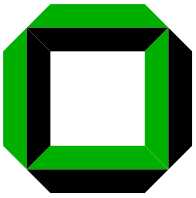

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

In-System-Debugging von Java-Programmen in der AmbiComp-Virtual-Machine

Wolf-Dennis Pahl

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Johannes Eickhold

3. August 2009

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 3. August 2009

_____

Wolf-Dennis Pahl

Abstract

Immer mehr Mikrocontroller lassen sich mit Java programmieren und steuern. Die Entwicklung der Java-Programme findet dabei auf Entwicklungscomputern und nicht auf den Mikrocontrollern statt, da die Ressourcen von Mikrocontrollern in der Regel für die Entwicklungstools zu gering sind.

Da sich die Java-VMs auf Entwicklungsrechner und Mikrocontroller in ihrem Funktionsumfang unterscheiden, ist es für Entwickler interessant das Verhalten von Java-Programmen auf der Zielhardware zu beobachten. Aus diesem Grund bieten einige Debugger die Möglichkeit des Remote-Debuggings. Remote-Debugging bedeutet, dass Debugger und zu debuggendes Programm auf unterschiedlichen Systemen laufen.

Das von Sun Microsystems für die Kommunikation zwischen Debugger und Java-VM vorgesehene Protokoll ist nicht auf geringen Ressourcenverbrauch optimiert. Es ist deshalb für die direkte Kommunikation mit einer Java-VM auf einem Mikrocontroller ungeeignet.

Eine mögliche Lösung, um die Kommunikation zwischen Debugger und Java-VM zu ermöglichen, ist ein Debug-Proxy, der zwischen dem Debugger und der Java-VM vermittelt. Im AmbiComp-Projekt wird dieser Ansatz gewählt, um In-System-Debugging von Java-Programmen auf der AmbiComp-Virtual-Machine zu ermöglichen.

Es existieren noch andere Lösungsmöglichkeiten, die im Falle von AmbiComp aber aus unterschiedlichen Gründen ungeeignet sind. Diese Arbeit erläutert naheliegende Lösungsansätze und zeigt ihre Vor- und Nachteile.

Die restliche Arbeit geht genauer auf die gewählte Lösung mittels AmbiComp-Debug-Proxy (ADP) ein. Die Implementierung des ADP wird vorgestellt. Konkrete Probleme, die beim Realisieren des Debug-Proxys auftreten, werden aufgezeigt und jeweils eine mögliche Lösung präsentiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die im Rahmen des Ambicomp-Projektes entwickelte Java-VM wurde für ein ressourcenbeschränktes Mikrocontrollersystem entwickelt. Aufgrund der daraus resultierenden Einschränkungen, ergeben sich eine Vielzahl von Herausforderungen, um einem Entwickler das Debuggen von Java-Programmen auf der AmibComp-VM (ACVM) zu ermöglichen.

Die wichtigste Einschränkung ist der relativ geringe permanente Speicher in den Mikrocontrollern. Um ihn zu schonen, werden die JAVA-Programme auf der Hardware in einem komprimierten Format gespeichert. Java-Programme in diesem Format enthalten jedoch nicht genügend Informationen um ein direktes Debugging mit einem JAVA-Debugger zu ermöglichen.

Die direkte Kommunikation ist auch wegen der geringen Rechenleistung der Mikrocontroller unerwünscht. Der im AmbiComp-Projekt verwendete 8-Bit Mikrocontroller müsste bei einer Debugsession viel Zeit mit der sich ergebenden Debugkommunikation verbringen, was den eigentlichen Programmablauf extrem verlangsamen würde.

Diese Arbeit zeigt die beim In-System-Debugging von Java-Programmen in der AmbiComp-Virtual-Machine auftretenden Probleme und stellt einen möglichen Lösungsansatz vor.

1.2 Gliederung

Der folgenden Text beginnt zuerst mit einem Überblick über das AmbiComp-Projekt. Die Grundlagen bezüglich des Debuggings im Allgemeinen und im Speziellen unter Java werden erläutert.

Die Herausforderungen von AmbiComp bezüglich des Debuggings werden aufgezeigt. Es folgt eine Formulierung der Anforderungen an ein Kommunikationsprotokoll für die Kommunikation mit der AmbiComp-VM.

Um dem Leser einen Überblick über ähnliche Programme zu geben, stellt der Text die Konzepte des Squawk-Projektes vor.

Nachdem die Probleme aufgezeigt sind, folgt die Vorstellung der gewählten Lösung

einschließlich Begründung.
Die Implementierung der Lösung wird sehr ausführlich erläutert.
Als letztes folgt eine Zusammenfassung und ein möglicher Ausblick.

Kapitel 2

Grundlagen

2.1 AmbiComp

Das AmbiComp-Projekt hat das Ziel, mehrere verteilte Kleinstcomputer zusammen zu schalten, um damit eine intelligente Umgebung zu schaffen. Unter einer intelligenten Umgebung versteht man hierbei eine selbstständig auf Ereignisse reagierende Umgebung. Dem Benutzer sollen im Idealfall die Kleinstcomputer verborgen bleiben. Die Idee die Umwelt des Menschen mit Computern auszustatten, die den Menschen bei seinen Aktivitäten unterstützen, nennt man auch Ambient-Computing und Ambient-Intelligence [1].

Die verteilten Kleinstcomputer werden als Ambient-Intelligence-Control-Units (AICUs) bezeichnet. Eine AICU besteht aus mehreren Mikrocontrollerboards mit jeweils unterschiedlichen Schnittstellen. Ein anderer Name für die Mikrocontrollerboards ist Sandwich-Module (SM). Eine AICU interagiert durch diese Schnittstellen der SMs mit ihrer Umgebung und anderen AICUs. Eine genauere Beschreibung der Hardware des AmbiComp-Projektes befindet sich in Anhang A.

Zur einfachen Steuerung einzelner AICUs wurde die AmbiComp-Virtual-Machine (ACVM) entwickelt. Die ACVM ist eine Java-Virtual-Machine, die es ermöglicht, Java-Code zur Steuerung der Hardware einer AICU ablaufen zu lassen.

Für den Entwickler von Programmen für die ACVM ist es wünschenswert, Java-Programme auf der ACVM zu beobachten. Dadurch wird es möglich, Fehlverhalten eines Programms festzustellen. Das Beobachten des Verhaltens von Programmen während deren Abarbeitung wird als Debugging bezeichnet.

2.2 Debugging von Programmen

Der Zweck des Debuggens ist das Finden und Reduzieren von Fehlern in einem Programm. Bei Software ist das Einfügen von Debugausgaben eine Möglichkeit, ein Programm während seiner Ausführung zu beobachten.

Debugausgaben sind Textausgaben auf der Konsole, die explizit zum Debuggen in den Quellcode eingebaut werden, um beim Programmablauf das Erreichen der entsprechenden Programmstellen oder die Werte von Variablen anzuzeigen. Das Einfügen

ermöglicht es, bei geeigneter Wahl der Ausgabestellen, Fehler zu finden oder einzugrenzen. Der Nachteil dieses Verfahrens ist eine notwendige Änderung am Quellcode, welche für die eigentliche Funktionalität eines Programms unnötig oder sogar störend ist. Bei verteilten Mikrocontrollern, wie im AmbiComp Projekt, ist dieser Ansatz aufgrund der geringen Ressourcen der Hardware ungeeignet. Ein diese Nachteile vermeidender Ansatz ist die Verwendung eines Debuggers, welcher es erlaubt ein Programm während der Ausführung zu beobachten, ohne dessen Quellcode zu verändern.

Ein Debugger ist ein Programm, das den Programmierer beim Finden von Fehlern in der Software hilft. Eclipse und viele andere Integrated-Development-Environments (IDE) besitzen einen integrierten Debugger.

Ein Sourcelevel-Debugger wie in Eclipse zeigt bei einem Absturz des zu debuggenden Programms die Stelle des Absturzes im Quellcode.

Ein Low-Level-Debugger zeigt dagegen den disassemblierten Code der abgestürzten Programmstelle.

Viele Debugger bieten dem Benutzer die Möglichkeit, Programme an jeder beliebigen Stelle anzuhalten und sich den Status (Quellcode-Position, Variablen) anzuschauen. Sie ermöglichen das Setzen von Breakpoints, an denen der Programmablauf unterbrochen wird. Bei angehaltenen Programmabläufen ist auch das aktive Eingreifen in das Programm möglich. So lassen sich zum Beispiel Variableninhalte manipulieren.

Programmfehler lassen sich mit einem Debugger schneller, einfacher und ressourcensparender finden, als durch Einfügen von Debugausgaben, die zusätzlichen Programmcode und somit zusätzliche Ressourcen benötigt würden.

2.3 Debugging von Java-Programmen

Bei der Programmiersprache Java werden Programme beim Übersetzen nicht direkt in die Maschinensprache des Zielsystems, sondern in eine Zwischensprache, den sogenannten Java-Bytecode, umgewandelt. Der Java-Bytecode ist vom ausführenden System unabhängig. Dies ermöglicht es Java-Programme auf unterschiedlichen Systemen zur Ausführung zu bringen.

Da jedes Prozessor nur die eigene Maschinensprache versteht, kann der Java-Bytecode vom System nicht direkt ausgeführt werden. Um den Java-Bytecode auf dem Prozessor auszuführen, kann ein virtuelles System verwendet werden. Dieses virtuelle System bezeichnet man als Java-Virtual-Machine (JVM). Es interpretiert den Java-Bytecode und führt ihn auf dem eigentlichen Prozessor aus. Für jede Prozessorarchitektur ist eine speziell angepasste JVM nötig.

Da die JVM alle Java-Programme ausführt, muss sich ein Java-Debugger zum Debuggen mit ihr verbinden. Befinden sich die JVM und der Java-Debugger auf der selben Maschine, so findet die Kommunikation zwischen Beiden mittels Inter-Process-Communication (IPC) statt. Wenn die JVM und der Java-Debugger auf unterschiedlichen Systemen laufen, ist IPC nicht möglich. In solch einem Fall kann man Remote-Debugging verwenden. Remote-Debugging bedeutet, dass der Debugger und das zu debuggende Programm (Debuggee) auf unterschiedlichen Systemen laufen. Um dies zu ermöglichen, baut der Debugger zum Debuggee eine Netzwerkverbindung auf, über die die Debuginformationen kommuniziert werden. Das von Sun für diese Zwecke spezifizierte Protokoll ist das Java-Debug-Wire-Protokoll (JDWP [2]). Der Aufbau dieses Protokolls wird im Anhang B genauer beschrieben.

Ablauf einer Debugsession

Beim Debugging eines Java-Programms auf einer JVM verbindet sich der Debugger vor dem Start des zu debuggenden Programms mit der wartenden JVM. Er fragt Informationen über laufbereite Programme ab und registriert sich für Ereignisse innerhalb der JVM. Wenn alle initial benötigten Informationen über das auszuführende Programm und die JVM abgefragt sind, sendet der Debugger der JVM das Kommando zum Starten der laufbereiten Java-Programme. Diese startet daraufhin alle ihre laufbereiten Java-Programme. Beim Auftreten eines registrierten Ereignisses verständigt die JVM den Debugger und hält, falls bei der Registrierung des auftretenden Events angegeben, den kompletten Programmablauf oder Teile davon an. Der Debugger ruft Informationen über die angehaltenen Programmteile von der JVM ab und präsentiert sie dem Anwender. Der Anwender hat die Möglichkeit die Daten der angehaltenen Programmteile zu verändern. Das Programm kann von ihm beendet oder fortgesetzt werden. Wenn das Ende des Programms erreicht ist, beendet sich die VM und die Debugsession ist zuende.

