „Paketverzögerung“

```
1  model {
2      namespace net {
3          object device[str] {
4              source txBytes : int;
5              source rxBytes : int;
6              event onRX() : packetType;
7              event onTX() : packetType;
8              event onEnqueue() : packetType;
9          }
10         type packetType {
11             byte[] macHdr;
12             byte macProtocol;
13             byte[] networkHdr;
14             byte networkProtocol;
15             byte[] transportHdr;
16             byte transportProcotol;
```

```
17              int dataLength;
18              process.process.sockets *socket;
19          }
20          object socket[int] {
21              source int type;
22              source int flags;
23          }
24          source delayTolerance : int;
25      }
26      namespace process {
27          object process[int] {
28              source utime : int;
29              source stime : int;
30              source
31          }
32      }
33      namespace ui {
34          source app *foregroundApp;
35          object app[string] {
36              source process *processes[];
37          }
38      }
39  }
```

5.2.2 Automatische Umschaltung 2G/3g

Für das Fallbeispiel „Automatische Umschaltung 2G/3G“ist ebenfalls erforderlich, alle eingehenden und ausgehenden Netzwerkpakete nebst der dazugehörigen Prozesse respektive Applikationen aufzuzeichnen. Daher enthält dieses Datenmodell auch einen Namensraum *net* sowie *process*. Diese sind identisch mit denen aus dem Listing 5.2, abgesehen von dem Ereignis onEnqueue und der einfachen Datenquelle *delayTolerance*. Mit den daraus gewonnenen Daten kann der erzeugte Datenverkehr mit dem Prozess sowie der Applikation annotiert werden. Unter anderem anhand dieser Information soll in diesem Szenario die Umschaltung von 2G auf der 3G bzw. umgekehrt ausgelöst werden. Applikationen mit einem hohen Datenaufkommen sollen eine 3G-Verbindung erhalten, um schnell Daten zu übertragen. Anwendungen mit einem geringen Verbrauch hingegen genügt eine 2G-Verbindung. Diese benötigt weniger Energie.

Ein weiterer Indikator für eine Umschaltung zwischen den Netztechnologien ist der Zustand des Displays eines Smartphones. Ist es eingeschaltet, ist die Wahrscheinlichkeit wesentlich höher, dass eine Applikation ein größeres Datenaufkommen erzeugt. Wohingegen bei ausgeschaltetem Display vorwiegend Hintergrundapplikationen aktiv sind, denen eine langsamere 2G-Verbindung genügt. Deshalb wird der Namensraum *ui* um den Namensraum *screen* erweitert. Er enthält lediglich das Ereignis *screenState*. Dies wird ausgelöst, sobald sich der Zustand des Displays ändert. Das Ereignis liefert den Wert 1 für „eingeschaltet“und 0 für „ausgeschaltet“in Form eines Integers zurück.

Listing 5.3: Datenmodell für das Fallbeispiel „Automatische Umschaltung zwischen 2G und 3G“

```
1  model {
2      namespace net {
3          object device[str] {
4              source txBytes : int;
5              source rxBytes : int;
6              event onRX() : packetType;
7              event onTX() : packetType;
8          }
9          type packetType {
10             byte[] macHdr;
11             byte macProtocol;
12             byte[] networkHdr;
13             byte networkProtocol;
14             byte[] transportHdr;
15             byte transportProcotol;
16             int dataLength;
17             process.process.sockets *socket;
18         }
19         object socket[int] {
20             source int type;
21             source int flags;
22         }
23     }
24     namespace process {
25         object process[int] {
26             source utime : int;
27             source stime : int;
28             source net.socket *sockets;
29         }
30     }
31     namespace ui {
32         source app *foregroundApp;
33         object app[string] {
34             source process *processes[];
35         }
36         namespace screen {
37             event screenState : int;
38         }
39     }
40 }
```

5.3 Fazit

Im Abschnitt 5.1 ist dargestellt worden, wie mithilfe des in Listing 5.1 beschriebenen Metamodells ein Modell zur Beschreibung von Daten und deren Beziehungen erstellt werden. Wie bereits erläutert worden ist, gibt es in Linux verschiedene Punkte, um eingehende Netzwerkpakete abzufangen. Für eine Datenerhebung ist dies jedoch nicht von Belang. Daher wird beispielsweise in Listing 5.2 von dieser Tatsache abstrahiert, in dem es lediglich ein Ereignis *onRx()* existiert. Dies subsummiert alle beschriebenen

Funktionen in einem Ereignis.

Ferner erlaubt die Darstellung über ein explizites Datenmodell das Auslassen von nachrangigen Informationen. Die kernelinterne Repräsentation eines Netzwerkpakets *struct sk_buff* verfügt über eine Vielzahl von Feldern. In den vorgestellten Modellen wird jedoch nur ein Bruchteil eben jener verwendet. Es werden lediglich die Informationen für einzelnen Netzwerkschichten sowie das dazugehörige Socket weitergegeben. Da die Beschreibung des Datentyps ebenfalls über das Datenmodell erfolgt, ist es allerdings auf einfachem Wege möglich, diesen zu erweitern.

Im Anschluß an dieses Kapitel sollen nun in Kapitel 6 eine Anfragesprache vorgestellt werden, mit der auf eben diesem Datenmodellen operiert werden kann. Außerdem erfolgt ein Vergleich bzw. Abgrenzung gegenüber bereits existierenden Anfragesprachen.

6 Anfragesprache

Im Rahmen dieser Arbeit soll - wie bereits in Abschnitt 4.1 ausführlich erläutert worden ist - eine Anfragesprache zur Interaktion mit dem Datenmodell entworfen werden. Hierzu ist es zunächst erforderlich, den Blick auf die hinter der Sprache stehende Datenhaltung zu richten. In einem ersten, naiven Ansatz lassen sich beispielsweise die Beziehung zwischen den Elementen *process* und *socket* aus dem Datenmodell zum Fallbeispiel „Paketverzögerung“als Relation mittels eines *Entity-Relationship*-Diagramm modellieren. Eine Instanz des Objektes *process* kann n Verweise auf Instanzen des Objekts *socket* besitzen. Diese Relation ist ohne Weiteres z. B. mit einer *MySQL*-Datenbank realisierbar. Somit ist die Anfragesprache mit *MySQL* implizit vorgegeben. Allerdings setzt dies drei Bedingungen voraus:

- Der Datenbestand muss zum Zeitpunkt des Zugriffs fix sein. D.h. es existiert ein definierter Anfang und ein definiertes Ende.
- Die Zugriffe müssen über eine Schnittstelle erfolgen und synchronisiert werden, so dass die Daten immer kohärent sind.

Im Falle des Erzeugen von Prozessen beispielsweise fällt es sich jedoch so, dass der Datenbestand - hier die Menge der existierenden Prozesse - ohne Kenntnis der Datenhaltungsschicht modifiziert wird. Die Änderung erfolgt nicht durch die Schnittstelle zur Datenhaltung. Ferner ist aufgrund des hochfrequenten Charakters des Ereignisses nicht sichergestellt, dass der Datenbestand fix ist. Vielmehr erzeugt das Ereignis „Erzeugen von Prozessen“einen kontinuierlichen Datenstrom. Eine Interaktion findet immer nur mit einem Ausschnitt statt. Ein Datenstrom entspricht auch der in Kapitel 4 dargelegten Anforderungen.

Ein erster Ansatz zur Lösung der geschilderten Problematik stellt *PiCO QL* dar [10]. Hierbei werden, wie eingangs in Kapitel 2 bereits ausgeführt wurde, die Datenstrukturen des Linux-Kernels auf virtuelle Tabellen abgebildet. Ein Linux-Kernelmodul wird hierzu in den Betriebssystemkern geladen und erlaubt über eine virtuelle Datei im *proc*-Dateisystem die Interaktion mit der Datenbank. Über die Sprache *MySQL* ist es dem Nutzer möglich, Anfragen zu stellen. Zur Ausführung eine Anfrage wird der komplette Systemzustand eingefroren, der Zustand der Datenstruktur in virtuelle Tabellen überführt und die Anfrage auf ebendiesen ausgeführt. Weitere technische Informationen zu diesem Aspekt von *PiCO QL* finden sich in Kapitel 7. Durch das Einfrieren des Systemzustandes wird sichergestellt, dass die Datenbank auf einem festen Datenbestand operiert. Außerdem wird hierdurch die Kohärenz der Daten gewährleistet. Mittels periodischem Senden der immer gleichen Anfrage wäre es möglich, einen Datenstrom zu emulieren. Allerdings ist dies, aufgrund der erläuterten Schritte bei der Verarbeitung

eine Anfrage, sehr zeitaufwendig. Darüber hinaus ist die Wahrscheinlichkeit, Elemente - z. B. Prozesse kurzer Lebensdauer - zu verpassen, recht hoch. Eine kontinuierliche Datenquelle, wie sie in den verschiedenen Datenmodellen in Abschnitt 5.2 vorkommt, wäre somit zu realisieren. Jedoch sind Ereignisquellen und Objektereignisquellen nicht Bestandteil des Konzepts. Um sie mit *PiCO QL* nachzubilden, ist es erforderlich zu jedem im Datenmodell vorgesehenen Ereignis, die anfallenden Daten im Vorhinein zu erheben und zwischen zu speichern. Dies überschreitet je nach der Frequenz der Ereignisse nach absehbarer Zeit die Speicherkapazität des Betriebssystems. Abschließend ist festzuhalten, dass sowohl der Ansatz als auch die Anfragesprache von *PiCO QL* für den vorgesehenen Zweck ungeeignet sind.

Daher sollen Abschnitt 6.1 Anforderungen vorgestellt werden, nach denen ein adäquate Anfragesprache zu entwerfen ist. Eine passende Sprache wird in Abschnitt 6.2 vorgestellt. Den Abschluß bildet der Abschnitt 6.4 mit der Vorstellung der Anfragen für die jeweiligen Fallbeispiele.

6.1 Anforderungen

Die Datenmodelle, wie sie in Abschnitt 5.2 dargestellt worden sind, erlauben lediglich die Spezifikation der Daten sowie deren Zusammenhänge. Zur Interaktion mit diesen ist eine Sprache erforderlich, um auszudrücken, welches Element als Datenquelle dient und welche Art von Operationen durchzuführen sind - die sogenannte Anfragesprache. Bei der Spezifikation der Datenquelle muss ausgedrückt werden, welches Element des Datenmodells als Quelle dienen soll. Ebenso müssen die Charakteristika der verschiedenen Datenquellen berücksichtigt werden. Bei einem Objektereignis ist anzugeben, ob auf die Erzeugung oder das Zerstören reagiert werden soll, oder ob die Menge aller Objektinstanzen von Interesse sind. Da eine einfache Datenquelle - oder auch *source* - aktiv abgefragt werden muss, ist die Angabe einer Periode bzw. die Frequenz vonnöten. Für eine Ereignisquelle sind keine weiteren Angaben erforderlich.

Zur adäquaten Umsetzung der Fallbeispiele aus Abschnitt 4.2 sind offenkundig Operationen auf den erhobenen Daten erforderlich. Für die dargestellten Beispiele sind folgende Typen von Aktionen nötig:

- die Verknüpfung von Daten,
- die Einschränkung der Menge an Attributen und
- die Selektion von Daten.

Die Operationen werden auf einer Menge von Elementen durchgeführt, wobei jedes Element verschiedene Attribute besitzt. Die Verknüpfung dient der Anreicherung mit weiteren Daten. Im Falle des Fallbeispiels „Paketverzögerung“bedeutet dies, die ein- und ausgehenden Netzwerkpakete mit dem dazugehörigen Prozess zu verknüpfen. Bei der Einschränkung sollen die überflüssigen Attribute herausgefiltert werden. Als Beispiel ist hier ebenfalls die Paketverzögerung zu nennen. Hier befindet sich nach der Verknüpfung

des Netzwerkpakets mit dem Prozess und anschließend mit der Applikation noch die Prozess-ID im Datenstrom. Da hier aber nur die Applikation von Belang ist, kann die Prozess-ID entfernt werden. Die Selektion dient der Auswahl einzelner Elemente aus einer Ergebnismenge. Zu diesem Zweck müssen Prädikate angegeben werden, anhand derer geprüft wird, ob ein Element in der Menge verbleibt.

6.2 Existierende Sprachen

Bevor im folgenden Abschnitt die entworfene Sprache vorgestellt wird, soll in diesem Abschnitt der Blick auf verwandte Arbeiten gerichtet werden. Hierbei werden die Sprachen der Instrumentierungswerkzeuge *SystemTap* und *Fay* sowie die Datenbank für Sensornetze *TinyDB* hinsichtlich der oben genannten Kriterien untersucht. Abschließend wird die Datenstromsprache *Aurora* vorgestellt. Das Werkzeug *PiCO QL* verwendet als Sprache gewöhnliches *SQL*. Da die Sprache von *TinyDB* auf *SQL* aufbaut, wird *PicoQL* in diesem Kapitel nicht gesondert betrachtet.

6.2.1 Fay

Die grundlegende Funktionalität von *Fay* wurde vorab in Abschnitt 2.3 erläutert. Daher soll in diesem Abschnitt das Augenmerk auf die verwendete Anfragesprache gerichtet werden. Im Rahmen der Arbeiten für *Fay* ist eine eigene Anfragesprache entworfen worden - die sogenannte *FayLINQ*. Sie beruht auf der sogenannten *Language Integrated Query*[1] des *.NET* Frameworks nebst dessen Erweiterung *DryadLINQ*. Bei *LINQ* handelt es sich um eine Anfragesprache, die in die Programmiersprache - in diesem Fall *C#* - integriert ist.

Listing 6.1: Beispiel für ein *Language Integrated Query* (entnommen aus [18])

```
1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4
5  class app {
6    static void Main() {
7      string[] names = { "Burke", "Connor", "Frank",
8                         "Everett", "Albert", "George",
9                         "Harris", "David" };
10
11     IEnumerable<string> query = from s in names
12                                 where s.Length == 5
13                                 orderby s
14                                 select s.ToUpper();
15
16     foreach (string item in query)
17       Console.WriteLine(item);
18   }
```

[1] Kurzform: *LINQ*

```
19 }
```

In Listing 6.1 ist ein Beispiel für eine Abfrage mit *LINQ* dargestellt. Als Datenquelle dient in diesem Fall ein Feld aus Zeichenketten - vgl. Listing 6.1, Zeile 8-10. Die darauf ausgeführte Anfrage wird hierbei nicht, wie z. B. aus *SQL*[2] bekannt, in Form einer Zeichenkette spezifiziert. Vielmehr sind die dafür nötigen Schlüsselwörter Bestandteile der Programmiersprache - vgl. Listing 6.1, Zeile 12-15. Die Anfrage wird implizit bei der Zuweisung zu der Ergebnisvariablen *query* ausgewertet.

Die Erweiterung *DryadLINQ* ermöglicht es, eine Anfrage automatisch, parallel auf mehreren Rechnern auszuführen [19]. Diese beiden Elemente bilden die Grundlage für *FayLINQ*. Für die Erhebung von Betriebssystemdaten wurde die Sprache um Schlüsselwörter für die Definition von Markern und den Zugriff auf Funktionsargumente erweitert - siehe dazu Listing 6.2, Zeile 1 respektive Zeile 4.

Listing 6.2: Auszug aus einem Beispiel für eine *FayLINQ*-Anfrage (entnommen aus [9])

```
1 var kernelAllocations = cluster.Function(kernel, "ExAllocatePool");
2 var allocIntervalSizePairs = from event in kernelAllocations
3 where event.time < Now.AddMinutes(10)
4 let allocSize = event.GetArg(2)
5 select new { interval = event.time/period, size = log2(allocSize) };
```

Bei der Anfragesprache liegt das Hauptaugenmerk hauptsächlich auf der Verarbeitung der Daten, wie an der exemplarischen Anfrage in Listing 6.2 zu sehen ist. Mit der Sprache *LINQ* steht ein potentes Mittel zur Datenverarbeitung zur Verfügung, wie in dem Beispiel in Zeile 3-5 zu sehen ist. Daher eignet sich dieser Aspekt der Sprache gut für die geforderten Ziele. Außerdem ist das Zusammen von Markern möglich und erlaubt so eine Abstraktion von den konkreten Funktionsnamen - vgl. Listing 6.2, Zeile 1. Dennoch ist die Anfragesprache als Ganzes nicht für das gedachte Szenario geeignet, da die Datenquelle immer noch zu nah an dem konkreten Programmcode sind. Ferner erlaubt die Sprache keine Verknüpfung von Datenquellen.

6.2.2 TinyDB

TinyDB ist eine relationale Datenbank zur Datenerhebung in Sensornetzwerken [20]. In diesem Kontext wird nur die Sprache zur Erhebung der Daten betrachtet. Weiterführende Aspekte von *TinyDB*, wie die technische Umsetzung oder der Transport der Daten von den Sensorknoten zum Anfragenden, sind nicht Gegenstand dieser Untersuchung.

Die Sensorknoten werden in einer dedizierten Tabelle mit den Namen *sensors* abgebildet. Die Spalten stellen die verschiedenen Arten von Sensoren dar, die in dem Netzwerk verfügbar sind Die Zeilen entsprechen den unterschiedlichen Knoten. Verfügt ein bestimmter Knoten nicht über eine Sensorart, so weist die Zeile in der betreffenden Spalte das Schlüsselwort *NULL* auf. Die Anfragen werden in der Sprache *SQL* verfasst und an alle Knoten verteilt. Die Sprache wurde um Elemente erweitert, um die zeitliche Veränderung der Sensorwerte zu berücksichtigen. Eine Anfrage, wie sie in Listing 6.3 zu sehen

[2] Kurzform für *Structured Query Language*

ist, wird auf einem Knoten ausgeführt und die Werte werden über das Sensornetz sofort zum Anfragenden übermittelt.

Listing 6.3: Exemplarische Anfrage an ein Sensornetzwerk (entnommen aus [20])

```
1 SELECT nodeid, light, temp
2 FROM sensors
3 SAMPLE PERIOD 1s FOR 10s
```

Wie außerdem zu erkennen ist, kann über die Schlüsselwörter *SAMPLE PERIOD* angegeben werden, dass die Daten in einem Intervall über eine bestimmte Dauer abgefragt werden. In diesem Beispiel werden die Temperatur, die Lichtstärke sowie die Knotennummer einmal die Sekunde für eine Dauer von zehn Sekunden ermittelt. Darüber hinaus können die Daten auf einem Knoten lokal zwischengespeichert werden. Dazu bedarf es der Erzeugung eines Zwischenspeichers mit dem Schlüsselwort *STORAGE POINT*. Auf diesem Weg kann die Historie der Messwerte erhoben werden. Oftmals sollen Daten in einem Sensornetz nur bei Eintreten eines bestimmten Ereignisses ermittelt werden. In der Sprache ist dafür das Schlüsselwort *ON EVENT* vorgehen. Dies wird eine Anfrage vorangestellt. Es sorgt dafür, dass eine Anfrage nur bei dem Ereignis ausgeführt wird. Das Ereignis selbst kann von dem Knoten generiert wird, oder aber auch aus einer anderen Anfrage heraus.

Abschließend ist festzuhalten, dass mit *SQL* eine gute Möglichkeit der Datenverarbeitung geboten wird. Ferner erlauben die Erweiterungen eine Aufzeichnung verschiedener Werte einer Spalte. Jedoch besitzt dieser Ansatz immer noch einen relationalen Charakter. D.h. es gibt immer noch einen Zeitpunkt, zu dem der Datenbestand fixiert wird und anschließend die Auswertung durchgeführt wird. Er bietet lediglich Erweiterungen, um mit Datenströmen zu arbeiten.

6.2.3 Aurora

Abschließend soll auf eine weitere Arbeit eingegangen werden, die ein datenstromorientiertes System zu Verarbeitung von Daten vorgestellt [21]. Das System trägt den Namen *Aurora*. Es umfasst den Fluss der Daten währen der Verarbeitung, die Definition einer eigenen Algebra zur Formulierung von Anfragen sowie die Ausführung selbiger. Darüber hinaus werden Verfahren zur Optimierung von Anfragen und der Abarbeitungsreihenfolge vorgestellt. Jedoch geht es bei den folgenden Betrachtungen lediglich um die Anfragesprache.

Die Algebra ist für die Verarbeitung von Datenströmen konzipiert. Dies bedeutet, die einzelnen Operatoren sind verkettet, so dass die Daten sie sequentiell durchlaufen. Ferner berücksichtigen einige Operatoren, dass Datenströmen kein definierten Anfang bzw. kein definiertes Ende aufweisen. Daher müssen bei manchen Operatoren Fenster angegeben werden, auf denen operiert werden soll. Bei der Definition der Algebra wurden bereits viele Operatoren berücksichtigt, wie z. B. das Verknüpfen von Datenströmen, das Filtern von Tupeln aus einem Datenstrom oder aber die Aggregation von Tupeln. Allerdings ist die Datenquelle nicht Bestandteil der Anfrage. Bei *Aurora* wird davon ausgegangen, dass die Daten beispielsweise von einer externen Anwendung geliefert werden.

6.3 Entwurf der Sprache

Zu Beginn dieses Kapitels ist die Bewältigung der Problemstellung mittels existierender Lösungen angedeutet worden. Dabei ist zum Ausdruck gekommen, dass die genannten Möglichkeiten unzureichend bzw. zu teuer hinsichtlich der Laufzeit sind. Aus diesem Grund ist auf der Basis der Algebra des Datenstromsystems *Aurora* ein neuer Dialekt entwickelt worden [21]. Dabei werden einige Operatoren übernommen, andere modifiziert bzw. neu entworfen. Bei dem Entwurf von *Aurora* sind auf die Paradigmen *human-active, DBMS-passive* und *DBMS-active, human passive* zurückgegriffen worden, die ebenfalls im Rahmen von *Aurora* eingeführt worden sind. Unter dem ersten Paradigma *human-active, DBMS-passive* sind gewöhnliche, relationale Datenbank zu verstehen. Das Datenbankverwaltungssystem agiert hierbei passiv. Es führt die Aktionen des Nutzers aus und hält lediglich den aktuellsten Wert eines Attributs vor. Der Benutzer übernimmt unterdies die aktive Rolle. Er initiiert Anfragen, erzeugt neue Elemente oder verändert den Wert bereits existierender Elemente. Dem gegenüber steht der *DBMS-active, human passive* Ansatz. Das Verwaltungssystem übernimmt die zentrale Rolle. Es führt z. B. eine Historie über die verschiedenen Werte von einzelnen Attributen oder führt vom Nutzer definierte Anfragen automatisch bei neu eintreffenden Werten aus. Der Nutzer obliegt die Rolle des Beobachters, der über Ereignisse und neu eingetroffene Daten informiert wird. Er kann lediglich spezifizieren, welche Operationen auf den Daten durchgeführt werden sollen.

Die Architektur von *Aurora* sieht vor, dass die Eingabedaten von einer externen Quelle, wie z. B. Sensoren oder Programme, stammen. Seine Aufgabe beschränkt sich daher auf das Verarbeiten der Daten. Die im Rahmen dieser Arbeit entworfene Sprache hingegen berücksichtigt auch die Datenquelle. Sie ist Bestandteil der Sprache, wie im Folgenden noch erläutert wird. Jedes Element einer Anfrage wird als Operator bezeichnet. Es existieren verschiedene Arten von Operatoren, wie in Abschnitt 6.3.3 noch ausführlich dargelegt wird. Die Wurzel einer jeden Anfrage ist ein Operator zur Erzeugung des Datenstroms. Er darf genau einmal zu Beginn einer Anfrage vorkommen. Je nach Typ des Stroms wird in regelmäßigen oder unregelmäßigen Abständen ein Datum erzeugt. Danach können, wie der Abbildung zu entnehmen ist, null bis n weitere Operatoren folgen. Ein Datum, von dem Ursprungsknoten erzeugt, propagiert ähnlich einem Datenflussgraphen durch die Kette von Operatoren. Je nach Art eines Knotens kann ein einzelnes Datum absorbiert werden.

Eine detailliertere Betrachtung der diversen Sprachelemente soll in den folgenden Abschnitten erfolgen. Dazu werden zunächst in Abschnitt 6.3.1 die unterstützten Datentypen vorgestellt. In Abschnitt 6.3.2 wird das Format der Daten, wie sie im Datenstrom vorkommen, eingeführt. Abschließend werden in Abschnitt 6.3.3 die Operatoren zur Erzeugung von Datenströmen und zur Verarbeitung der Daten präsentiert.

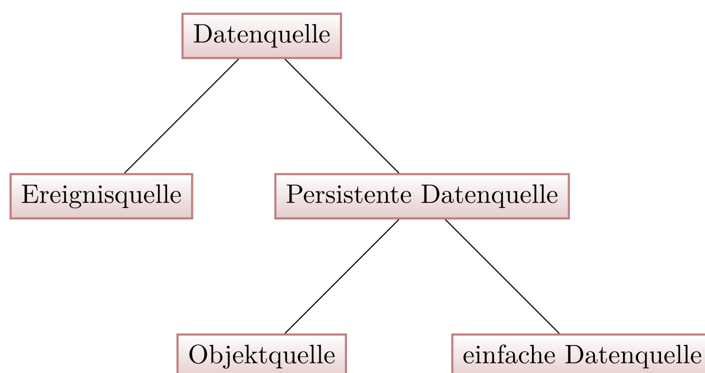


Abbildung 6.1: Hierarchie der Datenquellen

6.3.1 Datentypen

Zur Ausführung einer einzelnen Anfrage müssen die einzelnen Bestandteile - Operatoren genannten - abgearbeitet werden. Hierbei werden die Operatoren gemäß ihrer Reihenfolge in der Anfrage auf einem Datum aus dem Datenstrom ausgeführt. Für eine korrekte Anwendung eines spezifischen Operators ist es erforderlich, die Menge der potentiellen Datentyp, die ein Operand annehmen kann, zu kennen. Daher sollen ebendiese im Folgenden vorgestellt werden.

Als erstes sind die elementaren Datentypen zu nennen, die bereits Bestandteil des Metamodells[3] sind: *int*, *byte*, *float*, *string* und *array*. Darüber hinaus können Operanden auch auf komplexen Datentypen ausgeführt werden. Hierzu muss deren Definition im Datenmodell vorhanden, um zur Laufzeit die für eine Operation nötigen Werte extrahieren zu können. Als Beispiel ist der aus Listing 5.2 bekannte Typ *packetType* zu nennen.

Des Weiteren ist, wie im übernächsten Abschnitt noch verdeutlicht wird, die Interaktion mit anderen Quellen aus einem Datenmodell innerhalb einer Anfrage vorgesehen. Zu diesem Zweck werden die Datenquellen als eigene Gruppe von Datentypen definiert. Wie in Abbildung 6.1 zu sehen ist, steht die allgemeine Datenquelle an der Spitze der Hierarchie. Darunter folgen die Ereignisquellen und die persistenten Quellen. Bei ersteren ist der Zeitpunkt der Verfügbarkeit eines Datums nicht bestimmbar. Sie treten zu beliebigen Zeitpunkten und in beliebiger Häufigkeit auf. Ferner steht der Rückgabetyp einer Ereignisquelle nur im Moment des Auftretens des Ereignisses zur Verfügung. Daher werden sie als flüchtig bezeichnet. Wohingegen persistente Quellen immer abrufbar sind. Zu dieser Gruppe gehören einfache Datenquellen und Objektquellen. Obwohl ein Objekt, wie z. B. ein Prozess[4], eine begrenzte Lebensdauer hat, so ist die Menge aller existierender Instanzen instantan bestimmbar. Folglich kann innerhalb einer Anfrage nicht mit den Ereignissen „erzeugt“und „zerstört“eines Objektes interagiert werden. Es steht lediglich der Status eines Objektes zur Verfügung, wie in Abschnitt 6.3.3 noch gezeigt wird.

[3]vgl. Abschnitt 5.1
[4]vgl. Listing 5.2

6.3.2 Datenformat

Die Beschreibung einer Anfragesprache umfasst ebenfalls die Festlegung des Formats für die Ein- und Ausgabedaten. Sie liegen nicht als abgeschlossene Menge vor. Vielmehr erhält jeder Operator einen Datenstrom als Eingabe und erzeugt seinerseits einen Datenstrom als Ausgabe. Ein Element aus einem solchen Datenstrom wird als Tupel bezeichnet.

$$(\text{TS } \mu\text{sec}, \{\text{Datum}_1, \text{Datum}_2, \cdots, \text{Datum}_m\}) \tag{6.1}$$

Ein Tupel enthält einen UNIX-Zeitstempel TS. Dieser wird allerdings nicht in Sekunden, sondern in Mikrosekunden angegeben. Zudem besitzt ein Tupel eine m-elementige Menge - wie in Definition 6.1. Jedes Element auch - Datum genannt - ist ein Schlüssel-Wert-Paar. Der Schlüssel ist eine Zeichenkette, die den Pfad innerhalb des Datenmodells zu einem Knoten beschreibt. Dieser gibt aus Auskunft über den Typ des Datums. Für das Fallbeispiel „Paketverzögerung“[5] sieht ein Schlüssel, der einen Prozess beschreibt, wie folgt aus: $process.process$. Der Wert eines Datums darf selbst wieder ein Datum - sprich ein Schlüssel-Wert-Paar - enthalten. Ein Beispiel hierfür ist der komplexe Datentyp $packetType$[6].

Die Ordnung der Tupel innerhalb des Datenstroms ist durch ihre Reihenfolge im Strom implizit vorgegeben. Der Zeitstempel eines Tupels kann nicht zur Ordnung der Tupel herangezogen werden, da die aktuelle Uhrzeit nicht exakt synchron zwischen den Prozessoren ist und die globale Zeit außerdem durch Synchronisationsdienste wie das *Network Time Protocol* manipuliert werden können.

6.3.3 Operatoren

Bisher die verwendeten Datentypen sowie das Datenformat beschrieben. Im Folgenden soll der Blick nun weg von eben diesen hin zu den Operatoren gerichtet werden. Allerdings ist es zunächst von Nöten die Konventionen zur Referenzierung von Daten aus dem Datenstrom und anderen Quellen innerhalb einer Anfrage vorzustellen: Ein Datum aus dem Datenstrom wird in einer Anfrage mit dem Präfix *Stream* versehen. Danach folgt, durch einen Punkt getrennt, der Schlüssel des Datums. Eine Bedingung kann z. B. wie folgt aussehen: $Stream.proces.process \geq 1$. Bezieht sich die Pfadangabe auf eine einfache Datenquelle oder eine Ereignis- bzw. Objektquelle, so wird der Rückgabetyp respektive der Typ des Identifikators verwendet. In der zuvor genannten Bedingung bezieht sich der Vergleich auf den Identifikator des Objekts *process*. Handelt es sich allerdings um einen komplexen Datentyp wie z. B. *packetType*, so muss ein Attribut als Bestandteil der Bedingung verwendet werden. Der komplexe Datentyp selber ist hier nicht zulässig.

Es sind die folgenden Komparatoren vorgesehen: $==$, $\leq$, $\leq$, $\geq$ und $\geq$.

[5] vgl. Listing 5.2
[6] vgl. Listing 5.2

GenStream

Der Ausgangspunkt für jede Anfrage ist ein Operator zur Erzeugung eines Datenstroms. Dies erfolgt mithilfe des *GenStream*-Operators.

$$GenStream(\text{StreamType } t, \text{Datasource } e, \text{Extraparameter } ex) = \text{Stream } s \tag{6.2}$$

Wie in Definition 6.2 zu sehen ist, erzeugt er einen neuen Datenstrom, hier mit s bezeichnet. Dieser kann als Eingabe für weitere Operatoren verwendet werden. Als Datenquelle des Datenstroms wird die Quelle e aus dem Datenmodell verwendet. Der letzte Parameter - ex - beinhaltet datenstrom-spezifische Informationen. Darauf wird im Folgenden noch eingegangen. Der erste Parameter bestimmt die Art des Datenstroms. Dabei ist zwischen den folgenden drei Arten zu differenzieren:

- *EVENT_TYPE*
- *SOURCE_TYPE*
- *OBJECT_TYPE*

Zur Vereinfachung existieren Spezialisierungen des *GenStream*-Operators. Zur Erzeugung eines Ereignisstroms dient der *Event*-Operator - vgl. Definition 6.3. Hierzu ist lediglich die Angabe der Quelle im Datenmodell - evt - und die Dringlichkeit - u - erforderlich.

$$\begin{aligned} &Event\,(\text{Event evt}, \text{Urgency u}) = \\ &GenStream\,(\text{EVENT_TYPE, evt}, u, \emptyset) = \\ &\text{Stream s} \end{aligned} \tag{6.3}$$

Bei dem Datenstrom einer einfachen Datenquelle ist zusätzlich noch die Angabe eine Periode nötig, wie in Definition 6.4 zu sehen ist. Sie legt fest, wie oft die entsprechende Quelle abgefragt wird. Die Angabe erfolgt in Mikrosekunden.

$$\begin{aligned} &Source\,(\text{Source src}, \text{Urgency u}, \text{Period p}) = \\ &GenStream\,(\text{SOURCE_TYPE}, src, u, p) = \\ &\text{Stream s} \end{aligned} \tag{6.4}$$

Abschließend ist noch auf den Objektstrom einzugehen. Dabei muss angegeben werden, auf welche Ereignisse dieses Objektes zu reagieren sind - vgl. Definition 6.5.

$$\begin{aligned} &Object\,(\text{Object obj}, \text{Urgency u}, \text{ObjectEvents oevt}) = \\ &GenStream\,(\text{OBJECT_TYPE}, obj, u, oevt) = \\ &\text{Stream s} \end{aligned} \tag{6.5}$$

Ein Objekt kann Daten bei drei verschiedenen Ereignissen auf dem spezifizierten Objekt *obj* generieren:

- $CREATE \Rightarrow$ bei der Erzeugung einer neuen Instanz
- $DELETE \Rightarrow$ bei der Zerstörung einer Instanz und
- $STATUS \Rightarrow$ gibt alle Instanzen des Objektes zurück.

Das Ereignis *STATUS* ist kein reales Ereignis. Vielmehr wird unmittelbar nach der Registrierung einer Anfrage, die daran interessiert ist, eine Liste aller Instanzen des Objekts erzeugt und verarbeitet. Die Werte können mittels eines Oders verknüpft werden.

Liegen auf dem Pfad zu der Quelle eine oder mehrere Objekte, so muss für jedes Objekt spezifiziert werden, welche Instanz des jeweiligen Objektes angesprochen werden soll. Die Angabe erfolgt innerhalb des Pfads unmittelbar nach dem Namen des betreffenden Objektes. Dabei wird der Bezeichner der Instanz in eckige Klammern angehängt. Sollen alle Instanzen angesprochen werden, so ist ein Stern anstelle des Bezeichners zu verwenden. In diesem Fall enthält das erzeugte Tupel ein Datum, dass die jeweilige Instanz beschreibt, zu die Daten gehören.

Select

Der *Select*-Operator dient der Auswahl eines oder mehrerer Daten aus einem Eingabestrom *s*. Der Operator enthält eine Liste von Schlüsseln der jeweiligen Daten, die zu selektieren sind. Daraus resultiert ein Ausgabedatenstrom *t*, wie in Definition 6.6 zu sehen ist.

$$Select\,(\text{Stream s}, \text{Stream.Datum}_i, \cdots) = \text{Stream t},\ \text{i}\ \in [0, m] \tag{6.6}$$

Der *select*-Operator entspricht der Projektion aus der relationalen Algebra.

Filter

Zur Auswahl bestimmter Tupel gemäß verschiedener Kriterien aus dem Eingabestrom s dient der *Filter*-Operator. Er erhält bis zu n Prädikate als Parameter - siehe Definition 6.7. Damit ein Tupel an den Ausgabestrom t weitergereicht wird, muss es allen Prädikaten genügen. Ist mindestens ein Prädikat nicht anwendbar, so wird das Tupel verworfen.

$$Filter\,(\text{Stream s}, \text{Predicate P}_1, \cdots, P_n) = \text{Stream t} \tag{6.7}$$

Ein Prädikat besteht aus einem linken sowie einem rechten Operanden und einem der oben genannten Vergleichsoperatoren. Als Operanden können sowohl echte Werte als auch Elemente aus einem Tupel benutzt werden. Zur Adressierung von Elementen ist das Präfix *Stream* zu verwenden, gefolgt von dem Schlüssel des jeweiligen Datums.

Join

Der *Join*-Operator verknüpft jedes Tupel des Eingabestroms s mit einem Tupel aus einer persistenten Datenquelle pe - siehe Definition 6.8. Dabei müssen beide Tupel den Prädikaten P_i bis P_n genügen. Sind alle Prädikate anwendbar, so werden alle Daten aus dem Tupel der persistenten Quelle in das ursprüngliche Tupel übernommen. Doppelte Daten werden dabei verworfen. Ist mindestens ein Prädikat nicht anwendbar, so werden beide Tupel verworfen.

$$Join\,(\text{Stream s}, \text{PersistentDatasource pe}, \text{Predicate p}_1, \cdots) = \text{Stream t} \tag{6.8}$$

Enthält die Pfadangabe der Quelle pe ein oder mehrere Objekte, so muss auch hier die konkrete Instanz angegeben werden, die angesprochen werden soll.

Sort

Der *Sort*-Operator sortiert die Tupel aus dem Eingabestrom s gemäß n Daten. Dem Operator wird hierzu eine Liste aus Schlüsseln der Daten übergeben, nach denen sortiert werden soll - vgl. Stream.Datum$_i$ in Definition 6.9.

$$Sort\,(\text{Stream s}, \text{Size z}, \text{Stream.Datum}_i, \cdots) = \text{Stream t} \tag{6.9}$$

Da ein Datenstrom kein definiertes Ende bzw. definierten Anfang besitzt, ist es nicht möglich, eine abgeschlossene Menge zu erstellen, auf der eine Sortierung vorgenommen werden kann. Um diesen Umstand zu beseitigen, erfordert der *Sort*-Operator die Angabe einer Fenstergröße z. Der Operator wartet zunächst bis z Tupel aus dem Eingabestrom eingetroffen sind, sortiert diese und reicht sie anschließend an den Ausgabestrom weiter. Bevor nicht z Tupel vorliegen, erfolgt keine Weitergabe an den Ausgabestrom.