Kapitel 3

Analyse

3.1 Herausforderungen beim Debugging im AmbiComp-Project

Wie im Abschnitt 2.1 zu AmbiComp beschrieben, arbeitet die ACVM auf den AICUs. Da die AICUs aus Microkontrollerboards bestehen, haben sie nur eine beschränkte Anzahl an Ressourcen, wie Rechenleistung und Speicher.

Zum Beispiel hat das Microkontrollerboard IOSM eine Rechenleistung von 16MHZ, 8KB flüchtiger Speicher und 256KB permanenter Speicher. Die ACVM für dieses Modul benötigt 120KB permanenten Speicher. Somit bleiben 136KB permanenter Speicher für auf der ACVM auszuführende Programme. Eine genauere Beschreibung der AmbiComp-Hardware befindet sich in Anhang A.

Ein normales Debugging, bei dem der Debugger auf dem ausführenden System läuft, ist wegen der Ressourcen nicht sinnvoll. Um debuggen zu können, verfolgt die vorliegende Arbeit deshalb den Ansatz des Remote-Debuggings.

Java-Programme werden aus Speicherplatzgründen von der ACVM in Form einer BLOB-Datei eingelesen. Ein BLOB ist eine komprimierte Form aller Java-Class-Files (Bytecode-Dateien) eines Programms. Er enthält den Bestandteil der Java-Class-Files, der für die Ausführung des Programms auf der ACVM benötigt wird. Java-Class-Files sind die Standardform in der Java-Programme nach ihrer Kompilierung durch den Java-Compiler gespeichert werden. Sie enthalten außer den für die Programmausführung notwendigen Daten noch weitere Information (Klartextnamen).

BLOBs werden mit Hilfe eines Transcoders aus den Java-Class-Files erzeugt. Ein Transcoder wird verwendet, um einen kodierten Eingabestrom in einen anders kodierten Ausgabestrom zu wandeln. Der Transcoder des AmbiComp-Projektes verwendet als Eingabe die Java-Class-Files eines Java-Programms einschließlich der für das Programm benötigten Bibliotheksklassen. Als Ausgabe erzeugt er eine BLOB-Datei.

Die im BLOB enthaltenen Informationen reichen jedoch nicht zum Debuggen aus, da zum Beispiel Klartextnamen von Methoden und Klassen fehlen. Die ACVM hat somit nicht alle benötigten Informationen, um mit einem Java-Debugger mittels JDWP zu kommunizieren.

Ein Problem im Bezug auf Java-Debugging ist die fehlende Unterstützung des Java-Debug-Wire-Protocols in der ACVM. JDWP ist das Standardprotokoll für die Kom-

munikation zwischen JVM und Java-Debugger und wird von allen Remote-Debugging fähigen Java-Debuggern unterstützt. Die Integration des JDWP in die ACVM würde deren Verwaltungsaufwand vergrößern, da zusätzliche Informationen gespeichert werden müssten, wie etwa Registrierungen des Debuggers für Ereignisse in der JVM. Die ACVM würde dadurch mehr Speicherplatz verbrauchen und langsamer werden. Es wird absichtlich versucht, die ACVM so einfach wie möglich zu halten, um die Ressourcen zu schonen.

3.2 Anforderungen an das Protokoll für die Kommunikation mit der ACVM

Die Debugkommunikation mit der ACVM läuft nicht wie bisher vereinfacht dargestellt über eine direkte TCP/IP-Verbindung. Die ACVM ist in Wirklichkeit nur über den Systembus erreichbar. Um die Kommunikation aber trotzdem zu ermöglichen, gibt es eine weitere Softwarekomponente für die Umsetzung von TCP/IP auf das Protokoll des Systembusses. Der Name dieser Komponente ist OTTO (siehe Anhang C.1). Die Aufgabe des OTTO ist es, den Zugriff auf den Systembus mittels Socket zu ermöglichen. Der OTTO läuft auf dem Entwicklungssystem und nicht auf der Hardware. Wenn eine AICU per Kabel an den Entwicklungsrechner angeschlossen wird, meldet sie sich beim OTTO an. Auch emulierte AICUs melden sich über den Systembus beim OTTO. Der OTTO kennt somit alle erreichbaren ACVMs.

Alle TCP/IP-Kommunikation zur ACVM auf einer AICU muss somit zuerst mittels TCP/IP zum OTTO. Der OTTO überträgt sie weiter über den Systembus an die gewünschte ACVM. Die Auswahl einer ACVM ist mit Hilfe des OTTO-Protokolls möglich. Das Kommando zur Auswahl einer dem OTTO bekannten ACVM ist *select <aicu>,<sm>,<Endpunkt>*. Es wird als erstes Kommando über die neu aufgemachte TCP/-IP Verbindung an den OTTO übertragen. Nach dem Ausführen des Kommandos wird alle Kommunikation, die über diese TCP/IP-Verbindung geht, direkt an das Sandwich-Module <sm> auf der AICU <aicu> übertragen. Der Endpunkt muss für die Debugkommunikation mit der ACVM immer 5 sein.

Für den Systembus müssen die unteren ISO/OSI-Schichten der Debug-Nachricht ausgetauscht werden. Um dies zu ermöglichen, muss dass Debug-Protokoll auf der Anwendungsschicht des ISO/OSI-Schichtenmodells sein.

Das Debug-Protokoll für die Kommunikation mit der ACVM muss drei Aufgaben erfüllen:

1. **Steuerung der Virtuellen-Maschine:** Es muss möglich sein der ACVM Steuerbefehle, wie z.B. *resume* oder *suspend*, zu senden.

2. **Abfrage von Daten in der ACVM:** Daten sind hierbei alle Informationen, die für das Debugging gebraucht werden. Dies können zum Beispiel Variableninhalte oder Threadzustände sein.

3. **Transport von Events:** Events, die in der VM Anfallen, müssen mittels Protokoll transportiert werden. Als Beispiel sei hier auf das Erreichen eines Breakpoints verwiesen.

Das JDWP würde die Anforderungen erfüllen (siehe B). Es ist aber zu ressourcenverschwendend, da es mehr Funktionalität bietet als benötigt wird. Beispielsweise besitzt

das JDWP im Header ein Feld ID, welches eine eindeutige Zuordnung von Request und Reply-Nachricht bei asynchroner Kommunikation ermöglicht. Da die Kommunikation zur ACVM aber synchron sein kann, wird ein ID-Feld nicht benötigt. Für weitere Gründe gegen den Einsatz von JDWP sei auf Abschnitt 4.3.1 verwiesen.

3.3 Verwandte Arbeiten: Debugging in Squawk

Eine VM von Sun, die der AmbiComp-VM (ACVM) ähnelt, ist Squawk [7]. Ziele von Squawk sind die Programmierung des größten Teils der Java-VM in der Programmiersprache Java selbst und die Verwendung der VM auf eingebetteter Hardware mit beschränkten Ressourcen und ohne Betriebssystem.

Das Ziel, die entwickelte Java-VM auf einer eingebetteten Hardware mit beschränkten Ressourcen zu verwenden, stimmt beim AmbiComp-Projekt und beim Squawk-Projekt überein. Aus diesem Ziel ergibt sich der gemeinsame Wunsch, den benötigten Speicherplatz auf der Hardware so gering wie möglich zu halten und die VM beim Debuggen maximal zu entlasten.

Die Squawk-VM bekommt, wie die ACVM, modifizierte Dateien zur Abarbeitung und keine Java-Class-Files. Um den Speicherplatz auf der Hardware zu schonen, benutzt das Squawk-Projekt ein Speicherformat, dass fast nur die für die Ausführung eines Java-Programms notwendigen Daten enthält. Um eine Zuordung von Klassennamen und Klassennummern zu ermöglichen, muß das Squawk-Speicherformat die Klassennamen enthalten, was zur Ausführung eigentlich unnötig ist. Zur Erzeugung der Dateien für die Hardware wird ein Programm verwendet. Das Programm wird Translator genannt. Es hat die gleiche Funktion, wie der Transcoder im AmbiComp-Projekt (siehe 3.1).

Der Translator für die Squawk-VM nimmt als Eingabestrom die vom Java-Compiler erzeugten Java-Class-Files. Diese Java-Class-Files werden analysiert und die für die Ausführung auf der Hardware benötigten Teile in das projektspezifische Ausgabeformat geschrieben. Das Ausgabeformat des Squawk-Transcoders ist ein Squawk-Suite-File. Es ist möglich mehrere Suitefiles auf die Hardware zu laden. Suitefiles können untereinander Abhängigkeiten haben. Damit wird es ermöglicht gemeinsam genutzte Klassen nur in einem Suite-File zu speichern, was den Speicherverbrauch auf der Hardware senkt.

Als Nachteil dieser Minimierung des Java-Programms ergibt sich das Problem, dass die Daten im Suite-File nicht mehr ausreichen, um direkt mit einem Java-Debugger mittels JDWP kommunizieren zu können. Dem Java-Programm werden bei der Transcodierung viele zum Debuggen benötigte Informationen wie etwa Klartextnamen von Methoden und Variablen genommen. Diese Informationen sind aber für die direkte Kommunikation mit einem Java-Debugger mittels JDWP notwendig.

Die Verkleinerung der Programmdateien verursacht im Squawk-Projekt somit die gleichen Probleme bezüglich Debuggen, wie im AmbiComp-Projekt. Squawk löst die Probleme mit Hilfe eines Proxys.

Der Squawk-Debug-Proxy (SDP)

Um das Debuggen von Java-Programmen auf der Hardware mit einem Java-Debugger trotzdem zu ermöglichen, wird beim Squawk-Projekt ein Debug-Proxy zwischen die

Squawk-VM und den Debugger geschaltet. Der Debug-Proxy beantwortet den Teil der JDWP-Kommandos, die die Squawk-VM aufgrund der reduzierten Information im Suite-File nicht beantworten kann. Die JDWP-Kommandos, die direkt von der Squawk-VM beantwortet werden, leitet der Debug-Proxy unverändert an die Squawk-VM weiter.

Der Debug-Proxy ermöglicht durch die teilweise Beantwortung der Anfragen des Debuggers eine Reduzierung der auf der VM zum Debuggen benötigten Ressourcen (CPU, Speicher).

Für den Java-Debugger sieht es trotz und wegen der indirekten Kommunikation über den Proxy so aus, als würde er direkt mit einer JDWP fähigen JVM kommunizieren.

Für die Kommunikation zwischen Debug-Proxy und Squawk-VW wird das Squawk-Debug-Wire-Protocol (SDWP) [7] verwendet. Es besteht zum größten Teil aus einer echten Untermenge des JDWP. Der Rest sind zusätzliche Kommandos, die zur Aktualisierung der Wissensbasis des Proxys über die Squawk-VM dienen. Die zusätzlichen Kommandos des SDWP haben den gleichen Paketaufbau, wie die aus dem JDWP übernommenen Kommandos. Das JDWP ermöglicht in seiner Spezifikation ausdrücklich das Hinzufügen von vendorspezifischen Kommandos.

Der Debug-Proxy kommuniziert nicht direkt mit der VM, sondern mit dem Squawk-Debug-Agent (SDA). Dieser Agent realisiert das SDWP für die VM. Er ist ein eigenständiges Java-Programm, dass bei einer Debugsession auf der Squawk-VM gestartet werden muss. Als Parameter wird ihm der Name der Klasse mit der Main-Methode des zu debuggende Java-Programm übergeben (siehe Anhang D). Wenn der SDA nicht auf der Squawk-VM gestartet ist, kann kein Debugging stattfinden.

Der Proxy des Sqauwk-Projektes bezieht seine Zusatzinformationen direkt aus den Java-Class-Files, welche er als Startparameter bekommt. Da die Squawk-VM intern aber Klassennummern anstatt Klassennamen verwendet, muss ihm die Squawk-VM eine Zuordnung von Signatur (z.B: Ljava/lang/Object;) zu interner Klassennummer der VM mitteilen. Daraus folgt, dass die Signaturen der Klassen in der Datei für die VM (Hardware) enthalten sein müssen, wodurch diese größer wird. Die Mitteilung der Zuordnung geschieht durch den SDA direkt nach dem Aufbau der Verbindung zwischen Debug-Proxy und SDA.