Aggregate

Ein weiterer Bestandteil der Sprache ist eine Aggregatsfunktion zur Reduktion von Daten. Der entsprechende *Aggregate*-Operator fasst eine Menge von z Tupel aus dem Eingabestrom s zusammen und führt darauf Operationen aus. Auch in diesem Fall gilt, dass die Operationen nicht auf dem gesamten Datenstrom durchgeführt werden, sondern lediglich auf einem z großen Fenster aus Tupeln. Die zu realisierende Funktion wird über die drei OP Parameter angegeben: Alle Schritte, die zur Initialisierung dieses Operators nötig sind, werden mit dem Parameter OP_{init} spezifiziert. Beim Eintreffen eines weiteren Tupels werden die in OP_{next} angegebenen Operationen aufgeführt. Bei Abschluss eines Fensters werden die Schritte aus OP_{finish} abgearbeitet. Darüber hinaus bietet der Operator die Möglichkeit, nach Abschluss einer Operation um a Tupel das Fenster voran zu schieben. Dabei werden die a letzten Tupel verworfen. Welche Tupel in den Ausgabestrom emittiert werden, muss in der Menge OP_{finish} angegeben werden.

$$\begin{aligned}&Aggregate(\text{Stream s}, OP_{init},\ OP_{next},\ OP_{finish},\\&\text{Size z}, \text{Advance a}, \text{Stream.Datum}_i, \cdots) = \text{Stream t}\end{aligned} \tag{6.10}$$

Mit der in Definition 6.10 dargestellten Schreibweise lassen sich verschiedene Arten der Aggregation realisieren. Aktuell sind die folgenden Funktionen vorgesehen: *Avg*, *Min* und *Max*. In Definition 6.11 ist die Mittelwertsfunktion mithilfe des *Aggregate*-Operators definiert. Hierbei ist zu sehen, dass zu Beginn die beiden Variablen mit Null initialisiert werden - siehe OP_{init}. Bei jedem weiteren Tupel wird der Zähler inkrementiert und der Wert des zu aggregierenden Datums aufsummiert.

$$\begin{aligned} &Avg\,(\text{Stream s}, \text{Size z}, \text{Advance a}, \text{Stream.Datum}_i, \cdots) = \text{Stream t} \Leftrightarrow \\ &Aggregate\,(\text{Stream s}, \{start = \text{ Stream.TS}; sum = count = 0;\}\,, \\ &\{sum{+}= \text{ Stream.Datum}_i; count++;\}\,, \\ &\{out \Leftarrow (start, \{\text{ Stream.Datum}_i \rightarrow sum/count\})\}\,, z\ , a) = \text{Stream t} \end{aligned} \tag{6.11}$$

Bei Erreichen der Fenstergröße z wird der Mittelwert in Form eines Tupels in den Ausgabestrom geschrieben. Als Zeitstempel des Ausgabetupels wird der Zeitstempel des ersten Tupels aus dem aktuellen Fenster verwendet. Der Mittelwert selber wird als Datum in das Tupel geschrieben, dessen Schlüssel dem des zu aggrigierenden Elements entspricht. Die Beschreibung der Minimumsfunktion ist in Definition 6.12 dargestellt. Hier wird bei jedem neu eintreffenden Tupel überprüft, ob der Wert des i-ten Datums kleiner als der aktuell gespeicherte Wert ist. In dies der Fall, so wird der neue Wert abgespeichert. Bei Erreichen der kompletten Fenstergröße wird ebenfalls ein Tupel erzeugt, allerdings enthält dies den kleinsten Wert des entsprechenden Datums.

$$\begin{aligned} &Min\,(\text{Stream s}, \text{Size z}, \text{Advance a}, \text{Stream.Datum}_i, \cdots) = \text{Stream t} \Leftrightarrow \\ &Aggregate\,(\text{Stream s}, \{start = \text{ Stream.TS}; min = MAX_INT;\}\,, \\ &\{min = (Stream.e \leq min ?\, Stream.e : min)\}\,, \\ &\{out \Leftarrow (start, \{\text{ Stream.Datum}_i \rightarrow min\})\}\,, z\ , a) = \text{ Stream t} \end{aligned} \tag{6.12}$$

Analog dazu ist in Definition 6.14 die Maximumsfunktion definiert. Abschließend ist auf die *group*-Funktion als Spezialfall der Aggregation einzugehen. Beim Gruppieren werden Daten zusammengefasst, ohne dass eine spezielle Operation auf ihnen ausgeführt wird.

$$Max\,(\text{Stream s}, \text{Size z}, \text{Advance a}, \text{Stream.Datum}_i, \cdots) \tag{6.13}$$

$$\begin{aligned} &= Stream\ t \Leftrightarrow \\ &Aggregate\,(\text{Stream s}, \{start = \text{ Stream.TS}; max = MIN_INT;\}\,, \\ &\{max = (Stream.e \leq max ?\, Stream.e : max)\}\,, \\ &\{out \Leftarrow (start, \{\text{ Stream.Datum}_i \rightarrow max\})\}\,, z\ , a) = \text{ Stream t} \end{aligned} \tag{6.14}$$

Ferner entspricht Fortschrittsparameter *Advance* der Fenstergröße, wie in Definition 6.15 zu sehen ist.

$$\begin{aligned} &Group\,(\text{Stream s}, \text{Size z}, \text{Stream.Datum}_i, \cdots) = \text{Stream t} \Leftrightarrow \\ &Aggregate\,(\text{Stream s}, \emptyset, \emptyset, \emptyset, z, z) = \text{ Stream t} \end{aligned} \tag{6.15}$$

6.4 Exemplarische Anfragen

Nachdem im vorangegangen Abschnitt die Anfragesprache ausführlich vorgestellt worden ist, sollen nun anhand der Fallbeispiel aus Abschnitt 4.2 Anfragen formuliert werden. Dies dient der Veranschaulichung der formalen Definition.

6.4.1 Paketverzögerung

Zur Verzögerung von Netzwerkpaketen muss zunächst das Datenaufkommen pro Prozess bzw. pro Applikation bestimmt werden, um das Lastprofil einer Applikation lernen zu können. Hierzu müssen alle eingehenden und ausgehenden Netzwerkpakete erfasst werden und die dazugehörige Applikation ermittelt werden. Daher werden zwei Anfragen formuliert, die sich für das Ereignis *net.device.onTx* respektive *net.device.onRx* registrieren. Da die Pakete aller Netzwerkgeräte von Interesse sind, werden die Anfragen für alle vorhandenen Netzwerkgeräte - ausgedrückt durch *[*]* - registriert. Anschließend werden das Socket des ein- bzw. ausgehenden Pakets mit dem dazu passenden Prozess verknüpft, wie in Zeile 1-2 und 6-7 des Listing 6.4 zu sehen ist. Darauffolgend wird der Prozess mit der korrespondierenden Applikation verknüpft. Abschließend wird aus dem Datenstrom nur das Datenpaket, der Prozess-ID und der Name der Applikation selektiert - vgl. Zeile 4 bzw. 9.

Listing 6.4: Anfragen für die Lernphase des Fallbeispiels „Paketverzögerung“

```
1 v = Event(net.device[*].onTx,0)
2 v = Join(v,process.process[*].sockets, Stream.net.packetType.socket ==
     Join.process.process.sockets)
3 v = Join(v,ui.app[*].processes, Stream.process.process ==
     ui.app.processes)
4     Select(v,Stream.net.packetType,Stream.ui.app,Stream.process.process)
5
6 x = Event(net.device[*].onRx,0)
7 x = Join(x,process.process[*].sockets, Stream.net.packetType.socket ==
     Join.process.process.sockets)
8 x = Join(x,ui.app[*].processes, Stream.process.process ==
     ui.app.processes)
9     Select(x,Stream.net.packetType,Stream.ui.app,Stream.process.process)
```

Nachdem erste Lastprofile ermittelt worden sind, kann aktiv in das System eingegriffen werden. Dazu wird eine Anfrage für das Ereignis *net.device.onEnqueueTx* registriert. Dieses Ereignis wird ausgelöst, sobald ein Netzwerkpaket zum Versand in die Warteschlange für das entsprechende Netzwerkgerät eingehängt wird. Wie in Zeile 2-4 des Listing 6.5 zu sehen ist, wird auch hier zu jedem Datenpaket der zugehörige Prozess bzw. die passende Applikation bestimmt. Mit dieser Information kann die Lernkomponente erkennen, welche Applikationen im Begriff sind Daten zu senden und Vorhersagen für die Zukunft zu treffen. Daraus kann eine maximale Verzögerung abgeleitet werden. Die Vorhersagen greift eine steuernde Komponente mit der entsprechenden Anfrage aus Zeile 6 ab. Hier wird alle 50 *ms* die aktuelle Verzögerungstoleranz abgefragt. Dieser Wert wird im Netzwerksubsystem zur Verzögerung wartender Datenpakete benutzt.

Listing 6.5: Anfragen zur Systemkontrolle für das Fallbeispiel „Paketverzögerung“

```
1 x = Event(net.device[*].onEnqueueTx,0)
2 x = Join(x,process.process[*].sockets, Stream.net.packetType.socket ==
      Join.process.process.sockets)
3 x = Join(x,ui.app[*].processes, Stream.process.process ==
      ui.app.processes)
4      Select(x,Stream.net.packetType,Stream.ui.app,Stream.process.process)
5
6 y = Source(net.delayTolerance,0,50000)
```

6.4.2 Automatische Umschaltung 2G/3g

Im Kontext des Fallbeispiels „Automatische Umschaltung 2G/3G“soll sowohl anhand des applikations-spezifischen Netzwerkverkehrs als auch durch den Zustands des Displays entschieden werden, welche Mobilfunktechnologie zum Einsatz kommt. Deshalb wird, ebenfalls wie beim vorangegangenen Fallbeispiel, jeweils eine Anfrage zu Ermittlung der ein- bzw. ausgehenden Netzwerkpakete und der passenden Anwendungen registriert. Da der Verbleib des Datenpakets vor dem Senden irrelevant ist, wird auf weitere Anfragen, wie z. B. auf das Ereignis *net.device.onEnqueueTx*, verzichtet. Zur Bestimmung des Zustands des Displays wird eine Anfrage auf das Ereignis *ui.screen.screenstate* registriert - vgl. Zeile 11 in Listing 6.6.

Listing 6.6: Anfrage für das Fallbeispiel „Automatische Umschaltung zwischen 2G und 3G“

```
1 v = Event(net.device[*].onTx,0)
2 v = Join(v,process.process[*].sockets, Stream.net.packetType.socket ==
      Join.process.process.sockets)
3 v = Join(v,ui.app[*].processes, Stream.process.process ==
      ui.app.processes)
4      Select(v,Stream.net.packetType,Stream.ui.app,Stream.process.process)
5
6 x = Event(net.device[*].onRx,0)
7 x = Join(x,process.process[*].sockets, Stream.net.packetType.socket ==
      Join.process.process.sockets)
8 x = Join(x,ui.app[*].processes, Stream.process.process ==
      ui.app.processes)
9      Select(x,Stream.net.packetType,Stream.ui.app,Stream.process.process)
10
11 z = Event(ui.screen.screenstate,0)
```

7 Implementierung

Die Ziele dieser Arbeit sind in Kapitel 4 ausführlich vorgestellt worden. Als erster Schritt in Richtung ebendieser Ziele ist in Kapitel 5 ein Metamodell zur Erstellung von Datenmodellen für die jeweiligen Anwendungsfälle präsentiert worden. Ferner sind auch Beispiele für Datenmodelle demonstriert worden. Der zweite Schritt ist die Definition einer Anfragesprache in Kapitel 6. Dabei ist es um die Interaktion mit dem Datenmodell gegangen. Bei den genannten Phasen liegt der Fokus vor allem auf der formalen Definition. Der letzte Schritt zur Umsetzung der angestrebten Ziele lenkt das Hauptaugenmerk auf die konkrete Realisierung. Hierbei geht es um die Implementierung des Datenmodells, der Anfragesprache sowie der dafür nötigen Hilfsmittel. Aus diesem Grund werden zunächst in Abschnitt 7.1 die Anforderungen an die Implementierung spezifiziert. In Abschnitt 7.2 wird auf die Umsetzung der Schnittstellen für externe *Provider* wie auf die drei Teilbereiche Datenmodell, Anfragesprache und Tupel eingegangen. Den Abschluss bildet in Abschnitt 7.3 die Vorstellung von Problemstellungen bei der Implementierung auf der Kern- und Benutzerebene.

7.1 Anforderungen

Die Abb. 4.1 in Kapitel 4 veranschaulicht bereits erste Anforderungen an die Implementierung. Es soll eine Kernkomponente geben, die aus zwei Teilen besteht: einen Kernteil und einen Teil auf der Benutzerebene. Beide Teile sollen funktional identisch seien. D.h. für einen Anbieter von Daten soll es bei der Kommunikation mit dem *Claosi* grundsätzlich keinen Unterschied geben, ob er sich im Kern- oder Benutzerbereich befindet. Die Unterschiede bei der Programmierung im Linux-Kernel und auf Benutzerebene sind hiervon ausgeklammert. Die *Claosi* soll selber keine Elemente eines systemweiten Datenmodells bereitstellen. Es besitzt initial ein leeres Datenmodell, das nur aus einem *model*-Knoten ohne Kinder besteht. Somit soll es lediglich ein Grundgerüst darstellen, dessen Funktionalität von Anbietern sowie Nutzern verwendet werden kann. Da Teile von *Claosi* auf der Benutzerebene laufen, ist zu prüfen, welche Programmiersprache zum Einsatz kommen soll.

Zwischen den beiden Komponenten soll zudem eine Kommunikation stattfinden. Hierbei ist ein Kanal mit niedriger Latenz vorgesehen, der für eine dringende Nachrichten bzw. Daten gedacht ist. Dabei spielt der Mehraufwand für Übertragung einer Nachricht eine untergeordnete Rolle. Es kommt darauf an, dass diese Nachricht möglichst schnell auf die andere Ebene transportiert wird. Außerdem soll es einen Kommunikationskanal für den Massentransport geben. Es werden alle zu sendenden Nachrichten gesammelt und sobald eine Schwelle, z. B. die Anzahl Nachrichten oder die kumulierte Größe der

Nachrichten in Bytes, überschritten ist, findet ein Transport auf die andere Ebene statt. Des Weiteren muss eine Synchronisation zwischen den beiden Komponenten bezüglich des Zustands des systemweiten Datenmodells erfolgen, da es für einen Nutzer des Datenmodells vollkommen transparent ist, wo sich der jeweilige Anbieter befindet.

Das Werkzeug *Claosi* soll so konzipiert werden, dass zur Laufzeit sowohl Anfragen als auch Anbieter registriert und auch wieder abgemeldet werden können. Zu diesem Zweck ist der Entwurf einer Schnittstelle erforderlich, die eben genau jene Funktionalität umsetzt. Um überhaupt einem Anbieter die Registrierung zu ermöglichen, muss zuerst eine Darstellung eines Datenmodells in der gewählten Programmiersprache entworfen werden. Dabei muss die Baumstruktur eines Datenmodells abgebildet werden. Die entworfene Datenstruktur muss das Traversieren des Baums sowohl aufwärts als auch abwärts erlauben. Ferner müssen je nach Elementtyp Zusatzinformationen, wie z. B. der Rückgabewert eines Ereignisses, festgehalten werden. Zu eben jenen Informationen zählen unter anderem die Möglichkeiten zur Abfrage einer Quelle und die Aktivierung bzw. Deaktivierung eines Ereignisses. Bei der Registrierung bietet der Anbieter eine Erweiterung des Datenmodells an. Deren Syntax muss zunächst überprüft werden. Anschließend muss geprüft werden, ob diese mit dem aktuellen Datenmodell vereinbar ist. Erst wenn beide Überprüfungen positiv verlaufen, ist eine Vereinigung mit dem bestehenden Modell möglich. Daher sind die Funktionen essentiell für die *Claosi*. Damit ein Anbieter das Auftreten eines Ereignisses oder die Veränderung eines Objektes vermelden kann, müssen ebenfalls Schnittstellen zur *Claosi* vorgesehen werden.

Selbstverständlich sind ebenfalls Schnittstellen vorzusehen, um Anfragen an- bzw. abzumelden. Zur Formulierung dieser Anfragen ist die Abbildung der in Kapitel 6 definierten Sprache in der gewählten Programmiersprache erforderlich. Hierbei muss eine Verkettung der Operatoren umgesetzt werden, so dass eine sequentielle Ausführung ermöglicht wird. Die Datenstrukturen sind so zu wählen, dass Gemeinsamkeiten von Operatoren zusammengefasst werden. Als ein Beispiel ist der *GenStream*-Operator zu nennen, der eine Oberklasse der drei konkreten Operatoren *Event*, *Source* und *Object* darstellt. Für die dynamische Registrierung einer Anfrage muss zunächst eine Syntaxüberprüfung realisiert werden. Diese soll u.a. gewährleisten, dass alle zum Erzeugen eines Datenstrom erforderlichen Informationen gegeben sind. Außerdem soll kontrolliert werden, ob alle in der Anfrage referenzierten Elemente des Datenmodells auch tatsächlich präsent sind.

Eine weitere Aufgabe ist, die Ausführung der Anfragen auf dem jeweiligen Tupel. Dazu gehört die Interpretation der einzelnen Operatoren nebst deren Anwendung auf die Tupel im Datenstrom. Auch in diesem Fall ist zunächst eine geeignete, generische Datenstruktur zur Repräsentation eines Tupels zu entwerfen. Dabei müssen gemäß des in Abschnitt 6.3 definierten Formats ein Zeitstempel sowie eine beliebige Anzahl von Daten in einem Tupel abgelegt werden können. Ferner muss jedes Datum aus einem Schlüssel und einem generischen Teil, der den Wert des Datums enthält, bestehen. Die Größe des für den generischen Teil erforderlichen Speichers ist aus dem Datenmodell zu ermitteln. Darüber hinaus sind Funktionen zur Erzeugung und der Manipulation von Tupeln von Nöten. Die Manipulation von Tupeln umfasst, das Setzen und Abfragen des Wertes eines Datums sowie das Löschen einzelner Daten oder das Tupel als Ganzes.

Die Operation auf den Werten ist u.a. für den *join*- und *filter*-Operator von Interesse. Der *select*-Operator benötigt das Löschen eines einzelnen Datums. Außerdem müssen zwei Tupel im Falle der Verknüpfung eines Datenstroms mit einer weiteren Datenquelle[1] zusammengeführt werden können.

7.2 Kernkomponenten

Das Werkzeug *Claosi* soll gemäß Anforderungen ebenenübergreifend Daten bereitstellen. Daher ist es unabdingbar, dass es sowohl eine Komponente im Betriebssystemkern als auch im Benutzerbereich gibt. Die Art der Programmierung in den beiden Bereichen unterscheiden sich jedoch massiv. Somit kann es keine einheitliche Codebasis geben, die in beiden Bereichen übersetzt werden kann. Aus diesem Grund wird zwischen zwei Teilen unterschieden: dem Kernteil sowie dem ebenenspezifischen Teil. Ersterer umfasst alle Funktionalität für den Umgang mit dem Datenmodell, das Interpretieren von Anfragen und die Erzeugung und Modifikation von Tupeln. Der ebenenspezifische Teil beinhaltet Programmcode, der abstrakte Konzepte, wie das Verwalten und zur Ausführungen bringen von Anfragen, auf die jeweilige Ebene abbildet. Die beschriebene Struktur schränkt die Auswahl der möglichen Programmiersprachen bereits stark ein. Aufgrund der Ausführung im Betriebssystemkern kommen nur native Sprachen wie *C* oder *C++* in Frage. Grundsätzlich ist es möglich, in *C++* geschriebenen Programmcode durch geeignete Kapselung und Schnittstellen, in einer reinen *C*-Umgebung wie dem Linux-Kern ausführen. Der genaue Aufwand hierfür ist jedoch nicht vollständig abschätzbar, so dass die Wahl schlussendlich auf die Programmiersprache *C* gefallen ist.

Bei der Implementierung des Kernteils ist darauf geachtet worden, dass der Programmcode ebenenunabhängig gehalten wird. Da dies jedoch nicht immer ohne Weiteres möglich ist, ist an den entsprechenden Stellen auf Makros zurückgegriffen worden. Als ein Beispiel ist das Holen eines *Locks* zu nennen. Die dazu nötige Funktion und die erforderlich Datenstruktur unterscheiden sich je nach Ebene stark. Daher wird im Kernteil das Makro *ACQUIRE_WRITE_LOCK* zum Holen und *RELEASE_WRITE_LOCK* zum Freigeben verwendet. Das Marko selber wird im ebenenspezifischen Teil implementiert. Somit kann der Kernteil ohne Probleme mit den Werkzeugen sowohl für den Betriebssystemkern als auch für den Benutzerbereich übersetzt werden. Die Komponente für den Betriebssystemkern wurde als Linux-Kern-Modul implementiert, so dass sie zur Laufzeit nachgeladen werden kann. Sie wurde für den Linux-Kern in Version 3.14.14 entwickelt. Die Komponente für den Benutzerbereich läuft als eigenständiger Prozess. Auf die Kommunikation zwischen beiden Komponenten wird noch gesondert in Abschnitt 7.3.2 eingegangen. Im Rahmen der Festlegung der Ziele in Kapitel 4 wird das Betriebssystem *Android* als ein Einsatzgebiet vorgestellt. Die Implementierung erfolgte auf einem Arbeitsplatzrechner, der mit einer gewöhnlichen Linux-Distribution versehen ist. In *Android* kommt ebenfalls ein Linux-Kern zum Einsatz. Ferner ist der Großteil des sogenannten *Android Frameworks* in *C* bzw. *C++* geschrieben. Somit ist eine Integrati-

[1] vgl. Abschnitt 6.3 - *join*-Operator

on der hier vorgestellten Lösung in *Android* sowohl im Betriebssystemkern als auch auf der Benutzerebene möglich.

Die in Abschnitt 7.1 geforderte Schnittstelle für externe Anbieter ist in Gestalt von vier Funktionen umgesetzt worden. Bei der folgenden Erläuterung liegt der Fokus weniger auf der korrekten Darstellung der Parameter und deren Typen. Vielmehr geht es um eine abstraktere Beschreibung der Funktionalität.

- **registerProvider(Datamodel dm, Query q):** Die Funktion dient der Registrierung eines Datenmodells sowie einer Menge von Anfragen. Der erste Parameter enthält die Beschreibung eines Datenmodells, mit dem das bereits existierende Datenmodell erweitert werden soll. Zunächst wird die Syntax überprüft der Beschreibung überprüft, gefolgt von einer Prüfung, ob die Erweiterung mit dem bestehenden Modell vereinbar ist. Fallen beide Prüfungen positiv aus, so wird das bereitgestellte Datenmodell integriert und steht sofort zur Verfügung. Über den zweiten Parameter kann der Nutzer direkt eine Liste von Anfragen übergeben, die unmittelbar nach der Registrierung des Datenmodells ebenfalls registriert werden. Hier findet gleichfalls eine Überprüfung der Syntax statt. Einer der beiden Parameter darf weggelassen werden. Die Funktion gibt nach erfolgreicher Durchführung aller Schritte Null zurück. Im Falle eines Fehlers deutet der Rückgabewert auf die Art des Fehlers innerhalb der Anfrage bzw. des Datenmodells hin. Die Stelle bezieht sich entweder auf das Datenmodell oder auf eine der Anfragen. Tritt der Fehler bei der Prüfung einer der Anfragen auf, ist das bereitgestellte Datenmodell bereits registriert. Daher ist es unter Umständen erforderlich, dass Datenmodell wieder zu entfernen.

- **registerQuery(Query q):** Diese Funktion dient lediglich der Registrierung einer Menge von Anfragen. Hierbei findet zunächst eine Prüfung der Syntax aller Anfragen statt. Ist diese erfolgreich, werden alle Anfragen bei dem *Claosi* angemeldet. Falls erforderlich, werden die entsprechenden Datenquellen aktiviert. Auch hier gibt der Rückgabewert im Falle eines Fehlers, die Art des Fehlers an.

- **objectChanged(String path, ObjectEvent evt, Tuple tuple):** Diese Funktion ermöglicht einem Anbieter, zu signalisieren, dass sich ein Objekt verändert hat. Der erste Parameter bezeichnet durch eine Zeichenkette den Pfad im Datenmodell zu diesem Objekt. Der zweite Parameter gibt die Art der Änderung an - vgl. Abschnitt 6.3.3. Der letzte Parameter ist ein Tupel, welches die jeweiligen Daten beinhaltet.

- **eventOccured(String path, Tuple tuple):** Mithilfe der Funktion können Anbieter dem *Claosi* mitteilen, dass ein Ereignis eingetreten ist. Die Zeichenkette, welche als ersten Parameter übergeben wird, gibt den Pfad im Datenmodell zu dem Ereignis an. Der zweite Parameter ist das Tupel mit dem Rückgabewert, der gemäß Datenmodell erwartet wird.

Zu den ersten beiden Funktionen existiert das jeweilige Gegenstück, das Teile des Datenmodells löscht bzw. Anfragen aus dem System entfernt. Das Vorgehen dabei entspricht

dem der erläuterten Funktionen abgesehen von der konkreten Operation des Löschens.

Für die beschriebenen Operationen ist es nötig, die internen Datenstrukturen gegen konkurrierenden Zugriff abzusichern. Deshalb wird auf jeder Ebene eine Sperre zur Gewährleistung von gegenseitigem Ausschluss eingesetzt. Für diesen Anwendungsfall wird eine Leser-Schreiber-Sperre verwendet. Diese erlaubt das gleichzeitige holen der Lesesperre durch eine beliebiger Anzahl von Nutzern. Jedoch ist nur ein Nutzer berechtigt die Schreibersperre zu belegen. Die oben beschriebenen Funktionen zum An- und Abmelden von Datenmodellen bzw. Anfragen holen bei Funktionseintritt die Schreibersperre und geben diese erst vor Rückkehr wieder frei. Somit ist gewährleistet, dass beispielsweise keine Anfrage abgearbeitet wird, die möglicherweise auf ein Element im Datenmodell verweist, das gerade entfernt wurde. Alle anderen Operationen, wie z. B. das Ausführen einer Anfrage oder das Erstellen eines Tupels durch einen Anbieter, erfordern lediglich eine Lese-Sperre. Zusätzlich wird bei jeder Sperre, die im Betriebssystemkern geholt wird, die Unterbrechungen abgeschaltet. Die hat zur Folge, dass die entsprechende Aktion atomar ausgeführt wird. Durch das beschriebene Vorgehen ist sichergestellt, dass Modifikationen u.a. an dem Datenmodell nur durch einen Nutzer gleichzeitig erfolgen. Der lesende Zugriff, z. B. in Form von Erstellen eines Tupels oder ausführen einer Anfrage, darf hingegen von mehreren Stellen gleichzeitig erfolgen. Hierdurch wird der Grad der Parallelität so wenig wie möglich beeinflusst.

7.2.1 Handhabung des Datenmodells

Bei der Abbildung eines Datenmodells, wie es z. B. in Abschnitt 5.2 vorkommt, wird von einer zentralen Datenstruktur ausgegangen. Diese bildet den Knoten eines Datenmodells ab und wird je nach Typ mit eine definierten Menge an Informationen angereichert. Die Datenstruktur heißt *DatenModelElement* und ist in Listing 7.1 dargestellt. Der Name eines jeden Knoten ist in dem Attribut *name* abgelegt. Er beinhaltet nur den Namen und nicht den vollständigen Pfad. Die Attribute *childrenLen* und *children* ermöglichen das Anhängen einer beliebigen und veränderlichen Zahl von Kindknoten. Ferner ist über einen Zeiger auf den Vaterknoten - siehe Attribut *parent* - ist eine Traversierung des Baumes in beide Richtungen möglich. Der Typ eines Knoten wird in *dataModelType* festgehalten. Dabei zeigt *typeInfo* je nach Typ auf einen Speicherbereich, der zusätzliche Informationen enthält. Diese umfassen bei einem Ereignis beispielsweise Zeiger auf Funktionen zur Aktivierung bzw. Deaktivierung der Datenquelle sowie den Rückgabetyp. Außerdem werden in den Zusatzinformation zu Objekten, Ereignissen und einfachen Datenquelle die Menge der für diese Quelle registrieren Anfragen verwaltet. Über das Attribut *layerCode* wird festgehalten, auf welche Ebene der Ursprung eines Knoten liegt. Dies spielt z. B. bei der Aktivierung einer Quelle eine Rolle, wie in Abschnitt 7.3.3 noch tiefgehender erläutert wird.

Listing 7.1: captionDefinition der Datenstruktur *DatenModelElement*

```
1 struct DataModelElement{
2     char name[45];
3     struct DataModelElement *parent;
4     int layerCode;
```

```
5       int childrenLen;
6       struct DataModelElement **children;
7       unsigned int dataModelType;
8       void *typeInfo;
9   };
```

Auf Basis der genannten Datenstruktur sind einige Operationen implementiert worden: Dazu gehört zunächst die Überprüfung der Syntax eines zur Registrierung übergebenen Datenmodells. Bei der Überprüfung wird untersucht, ob das gegebene Datenmodell konform mit dem Metamodell aus Listing 5.1 ist. Dies umfasst u.a. die Prüfung der Anzahl und des Typs der Kindknoten. Zusätzlich wird bei der Verwendung von komplexen Datentypen als Rückgabetyp verifiziert, ob diese Bestandteil des bereits vorhandenen Datenmodells oder des zukünftigen Datenmodells sind.

Zur Vereinigung des bestehenden Datenmodells mit einer durch einen Anbieter bereitgestellten Erweiterung existiert ebenfalls eine Funktion, die zunächst sicherstellt, ob beide Modelle vereinbar sind. Dafür dürfen Ereignisse, einfache Datenquelle und komplexe Datentyp nicht doppelt existieren. Ferner müssen zwei gleiche Objekte den gleichen *Identifier*-Typ aufweisen, damit sie vereint werden dürfen. Ebenso existiert eine Funktion zur Entfernung eines Teils aus einem Datenmodell, wie es bei der Abmeldung eines Anbieters der Fall ist. Hierzu erfolgt die Beschreibung des zu löschenden Teildatenmodells über die oben genannte Datenstruktur. Der Löschvorgang entfernt anhand der Beschreibung alle existenten Knoten und vereinfacht das verbleibende Datenmodell soweit wie möglich.

Für die Allokation von Speicherplatz zum Ablegen einer Instanz eines komplexen Datentyps ist es unabdingbar die Größe des Datentyps zu ermitteln. Die für diesen Zweck entworfene Funktion bestimmt zur Laufzeit anhand einer Pfadangabe die Größe des Typs. Für den Zugriff auf einzelne Attribute steht ebenfalls eine Funktion zur Berechnung des Offsets eines Attributs innerhalb der Datenstruktur.

7.2.2 Erzeugung und Verarbeitung von Tupeln

Das Tupel ist bereits in Abschnitt 6.3.2 definiert worden. In diesem Abschnitt geht es um die Abbildung eines Tupels in der Programmiersprache *C*. Eine Instanz der Datenstruktur Tupel enthält lediglich den Zeitstempel und einen Zeiger auf ein Feld mit Zeigern auf die Daten eines Tupels. Diese Indirektion ermöglicht es, die Anzahl von Daten pro Tupel zur Laufzeit einfach zu verändern. Wie in Abb. 7.1 zu sehen ist, zeigt jedes Element des Feldes auf eine Instanz von *Item*. Ein *Item* entspricht einem Datum eines Tupels. Das Attribut *name* repräsentiert den Schlüssel eines Datums und enthält den Pfad zu dem Knoten, der den Typ dieses Datums beschreibt. Ferner ist ein Zeiger vorhanden, der auf einen Speicherbereich zeigt, in dem der Wert eines Datums abgelegt ist. Die Größe des Speicherbereich richtet sich nach dem jeweiligen Datentyp. Dessen Größe kann aus dem Datenmodell ermittelt werden.

Es werden Funktionen zur Erzeugung und Modifikation eines Tupels bereitgestellt. Die Erzeugung erfolgt mehrstufig: Zunächst ist ein Tupel der Größe n zu allozieren, wobei n die initiale Anzahl von Daten bestimmt. Anschließend ist eine sequentielle Allokation der

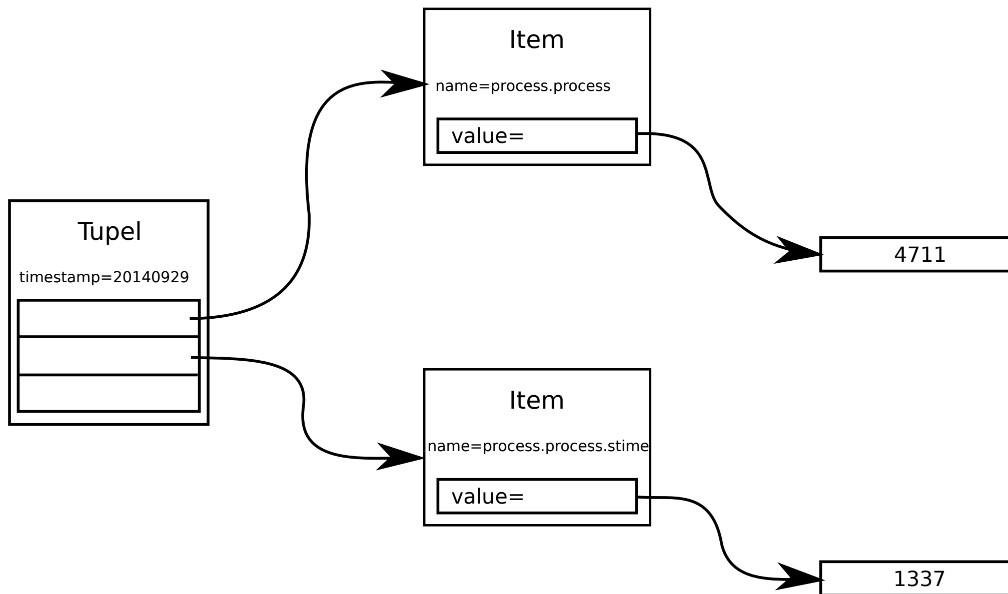


Abbildung 7.1: Struktur eines Tupels

einzelnen Daten erforderlich. Dabei muss der Typ des Datums in Form eines Pfades zu dem entsprechenden Knoten angegeben werden. Die dafür vorgesehene Funktion alloziert ein Datum und weißt es dem Tupel zu. Die für ein Datum nötige Menge an Speicher wird zur Laufzeit aus dem Datenmodell anhand des Pfads abgeleitet.

Beim Zugriff auf den Wert eines Datums gibt der Nutzer den absoluten Pfad des betreffenden Elements an sowie das betreffende Tupel. Der Algorithmus bestimmt automatisch, um welches Datum es sich handelt. Anschließend wird der Inhalt des Speicherbereich ausgelesen und gemäß des Typs des Datums interpretiert. Handelt es sich dabei um einen komplexen Datentyp, wird anhand des Pfads der Offset des Attributs bestimmt und die passende Speicherstelle ermittelt. Somit ist mit einem Funktionsaufruf auch der Zugriff auf komplexe Datentypen möglich, wie z. B. auf den Inhalt von *net.packetTyp* aus dem Fallbeispiel „Paketverzögerung“aus Abschnitt 5.2.1. Für den Zugriff auf das Attribut *dataLength* genügt die Angabe des Pfads *net.packetType.dataLength*.

Es existieren ebenfalls Funktionen, die bei der Umsetzung der Verknüpfung eines Datenstroms mit einer weiteren Datenquelle helfen. Soll eine Vereinigung von Tupel B mit Tupel A erfolgen, so muss zunächst die Anzahl der Daten aus Tupel B ermittelt werden, die nicht in A vorhanden sind. Danach wird Tupel A um eben jene Anzahl vergrößert und die Zeiger auf die entsprechenden Daten Tupel A zugewiesen. Die redundanten Daten aus Tupel B sowie das Tupel selbst werden gelöscht.

7.2.3 Verarbeitung von Anfragen

Die Implementierung von Anfragen und Operatoren, wie in Abschnitt 6.3.3 beschrieben wurden, erfordern zunächst zwei praktische Eigenschaften: Jeder Operator benötigt einen Zeiger auf seinen Nachfolger, um eine aus einer Liste von Operatoren eine Anfrage

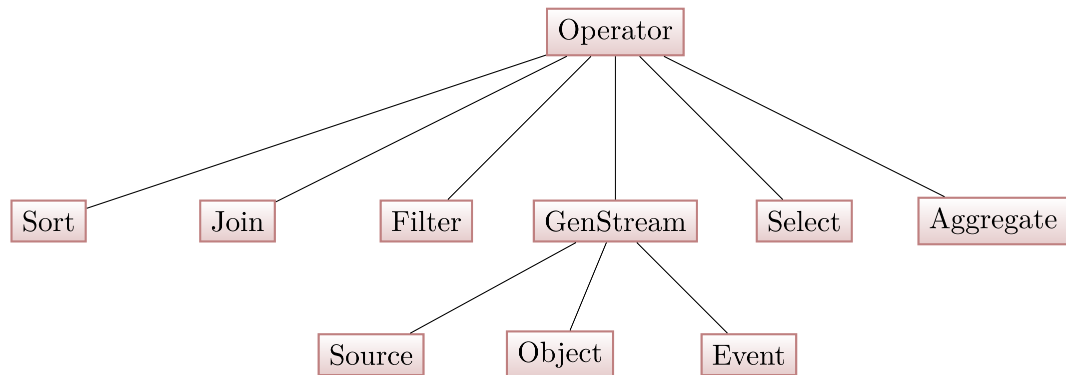


Abbildung 7.2: Hierarchie der Datenstrukturen zur Darstellung von Anfragen

	Event	*Object*	*Source*	*Select*	*Join*	*Filter*	*Sort*	*Aggregate*
Status	x	x	x	x	x	x	-	-

Tabelle 7.1: Übersicht über die implementierten Operatoren

zu formen. Darüber hinaus muss in jedem Operator sein konkreter Typ vermerkt werden, da in der Programmiersprache *C* keine Klassen existieren, sondern lediglich komplexe Datenstrukturen. Zum Auslesen der der Typinformation ist daher ein dediziertes Attribut nötig. Somit ergibt sich zwischen den konkreten Operatoren eine Hierarchie, wie in Abb. 7.2 zu sehen ist. Jeder Operator benötigt ein Attribut *child* sowie ein Attribut *type*. Die Abbildung zeigt außerdem, dass die Operatoren *Event*, *Object* und *Source* gemeinsame Attribute aufweisen, die im *GenStream*-Operator zusammengefasst sind. Auch dieser Umstand wurde bereits in Abschnitt 6.3.3 bei der Definition verdeutlicht.