Wenn der Quellcode im Debugger, die Class-Files im SDP und die Daten im Squawk-File nicht zueinander passen, kann es beim Debuggen zu Fehlern oder der Beendigung der Debugsession kommen. Eine Überprüfung findet nicht statt.

Kapitel 4

Entwurf

Wie in den vorherigen Kapiteln gezeigt, besteht die Herausforderung darin, die Kommunikation zwischen ACVM und Java-Debugger zu ermöglichen. Es gibt 3 Lösungsmöglichkeiten:

1. JDWP in ACVM implementieren (siehe Abbildung 4.1 Mitte)
2. Debugger anpassen (siehe Abbildung 4.1 links)
3. Debug-Proxy verwenden (siehe Abbildung 4.1 rechts)

Im vorherigen Kapitel ist bereits klar geworden, dass das Implementieren des JDWP in die ACVM keine Option ist, da die Unterstützung des JDWP die ACVM vergrößern würde. Auch wären zusätzliche Informationen im BLOB auf der Hardware nötig. Die ACVM müsste außer der Abarbeitung des zu debuggenden Programms auch die komplette Kommunikation mit dem Java-Debugger durchführen, was CPU, Speicher und Netzwerkressourcen verbrauchen würde. Da die Ressourcen auf der Zielhardware beschränkt sind, ist das Anpassen der ACVM somit keine Lösung.

Ziel muss die maximale Entlastung der Hardware sein. Daraus folgt, dass auch die ACVM bezüglich Debugging so einfach wie möglich sein sollte.

Das Anpassen des Debuggers ist nicht erwünscht, da es dem Entwickler das Verwenden eines bestimmten Debuggers vorschreiben würde. Ziel muss es sein, dem Entwickler die größt mögliche Freiheit bei der Entwicklung zu ermöglichen.

Somit bleibt nur die dritte Lösungsmöglichkeit, wie sie auch im Squawk-Projekt gewählt wurde, übrig. Zwischen dem Debugger und die ACVM wird ein Debug-Proxy positioniert und die ACVM minimal erweitert. Die minimale Erweiterung der ACVM ist notwendig, um die Kommunikation zwischen ADP und ACVM mittels eines für diesen Zweck neu spezifizierten ACDWP zu unterstützen.

Der Proxy ermöglicht die Verwendung eines beliebigen Java-Debuggers und entlastet die ACVM, da diese nicht das JDWP unterstützen muss.

4.1 Der AmbiComp-Debug-Proxy (ADP)

Um mit einem Java-Debugger eine Anwendung auf der Hardware debuggen zu können, schaltet man einen Proxy zwischen den Java-Debugger und die ACVM (siehe Abbil-

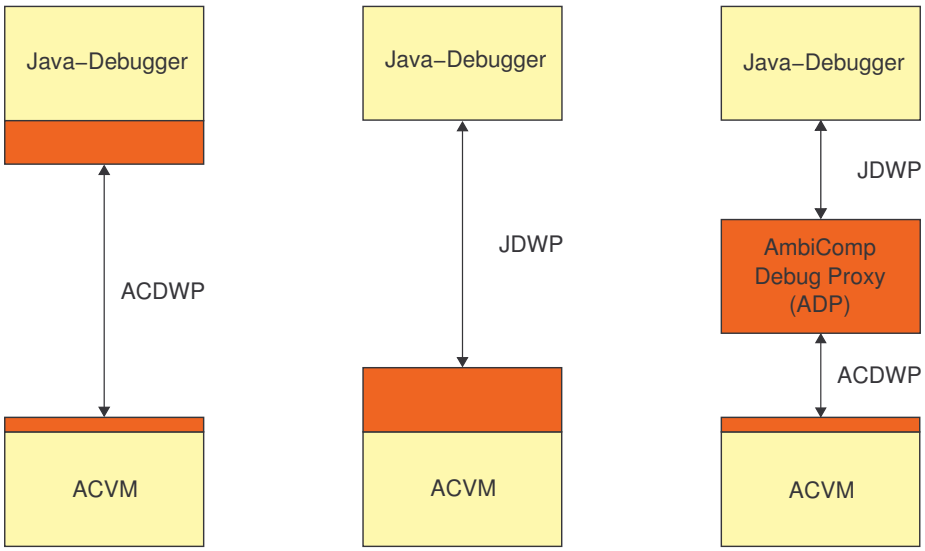

Abbildung 4.1: Drei Entwurfsmöglichkeiten: Debugger anpassen (links), JDWP in ACVM implementieren (Mitte), Debug-Proxy verwenden (rechts). Die markierten Bereiche zeigen die jeweils notwendigen Anpassungen.

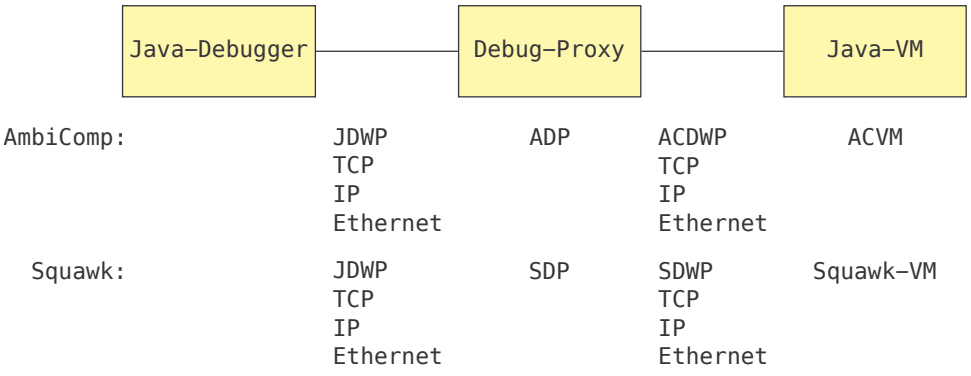

Abbildung 4.2: Debugging in AmbiComp und Squawk mit Komponenten und Kommunikationsprotokollen

dung 4.3). Dieser Proxy kommuniziert mit dem Java-Debugger über das JDWP [2]. Für die Kommunikation mit der ACVM wird das für diese Aufgabe entwickelte ACDWP (4.3.2) genutzt, welches die Einschränkungen der Hardware, wie Ressourcenmangel (CPU, Speicher, Netzwerk) berücksichtigt.

Der Java-Debugger erkennt nicht, dass er mit einem Proxy verbunden ist. Für ihn sieht es so aus, als würde er mit einer wirklichen JVM kommunizieren.

Die Squawk VM hat wie bereits erwähnt den Nachteil, dass das Suite-File auf der Hardware die Klartextnamen aller Klassen enthalten muss. Dies ist notwendig, um während des Programmablaufs eine Zuordnung zwischen verwendeten Nummern und den Klassen-Signaturen herzustellen.

Um diesem Nachteil zu entgehen, werden beim AmbiComp-Projekt zwei Dateien (BLOBs) beim Transcodieren mittels Transcoder erzeugt. Einer von beiden BLOBs besitzt erweiterte Debuginformationen und wird vom Proxy verwendet. Er enthält sowohl die Klassennummern für die Kommunikation mit der ACVM, als auch die Signaturen für die Kommunikation mit dem Java-Debugger. Mit seiner Hilfe ist die Zuordnung zwischen Klassennummer und Signatur ohne Kommunikation mit der ACVM im Proxy möglich. Dieser BLOB wird im weiteren Text als Debug-BLOB bezeichnet. Der andere vom Transcoder erzeugte BLOB ist auf minimale Größe optimiert. Er enthält nur die für den Programmablauf nötigen Daten und ist für die Programmausführung auf der Hardware gedacht. Im weiteren Text wird dieser BLOB als ACVM-BLOB bezeichnet. Durch diesen Ansatz ist es möglich, die Zuordnung nur im Debug-BLOB und somit nicht auf der AICU-Hardware zu speichern.

Ein weiterer Vorteil des Proxy im Vergleich zu einer direkten Kommunikation, ist die Möglichkeit Informationen an den Debugger zu liefern, ohne diese von der ACVM abzufragen. Viele Anfragen können vom Proxy gefiltert und zum Teil oder ganz beantwortet werden. Zur Beantwortung verwendet der ADP unterschiedliche Informationsquellen, unter anderem den Debug-BLOB. Die Entlastung ermöglicht eine schnellere Ausführung des zu debuggenden Java-Programms in der ACVM, da diese weniger Anfragen des Debuggers abarbeiten muss.

4.2 Grundlegende Funktionsbeschreibung des ADP

Nach dem Aufbau der Verbindungen zu ACVM und Debugger beginnt der ADP mit seiner eigentlichen Arbeit. Es gibt zwei Szenarios:

- Das erste und häufigere Szenario ist der Empfang eines JDWP-Kommandopaketes vom Debugger. Ein JDWP-Kommandopaket kann sowohl die Anfrage nach Informationen als auch ein Kommando an die ACVM enthalten.

  Kommandos an die ACVM werden in entsprechende ACDWP-Requests umgesetzt und an die ACVM weitergeleitet. Eine eventuelle Antwort der ACVM wird vom ADP empfangen und in das entsprechende JDWP-Reply-Paket umgesetzt. Ein JDWP-Reply-Paket ist das in JDWP spezifizierte Antwortformat.

  Enthält das JDWP-Kommandopaket des Debugger eine Informationsabfrage, so kann und muss die Anfrage nicht immer von der ACVM beantwortet werden. Der Grund hierfür ist der gewählte Lösungsansatz, welcher in 3.1 und 4.1 beschrieben ist. Wenn es möglich ist, beantwortet der ADP Informationsanfragen des Debuggers teilweise oder komplett ohne die ACVM. Dies entlastet die CPU

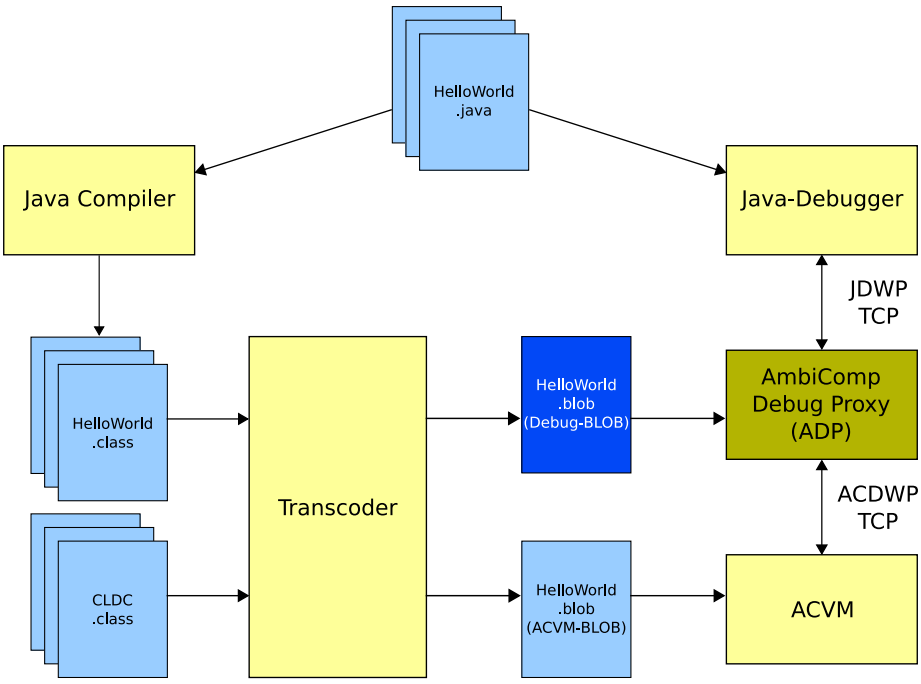

Abbildung 4.3: HelloWorld als Beispiel

und den Speicher des ausführenden SMs. Es erhöht sich auch die Antwortgeschwindigkeit für den Debugger, da die Kommunikation mit der ACVM für diese Anfragen entfällt.