Eine Vererbung, wie sie aus den objektorientierten Programmiersprachen bekannt ist, existiert in *C* nicht. Allerdings lässt sie sich nachbauen, indem in jeder Datenstruktur, die eine Unterklasse repräsentieren soll, als erstes Attribut eine Variable vom Typ der Oberklasse platziert wird. Dies wird in Listing 7.2 nochmal verdeutlicht. Die abgeleitete Klasse heißt in diesem Fall *Derived* und enthält als erstes Attribut eine Variable vom Typ *Base*.

Listing 7.2: Vererbung in *C*

```
1 struct Base {
2     int a;
3 };
4
5 struct Derived {
6     struct Base base;
7     int b;
8 };
```

Nach diesem Prinzip wurden die Datenstrukturen für alle im Abschnitt 6.3.3 definierten Operatoren implementiert. Daher können alle Operatoren bereits in Anfragen benutzt werden. Allerdings ist es noch nicht möglich, alle Operatoren auszuführen. In Tab. 7.1 ist eine Übersicht über alle Operatoren und deren Status gegeben. Ein X in der ent-

sprechenden Zelle zeigt an, dass der betreffende Operator von dem *Claosi* ausgeführt werden kann. Ein Bindestrich steht für einen Operator, der lediglich in einer Anfrage ausgedrückt werden kann, aber noch nicht ausgeführt wird.

Die bisher vorgestellten Datenstrukturen dienen lediglich der Darstellung des Rumpfes einer Anfrage. Zur vollständigen Formulierung einer Anfrage gehören noch weitere Verwaltungsinformationen. Diese werden in der Struktur *Query* zusammengefasst. Neben verschiedenen Attributen, die zur internen Verarbeitung durch das *Claosi* benötigt werden, enthält sie drei wichtige Attribute: *queryID*, *onQueryCompleted* und *root*. Bei dem erstgenannten Attribut handelt es sich um eine eindeutige Identifikationsnummer, die ebenenübergreifend eindeutig ist. Sie wird nach der erfolgreichen Registrierung der Anfrage gesetzt. Das Attribut *onQueryCompleted* ist ein Zeiger auf eine Funktion, die aufgerufen wird, sobald eine Anfrage auf einem Tupel erfolgreich beendet wurde. Sie bekommt als ersten Parameter die Identifikationsnummer und als zweiten Parameter einen Zeiger auf das Tupel übergeben. Bei dem letzten Attribut handelt es sich um einen Zeiger auf den ersten Operator der Anfrage. Dieser muss einer der drei datenstrom-erzeugenden Operatoren sein. Die vollständigen Definitionen der Datenstrukturen finden sich in Anhang A.1.

7.3 Ebenenübergreifende Belange

Nach der Vorstellung des Kernteils, der auf beiden Ebenen identisch ist, geht es in dem folgenden Abschnitt um die Präsentation von ebenenübergreifenden Belangen. Dabei geht es sowohl um das Zusammenspiel der beiden Ebenen als auch um Problemstellungen, die auf beiden Ebenen anzutreffen sind.

7.3.1 Implementierung eines Zeitgebers

Die Realisierung von Ereignis- und Objektquellen erfordert seitens des *Claosi* wenig Implementierungsaufwand, da sie lediglich aktiviert bzw. deaktiviert werden müssen. Einmal aktiviert produzieren sie selbstständig Daten. Vor allem bestimmt der Anbieter einer Quelle über seine Implementierung, den Zeitpunkt der Erzeugung von Tupeln für die jeweilige Quelle. Im Gegensatz dazu muss eine Datenquelle nur in dem vom Nutzer spezifizierten Intervall abgefragt werden, damit Daten für den Datenstrom erzeugt werden. Dabei ist zu berücksichtigen, dass es eine beliebige Anzahl von Anfragen geben kann, deren Ursprung eine einfache Datenquelle ist. Zusätzlich können alle Anfragen voneinander verschiedene Intervalle aufweisen, so dass für jede Anfrage ein separater Zeitgeber nötig ist. Zur Lösung der Problemstellung wurden zwei Ansätze erarbeitet, die im Folgenden vorgestellt werden.

Variante 1

Der erste Lösungsansatz geht von einem zentralen Zeitgeber aus. Dieser wird je nach Ebene durch den dortigen Zeitgeber realisiert. Er verwendet ein möglichst niedriges

Basisintervall, um ein hohe, zeitliche Auflösung zu ermöglichen. Das Intervall einer jeden Anfrage wird auf ein vielfaches des Basisintervalls aufgerundet. Zusätzlich ist eine zentrale Datenstruktur nötig, um alle Anfragen inklusive ihrer Intervalle zu verwalten. Dabei kommt es besonders auf die effiziente Umsetzung an, da eine Anfrage, nachdem ihr Intervall abgelaufen ist, sofort erneut eingefügt wird. Denn eine einfache Datenquelle liefert kontinuierlich Daten. Für diesen Zweck bieten sich *Timing Wheels* an. Das ist eine Datenstruktur zur Verwaltung von *Timern*. Die Diskussion der verschiedenen Implementierung eines *Timing Wheels* ist nicht Gegenstand dieser Arbeit. Dies wurde bereits ausführlich im Kontext von anderen Veröffentlichungen abgehandelt [22]. Hierbei geht es lediglich um die Notwendigkeit, eine derartige Datenstruktur zu verwenden und ggf. zu implementieren. Jede Anfrage, die auf einer einfache Datenquelle aufsetzt, wird als ein *Timer* in der Datenstruktur vermerkt. Nach Ablauf des Intervalls wird die Quelle abgefragt, ein Tupel in den Datenstrom gegeben und ein neuer *Timer* mit dem gleichen Intervall erzeugt. Somit lassen sich beliebige Anfragen mit verschiedenen Intervallen umsetzen. Aufgrund des dargestellten Ansatzes ergeben sich verschiedene Vor- und Nachteile: Da es nur einen zentralen Zeitgeber gibt, erfolgt das Abfragen derselben einfachen Datenquelle durch mehrere Anfragen implizit sequentiell. Dadurch verringert sich der Synchronisationsaufwand. Allerdings ist es nötig, die Datenstruktur *Timing Wheel* eigens dafür zu implementieren bzw. eine vorhandene Implementierung zu verwenden. Dieser Schritt ist jedoch überflüssig, da der Linux-Kern bereits über eine ähnliche Infrastruktur bereits verfügt - wie im Folgenden noch erläutert wird. Ferner sinkt durch die angesprochene sequentielle Abarbeitung der Grad der Parallelität bei Anfragen, die verschiedene einfache Datenquellen abfragen. Gerade auf einem Multikernsystem könnten eben jene Anfragen parallel abgearbeitet werden.

Variante 2

Der zweite Lösungsansatz sieht eine Realisierung auf Basis des ebenenspezifischen Zeitgebers vor. Hierbei wird pro Anfrage, die eine einfache Datenquelle als Ursprung hat, ein *Timer* mit dem entsprechenden Intervall als Ablaufzeit erstellt und auf der jeweiligen Ebene aktiviert. Es wird keine zusätzliche Datenstruktur zur Verwaltung der einzelnen *Timer* benötigt. Stattdessen wird auf die existierende Infrastruktur der jeweiligen Ebene zurückgegriffen. Hierdurch wird der Implementierungsaufwand maßgeblich reduziert. Ferner ist davon auszugehen, dass die Infrastruktur sowohl im Linux-Kern als auch auf der Benutzerebene hinreichend effizient implementiert ist. Da jede Anfrage ihren eigenen *Timer* zugewiesen bekommt, wird der Grad der Parallelität gesteigert. Sofern die zugrunde liegende Infrastruktur dies unterstützt, kann auf einem Multikernsystem bei Ablauf eines *Timers* jede Quelle auf einem separaten Kern abgefragt werden. Jedoch ergibt sich hieraus auch die Notwendigkeit der Synchronisation bei der Abfrage der gleichen Quelle. Hierzu ein *Lock* erforderlich, dass sicherstellt, dass jede Quelle zeitgleich nur von einem Kontext aus abgefragt werden kann. Damit es allerdings zu Wettkampfbedingungen um eine Quelle kommt, müssen die Intervalle zweier Anfragen vielfacher voneinander sein und beide *Timer* müssen gleichzeitig bzw. ein *Timer* muss bei Ablaufen des anderen gestartet werden. Die Wahrscheinlichkeit für das Zustandekommen dieser Konstellation

wird jedoch als gering angesehen.

Fazit

Bei Umsetzung des Eingangs erläuterten Problems wurde Variante 2 bevorzugt, da diese einen geringeren Implementierungsaufwand beinhaltet sowie einen höheren Grad an Parallelität ermöglicht. Die Implementierung im Linux-Kern nutzt die sogenannten *High-Resolution Timer*[2] als zugrunde liegenden Zeitgeber. Diese sind besonders für Szenarien gedacht, in denen ein *Timer* in der Regel auslöst und nicht vor Ablauf seines Intervalls gelöscht wird. Das traditionelle *Timer*-Subsystem des Linux-Kerns ist hingegen für *Timer* ausgelöst, die in der Regel nicht auslösen. Sie dienen nur der Signalisierung, dass eine bestimmte Zeit verstrichen ist, wie es bei nicht-bestätigen Netzwerkpaketen der Fall ist. Die Kosten für das Aktivieren bzw. Deaktivieren sind besonders gering. Das Verwalten eines *Timers* zur Laufzeit, der mit hoher Wahrscheinlichkeit auslöst, kostet dagegen sehr viel Zeit [23]. Die Verwendung dieser Infrastruktur ist somit für den dargestellten Zweck ungeeignet, da der *Timer* für eine Anfrage immer auslöst und sofort wieder aktiviert wird.

Auf Benutzerebene werden die sogenannten *POSIX-Timer* als zeitgebende Infrastruktur eingesetzt. Wie der Dokumentation[3] des Linux-Kerns zu entnehmen ist, werden diese im Betriebssystemkern ebenfalls durch *High-Resolution Timer* realisiert [24].

7.3.2 Kommunikation

Ein weiterer, wesentlicher Teil dieser Arbeit ist die ebenenübergreifende Realisierung des *Claosi*. Es soll keinerlei Rolle spielen, auf welche Ebene eine Anfrage registriert wird und auf welche Ebene die einzelnen Bestandteile angesiedelt sind. Daher müssen beide Komponenten miteinander kommunizieren, um z. B. Informationen über Veränderungen an dem Datenmodell auszutauschen oder z. B. Tupel an die andere Ebene zuzustellen. Ein erster Lösungsansatz sieht dazu zwei virtuelle Dateien im *proc*-Dateisystem vor: *send* und *receive*. Die *send*-Datei dient der Komponente im Benutzerbereich zum Senden von Nachrichten in den Betriebssystemkern. Der Prozess schreibt die Nachricht sowie die Adresse für die Nutzlast in einen Puffer und übergibt die Adresse des Puffers dem Systemaufruf *write*[4]. Der Betriebssystemkern liest die Nachricht und die besagte Adresse aus dem Puffer. Vielfach liegt jedoch die Nutzlast nicht in einem großen Block Speicher. Vielmehr ist sie über mehrere Indirektionen über den gesamten Speicherbereich verteilt, wie am Beispiel eines Tupels in Abb. 7.1 zu sehen ist. Deshalb muss zunächst jede Indirektion aufgelöst werden, für den entsprechenden Teil Speicher im Kern alloziert werden und abschließend der Inhalt kopiert werden. Danach kann die Nachricht nebst Nutzlast verarbeitet werden. Über die *receive*-Datei kann der Prozess mithilfe des Systemaufrufs *read*[5] Nachrichten vom Kern empfangen. Allerdings muss er dazu

[2] siehe Linux-Quellen *Documentation/timers/hrtimers.txt*
[3] siehe Linux-Quellen *Documentation/timers/hrtimers.txt*
[4] vgl. *man 2 write*
[5] vgl. *man 2 read*

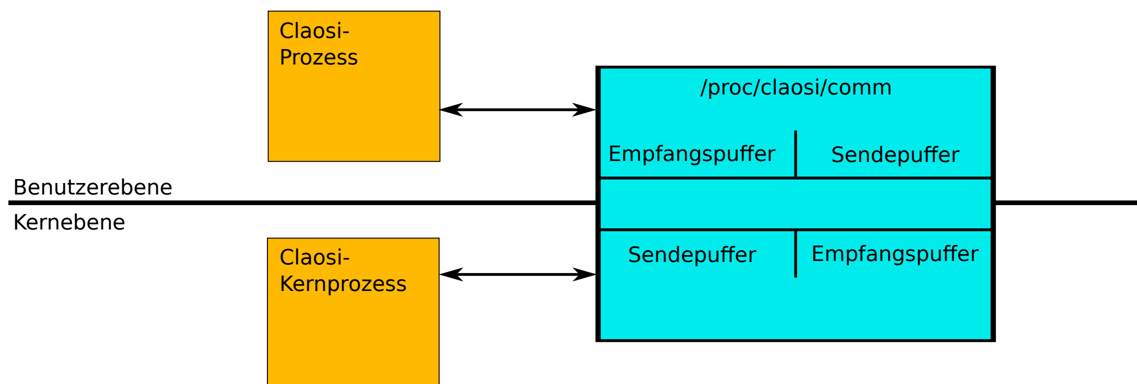


Abbildung 7.3: Aufbau der ebenenübergreifenden Kommunikation

dem Systemaufruf die Adresse eines hinreichend großen Puffers übergeben, in dem die gesamte Nachricht inklusive Nutzlast abgelegt werden kann. Die maximale Nachrichtenlänge ist jedoch nicht festgelegt, da beispielsweise die Größe eines Tupels je nach Inhalt schwanken kann. Hinzu kommt, dass der Benutzerprozess nicht auf den Adressraum des Kerns zugreifen kann. Somit ist ein Vorgehen wie beim Senden von Nachrichten in den Kern nicht möglich. Daher muss der Betriebssystemkern zunächst alle Daten in den bereitgestellten Puffer kopieren. Zur Reduktion des Mehraufwands durch den Systemaufruf ist es denkbar, Nachrichten zusammenzufassen und als Block zu übertragen. Dies hat allerdings Einfluss auf die Latenz einer einzelnen Nachricht. Sie wird im Zweifel erheblich verzögert in Abhängigkeit von der Gesamtzahl an Nachrichten, die mit einem Systemaufruf übertragen werden. Dieser Ansatz wurde jedoch aufgrund des Mehraufwands pro Systemaufruf und der Problematik mit der Nachrichtengröße beim Empfangen auf der Benutzerebene zugunsten einer asynchronen Lösung verworfen. Der zweite Lösungsansatz ist symmetrisch. Daher wird hier nur die Seite im Betriebssystemkern erläutert. Der Ausgangspunkt ist ein 65 Seiten umfassender gemeinsamer Speicher, der über eine virtuelle Datei im *proc*-Dateisystem auf die Benutzerebene exportiert wird. Der Benutzerprozess blendet die virtuelle Datei mithilfe des Systemaufrufs *mmap*[6] in seinen Adressraum ein. Jeweils 32 Seiten werden als Empfangs- bzw. Sendepuffer verwendet. Wie Abb. 7.3 zu entnehmen ist, sind die Bezeichnungen für die Puffer auf der Benutzer- und Kernebene vertauscht. In den Puffern werden nur die Nutzdaten einer Nachricht abgelegt. Der Nachrichtenaustausch selbst erfolgt über zwei Ringpuffer. Beide besitzen eine feste Größe. Ein Eintrag in dem Ringpuffer enthält lediglich den Typ der Nachricht sowie in Abhängigkeit vom Typ die Adresse innerhalb des Sendepuffers aus Sicht des Sendenden. Die Ringpuffer-Datenstrukturen liegen beide in der ersten Seite des gemeinsamen Speichers. Die Implementierung des Ringpuffers wurde so gewählt, dass er gegen den konkurrierenden Zugriff abgesichert ist.

Zum Empfangen von Nachrichten wird eigens dafür vorgesehener leichtgewichtiger Prozess erstellt. Er versucht aus dem Empfangsringpuffer zu lesen. Schlägt dies fehl, so legt er sich für eine definierbare Zeit schlafen und versucht es anschließend erneut.

[6]vgl. *man 2 mmap*

Ist jedoch eine Nachricht vorhanden, wird sie ausgelesen und interpretiert. Hängt der Nachricht eine Nutzlast an, müssen alle darin enthaltenen Zeiger zunächst umgeschrieben werden. Dies ist bedingt durch die Trennung des Benutzer- und Kernadressraums in Linux. Jeder dynamisch im Kern allozierter Speicher besitzt eine Adresse, die größer als drei Gigabyte ist. Dieser Bereich ist für den Benutzerprozess nicht zugreifbar. Der Adressraum des Benutzerprozesses ist im Kern grundsätzlich zugreifbar. Da jedoch das Lesen aus dem Ringpuffer in einem eigenen leichtgewichtigen Prozess erfolgt, ist der Adressraum des betreffenden Benutzerprozesses nicht eingeblendet. Deshalb muss auch in diesem Fall ein Umschreiben der Zeiger durchgeführt werden. Unter Umständen muss vor der Interpretation der Nachricht noch die Nutzlast in einen eigenen Speicherblock kopiert werden. Abschließend wird der Lesezeiger des Ringpuffers weitergesetzt und die entsprechende Nachricht damit als gelesen markiert.

Das Schreiben in den jeweiligen Senderingpuffer erfolgt synchron zum Kontrollfluss, der das Senden einer Nachricht erfordert. Zunächst wird, falls der Nachrichtentyp dies erfordert, aus dem Sendepuffer eine passende Menge Speicher alloziert. Dazu wird die Bibliothek *liballoc* verwendet [25]. Die Größe der Nutzlast ist abgesehen von der Größe des gemeinsamen Speichers nicht begrenzt. Nachdem der jeweilige Inhalt in den eben allozierten Bereich kopiert wurde, wird versucht die Nachricht in den Senderingpuffer zu schreiben. Schlägt dies fehl, wird aktiv gewartet bis die Nachricht geschrieben werden kann. Ein entsprechend dimensionierter Ringpuffer in Kombination mit einem leichtgewichtigen Prozess, der auf der Gegenseite kontinuierlich Nachrichten ausliest, führt zu einer geringen Wahrscheinlichkeit, dass der Senderpuffer vollläuft. Unmittelbar vor dem Schreiben in den Senderingpuffer wird, nachdem die Hälfte aller Elemente beschrieben wurde, über alle bereits gelesenen Einträge iteriert und der darin vermerkte Speicherbereich freigegeben. Somit ist gewährleistet, dass stets genug Speicher im Sendepuffer zur Verfügung steht.

Das vorgestellte Verfahren zur Kommunikation erlaubt beiden Komponenten vollständig unabhängig voneinander zu operieren. Es existieren keinerlei blockierende Synchronisationspunkte wie Systemaufrufe. Ausgehende Nachrichten können asynchron zum lesenden *Thread* abgesendet werden. Ferner wird das Problem der unbestimmten Nachrichtenlänge durch eine senderseitige dynamische Speicherallokation umgangen. Der Aufwand für einen Systemaufruf wird ebenfalls vermieden. Dies erfolgt allerdings auf Kosten eines leichtgewichtigen Prozesses, der regelmäßig abfragt, ob Nachrichten verfügbar sind. Als zukünftige Optimierung ist es möglich, die Schlafzeit der Last entsprechend zu regulieren. Damit wird die Aktivitätsdauer des lesenden Prozesses bei sehr geringer Last weiter reduziert. Abschließend ist noch zu erwähnen, dass neuere Versionen des Linux-Kerns über einen ähnlichen Kommunikationsmechanismus[7] verfügen. Hierbei wird eine Verbindung zum Betriebssystemkern über die sogenannte *netlink*-Schnittstelle aufgebaut. Die eigentliche Kommunikation erfolgt jedoch über einen im Benutzerprozess eingeblendeten Speicherbereich, der ebenfalls einen Ringpuffer enthält. Der Benutzerprozess wartet mithilfe des Systemaufrufes *poll*[8], bis der Kern den Eingang weiterer

[7] Siehe Linux-Quellen *Documentation/networking/netlink_mmap.txt*
[8] vgl. man 2 poll

Nachrichten signalisiert. Von dem Verfahren wurde jedoch Abstand genommen, da es eine feste Nachrichtenlänge benutzt und durch den Systemaufruf *poll* trotzdem Übergänge in den Betriebssystemkern erforderlich sind. Außerdem ist es nicht für den späteren Einsatz auf dem Betriebssystem *Android* geeignet, da die Geräte meist eine ältere Version des Linux-Kerns einsetzen, der diese Funktionalität noch nicht enthält. Dennoch beeinflusste diese Variante den Entwurf der oben vorgestellten Lösung.

7.3.3 Synchronisation des Datenmodells

Das *Claosi* muss ein systemweites Datenmodell verwalten, in dem alle Anbieter vermerkt sind. Da es eine Komponente im Betriebssystemkern und eine auf der Benutzerebene gibt, muss sichergestellt sein, dass beide eine konsistente Sicht auf das Datenmodell haben. Im Folgenden werden zwei Ansätze vorgestellt, die sich der Lösung dieses Problems widmen.

Ein geteiltes Datenmodell

Dieser Ansatz sieht ein physikalisches Datenmodell, das ebenfalls in dem bereits erwähnten gemeinsamen Speicher liegt. Aufgrund der Trennung der Adressräume zwischen Betriebssystemkern und Benutzerebene, wie sie in Abschnitt 7.3.2 bereits erwähnt wurde, dürfen die Zeiger innerhalb des Datenmodells nur Adressen relativ zum Start des gemeinsamen Speichers enthalten. Es muss bei jedem Zugriff auf das Datenmodell für jeden Zeiger zunächst die absolute Adresse durch Addition der Basisadresse berechnet werden. Die Synchronisation der beiden Komponenten beim Zugriff auf den Speicher erfolgt durch ein *Lock* auf Betriebssystemebene. Diese wird über eine virtuelle Datei im *proc*-Dateisystem für den Benutzerprozess exportiert [26]. Durch Schreiben einer Eins in die Datei seitens des Benutzerprozesses wird im Kern ein leichtgewichtiger Prozess gestartet, der stellvertretend für den Benutzerprozess das *Lock* hält. Der Systemaufruf kehrt allerdings erst zurück, wenn der Stellvertreter-Prozess das *Lock* geholt hat. Erst durch Schreiben einer Null in die gleiche Datei gibt der leichtgewichtige Prozess das *Lock* frei und terminiert. Auf diese Weise kann nur eine Komponente schreibend auf das Datenmodell zugreifen.

Ein verteiltes Datenmodell

Der verteilte Ansatz sieht zwei Datenmodelle vor. Jede Komponente besitzt ihre eigene Kopie sowie ihre eigenes *Lock* zur Absicherung der Operationen auf der Datenmodell. Die Anmeldung eines neuen Anbieters erfolgt zunächst auf der eigenen Ebene. Ist sie erfolgreich abgeschlossen, wird die Erweiterung des Datenmodells in den Sendepuffer kopiert und die andere Ebene mit einer Nachricht darüber informiert. Diese holt zunächst ihr eigenes *Lock* und prüft anschließend, ob das empfangene Datenmodell mit ihrem aktuellen Modell vereinbar ist. Nach erfolgreicher Prüfung wird die Erweiterung mit dem vorhandenen Datenmodell vereint. Abschließend wird das *Lock* wieder freigegeben. Das Abmelden eines Anbieter läuft analog ab: Hier wird zunächst der Anbieter lokal

abgemeldet und anschließend die andere Ebene darüber informiert. Diese Lösung birgt grundsätzlich das Risiko, dass das ein von beiden Datenmodellen nicht mehr synchron ist. Hierfür müssten zwei Erweiterungen - je eine auf einer Ebene - anmelden werden, die sich gegenseitig ausschließen. In diesem Fall würde erst die lokale Anmeldung erfolgen und anschließend die entfernte Anmeldung initiiert, welche fehlschlägt. Somit enthalten beide Ebenen, die bei ihnen angemeldete Erweiterung. Ihnen ist die jeweils andere jedoch nicht bekannt. Dieser Umstand stellt ein Problem dar, dessen Wahrscheinlichkeit als vernachlässigbar angesehen wird.

Fazit

Der Vorteil der ersten Lösung besteht in der Existenz genau eines Datenmodells, das zu jedem Zeitpunkt synchron ist. Die Wahrscheinlichkeit für ein inkonsistentes Datenmodell ist null. Die Art der Synchronisation ist als Nachteil zu werten, da eine Komponente blockiert, bis die andere ihre Arbeit an dem Datenmodell verrichtet hat. Ferner hat das Berechnen der absoluten Adressen zur Laufzeit Einfluss auf die Performanz. Die zweite Lösung hingegen erlaubt es beiden Komponenten unabhängig voneinander zu agieren. Jede Komponente arbeitet auf ihrer eigenen Kopie des Datenmodells. Daher wurde sie implementiert. Das Risiko eines inkonsistenten Datenmodells wird dabei in Kauf genommen.

7.3.4 Ebenenübergreifende Ausführung von Anfragen

Es wurde bereits mehrfach ausgeführt, dass es für eine Anfrage vollkommen transparent ist, auf welcher Ebene die einzelnen Knoten angesiedelt sind, die in einer Anfrage involviert sind. Daher ist es erforderlich, eine Anfrage in ihrer Ausführung zu pausieren und auf der anderen Ebene fortzusetzen, wenn beispielsweise die Verknüpfung mit einer Datenquelle erfolgen soll, die auf der aktuellen Ebene nicht vorhanden ist. Damit nicht bei jeder Unterbrechung der Ausführung die vollständige Anfrage mitgesendet werden muss, wird bei der Registrierung entschieden, ob die Anfrage ebenfalls auf der anderen Ebene zu registrieren ist. Hierüber entscheidet eine spezielle Funktion, die nach Belieben ersetzt werden kann. Die derzeitige Implementierung sendet jede Anfrage an die andere Ebene. Bei einer Anfrage deren Datenquelle sich auf der anderen Ebene befindet ist es z. B. unabdingbar, die Anfrage zu versenden, damit die Datenquelle überhaupt aktiviert werden kann. Die entfernte Registrierung wird die Anfrage lediglich in den internen Datenstrukturen abgelegt. Es erfolgt keine erneute Zuweisung einer Identifikationsnummer. Zusätzlich wird in der Anfrage vermerkt, dass eine entfernte Registrierung stattgefunden hat. Somit ist beim Löschen der Anfrage ersichtlich, dass die andere Ebene über diesen Vorgang auch zu benachrichtigen ist.

Die derzeitige Implementierung sieht vor, dass eine Anfrage solange auf einer Ebene ausgeführt wird, bis dies nicht mehr möglich ist. Ein Szenario, wo dies vorkommen kann, wurde eingangs bereits erläutert: Es handelt sich um die Verknüpfung eines Datenstroms mit einer weiteren Datenquelle aus dem Datenmodell. Die betreffende Quelle ist hierbei jedoch nicht auf der aktuellen Ebene angesiedelt. Des Weiteren findet eine Unterbrechung

statt, wenn die Anfrage ursprünglich von der anderen Ebene stammt und somit die Zustellung des Tupels an den Anfragenden dort erfolgen muss. Im schlimmsten Fall wird eine Anfrage so entworfen, dass nach der Ausführung jedes Operators die Ebene gewechselt wird. Dieses Szenario wird jedoch zunächst vernachlässigt. Es lässt Raum für weitere Optimierungen.

Der Zustand einer Anfrage umfasst u.a. eine eindeutige Identifikation der Anfrage. Es wird nicht die Adresse derselben Anfrage auf der anderen Ebene zur Identifikation benutzt, da es möglicherweise zu Wettkampfbedingungen zwischen dem Löschen einer Anfrage und dem Eintreffen des Auftrags zur Fortsetzung selbiger kommen kann. Stattdessen wird der Pfad im Datenmodell zur Datenquelle inklusive der Identifikationsnummer verwendet. Dies macht es bei Eintreffen der Nachricht erforderlich, die empfangene Beschreibung zu einer konkreten Adresse aufzulösen. Als weiteres Merkmal zur Definition des Zustands wird die Position des letzten, ausgeführten Operators mitgesendet. Damit ist es möglich bei der Fortsetzung der Ausführung, innerhalb der Anfrage vorzuspulen.

8 Evaluation

Nach der Präsentation der Implementierung des Werkzeugs *Claosi* soll in diesem Kapitel eine qualitative Bewertung durchgeführt werden. Dabei findet auch eine Einordnung gegenüber anderen Werkzeugen statt.

Im Abschnitt 8.1 werden die Gesichtspunkte dargestellt, unter denen die Evaluation durchgeführt wird. Der Abschnitt 8.2 stellt den Versuchsaufbau sowie die ermittelten Größe vor. Die Ergebnisse werden in Abschnitt 8.3 präsentiert. Das Kapitel schließt mit einer Gegenüberstellung der Ergebnisse und den Erwartungen in Abschnitt 8.4.

8.1 Ziele

Es wird eine makroskopische Betrachtung stattfinden. Hierbei soll die Laufzeit für einen Benchmark bestimmt werden. Parallel zu der Ausführung des Benchmarks werden im Hintergrund verschiedene Werkzeuge ausgeführt. Die verwendeten Benchmarks und Werkzeuge werden im folgenden Kapitel im Rahmen der Ausführungen zum Ablauf benannt. Aufgrund der generischen Verarbeitung der Daten durch *Claosi* ist zu erwarten, dass die Laufzeiten gegenüber der anderen Werkzeuge nicht geringer ausfällt. Die Bewertung soll die Kosten für eine generische Lösung ermitteln.

8.2 Durchführung

In diesem Abschnitt soll zum einen der technische Aufbau erläutert werden, zum anderen werden die eingesetzten Benchmarks vorgestellt. Des Weiteren werden die evaluierten Szenarien und ihre unterschiedlichen Konfigurationen dargelegt.

8.2.0.1 Technischer Aufbau

Die Evaluation wird auf einem handelsüblichen Arbeitsplatzrechner mit einem aktuellen Prozessor der Marke Intel durchgeführt. Der Prozessor besitzt vier Kerne, die auf 3,4 GHz getaktet sind. Die Taktrate ist während der gesamten Evaluation fixiert. Als Betriebssystem wird eine neue Ubuntu 14.04 Installation verwendet, die den Linux-Kern in Version 3.14.17 einsetzt. Da für manche Konfigurationen eine Datenübertragung von Nöten sein wird, ist der Rechner mit einem baugleichen Modell direkt über eine Gigabit-Schnittstelle verbunden. Pro Konfiguration werden 30 Durchläufe getätigt, um statische Schwankungen auszugleichen. Die Laufzeit eines Benchmark wird mit dem Linux-Werkzeug *time* bestimmt.

Werkzeug	Szenario A	Szenario B	Szenario C
SystemTap	x	x	x
PiCO QL	x	-	-
Claosi	x	x	x

Tabelle 8.1: Auflistung der evaluierten Konfigurationen

8.2.0.2 Benchmarks

Es werden zwei verschiedene Benchmark-Werkzeuge verwendet. Zum einen wird ein rein prozessorlastiger Benchmark zur Berechnung von Primzahlen verwendet. Es finden keinerlei Übergänge in das Betriebssystem seitens des Benchmarks statt. Er stammt aus der Werkzeugsammlung zur Bewertung von Betriebssystemparametern - genannt *SysBench* [27]. Er berechnet auf vier Kernen alle Primzahlen bis 650000.

Bei dem anderen Benchmark handelt es sich um das parallele Übersetzen des Linux-Kerns auf allen vier Prozessorkernen. Es wird der Linux-Kern in Version 3.14.17 mit der Standardkonfiguration aus einem aktuellen Debian-System verwendet. Dieser Benchmark zeichnet sich ebenfalls durch eine hohe Prozessorlast aus. Darüber hinaus erzeugt er beständig E/A-Last, die immer wieder für Sprünge in den Betriebssystemkern sorgt.

8.2.0.3 Szenarien

Es sollen drei verschiedene Situationen bewertet werden. In Tab. 8.1 wird eine Übersicht über die Szenarien gegeben.

Szenario A

In Szenario A sollen alle erzeugten Prozesse aufgezeichnet werden. Des Weiteren wird zu jedem Prozess der Programmname sowie die Rechenzeit, die der Prozess im Betriebssystemkern verbracht hat, bestimmt. In diesem Szenario werden alle drei Werkzeuge bewertet - vgl. Tab. 8.1. Zur Aufzeichnung der erzeugten Prozesse mit *SystemTap* wird ein Marker auf die Rückkehr der Funktion *do_fork* in Linux-Kern registriert. Bei Auslösen des Markers werden die gewünschten Informationen bestimmt. Eine Auflistung des Quellcodes hierfür ist im Anhang A.2 zu finden. Bei *PiCO QL* kann nicht auf Ereignisse reagiert werden. Daher wird in diesem Fall in Intervallen von 0-5 Sekunden eine entsprechende Anfrage an *PicoQL* gestellt. Die zur Aufzeichnung nötige Anfrage für *Claosi* ist in Listing 8.1 dargestellt.

Listing 8.1: Anfrage zum Mitschnitt von erzeugten Prozessen

```
1 x = Object("process.process",OBJECT_CREATE,0)
2 x =
    Join(x,"process.process.comm",Stream.process.process==Join.process.process)
3 x =
    Join(x,"process.process.stime",Stream.process.process==Join.process.process)
```

Szenario B und C

In den Szenarien B und C sollen die ein- und ausgehenden Netzwerkpakete aufgezeichnet werden. Zu diesem Zweck findet während aller Ausführungen im Hintergrund eine kontinuierliche Datenübertragung zu dem zweiten Rechner statt. Damit der Rechenaufwand für die Übertragung möglichst gering gehalten wird, werden mit dem Werkzeug *netcat* lediglich Nullen übertragen.

Die Unterschiede zwischen den Szenarien B und C lassen sich anhand der Anfragen für *Claosi* am besten aufzeigen. In Szenario B wird zu jedem Paket der korrespondierende Prozess ermittelt - vgl.Listing 8.2. Im Gegensatz dazu werden in Szenario C nur die Netzwerkpakete selbst aufgezeichnet - vgl. Listing 8.3. Außerdem findet eine Filterung nach der Paketgröße statt. Die Zeichenkette „p4p1“bezeichnet den systeminternen Namen der Netzwerkschnittstelle dessen Verkehr aufgezeichnet werden soll. Gleichzeitig handelt es sich um eben jene Schnittstelle, die direkt mit dem zweiten Rechner verbunden ist.

Listing 8.2: Anfrage zum Mitschnitt ein- und ausgehender Netzwerkpakete sowie der dazugehörigen Prozesse

```
1  v = Event(net.device[p4p1].onTx,0)
2      Join(v,process.process[*].sockets, Stream.net.packetType.socket ==
           Join.process.process.sockets)
3
4  x = Event(net.device[p4p1].onRx,0)
5      Join(x,process.process[*].sockets, Stream.net.packetType.socket ==
           Join.process.process.sockets)
```

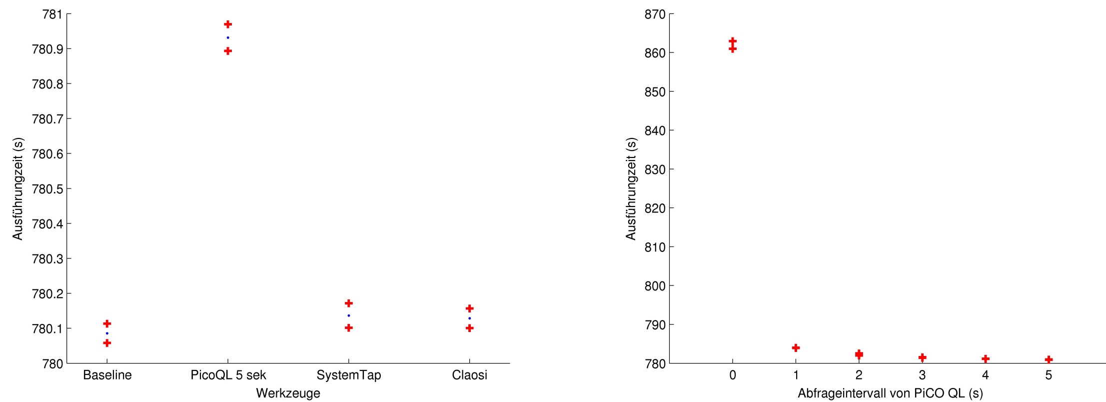
Da *PiCO QL* keine Möglichkeit für den Zugriff auf den Netzwerkverkehr bietet, wird es hierbei nicht betrachtet. Für *SystemTap* wurden Skripte äquivalent zu den dargestellten Anfragen geschrieben, die Marker an verschiedenen Stellen im Kern registrieren, um die Datenpakete aufzuzeichnen. Das genaue Listing hierzu ist im Anhang A.3 respektive Anhang A.4 aufgeführt.