Die für die Antwort benötigten Informationen bezieht der ADP dabei aus unterschiedlichen Quellen:

1. Eine Informationsquelle ist der Debug-BLOB, welcher auch Informationen enthält, die der ACVM-BLOB der ACVM nicht hat. Der Debug-BLOB ermöglicht eine Zuordnung von Nummern und Positionen zu Informationen, die sich auf Java-Class-Files oder Java-Source-Files beziehen, wie Klartextnamen von Klassen, Methoden, Variablen und Positionen in Java-Source-Files. Um Speicherplatz zu sparen entfernt der Transcoder diese Informationen aus dem ACVM-BLOB. Eine Beschreibung, wie man den Transcoder benutzen muss, um den Debug-BLOB und den ACVM-BLOB zu erstellen, befindet sich in Anhang C.2.
2. Eine weitere Informationsquelle zur Beantwortung einer Anfrage des Java-Debuggers sind Informationen, die die ACVM zu einem früheren Zeitpunkt als Events an den ADP gesendet hat und die dieser in einer internen Datenstruktur speichert. Threadzustandsänderungen werden auf diese Art dem ADP mitgeteilt. Der ADP speichert die Änderung des Zustandes eines Threads in seiner internen Datenstruktur und beantwortet spätere Anfragen des Java-Debuggers aus dieser Datenstruktur. Falls sich der Java-Debugger für die Art des erhaltenen Events beim ADP registriert hatte, wird er über das Eintreffen des Events mittels JDWP-Kommandopaket benachrichtigt.

3. Wenn die benötigte Information nur auf der ACVM vorhanden ist, wird sie von dieser abgefragt. Die Anfrage wird mittels ACDWP-Paket an die ACVM gesendet. Die ACVM sendet als Antwort an den ADP ein ACDWP-Paket mit den gewünschten Informationen. Die Antwort wird mittels JDWP-Reply-Paket an den Debugger gesendet. Informationen, die direkt mit dem Programmablauf zu tun haben, können nur auf diesem Weg erhalten werden. Inhalte von Variablen fallen unter diese Kategorie, da sie vom Programmablauf abhängig sind.

- Das zweite Szenario ist das Auftreten eines Events in der ACVM, was diese veranlasst eine Nachricht in Form eines ACDWP-Paketes an den ADP zu senden. Der ADP wertet die Nachricht aus und vergleicht sie mit den Registrierungen auf Ereignisse des Debuggers. Falls es passende Registrierungen des Debuggers für das empfangene Event gibt, so wird der Debugger mittels JDWP-Kommandopaket über das Event informiert. Je nach Event werden die von der ACVM erhaltenen Informationen vom ADP zwischengespeichert, um spätere Anfragen des Debuggers schneller und unabhängig von der ACVM beantworten zu können.

4.3 AmbiComp-Debug-Wire-Protocol (ADWP)

4.3.1 Warum ein eigenständiges Protokoll?

Die Verwendung eines eigenen Protokolls ermöglicht es der ACVM den Rechen- und Kommunikationsaufwand im Vergleich zu JDWP weiter zu verringern, da das Protokoll an die Eigenschaften der ACVM angepasst ist.

Im Vergleich zu JDWP, besitzt das ACDWP weniger Felder im Paketheader, was die Größe der Nachrichten verkleinert (siehe 5.6). Er besteht beim ACDWP nur aus *length* und *type*. Das JDWP-Paketformat sieht als Header dagegen folgende Felder für Request- und Reply-Pakete vor (siehe Abbildung B.1): *length, ID, flags, command set, command, error code*. Das Feld für die *ID* kann beim ACDWP wegen der synchronen Kommunikation weggelassen werden. Asynchrone Kommunikation ist zum Debuggen nicht notwendig. Der Proxy versendet immer nur einen Request gleichzeitig, was die Zuordnung der Response eindeutig macht. *Flags* werden beim JDWP verwendet, um Request- und Reply-Pakete zu unterscheiden. Dies ist beim JDWP nötig, da sich der Paketaufbau zwischen einem *CommandPacket* (Request) und einem *ReplyPacket* unterscheiden und Events im JDWP als Command-Paket versendet werden. Das ACDWP macht keine Unterscheidung im Paketaufbau, weshalb das Feld *Flags* im ACDWP nicht benötigt wird. Die Funktionalität der Felder *command set* und *command* im JDWP-Header wird beim ACDWP mit dem einen Feld *type* realisiert (siehe 5.6). Das Feld *error code* im JDWP-Reply-Paket wird verwendet, um Fehler bei der Abarbeitung innerhalb der JVM anzuzeigen. Im ACDWP existiert für Fehler bei der Abarbeitung ein extra Event, welches wegen der synchronen Kommunikation eindeutig einem ACDWP-Request zugeordnet werden kann.

Der sich aus der Realisierung eines eigenständigen Protokolls ergebende Nachteil, ist der erhöhte Aufwand im Proxy. Der Proxy muss die komplette Kommunikation übersetzen und kann nicht wie in Squawk einen Teil der Kommunikation unverarbeitet an die VM weiterleiten. Dies ist jedoch nicht wirklich ein Nachteil, da der Proxy nicht auf

der ressourcenbeschränkten Hardware läuft.

4.3.2 Design des ACDWP

Das Ambicomp-Debug-Wire-Protokoll (ACDWP) ist, wie das JDWP, ein Protokoll, welches auf der Anwendungsschicht im ISO/OSI-Schichtenmodell eingeordnet ist. Es ermöglicht weder eine zuverlässige Verbindung, noch die Adressierung des Kommunikationspartners. Für die Aufgabe der Adressierung des Kommunikationspartners benutzt der ADP das IP-Protokoll auf Schicht 3 des ISO/OSI Modells. Die gesicherte Verbindung wird mittels TCP auf Schicht 4 realisiert. Grundsätzlich ist es aber auch möglich ACDWP mit anderen Protokollen zu kombinieren, was der OTTO bei der Kommunikation zu einer AICU auch macht. Der OTTO ermöglicht den Zugriff auf den Systembus durch einen Netzwerk-Socket. Er verbirgt die wirkliche Position der ACVM und kümmert sich um die Umwandlung der unteren Protokollebenen. (siehe Abbildung C.1)

Obwohl alle ACDWP-Pakete im Gegensatz zu JDWP-Paketen das gleiche Paketformat haben (siehe Abbildung 5.7 und Abbildung B.1), lassen sich drei Arten von ACDWP-Paketen unterscheiden:

- **ACDWP-Requestnachricht:** Abfragenachricht an die ACVM. Sie kann entweder eine Informationsanfrage oder ein Kommando enthalten.
- **ACDWP-Responsenachricht:** Antwortnachricht auf eine Requestnachricht.
- **ACDWP-Eventnachricht:** Eventnachricht der ACVM an den ADP.

Eine ACDWP-Requestnachricht wird vom Proxy genutzt, um Daten abzufragen oder Kommandos an die ACVM zu senden.

Ein Beispiel für das Abfragen von Daten ist das Kommando *ACDWP.VERSION_REQUEST*, welches zur Abfrage der Versionsnummer der ACVM benötigt wird.

Das Kommando *ACDWP.VM_RESUME* stellt ein Beispiel für einen Request dar, der zum Steuern der ACVM genutzt wird. Sein Zweck ist, die VM und somit das Programm auf der ACVM zu starten.

Zu jeder ACDWP-Requestnachricht gibt es immer eine Antwort in Form einer ACDWP-Responsenachricht. Wenn der Request nur ein Kommando an die ACVM ist, kann der Inhalt der Responsenachricht leer sein.

Nur die ACVM kann ACDWP-Eventnachrichten senden. ACDWP-Eventnachrichten werden von der ACVM genutzt, um den ADP über Ereignisse in der ACVM zu informieren. Ereignisse können Threadzustandsänderungen, Breakpoints oder Fehler bei der Kommandoabarbeitung sein. ACDWP-Eventnachrichten werden vom ADP nie beantwortet.

Kapitel 5

Implementierung

5.1 Klasseneinteilung im AmbiComp-Debug-Proxy

Die Klasseneinteilung mit den wichtigsten Klassen sieht man in Abbildung 5.1.

Das Design des ADP orientiert sich im Wesentlichen am Design des Squawk-Debug-Proxies (SDP). Das Programm startet in der Klasse *ADP*. Diese erzeugt, die für die Steuerung der Kommunikation notwendigen, zwei Listener, sowie jeweils ein Objekt der Klassen *ADPEventManager*, *ADPThreadManager* und *ADPBlobReader* (siehe Abb. 5.1).

5.1.1 Listener

Pro Kommunikationskanal gibt es einen Listener, der als Thread läuft:

- **DebuggerListener:** Der DebuggerListener ist für die Abarbeitung eingehender Nachrichten vom Java-Debugger zuständig.

  Falls nötig erstellt er ACDWP-Requests und sendet sie über die Verbindung des ACVMListeners an die ACVM. Die Antwort auf den ACDWP-Request wird vom ACVMListener empfangen. Bis zur Benachrichtigung über den Eingang der Antwort durch den ACVMListener, legt sich der Thread schlafen.

  Wenn möglich wird die Anfrage des Java-Debuggers aber direkt vom Debugger-Listener ohne die ACVM beantwortet.

- **ACVMListener:** Der ACVMListener ist für die Eingehenden Nachrichten der ACVM zuständig. Er nimmt eingehende Events oder Antworten auf vom DebuggerListener versendete ACDWP-Requests an. Je nach Art der eingehenden Nachricht erfolgt eine andere Verarbeitung:

  Eingehende ACDWP-Events werden vom ACVMListener falls nötig zu Event-Objekten gewandelt und in einer Liste zu bearbeitender Events gespeichert, welche vom EventHandler abgearbeitet wird. Teilweise werden auch Informationen zur Erstellung und Aktualisierung der internen Datenstruktur genutzt. Hierunter fallen Informationen wie Threadstatus oder Frameinformationen.

  Antworten auf Anfragen des DebuggerListeners werden der passenden Anfrage zugeordnet und der DebuggerListener benachrichtigt.

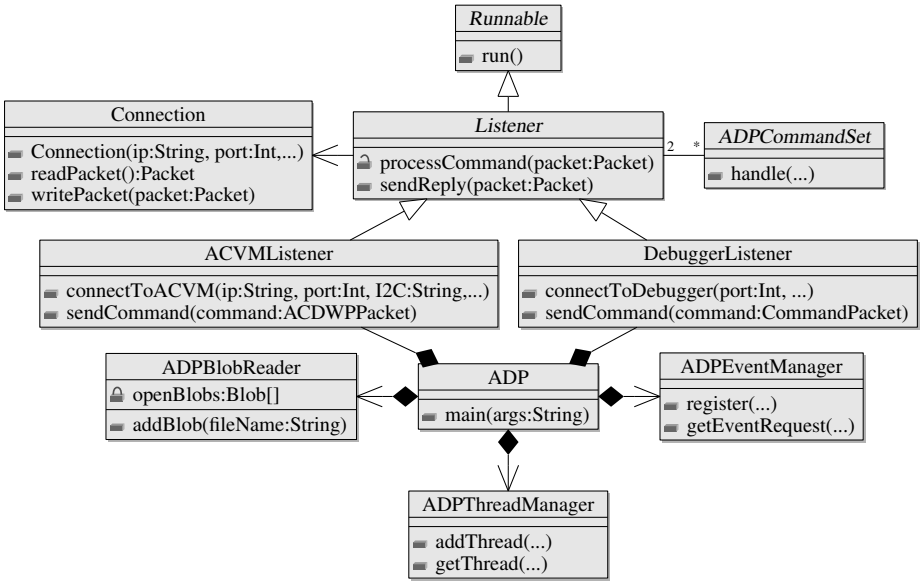

Abbildung 5.1: Die wichtigsten Klassen des ADP

5.1.2 Connection

Die grundlegenden Kommunikationsfunktionen, wie Verbindungsaufbau, Verbindungsabbau, Paketversand und Paketempfang finden in einem Objekt der Klasse *Connection* oder *ACVMConnection* statt. Jeder der beiden Listener nutzt ein Objekt dieser Klassen. Den Listenern ist es möglich ein Paket über ihre eigene Verbindung oder die des anderen Listeners zu senden. Zu diesem Zweck haben beide eine Referenz aufeinander.