Listing 8.3: Anfrage zum Mitschnitt ein- und ausgehender Netzwerkpakete

```
1  v = Event(net.device[p4p1].onTx,0)
2  v = Filter(v,net.packetType.dataLength >= 1000)
3      Select(v,net.packetType)
4
5  x = Event(net.device[p4p1].onRx,0)
6  x = Filter(x,net.packetType.dataLength >= 10)
7      Select(x,net.packetType)
```

8.3 Ergebnisse

Im folgenden Abschnitt werden die Messergebnisse der einzelnen Szenarien vorgestellt. Zu jeder Messreihe wurde das 95%-Konfidenzintervall berechnet. In den Diagrammen in den folgenden Abschnitten sind die Grenzen des Konfidenzintervalls mit roten Kreuzen gekennzeichnet. Anstatt alle Messwerte darzustellen wurde jeweils der Mittelwert im



(a) Laufzeiten bei Nutzung der verschiedenen Werkzeuge

(b) Laufzeiten bei verschiedenen Abfrageintervallen von *PiCO QL*

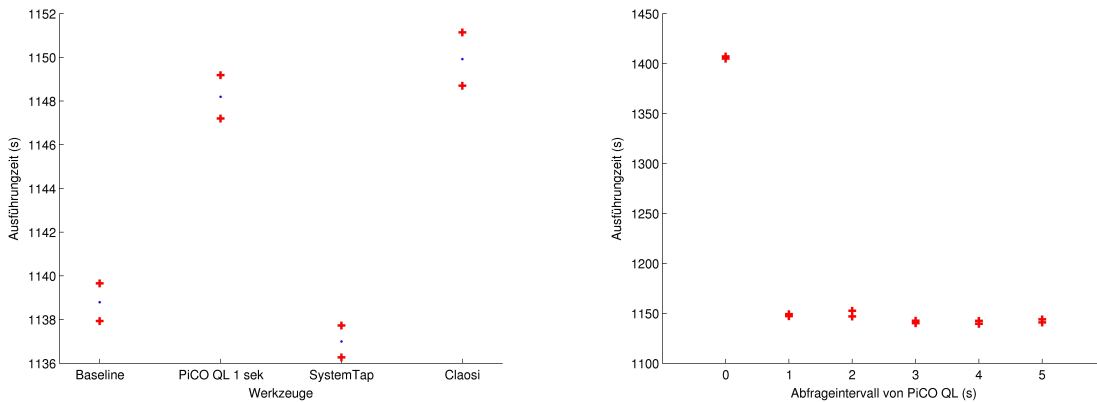
Abbildung 8.1: Laufzeiten des Benchmarks *SySBench* für Szenario A

Diagramm abgebildet. Dieser ist durch einen blauen Punkt zu erkennen. Auf der Y-Achse ist die Ausführungszeit in Sekunden abgetragen. Auf der X-Achse werden die einzelnen Werkzeuge abgebildet. Für die unterschiedlichen Abfrageintervalle von *PiCO QL* wurden jeweils nur 20 Messwerte erhoben. Da die Messwerte wenig streuen, spielt diese Tatsache nur eine untergeordnete Rolle.

Szenario A

Bei einer ersten Betrachtung der Ergebnisse in Abb. 8.1a fällt auf, dass die Laufzeiten für die jeweiligen Werkzeuge nicht stark voneinander abweichen. Allerdings ist in dieser Abbildung für das Werkzeug *PiCO QL* nur der Wert für ein Abfrageintervall von 5 Sekunden aufgetragen. Betrachtet man hingegen die Entwicklung der Laufzeiten für verschiedene Intervalle in Abb. 8.1b, so fällt auf, dass die Laufzeit bei einem Intervall 0 Sekunden um rund 11 % höher liegt. Dies bedeutet, dass erst ab einem Intervall von 5 Sekunden eine vergleichbare Laufzeit erreicht wird. Jedoch können dann nicht alle neu erzeugten Prozesse aufgezeichnet werden. Ganz im Gegensatz zu den Werkzeugen *SystemTap* und *Claosi*. Bezogen auf einen Durchlauf ohne Instrumentierung weisen beide Werkzeuge einen Mehraufwand von unter einem Prozent auf. Hinsichtlich der Bewertung von *Claosi* ist daher festzuhalten, dass es keinen nennenswerten Mehraufwand erzeugt. Dies gilt jedoch nur für die Beobachtung des Leerlaufbetriebs, da der Benchmark *SySBench* selbst keine neuen Prozesse erzeugt. Wobei der Begriff Leerlaufbetrieb nicht den Zustand des gesamten Systems beschreibt, sondern lediglich die Frequenz, mit der neue Prozesse erzeugt werden.

Anders stellt sich der Sachverhalt bei dem zweiten Benchmark dar. Hierbei werden, während des Übersetzen der Linux-Kerns, laufend neue Prozesse erzeugt. Dies spiegelt sich in den Laufzeiten, wie in Abb. 8.2a zu sehen ist, wider. Die Werkzeuge *Claosi* und *PiCO QL* erzeugen einen Mehraufwand von 0,97 % respektive 0,82 %. In Abb. 8.2b ist zu



(a) Laufzeiten bei Nutzung der verschiedenen Werkzeuge

(b) Laufzeiten bei verschiedenen Abfrageintervallen von *PiCO QL*

Abbildung 8.2: Laufzeiten des Benchmarks „Linux-Kern übersetzen“für Szenario A

erkennen, dass sich die Laufzeit für das Werkzeug *PiCO QL* nicht weiter abflacht, sondern auf einem Niveau verbleibt. Dies ist vermutlich damit zu begründen, zur Ausführung der Anfragen die Ausführung auf allen Kernen unterbrochen wird. Da das Übersetzen eines Linux-Kern vergleichsweise viel Interaktion mit dem Betriebssystemkern erfordert, äußert sich das genannte Vorgehen in dieser Form.

Szenario B

Bei ersten Tests zur Funktionsfähigkeit des Programmcodes für dieses Szenario stellte sich heraus, dass die Anfrage für das Werkzeug *Claosi* zu aufwendig ist, um eine Datenrate von 1 Gbit/s zu bewältigen. Dies äußerte sich darin, dass die Liste mit den abzuarbeitenden Anfragen stetig anwuchs. Zusätzlich stieg der Speicherverbrauch kontinuierlich an, da zu jeder ausstehende Anfrage ein Tupel gehört, das ein ein- oder ausgehendes Netzwerkpaket repräsentiert. Auch das Herabsenken der Verbindungsgeschwindigkeit auf 100 Mbit/s ergab keine Änderung in dem beobachtetem Verhalten. Erst eine Reduktion auf 10 Mbit/s erlaubte es dem Werkzeug *Claosi* die ausstehenden Anfragen abzuarbeiten. Da sich bei diesen Bedingungen adäquaten Vergleichswerte erzeugen lassen, wurde für dieses Szenario von einer Bewertung abgesehen.

Szenario C

In Abb. 8.3 sind die Messergebnisse für Szenario C aufgetragen. Hier fällt deutlich der gestiegene Mehraufwand von *Claosi* auf. Dieser beträgt gegenüber dem Referenzlauf 14,146%. Wohingegen das Werkzeug *SystemTap* nur eine Laufzeitsteigerung von 3,71% aufweist. Allerdings zeigt der Vergleich der Referenzmessungen von Szenario A und C, dass allein die permanente Datenübertragung die Laufzeit um 9,66 % erhöhen kann.

Für den zweiten Benchmark fällt der Unterschied zwischen *SystemTap* und *Claosi*

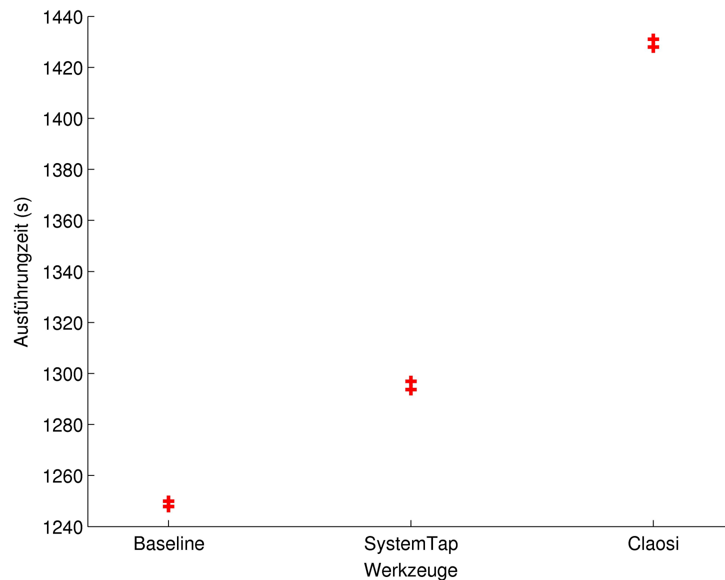


Abbildung 8.3: Laufzeiten des Benchmarks „Linux-Kern übersetzen“für Szenario C

ähnlich deutlich aus - vgl. Abb. 8.4. Das Werkzeug *Claosi* benötigt für einen Durchlauf ca. 4,74% länger als der Referenzlauf. Das Werkzeug *SystemTap* weist lediglich einen Unterschied von 0,5% auf. Auch hier fällt die Differenz zwischen den Referenzläufen aus Szenario A und C mit rund 4 % auf. Dies unterstreicht noch einmal unter welcher Last die Werkzeuge agieren müssen.

8.4 Fazit

Die Messergebnisse haben gezeigt, der generische Ansatz, wie das Werkzeug *Claosi* ihn verwendet, Mehraufwand mit sich bringt. Allerdings ist dies nicht in dem Umfang der Fall, wie zu erwarten wäre - zumindest für einfache Anfragen wie in Szenario A. Die Tatsache, dass für Szenario B keine Messwerte erhoben werden konnten, deutet darauf hin, dass noch viel Raum für Optimierungen ist. Erste Ansatzpunkte für Verbesserungen werden daher in Abschnitt 9.3 aufgezeigt. Das Szenario C zeigte, dass entsprechende Hochlastszenarien die Kosten einer generischen Lösung gut demonstrieren können.

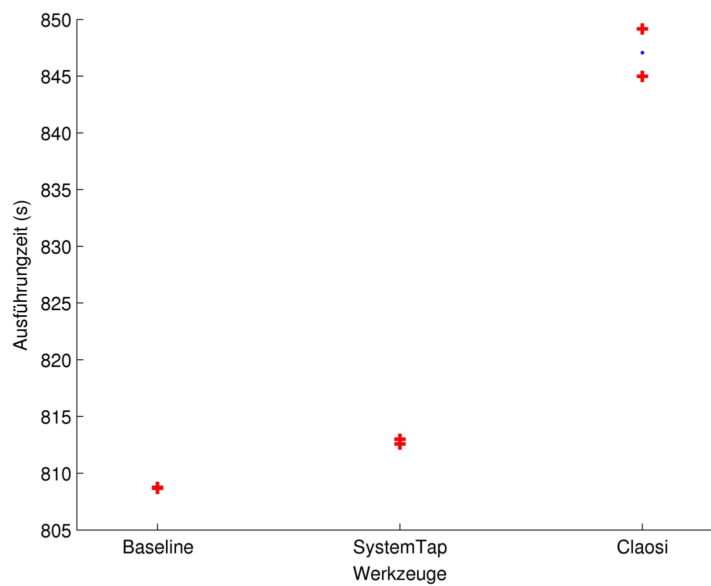


Abbildung 8.4: Laufzeiten des Benchmarks *SySBench* für Szenario C

9 Diskussion

Nach der reinen quantitativen Bewertung der Implementierung im vorangegangen Kapitel soll es nun um die Auseinandersetzung mit Entwurfsentscheidungen gehen. Ferner werden Anknüpfungspunkte für weitere Arbeiten über den Rahmen dieser Arbeit hinaus aufgezeigt.

Es soll zunächst ein Blick auf den Umgang mit Lernergebnissen durch das Werkzeug *Claosi* geworfen werden. Das Datenmodell nebst Anfragesprache wird in Abschnitt 9.2 begutachtet. Abschließend werden in Abschnitt 9.3 Verbesserungen bei der Implementierung vorgestellt.

9.1 Bereitstellung von Lernergebnissen

In welchem Umfang Lernergebnisse durch *Claosi* im Betriebssystem zur Verfügung stehen wurde bisher weder in Kapitel 5 noch in Kapitel 7 explizit erwähnt. Dies ist in der Struktur von Datenmodellen sowie der Programmierschnittstelle begründet. Es bedarf keiner gesonderten Behandlung. Das Datenmodell differenziert nicht zwischen inferierten und bereits existierenden Daten. Es gibt lediglich Datenquellen.

Zur Bereitstellung von Lernergebnissen genügt die Implementierung als ein gewöhnlicher Anbieter. Dieser registriert sich über die bekannte Programmierschnittstelle. Die inferierten Daten werden in Folge dessen als Erweiterung des Datenmodells im System bekanntgemacht.

9.2 Datenmodell und Anfragesprache

Die entworfene Anfragesprache operiert auf Datenströmen und berücksichtigt so den veränderlichen Charakter von Betriebssystemdaten. Allerdings ist rückblickend festzuhalten, dass für Betriebssystemdaten weder ein rein datenstrom-basierter noch ein rein relationaler Ansatz geeignet ist. Vielmehr sieht eine Kombination aus beidem vielversprechend aus. Denn obwohl die Daten in einem Betriebssystem sich ständig verändern und somit ein kontinuierlicher Datenstrom entsteht, besteht zwischen den Elementen in einem Betriebssystem ein relationaler Zusammenhang. Die Problematik wird durch das folgende Beispiel verdeutlicht:

Die Verknüpfung eines Datenstroms mit einem Objekt ist bisher so definiert, dass bei einer erfolgreichen Verknüpfung nur der Objektbezeichner zu dem Datenstrom hinzugefügt wird. Dies dient der Reduktion des Verarbeitungsaufwands. Würden stattdessen alle Kindelemente eines Objekts dem Strom hinzugefügt, müsste, da Objekte beliebig

verschachtelt werden können, zunächst die Potenzmenge aller Objektinstanzen gebildet werden. Sollen in einem Datenstrom die verschiedenen Kindelemente eines Objektes hinzugefügt werden, muss aufgrund dieser Entwurfsentscheidung für jedes Kind die Anfrage um einen weiteren *Join*-Operator ergänzt werden.

Die Semantik des *Join*-Operators in *SQL* würde die Anfrage erheblich verkleinern. Die Semantik sieht vor, dass zunächst alle Spalten der verknüpften Relation hinzugefügt. Jedoch führt dies zu dem bereits erwähnten Mehraufwand bei der Verarbeitung.

Ein möglicher Kompromiss besteht in dem automatischen Hinzufügen aller Kindelemente des Objektes, die selbst keine Objekte sind. Für verschachtelte Objekte muss eine separate Verknüpfung erfolgen.

9.3 Implementierung

Die Realisierung des Werkzeugs *Claosi* besteht auf vielen, kleinen Bausteinen, die alle implementiert worden sind. Dabei wurde jedoch oftmals in erster Näherung eine naive Implementierung gewählt, um einen lauffähiges Gesamtprojekt zu erzeugen. Dies führte mutmaßlich bei der Evaluation zu dem deutlichen Abstand zwischen der derzeitigen Implementierung und den übrigen Werkzeugen. Im Folgenden sollen nun drei Stellen aufgezeigt werden, die Optimierungspotential bieten.

9.3.1 Zugriff auf das Datenmodell

Der Zugriff auf das Datenmodell ist ein essentieller Bestandteil der *Claosi*. Dementsprechend häufig findet er statt. Dies gilt insbesondere für Operationen auf Tupeln, da an dieser Stelle das Datenmodell zur Auflösung der Datentypen herangezogen wird. Daher ist zu erwarten, dass eine Verbesserung entsprechenden Einfluss hat. Bisher geschieht der Zugriff durch stückweises Auflösen eines Pfades mithilfe von Zeichenkettenoperationen. Aus diesem Grund soll eine Idee zur Lösung dieses Problems aufgezeigt werden:

Der erste Ansatz sieht das Ablegen aller häufig abgerufenen Elemente in einem Zwischenspeicher vor. Dazu wird der Pfad sowie ein Zeiger auf die Datenstruktur im Zwischenspeicher vermerkt. Die Indizierung geschieht anhand des Pfads oder durch eine Prüfsumme, die über die Zeichenkette gebildet wird. Bei einem hinreichend dimensionierten Speicher entfällt somit das Untersuchen der Pfadangabe. Je nach Indizierung sind evtl. einfache Vergleiche von Zeichenketten erforderlich.

Alternativ kann die Prüfsumme nur über einen Teil des Pfades gebildet werden. Hierbei wird der Zeiger auf einen Teilbaum im Zwischenspeicher abgelegt. Diese Lösung begünstigt Zugriffe auf verschiedene Elemente eines Teilbaums.

9.3.2 Realisierung des Join-Operators

Die Anfragen, die zwei Datenquellen miteinander verknüpfen, schneiden bei der Evaluation besonders schlecht ab - vgl. Listing 8.2. Der Grund ist die aktuelle Implementierung des *Join*-Operators in Kombination mit der Art der Anfrage. Da in der Anfrage nicht

enthalten ist, zu welchem Prozess ein *Socket* gehört, muss bei jeder Ausführung eine Liste von Tupeln erstellt werden, die alle *Sockets* für alle Prozesse beinhaltet. In der Folge wird bei jedem Tupel aus dieser Liste geprüft, ob die angegebenen Prädikate anwendbar sind. Zur Verminderung des Aufwands können zwei Ansätze gewählt werden.

Eine Möglichkeit besteht in der Reduktion der Frequenz des Abfragens der Datenquelle. Dazu müssen die Werte zwischengespeichert werden. Parallel dazu muss im Betriebssystem beobachtet werden, ob Verbindungen geöffnet bzw. geschlossen werden. Ist dies der Fall, müssen die zwischengespeicherten Werte aktualisiert werden.

Eine weitere Möglichkeit ist, der Abfrage weitere Kontextinformationen mitzugeben, so dass von vornherein eine kleinere Liste von Tupeln generiert wird.

9.3.3 Priorisierung von Anfragen

Die Ausführung von Anfragen erfolgt derzeit nach dem Schema *First-come, first-served*. Jedoch ist es denkbar, dass je nach Szenario eine Priorisierung einzelner Anfragen oder gar ganzer Gruppen von Anfragen erfolgt. In Anbetracht der Evaluationsergebnisse erscheint es durchaus sinnvoll, Anfragen zu hochfrequenten Ereignissen Vorrang einzuräumen. Dies vermindert die Latenz einzelner Anfragen. Zusätzlich wird der Speicherplatz für die Tupel schneller wieder freigegeben, da diese zügiger verarbeitet werden.

Weiterhin können Anfragen mit einer hohen Ausführungszeit als verdrängbar kennzeichnet werden und ermöglichen so die kurzzeitige Unterbrechung, um kurze oder wichtigere Anfragen vorzulassen.

10 Zusammenfassung

In der vorliegenden Arbeit wurden die verschiedenen Bestandteile zur Umsetzung eines Werkzeugs zur ebenenübergreifenden Bereitstellung von Betriebssystemdaten und inferierten Daten dargestellt. Dabei sind die vier in Kapitel 4 dargelegten Aspekte von *Claosi* berücksichtigt worden. Dazu gehört zunächst ein Metamodell zur Modellierung von Daten in einem Betriebssystem. Dies ist so gestaltet worden, dass sowohl existierende als auch inferierte Daten dargestellt werden können. Die erzeugten Datenmodelle abstrahieren von dem zugrunde liegenden Programmcode, so dass eine Beschränkung auf das Wesentliche möglich ist. Damit ist die Abbildung der Datenquellen sowie der Zusammenhänge zwischen den Daten gemeint. Zusätzlich bietet es die Freiheit, eigene Datentypen zu definieren.

Darauf aufbauend wurde eine datenstromorientierte Anfragesprache entworfen. Die Anfragen beinhalten den vollständigen Fluss der Daten von der Quelle über die die Verarbeitung hin zum Ergebnis. Die drei unterschiedlichen Datenquellen sind ebenso in der Sprache vorhanden wie verschiedene Operatoren. Dazu zählt die Verknüpfung von Datenquellen, wie an den beiden Fallbeispielen deutlich wurde. Zusätzlich können die Tupel eines Datenstroms gefiltert werden und einzelne Daten aus einem Tupel selektiert werden. In Abschnitt 9.2 wurde bereits mit den Einschränkungen der Sprache zukünftige Forschungsthemen motiviert.