5.1.3 ADPCommandSet

Die eigentliche Verarbeitung der Kommandos findet in Unterklassen des Typs *ADPCommandSet* statt. Die Unterklassen von *ADPCommandSet* enthalten die Realisierung der ACDWP- oder JDWP-Kommandos. Jede Klasse steht dabei für eine Kommandomenge des JDWP.

Der Listener übergibt ankommende Pakete an die entsprechende Klasse, wobei er die richtige Klasse (Kommandomenge) anhand der Kommandomengennummer (commandSet) im JDWP-Paket bestimmt. Die Kommandonummer (command) im JDWP-Paket bestimmt die Kommandomethode, welche zur Verarbeitung des Paketes ausgeführt wird. Beide Nummern sind zur eindeutigen Bestimmung des auszuführenden Kommandos notwendig.

5.1.4 ADPBlobReader

Um Anfragen des Debuggers beantworten zu können, ist es dem DebuggerListener möglich, mit Hilfe des Objekts der Klasse *ADPBlobReader*, auf den Debug-BLOB zuzugreifen. Die Klasse ADPBlobReader erbt von der Klasse *transcoder.ReverseTranscoder*. Die Klasse *transcoder.ReverseTranscoder* bietet die Möglichkeit Debug-BLOBs

einzulesen und auf die enthaltenen Daten zuzugreifen.

Auf das Objekt der Klasse *ADPBlobListener* kann der DebuggerListener über seine Referenz zum Objekt der Klasse *ADP* zugreifen. Die Klasse *ADPBlobReader* hilft auch bei der Anpassung von Datenformatierungen zwischen BLOB und Debugger. Die Instanz (Objekt) der Klasse *ADPBlobReader* wird beim Programmstart vom ADP erstellt.

5.1.5 ADPEventManager

Event-Registrierungen des Debuggers werden im ADPEventManager gespeichert und nur im Falle von Breakpoints nach Umwandlung der Anfrage auch an die ACVM weitergeleitet. Gespeichert werden die Requests in einem Objekt der Klasse *ADPEventManager*. Dieses Objekt ist für die Verwaltung von Requests zuständig und wird beim Programmstart von ADP angelegt. Der einzelne Request wird in Form eines Objektes der Klasse *EventRequest* gespeichert, welches wiederum Objekte der Klasse *EventRequestModifier* besitzt. In diesen Modifiern werden Bedingungen der Requests gespeichert. Ein Event gehört nur dann zu einem EventRequest, wenn es alle Bedingungen erfüllt. Über Bedingungen lässt sich zum Beispiel festlegen, dass eine Event-Registrierung nur für einen bestimmten Thread gilt.

5.1.6 ADPThreadManager

Jede Threadzustandsänderung in der ACVM wird vom ADP gespeichert, um Threadzustandsanfragen des Debuggers ohne die ACVM beantworten zu können. Änderungen der gespeicherten Zustände können dabei entweder durch Mitteilungen der ACVM oder durch Kommandos des Debuggers ausgelöst werden. Für jeden bekannten Thread existiert im Objekt der Klasse *ADPThreadManager* ein Objekt *ACVMThread*. Die Klasse *ADPThreadManager* wird beim Start von ADP instantiiert. Bei in der ACVM laufenden Threads die suspended sind, baut der ADP falls erforderlich eine Frameliste auf, welche aus Objekten der Klasse *ACVMFrame* besteht und die Frameinformationen der in der ACVM laufenden Threads speichert.

5.1.7 EventHandler

Außer den beiden Listenern gibt es einen weiteren Thread, den EventHandler:

- **EventHandler:** Der EventHandler wird vom ACVMListener benachrichtigt oder gegebenenfalls gestartet, wenn der ACVMListener Events in die Liste der abzuarbeitenden Events einträgt. Er arbeitet die Liste solange ab, bis kein Event mehr in der Liste vorhanden ist und wartet dann auf eine erneute Benachrichtigung durch den ACVMListener.

  Der EventHandler ist nötig, um einen Deadlock zu verhindern und die zeitnahe Abarbeitung von eingehenden Nachrichten der ACVM sicherzustellen (siehe Abbildungen 5.1.7 und 5.1.7). Ohne ihn kann es beim Empfang des VM-Start-Events zu einem Deadlock oder einer Verzögerung der danach empfangenen Nachrichten kommen, da die Weiterleitung des VM-Start-Events erst nach bestimmten Vorbedingungen möglich ist. Für alle anderen Nachrichten von der

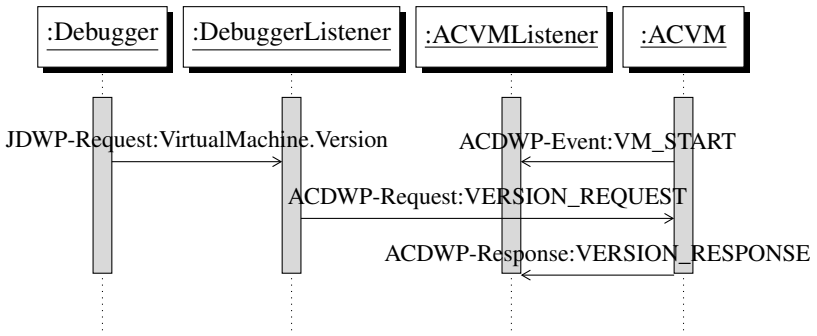

Abbildung 5.2: Deadlock: Der ACVMListener empfängt eine Eventnachricht von der ACVM. Gleichzeitig empfängt der DebuggerListener ein JDWP-Request vom Debugger. Der ACVMListener kann das Ereignis nicht an den Debugger senden, da dieser die Nachricht mit den ID-Größen noch nicht vom DebuggerListener bekommen hat. Ohne das Wissen über die ID-Größen kann das Event vom Debugger nicht verarbeitet werden. Der ACVMListener wartet solange, bis er die Nachricht senden kann. Der DebuggerListener sendet den empfangenen JDWP-Request nach der Umwandlung in einen ACDWP-Request an die ACVM. Nun wartet der DebuggerListener solange, bis die Antwort empfangen ist. Er kann also keine weitere Nachrichten vom Debugger empfangen. Die ACVM sendet die Antwort auf den ACDWP-Request des DebuggerListeners an den ACVMListener. Während der ACVMListener wartet, kann er die Antwort auf die Anfrage des DebuggerListeners nicht empfangen. Beide Threads des ADP sind blockiert. Der ACVMListener wartet darauf, dass er das Event senden kann. Der DebuggerListener wartet auf die Antwort des Requests.

ACVM bewirkt er einen geringen Geschwindigkeitsvorteil, da der ACVMListener sich nicht um die Versendung von Eventnachrichten an den Debugger kümmern muss. Er ist somit schneller wieder empfangsbereit für Nachrichten von der ACVM.

Ein Deadlock bezeichnet den Fall, dass Threads gegenseitig auf ein Ereignis des jeweils anderen Threads warten. Da alle Threads warten, gibt es keinen Fortschritt im Programmablauf. Das Programm ist abgestürzt.

5.1.8 Packet

JDWP- und ACDWP-Nachrichten werden in Objekten der Unterklassen von *Packet* gespeichert.

Das Klassendiagramm von Packet und seinen Unterklassen ist in Abbildung 5.4 zu sehen.

Für JDWP gibt es zwei Unterklassen von *Packet*, welche entweder für JDWP-Command-Pakete oder JDWP-Reply-Pakete zuständig sind. JDWP-Command-Pakete werden in Objekten der Klasse *CommandPacket* gespeichert. JDWP-Reply-Pakete werden in Objekten der Klasse *ReplyPacket* gespeichert.

Bei ACDWP-Paketen gibt es beim Paketaufbau keinen Unterschied zwischen Request und Response, weshalb beide Nachrichtenarten in Objekten des Typs *ACDWPPacket* gespeichert werden.

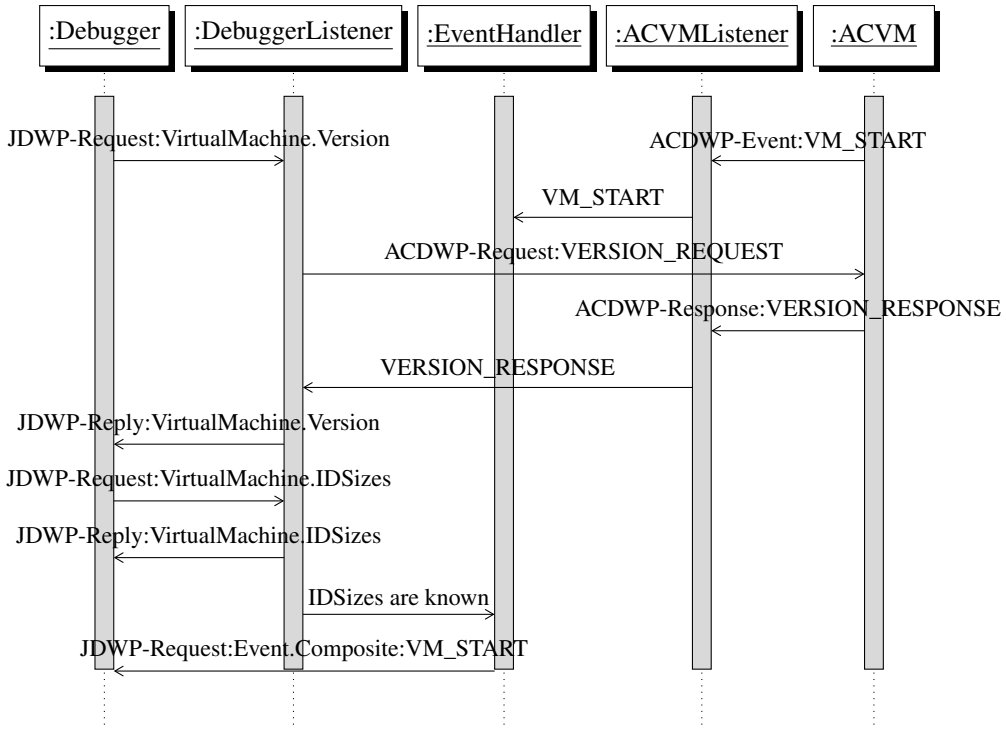

Abbildung 5.3: kein Deadlock: Der ACVMListener empfängt eine Eventnachricht von der ACVM. Gleichzeitig empfängt der DebuggerListener ein JDWP-Request vom Debugger. Der ACVMListener übergibt das Event an den EventHandler und wartet auf die nächste eingehende Nachricht von der ACVM. Der EventHandler kann das Event nicht an den Debugger senden, da dieser die Nachricht mit den ID-Größen noch nicht von DebuggerListener bekommen hat. Solange der EventHandler die Nachricht nicht senden kann, wartet er auf eine Benachrichtigung, dass die IDSizes dem Debugger mitgeteilt wurden. Der DebuggerListener sendet den empfangenen JDWP-Request nach der Umwandlung in einen ACDWP-Request an die ACVM. Nun wartet der DebuggerListener solange, bis die Antwort empfangen ist. Er kann zu diesem Zeitpunkt keine weitere Nachrichten vom Debugger empfangen. Die ACVM sendet die Antwort auf den ACDWP-Request des DebuggerListeners an den ACVMListener. Der ACVMListener empfängt die Nachricht mit der Antwort auf den Request und benachrichtigt den DebuggerListener. Der DebuggerListener sendet die Antwort an den Debugger. Der Debugger fragt im nächsten JDWP-RequestPaket an den DebuggerListener die IDSizes ab. Der DebuggerListener antwortet ihm. Zusätzlich informiert er den EventHandler darüber, dass die IDSizes nun bekannt sind. Der EventHandler kann nun das noch ausstehende VM_START-Event an den Debugger senden.