Die Implementierung erfolgte als Modul für den Linux-Kern. Somit kann es als integraler Bestandteil der Systemsoftware angesehen werden. Alle Bestandteile einer Anfrage werden zur Laufzeit ausgewertet und auf den Daten angewendet. Ferner erfolgt die Auflösung der Datentypen in einem Tupel ebenfalls zur Laufzeit anhand des Datenmodells. Damit ist eine vollständig generische Lösung zur Erhebung von Betriebssystemdaten gegeben. Allerdings wurden im Rahmen der Evaluation die Grenzen der derzeitigen Implementierung aufgezeigt. Daraus wurden in Abschnitt 9.3 weitere Aspekte für zukünftige Arbeiten abgeleitet.

Das Werkzeug *Claosi* wurde so gestaltet, dass es für eine Anfrage vollkommen transparent ist, von welcher Ebene die beteiligten Daten stammen. Es regelt selbständig den ebenenübergreifenden Transport von Anfragen respektive Tupeln. Darüber hinaus bietet es auf beiden Ebenen Programmierschnittstellen an, die von neuen Anbietern genutzt werden können, um Erweiterungen für das Datenmodell oder Anfragen zu registrieren.

Abschließend ist festzuhalten, dass die Ziele der vorliegenden Masterarbeit erfüllt sind.

Literaturverzeichnis

[1] TU Dortmund: *Collaborative Research Center SFB 876 - Providing Information by Resource-Constrained Data Analysis*. http://sfb876.tu-dortmund.de/index.html. Version: 09 2014

[2] Weiser, Mark: The Computer for the 21st Century. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 3 (1999), Juli, Nr. 3, 3–11. http://dx.doi.org/10.1145/329124.329126. – DOI 10.1145/329124.329126. – ISSN 1559–1662

[3] Gartner, Inc: *Gartner Says Worldwide Tablet Sales Grew 68 Percent in 2013*. http://www.gartner.com/newsroom/id/2674215. Version: 09 2014

[4] IDC Corporate USA: *Worldwide Smartphone Shipments Top One Billion Units for the First Time, According to IDC* . http://www.idc.com/getdoc.jsp?containerId=prUS24645514. Version: 09 2014

[5] Ferreira, Denzil ; Dey, AnindK. ; Kostakos, Vassilis: Understanding Human-Smartphone Concerns: A Study of Battery Life. Version: 2011. http://dx.doi.org/10.1007/978-3-642-21726-5_2. In: Lyons, Kent (Hrsg.) ; Hightower, Jeffrey (Hrsg.) ; Huang, ElaineM. (Hrsg.): *Pervasive Computing* Bd. 6696. Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–21726–5_2. – ISBN 978–3–642–21725–8, 19-33

[6] Red Hat, Inc.: *SystemTap*. https://sourceware.org/systemtap/. Version: 09 2014

[7] Prasad, Vara ; Eigler, Frank C. ; Keniston, Jim ; Cohen, William ; Hunt, Martin ; Chen, Brad: Locating System Problems Using Dynamic Instrumentation.

[8] Oracle Corporation: *Solaris Dynamic Tracing Guide*. http://docs.oracle.com/cd/E19253-01/817-6223/. Version: 09 2014

[9] Erlingsson, Úlfar ; Peinado, Marcus ; Peter, Simon ; Budiu, Mihai ; Mainar-Ruiz, Gloria: Fay: Extensible Distributed Tracing from Kernels to Clusters. In: *ACM Trans. Comput. Syst.* 30 (2012), November, Nr. 4, 13:1–13:35. http://dx.doi.org/10.1145/2382553.2382555. – DOI 10.1145/2382553.2382555. – ISSN 0734–2071

[10] Fragkoulis, Marios ; Spinellis, Diomidis ; Louridas, Panos ; Bilas, Angelos: Relational Access to Unix Kernel Data Structures. In: *Proceedings of the Ninth*

European Conference on Computer Systems. New York, NY, USA : ACM, 2014 (EuroSys '14). – ISBN 978–1–4503–2704–6, 12:1–12:14

[11] *SQLite*. http://www.sqlite.org/. Version: 09. – 2014

[12] Athivarapu, Pavan K. ; Bhagwan, Ranjita ; Guha, Saikat ; Navda, Vishnu ; Ramjee, Ramachandran ; Arora, Dushyant ; Padmanabhan, Venkat N. ; Varghese, George: RadioJockey: Mining Program Execution to Optimize Cellular Radio Usage. In: *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA : ACM, 2012 (Mobicom '12). – ISBN 978–1–4503–1159–5, 101–112

[13] Vergara, E.J. ; Sanjuan, J. ; Nadjm-Tehrani, S.: Kernel level energy-efficient 3G background traffic shaper for android smartphones. In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, 2013, S. 443–449

[14] Ikebe, Y. ; Nakayama, T. ; Katagiri, M. ; Kawasaki, S. ; Abe, H. ; Shinagawa, T. ; Kato, K.: Efficient Anomaly Detection System for Mobile Handsets. In: *Emerging Security Information, Systems and Technologies, 2008. SECURWARE '08. Second International Conference on*, 2008, S. 154–160

[15] Negi, A. ; Kishore, K.P.: Applying Machine Learning Techniques to Improve Linux Process Scheduling. In: *TENCON 2005 2005 IEEE Region 10*, 2005, S. 1–6

[16] Streicher, Jochen: Data Modeling of Ubiquitous System Software. Version: 07 2014. http://sfb876.tu-dortmund.de/PublicPublicationFiles/streicher_2014a.pdf. 2014. – Forschungsbericht

[17] Benvenuti, Christian: *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2005. – ISBN 0596002556

[18] Microsoft Corporation: *LINQ: .NET Language-Integrated Query*. http://msdn.microsoft.com/en-us/library/bb308959.aspx. – [abgerufen 25.08.2014]

[19] Microsoft Corporation: *LINQ: .NET Language-Integrated Query*. http://research.microsoft.com/en-us/projects/dryadlinq/. – [abgerufen 26.08.2014]

[20] Madden, Samuel R. ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei: TinyDB: An Acquisitional Query Processing System for Sensor Networks. In: *ACM Trans. Database Syst.* 30 (2005), M$\hat{}$b arz, Nr. 1, 122–173. http://dx.doi.org/10.1145/1061318.1061322. – DOI 10.1145/1061318.1061322. – ISSN 0362–5915

[21] Abadi, Daniel J. ; Carney, Don ; Çetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Aurora: A New Model and Architecture for Data Stream

Management. In: *The VLDB Journal* 12 (2003), August, Nr. 2, 120–139. http://dx.doi.org/10.1007/s00778-003-0095-z. – DOI 10.1007/s00778–003–0095–z. – ISSN 1066–8888

[22] VARGHESE, George ; LAUCK, Anthony: Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. In: *IEEE/ACM Trans. Netw.* 5 (1997), Dezember, Nr. 6, 824–834. http://dx.doi.org/10.1109/90.650142. – DOI 10.1109/90.650142. – ISSN 1063–6692

[23] GLEIXNER, Thomas ; NIEHAUS, Douglas: Hrtimers and Beyond: Transforming the Linux Time Subsystems. In: *Proceedings of the Linux Symposium, Ottawa, Canada* Bd. 1, 2006, 333–346

[24] BOVET, Daniel ; CESATI, Marco: *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. – ISBN 0596005652

[25] *liballoc - a memory allocator for hobbyist operating systems*. https://github.com/blanham/liballoc. Version: 09 2014

[26] LOVE, Robert: *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. – ISBN 0672329468, 9780672329463

[27] *SysBench: a system performance benchmark*. https://launchpad.net/sysbench. Version: 09 2014

Abbildungsverzeichnis

4.1 Struktur des *Claosi* für das Betriebssystem *Android* 12

6.1 Hierarchie der Datenquellen 31

7.1 Struktur eines Tupels 45
7.2 Hierarchie der Datenstrukturen zur Darstellung von Anfragen 46
7.3 Aufbau der ebenenübergreifenden Kommunikation 50

8.1 Laufzeiten des Benchmarks *SySBench* für Szenario A 58
8.2 Laufzeiten des Benchmarks „Linux-Kern übersetzen“für Szenario A . . . 59
8.3 Laufzeiten des Benchmarks „Linux-Kern übersetzen“für Szenario C . . . 60
8.4 Laufzeiten des Benchmarks *SySBench* für Szenario C 61

Tabellenverzeichnis

7.1 Übersicht über die implementierten Operatoren 46
8.1 Auflistung der evaluierten Konfigurationen 56

List of Listings

2.1 Beispiel für ein *SystemTap*-Skript 3

5.1 Metamodell zur Beschreibung eines Datenmodells 17
5.2 Datenmodell für das Fallbeispiel „Paketverzögerung“ 20
5.3 Datenmodell für das Fallbeispiel „Automatische Umschaltung zwischen 2G und 3G“ 21

6.1 Beispiel für ein *Language Integrated Query* (entnommen aus [18]) 27
6.2 Auszug aus einem Beispiel für eine *FayLINQ*-Anfrage (entnommen aus [9]) 28
6.3 Exemplarische Anfrage an ein Sensornetzwerk (entnommen aus [20]) . . . 29
6.4 Anfragen für die Lernphase des Fallbeispiels „Paketverzögerung“ 37
6.5 Anfragen zur Systemkontrolle für das Fallbeispiel „Paketverzögerung“ . . 38
6.6 Anfrage für das Fallbeispiel „Automatische Umschaltung zwischen 2G und 3G“ 38

7.1 captionDefinition der Datenstruktur *DatenModelElement* 43
7.2 Vererbung in *C* 46

8.1 Anfrage zum Mitschnitt von erzeugten Prozessen 56
8.2 Anfrage zum Mitschnitt ein- und ausgehender Netzwerkpakete sowie der dazugehörigen Prozesse 57
8.3 Anfrage zum Mitschnitt ein- und ausgehender Netzwerkpakete 57

A.1 *C*-Datenstruktur zur Repräsentation einer Anfrage 79
A.2 *SystemTap*-Skript zur Aufzeichnung von neu erzeugten Prozessen 80
A.3 *SystemTap*-Skript zur Aufzeichnung des Netzwerkverkehrs sowie des korrespondierenden Prozesses 81
A.4 *SystemTap*-Skript zur Aufzeichnung des Netzwerkverkehrs 84

Anhang

A.1 Definitionen der Datenstrukturen

Listing A.1: *C*-Datenstruktur zur Repräsentation einer Anfrage

```
1  typedef struct __attribute__((packed)) Query {
2      struct Query *next;
3      Operator_t *root;
4      unsigned short flags;
5      unsigned short idx;
6      unsigned int size;
7      unsigned int layerCode;
8      unsigned int queryID;
9      queryCompletedFunction onQueryCompleted;
10 } Query_t;
11
12 typedef struct __attribute__((packed)) GenStream {
13     Operator_t base;
14     char urgent;
15     DECLARE_BUFFER(name)
16     Selector_t *selectors;
17     int selectorsLen;
18 } GenStream_t;
19
20 typedef struct __attribute__((packed)) EventStream {
21     GenStream_t streamBase;
22 } EventStream_t;
23
24 typedef struct __attribute__((packed)) SourceStream {
25     GenStream_t streamBase;
26     int period;
27     void *timerInfo;
28 } SourceStream_t;
29
30 typedef struct __attribute__((packed)) ObjectStream {
31     GenStream_t streamBase;
32     int objectEvents;
33 } ObjectStream_t;
34
35 typedef struct __attribute__((packed)) Predicate {
36         unsigned short flags;
37         unsigned short type;
38         Operand_t left;
39         Operand_t right;
```

```
40 } Predicate_t;
41
42 typedef struct __attribute__((packed)) Filter {
43     Operator_t base;
44     unsigned short predicateLen;
45     Predicate_t **predicates;
46 } Filter_t;
47
48 typedef struct Select {
49     Operator_t base;
50     unsigned short elementsLen;
51     Element_t **elements;
52 } Select_t;
53
54 typedef struct __attribute__((packed)) Sort {
55     Operator_t base;
56     unsigned short elementsLen;
57     Element_t **elements;
58     unsigned short sizeUnit;
59     unsigned int size;
60 } Sort_t;
```

A.2 SystemTap-Skripte für die Evaluation

A.2.1 Aufzeichnung der Prozesserzeugung

Listing A.2: *SystemTap*-Skript zur Aufzeichnung von neu erzeugten Prozessen

```
1 %{
2 #include <linux/jiffies.h>
3 %}
4
5 function jiffies_to_usecs(value:long) %{
6     STAP_RETVALUE = jiffies_to_usecs(STAP_ARG_value);
7 %}
8
9 probe kernel.function("do_fork").return {
10     task = pid2task($return);
11
12     if (task != 0) {
13         printf("process␣%d(retval=%d)␣created␣with␣stime␣%d␣and␣comm␣%s␣
               (timestamp=%u)\n",
14                 @cast(task,"task_struct")->pid,
15                 $return,
16                 jiffies_to_usecs(@cast(task,"task_struct")->stime),
17                 kernel_string(@cast(task,"task_struct")->comm),
18                 gettimeofday_us());
19     }
20 }
```

A.2.2 Aufzeichnung des Netzwerkverkehrs

Listing A.3: *SystemTap*-Skript zur Aufzeichnung des Netzwerkverkehrs sowie des korrespondierenden Prozesses

```
1  %{
2  #include <linux/if_ether.h>
3  #include <linux/list.h>
4  #include <linux/netdevice.h>
5  #include <linux/tcp.h>
6  #include <net/tcp.h>
7  #include <net/inet_hashtables.h>
8  #include <net/sock.h>
9  #include <net/udp.h>
10 #include <linux/fdtable.h>
11 #include <linux/rwlock.h>
12
13 rwlock_t *kernTaskListLock;
14 %}
15
16 probe begin{
17     if (resolveTaskListLock() == 1) {
18         error("Cannot resolve tasklist_lock\n");
19         exit();
20     }
21 }
22
23 function resolveTaskListLock:long() %{
24     kernTaskListLock = (rwlock_t*)kallsyms_lookup_name("tasklist_lock");
25     if (kernTaskListLock == NULL) {
26         STAP_RETVALUE = 1;
27     } else {
28         STAP_RETVALUE = 0;
29     }
30 %}
31
32 function getTask:long(skbPtr:long, prot:long) %{
33     unsigned long ptr = STAP_ARG_skbPtr, flags;
34     struct sk_buff *skb = (struct sk_buff*)ptr;
35     struct sock *sk = NULL;
36     const struct tcphdr *th = NULL;
37     struct iphdr *iph;
38     struct udphdr *uh = NULL;
39     struct fdtable *fdt = NULL;
40     struct file *file = NULL;
41     struct task_struct *curTask;
42     struct socket *sock = NULL;
43     int foo = 0, i = 0, found;
44     char lastFileEmpty = 0;
45
46     if (kernTaskListLock == NULL) {
47         STAP_RETVALUE = 0;
48         return;
49     }
50     if (STAP_ARG_prot == 0) {
```

```
51            sk =skb >sk;
52        } else if (STAP_ARG_prot == 1) {
53            // TCP packet
54            th = tcp_hdr(skb);
55            sk = __inet_lookup_skb(&tcp_hashinfo, skb, th >source, th >dest);
56        } else if (STAP_ARG_prot == 2) {
57            // UDP packet
58            iph = ip_hdr(skb);
59            uh = udp_hdr(skb);
60            sk = __udp4_lib_lookup(dev_net(skb_dst(skb) >dev), iph >saddr,
                  uh >source,iph >daddr, uh >dest, inet_iif(skb),&udp_table);
61        }
62        // No valid socket found. Abort.
63        if (sk == NULL || sk >sk_socket == NULL) {
64            STAP_RETVALUE = 0;
65            return;
66        }
67
68        //write_lock_irqsave(kernTaskListLock,flags);
69        if (read_trylock(kernTaskListLock) == 0 ) {
70            STAP_RETVALUE = 0;
71            return;
72        }
73        local_irq_save(flags);
74        for_each_process(curTask) {
75            get_task_struct(curTask);
76            if (curTask >files == NULL) {
77                put_task_struct(curTask);
78                continue;
79            }
80            // .. and read a process sockets
81            if (spin_trylock(&curTask >files >file_lock) == 0) {
82                put_task_struct(curTask);
83                continue;
84            }
85            fdt = files_fdtable(curTask >files);
86            found = 0;
87            for (i = 0; i < fdt >max_fds; i++) {
88                file = rcu_dereference_check_fdtable(curTask >files,
                      fdt >fd[i]);
89                if (file == NULL) {
90                    /*
91                     * Two empty fds in a row indicate the end of the used
                           area of fdt
92                     */
93                    if (lastFileEmpty > 0) {
94                        break;
95                    }
96                    lastFileEmpty++;
97                    continue;
98                }
99                lastFileEmpty = 0;
```

```
100                // Refers the current fd to a socket?
101                sock = sock_from_file(file,&foo);
102                if (sock == NULL) {
103                    continue;
104                }
105
106                if (SOCK_INODE(sock)>i_ino ==
                       SOCK_INODE(sk>sk_socket)>i_ino) {
107                    found = 1;
108                    break;
109                }
110            }
111            spin_unlock(&curTask>files>file_lock);
112            put_task_struct(curTask);
113            if (found == 1) {
114                break;
115            }
116        }
117        local_irq_restore(flags);
118        read_unlock(kernTaskListLock);
119        // Give the socket back to the kernel
120        if (STAP_ARG_prot != 0) {
121            sock_put(sk);
122        }
123        if (found == 1) {
124            STAP_RETVALUE=(unsigned long)curTask;
125        } else {
126            STAP_RETVALUE = 0;
127        }
128    %}
129
130    probe kernel.function("tcp_v4_rcv") {
131        task = getTask($skb,1);
132        if (task != 0) {
133            printf("Received␣packet␣on␣device␣%s.␣Forwarded␣to␣process␣%d␣
                   (%s)\n",kernel_string($skb>dev>name),@cast(task,"task_struct")>pid,kernel_str
134        }
135    }
136
137    probe kernel.function("udp_rcv") {
138        task = getTask($skb,2);
139        if (task != 0) {
140            printf("Received␣packet␣on␣device␣%s.␣Forwarded␣to␣process␣%d␣
                   (%s)\n",kernel_string($skb>dev>name),@cast(task,"task_struct")>pid,kernel_str
141        }
142    }
143
144    probe kernel.function("dev_hard_start_xmit") {
145        task = getTask($skb,0);
146        if (task != 0) {
147            printf("Transmitted␣packet␣on␣device␣%s.␣Forwarded␣to␣process␣%d␣
                   (%s)\n",kernel_string($skb>dev>name),@cast(task,"task_struct")>pid,kernel_str
```

```
148         }
149 }
```

Listing A.4: *SystemTap*-Skript zur Aufzeichnung des Netzwerkverkehrs

```
1  probe kernel.function("tcp_v4_rcv") {
2      if ($skb >len > 1000) {
3          printf("Received␣packet␣on␣device␣
              %s.\n",kernel_string($skb >dev >name));
4      }
5  }
6
7  probe kernel.function("udp_rcv") {
8      if ($skb >len > 1000) {
9          printf("Received␣packet␣on␣device␣
              %s.\n",kernel_string($skb >dev >name));
10     }
11 }
12
13 probe kernel.function("dev_hard_start_xmit") {
14     if ($skb >len > 10) {
15         printf("Transmitted␣packet␣on␣device␣
              %s.\n",kernel_string($skb >dev >name));
16     }
17 }
```