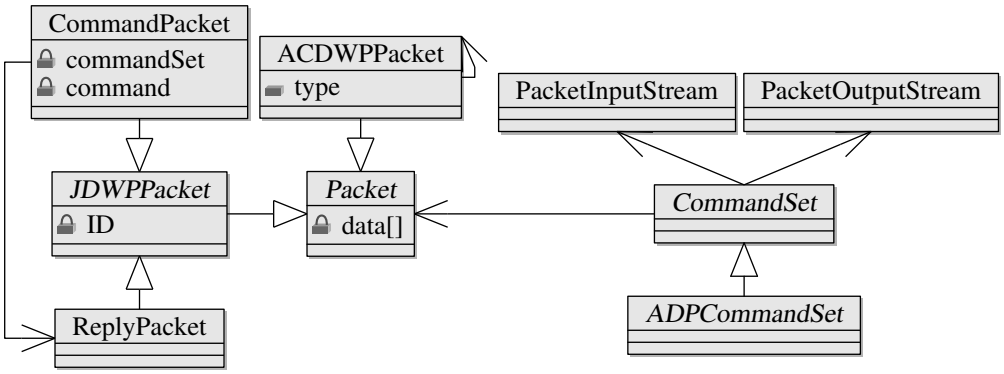

Abbildung 5.4: Packet-Class-Diagram

Innerhalb der Objekte liegen die Daten der Nachricht als Bytearray vor (siehe Abbildung 5.4). Die Felder der Header werden in Variablen innerhalb der Objekte gespeichert.

Um eine Nachricht zu erstellen, wird ein Objekt der passenden Klasse erzeugt. Bei JDWP-Requests werden die Felder *commandSet* und *command* entsprechend dem gewünschten Kommando der Nachricht initialisiert. Bei ACDWP-Requests bestimmt das Feld *type* das Kommando der Nachricht.

Ein Objekt vom Typ *PaketDataOutputStream* realisiert den schreibenden Zugriff auf die Daten einer zu sendenden Nachricht. Der lesende Zugriff auf die Daten einer empfangenen Nachricht erfolgt mit einem Objekt vom Typ *PaketDataInputStream*. Beide Klassen stellen Funktionen bereit, die es ermöglichen, primitive Datentypen (Byte, Integer) je nach Art in ein Bytearray zu schreiben oder von diesem zu lesen. Die Klasse *Packet*, von der alle anderen Pakettypen abgeleitet sind, hat Funktionen, die diese Objekte als Rückgabewert zurückgeben.

Zu sendende Nachrichtenobjekte (Pakete) werden dem *Connection*-Objekt des passenden Listeners übergeben, welches das Paket zur Gegenstelle sendet.

Zum Empfang eines Paketes von einer Gegenstelle erstellt das jeweilige Objekt der Klasse *Connection* auf Anfrage des Listeners ein Objekt einer der Unterklassen von *Packet*. Der Listener übergibt es dem passenden Handler (Unterklasse von ADPCommandSet), welcher es verarbeitet.

5.2 DebuggerListener-Thread

Dieser Abschnitt beschreibt, wie JDWP-Kommandos vom DebuggerListener-Thread verarbeitet werden. Der Ablauf ist in den Abbildungen 5.5 und 5.6 als Sequenzdiagramm dargestellt.

Zum Empfang eines Paketes vom Debugger ruft der DebuggerListener-Thread in der Abarbeitungsschleife die Funktion zum Empfangen eines neuen Paketes seines *Connection*-Objektes auf. Diese Funktion realisiert den Empfang eines JDWP-Paketes. Die Methode liest nacheinander die Felder des Headers eines JDWP-Paketes vom Eingabestrom der Netzwerkkarte.

Mit diesen Daten erzeugt die Methode ein neues Objekt der Klasse *CommandPacket*. Der Konstruktor dieser Klasse liest die kommandoabhängigen Daten des Paketes vom Eingabestrom. Das erzeugte Objekt wird als Rückgabeparameter an den DebuggerListener übergeben.

Die Abarbeitungsschleife des DebuggerListeners ruft nun die Methode zur Bestimmung des Nachrichtentyps auf. Diese Methode bestimmt mit Hilfe der Nummer des Feldes *commandSet* des empfangenen Paketes den für die Abarbeitung des Paketes zuständigen Handler in der alle Handler enthaltenden Hashtabelle des DebuggerListeners. Alle Handler werden bei der Initialisierung des DebuggerListeners erstellt und in dieser Hashtabelle mit ihrer Nummer als Key gespeichert. Der Typ der Handler ist *ADPCommandSet*, was die Oberklasse aller Kommandomengen ist (siehe Abbildung 5.4 und 5.1).

Nun wird die Abarbeitungsmethode des bestimmten Handlers aufgerufen.

Die Handlermethode initialisiert einige Instanzvariablen und holt den PacketInputStream des empfangenen Paketes und den PacketOutputStream des Antwortpaketes. Das Antwortpaket des PacketOutputStreams wird zu diesem Zweck zuvor erzeugt. Sowohl der PacketInputStream als auch der PacketOutputStream werden für die spätere Verwendung durch die Kommandomethoden in Instanzvariablen gespeichert. Kommandomethoden sind die Methoden in den Handlern, die die eigentliche Ausführung der Kommandos durchführen.

Zur Bestimmung der richtigen Kommandomethode, wird das Feld *command* im empfangenen Paket verwendet. Je nach Nummer wird eine andere Kommandomethode ausgeführt.

Abhängig vom Kommando werden kommandospezifische Daten vom PacketInputStream des empfangenen Paketes gelesen.

Der nun folgende Ablauf unterscheidet sich je nach Kommando:

- **Kommando an die ACVM (VirtualMachine.Resume) oder Informationsanfragen, die nur von der ACVM beantwortet werden können (StackFrame.GetValues):** Die gesammelte Information der Anfrage wird genutzt, um einen ACDWP-Request an die ACVM zu senden. Zu diesem Zweck wird ein ACDWP-Paket erstellt und mit den extrahierten Daten des JDWP-Requests beschrieben. Das Paket wird über den ACVMListener versendet und zusätzlich in die Liste der versendeten Request des ACVMListeners aufgenommen. Falls möglich wird die Information aus dem empfangenen JDWP-Paket auch zur Aktualisierung des proxyinternen Wissens über die Threadzustände der auf der ACVM laufenden Threads genutzt. Nach dem Senden des ACDWP-Paketes an die ACVM suspended sich der Thread des DebuggerListeners.

  Eine Antwort der ACVM wird vom ACVMListener in dessen Abarbeitungsschleife empfangen. Er erkennt anhand der Typnummer des ACDWP-Paketes, dass es sich um eine Antwort auf einen Request handelt und wählt aus der Liste der versendeten Requests den Richtigen aus. Der richtige Request bekommt eine Referenz auf das empfangene Antwortpaket. Nun benachrichtigt der ACVMListener den Thread des DebuggerListeners. Der ACVMListener beginnt daraufhin einen neuen Durchlauf seiner Abarbeitungsschleife.

  Der benachrichtigte Thread des DebuggerListeners fährt mit der Kommandomethode fort. Die Informationen des empfangenen ACDWP-Paketes werden ausgelesen. Wenn keine Umwandlung der empfangenen Daten notwendig ist, werden

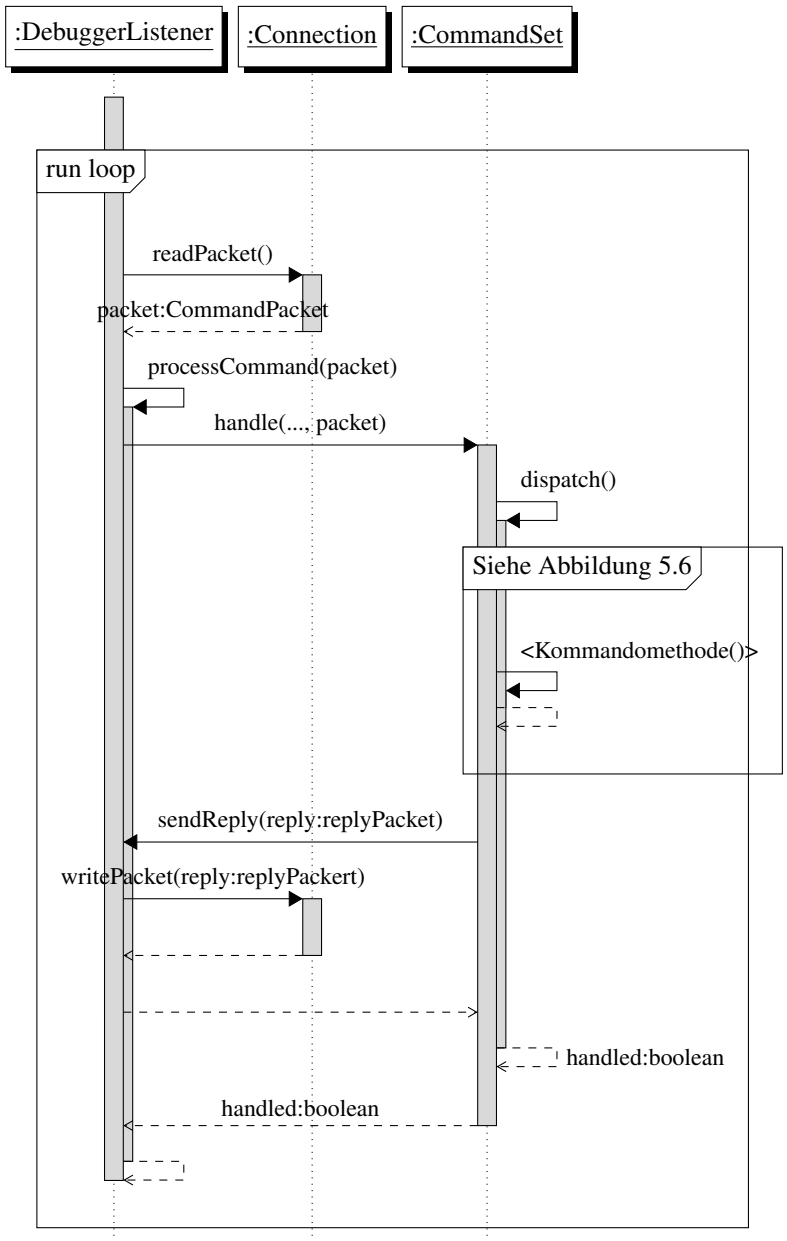

Abbildung 5.5: Abarbeitungsschleife für Kommandos vom Debugger

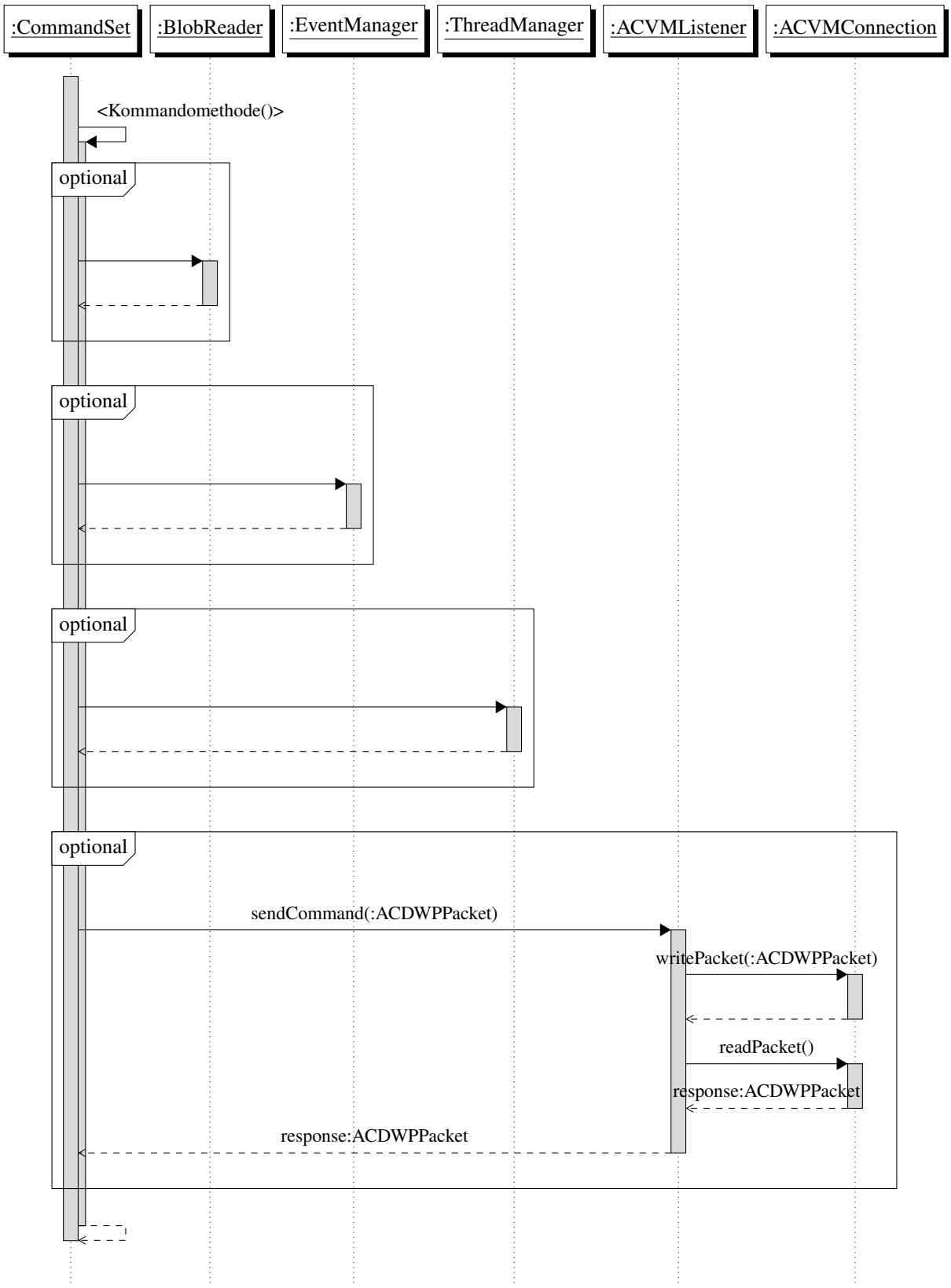

Abbildung 5.6: Kommandomethode

sie direkt in den Outputstream geschrieben. Andernfalls werden vor dem versenden der Antwort Informationen des Debug-BLOBs zur Transformation der Daten genutzt.

- **Informationsanfragen, die mit Informationen des Debug-BLOB beantwortbar sind:** Die gesammelten Informationen des JDWP-Requests werden genutzt, um mit Hilfe des ADPBlobReader die gewünschten Informationen aus dem Debug-BLOB zu extrahieren. Falls nötig findet eine Umwandlung des Datenformates statt. Die Antwort wird in den PacketOutputStream des Antwortpaketes geschrieben.

- **EventRequest (Breakpoint):** Wenn der empfangene EventRequest ein BreakPoint ist, wird ein ACDWP-Paket mit dem ACDWP-Kommando *ACDWP.BREAKPOINT* an die ACVM gesendet. Das Senden funktioniert, wie im ersten Fall beschrieben. Die Antwort der ACVM enthält die Breakpointnummer in der ACVM. Diese Nummer wird zusammen mit der Requestnummer in einem Objekt der Klasse *EventRequest* im ADPEventManager gespeichert. Das Objekt der Klasse *EventRequest* hat wiederum Objekte der Klasse *EventRequestModifier*, welche den Request genauer spezifizieren. Ein EventRequest eines Breakpoint hat zum Beispiel immer einen EventRequestModifier, der die Position des Breakpoints angibt.

  Zusätzlich wird ein *Event*-Objekt angelegt, welches der ACVMListener beim Auftreten des passenden Breakpoints verwendet, um den Debugger zu informieren. Das *Event*-Objekt des Breakpoints wird auch im ADPEventManager gespeichert. Als Antwort auf den JDWP-Request wird die Eventnummer in den PacketDataOutputStream des Antwortpaketes geschrieben.

- **EventRequest (ohne Breakpoint):** Ein zum Request passendes Objekt des Types *EventRequest* wird erstellt und im ADPEventManager gespeichert. Das Objekt der Klasse *EventRequest* enthält wiederum Referenzen auf Objekte der Klasse *EventRequestModifier*, welche den EventRequest genauer spezifizieren. Als Antwort auf den JDWP-Request wird die Eventnummer in den PacketDataOutputStream geschrieben.

Die Kommandomethode kehrt zurück zur Abarbeitungsmethode.

Die Methode *handler* schaut nach, ob der Request überhaupt eine Antwort benötigt und sendet diese gegebenenfalls durch Aufruf einer Methode des DebuggerListeners. Die Methode des DebuggerListeners ruft wiederum eine Methode des Connection-Objektes auf, die das JDWP-Reply-Paket der JDWP-Spezifikation entsprechend ins Netzwerk sendet.

Der Programmablauf kehrt zur Abarbeitungsschleife zurück, welche mit einem neuen Durchlauf beginnt und versucht das nächste Paket einzulesen.

5.3 ACVMListener-Thread

Zum Empfang eines Paketes von der ACVM ruft der DebuggerListener-Thread in der Abarbeitungsschleife die Funktion zum Empfangen eines neuen Paketes seines *Connection*-Objektes auf. Diese Funktion realisiert den Empfang eines ACDWP-Paketes. Die Methode liest nacheinander die Felder des Headers eines ACDWP-Paketes

(siehe 5.6) vom Eingabestrom der Netzwerkkarte. Das Feld *type* des ACDWP-Paketes wird genutzt um die Instanzvariablen *commandSet* und *Command* im Paketobjekt zu belegen. "*type*$/100$" ergibt den Wert für *commandSet*. "*type*$\%100$" ergibt den Wert für *command*. Somit bestimmt *commandSet*, ob das Paket ein Request, Response oder Event ist (siehe 5.6).

Mit diesen Daten erzeugt die Methode ein neues Objekt der Klasse *ACDWPPacket*. Der Konstruktor dieser Klasse liest die kommandoabhängigen Daten des Paketes vom Eingabestrom. Das erzeugte Objekt wird als Rückgabeparamter an den ACVMListener übergeben.

Wenn das empfangene Paket eine Antwort auf ein vom DebuggerListener versendetes Request-Paket ist, wird das zugehörige Request-Paket aus der Liste *sentCommands* herausgenommen. Das Request-Paket bekommt eine Referenz auf das empfangene Response-Paket. Der ACVMListener benachrichtigt den DebuggerListener über das Eintreffen einer Antwort. Die Abarbeitungsschleife beginnt von vorne.

Enthält das empfangene ACDWP-Paket ein ACDWP-Event, so wird die Methode zur Bestimmung des für die Paketverarbeitung zuständigen Handlers aufgerufen. Die Bestimmung des korrekten Handlers geschieht wie beim DebuggerListener mit Hilfe es Feldes *commandSet*. Da die ACVM dem ACVMListener nur Response- und Event-Nachrichten senden kann und Response-Nachrichten bereits verarbeitet wurden, gibt es momentan nur den Handler für Event-Nachrichten.

Die Handlermethode bereitet, wie im DebuggerListener, die Verarbeitung des empfangenen Paketes vor und startet dann die eigentliche Kommandomethode, welche die empfangene Nachricht verarbeitet. Je nach Kommandomethode erfolgt eine unterschiedliche Verarbeitung.

- **Event (Fehlermeldung)** Wenn das empfangene Event eine Fehlermeldung der ACVM ist, so wird aus dem in der Nachricht enthaltenen Fehlercode eine für den Menschen verständliche Mitteilung generiert. Die Mitteilung wird auf der Konsole ausgegeben und der ADP beendet.
- **Event (keine Fehlermeldung)** Alle Events, die keine Fehlermeldung der ACVM sind, werden zur Aktualisierung der im ADP gespeicherten Informationen verwendet.

  Falls sich der Debugger für ein empfangenes Event registriert hat, wird das Event in ein Objekte des Typs *Event* gewandelt und in der Liste *eventList* gespeichert. Der EventHandler-Thread wird benachrichtigt. Die Events in der Liste werden vom EventHandler-Thread mittels eines JDWP-Request-Paketes an den Debugger versendet.

Die Abarbeitungsschleife des ACVMListeners beginnt von vorne.

5.4 Verarbeitung von JDWP-Kommandos

Im JDWP sind Kommandos gruppiert. Die im folgenden aufgelisteten Kommandogruppen sind im ADP implementiert:

- VirtualMachine (1)

- ReferenceType (2)
- ClassType (3)
- Method (6)
- ObjectReference (9)
- StringReference (10)
- ThreadReference (11)
- ThreadGroupReference (12)
- ArrayReference (13)
- EventRequest (15)
- StackFrame (16)

Der Name der Gruppen identifiziert die Funktion der in der Gruppen enthaltenen Kommandos.

Die Liste enthält nur einen Teil der in der JDWP-Spezifikation enthaltenen Gruppen. Alle nicht aufgelisteten Gruppen werden vom ADP nicht unterstützt. Der ADP hat nur die Debugfunktionen implementiert, die zum Debuggen notwendig sind. Bestimmt wurden diese Gruppen mit Hilfe des Squawk-Systems und eigenen Versuchen mit der ACVM. Da es keine Liste mit umbedingt notwendigen JDWP-Kommandos gibt, war dies die einzige Möglichkeit, die benötigten Kommandogruppen zu bestimmen.

Bei Kommandogruppen ist die Nummer in den Klammern die eindeutige Kommandomengennummer.

Eine Liste aller im ADP implementierten JDWP-Kommandomengen und JDWP-Kommandos befindet sich im Anhang B.

Für jede Kommandomenge existiert ein Handler. Jeder Handler implementiert die zu seiner Gruppe gehörenden Kommandos. Für jedes Kommando existiert eine Kommandomethode. Die Kommandomethode realisiert die Funktion des Kommandos und erstellt die Antwort.

5.5 Implementierung einer Kommandomenge

Diesem Abschnitt zeigt am Beispiel der Kommandomenge *VirtualMachine*, wie Kommandomengen im ADP implementiert sind. Alle Kommandomengen erben vom Typ *ADPCommandSet*. Jede Kommandomengenklasse implementiert die zur eigenen Kommandomenge passenden Kommandos.

Im Quellcode befinden sich die Implementierungen der Komandomengen und Kommandos im jeweiligen Java-File des passenden Listeners. Der Quellcode der implementierten JDWP-Kommandomengen ist in der Datei *DebuggerListener.java*. Die ACDWP-Kommandomengen sind in der Datei *ACVMListener.java*. Die Bedeutung der einzelnen JDWP-Kommandos kann man in der Spezifikation des JDWP finden [2]. Die Bedeutung aller ACDWP-Kommandos steht im Anhang E.

Es folgt nun beispielhaft eine Erklärung der Implementierung einzelner JDWP-Kommandos.

Kommando: VirtualMachine.Version

Mit diesem JDWP-Kommando bestimmt der Debugger die Versionsnummern der ACVM und des JDWP.

Der ADP fragt die ACVM nach ihrer Versionsnummer. Dies geschieht mit dem ACDWP-Request *ACDWP.VERSION*.

Die Versionsnummer des JDWP ist 1.2, da diese Version des JDWP dem Funktionsumfang im ADP und der ACVM entspricht.

Die Versionsnummern werden dem Debugger mittels JDWP-Reply-Paket gesendet

Kommando: VirtualMachine.ClassesBySignature

Die Aufgabe dieses JDWP-Kommandos ist die Bestimmung aller zu einer Signatur passenden Klassen und Interfaces. Die Rückgabe besteht aus dem TypeTag und der eindeutigen ClassID der zur Signatur passenden Klassen und Interfaces. Der TypeTag gibt jeweils an, ob es sich um ein Interface oder eine Klasse handelt.

Zur Abarbeitung diese JDWP-Kommandos wird als erstes die Signatur aus dem JDWP-Request-Paket ausgelesen. Die Signatur der Klasse *Object* ist zum Beispiel *Ljava/lang/-Objekt;*

Um die Signatur mit den Signaturen der Klassen in den BLOBs vergleichen zu können, wird das *BaseType*-Zeichen *(L)* und der *;* aus dem String entfernt.

Der verbleibende String enthält nur den Paketpfad und den Klassennamen.

Die Signaturen aller Klassen in allen BLOBs werden mit dem String vergleichen. Der Zugriff auf die Klasseninformationen in den BLOBs erfolgt mittels ADPBlobReader. Klassen, deren Signatur mit dem String übereinstimmt, werden in eine Liste Aufgenommen.

Für alle Klassen in der Liste wird die classID bestimmt. Die ClassID wird folgendermaßen gebildet: $classID = classPosition + BLOBNumber * factor$. Sie ist eine eindeutige Identifikationsnummer. Die Umrechnung ist notwendig, da die *classPosition* bei mehreren BLOBs nicht mehr eindeutig ist. Die *classPosition* einer Klasse ist die Position der Klasse innerhalb eines BLOBs.

Der TypeTag und die ClassID aller Klassen in der Liste, wird an den Debugger mittels JDWP-Reply-Packet gesendet.

Kommando: VirtualMachine.AllClasses

Die Aufgabe dieses JDWP-Kommandos ist die Bestimmung aller Klassen und Interfaces. Die Rückgabe besteht aus dem TypeTag, der eindeutigen ClassIDs, der Signatur und dem Status jeder Klasse oder jedes Interfaces. Der TypeTag gibt jeweils an, ob es sich um ein Interface oder eine Klasse handelt. Der Status sagt aus, ob die Klasse von der VM bereits geladen wurde, was bei der ACVM immer der Fall ist.

Zur Abarbeitung diese JDWP-Kommandos wird mit Hilfe des ADPBlobReaders eine Liste mit alles Klassen in allen BLOBs erstellt. Für jede Klasse oder Interface wird der TypeTag, die ClassID, die Signatur und der Status bestimmt. Die dafür nötigen Informationen über die Klassen und Interfaces, werden mit Hilfe des ReverseTranscoder aus den Debug-BLOBs entnommen.

Die ClassID wird folgendermaßen gebildet: $classID = classPosition + BLOB - Number *$

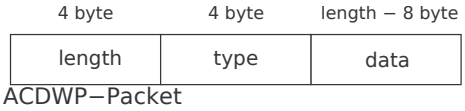

Abbildung 5.7: ACDWP-Paketformat

factor. Sie ist eine eindeutige Identifikationsnummer. Die Umrechnung ist notwendig, da die classPosition bei mehreren BLOBs nicht mehr eindeutig ist. Die *classPosition* einer Klasse ist die Position der Klasse innerhalb eines BLOBs.

Die ermittelten Daten aller Klassen und Interfaces werden mittels JDWP-Paket an den Debugger gesendet.

5.6 Aufbau eines ACDWP-Paketes

Ein ACDWP-Paket Abb.5.7 besteht aus zwei oder mehr Integer-Werten (1 Integer = 4 Byte). Die ersten beiden Integer-Werte sind bei allen drei Arten eines ACDWP-Paketes gleich. Das erste Feld (1 Integer) ist die Länge des gesamten Paketes in Integer einschließlich des Längenfeldes. Im Längenfeld steht somit immer eine Zahl größer oder gleich 2. Das zweite Feld ist der Typ. Der Typ bestimmt die Bedeutung und die Art des ACDWP-Paketes.

- Typ = 000-099: Ist der Wert des Typs kleiner als 100, so handelt es sich um ein Request-Paket. Ein Paket mit Typ = 001 ist zum Beispiel zur Abfrage der Versionsinformation der ACVM bestimmt.
- Typ = 100-199: Ist der Typ zwischen 100 und 199, so handelt es sich um eine Antwort auf einen Request. Ein Paket mit Typ = 101 ist die Antwort auf eine Anfrage mit Typ = 001. Die Typnummern sind so gewählt, dass der Typ der Response genau der Typ des Requests plus 100 ist.
- Typ = 200-255: Die Typnummern der Events beginnen bei 200 und enden bei 255. Das Ereignis mit dem Typ 255 ist das Error-Event (Fehlernachricht).

Die Bedeutung aller Felder, die sich eventuell hinter dem Typ befinden, ist nur mit Hilfe des Typs erkennbar. Eine genaue Auflistung der spezifizierten ACDWP-Kommandos und der zu ihnen gehörenden Felder befindet sich im Anhang E.

Kapitel 6

Zusammenfassung

6.1 Herausforderung und Lösung

Um Java-Programmen und die ACVM auf Kleinstcomputern mittels eines beliebigen Java-Debugger zu debuggen, gibt es einige Herausforderungen, die es zu bewältigen gilt.

Kleinstcomputer haben Beschränkungen bezüglich der Rechenleistung und der Speicherkapazität. Die Implementierung des für das Remote-Debugging von Sun entworfenen Java-Debug-Wire-Protocol (JDWP) in die ACVM ist deshalb aus Ressourcengründen keine geeignete Lösung. Das Schreiben eines neuen Java-Debuggers, der ein ressourcensparendes Protokoll zur Kommunikation mit der ACVM nutzt, widerspricht dem Ziel einen beliebigen Java-Debugger verwenden zu können.

Der einzige praktikable Ansatz ist die Verwendung eines Proxys zwischen Debugger und ACVM. Der Proxy läuft dabei nicht auf der beschränkten Hardware. Er kommuniziert mit dem Debugger mittels JDWP, was die Verwendung eines beliebigen Java-Debuggers ermöglicht. Für die Kommunikation mit der ACVM auf der ressourcenbeschränkten Hardware, wird das eigens für diesen Zweck entwickelte AmbiComp-Debug-Wire-Protocol(ACDWP) verwendet. Es ermöglicht deutlich kleinere Nachrichten.

Um die Ressourcen der Hardware noch mehr zu schonen, beantwortet der Proxy einen großen Teil der Anfragen des Debuggers mit Hilfe einer speziellen BLOB-Datei, welche extra für den Proxy erstellt wird. Dies ermöglicht es den BLOB für die Hardware auf die für die Ausführung des Java-Programms notwendigen Daten zu reduzieren, was den Speicherplatz auf der Hardware schont. Im Gegensatz zur BLOB-Datei für die Hardware, enthält der BLOB für den Proxy alle zum Debuggen wichtigen Informationen und kann somit als Informationsbasis dienen.

Die dritte Funktionalität des Proxys ist das Speichern von Zustandsinformationen über Threads. Anfragen des Debuggers werden wenn möglich mit den gesammelten Threadinformationen beantwortet. Dies verhindert unnötige Anfragen an die ACVM.

6.2 Ausblick

Der bisher entwickelte Proxy implementiert nur die JDWP-Version 1.2. Da die ACVM jedoch keine Java-Funktionen der neueren Versionen implementiert ist die Unterstützung für höhere JDWP Versionen bisher nicht nötig. Sollte jedoch die Funktionalität der ACVM erweitert werden, so ist gegebenfalls auch eine Erweiterung des ADP notwendig.

Es gibt keine Gewähr, dass alle benötigten JDWP-Kommandos im ADP implementiert sind. Dies ist der Fall, da zur Bestimmung der benötigten Befehle, die Kommunikation im Squawk-System zwischen Java-Debugger und SDP ausgewertet wurde. Es wurden mehrere Szenarien mit dem ADP und der ACVM durchgetestet, doch kann dadurch nicht ausgeschlossen werden, dass es Szenarien gibt bei denen noch nicht implementierte JDWP-Kommandos benötigt werden. Eine Aufgabe für die Zukunft ist somit die Bestimmung und Implementierung eventuell noch fehlender JDWP-Kommandos.

Anhang A

Die Hardware des AmbiComp-Projektes

Die Hardware des AmbiComp-Projektes besteht aus einzelnen Sandwich-Modulen (SM), welche zu einer AmbiComp-Control-Unit zusammengesteckt werden können. Die zusammengesteckten SMs kommunizieren mit Hilfe der Backplane. Für die Kommunikation zwischen AICUs werden die Schnittstellen der einzelnen SMs verwendet.

Im AmbiComp-Prokekt gibt es folgende SMs:

SMs:					
Name:	IOSM	BTSM	EtherSM	PoESM	PeriSM
CPU:	8 bit	8 bit	8 bit		
Clock:	16 MHZ	7,37 MHZ	8 MHZ		
Ext. Mem:		512 KB	512 KB		
Flash:	256 KB	256 KB	256 KB		
Comm.:		BT 2.0	Ethernet		
I/O:	16x digital 16x ADC 4x DAC				
Power:	5 V	3,3 V	3,3 V		
Power-Supply:				3,3 V 5 V	3,3 V 5 V

Für eine noch genauere Beschreibung der Hardware sei auf [6] verwiesen.

Anhang B

Java-Debug-Wire-Protocol (JDWP)

JDWP [2] ist das von Sun für die Debugkommunikation zwischen Java-Debugger und JVM vorgesehene Protokoll. Es gehört zu Schicht 7 des ISO/OSI-Schichtenmodells. Für die unteren Schichten wird normalerweise TCP/IP verwendet. JDWP geht von eine sicheren Verbindung aus. Es gibt keine Fehlerkorrektur auf Schicht 7.

B.1 Paketaufbau

Zwei Paketarten B.1 werden vom JDWP spezifiziert:

- **Request-Packet:** Wird vom Java-Debugger verwendet, um der JVM Kommandos zu erteilen, sich für Ereignisse in der VM zu registrieren oder Informationen abzufragen.

  Die JVM verwendet Request-Pakete um Ereignisse mitzuteilen.

- **Reply-Packet:** Im Reply-Paket stecken die Antwortdaten eines JDWP-Kommandos.

Der Header des JDWP-Request-Paketes besteht aus folgenden Feldern:

- **length:** Die Länge des kompletten Paketes einschließlich dem Feld length.
- **ID:** Eine eindeutige ID, welche zur Identifikation der Antwort benötigt wird.
- **flags:** Flags der Nachricht. Hat für das JDWP-Command-Paket keine Bedeutung.
- **command set:** Nummer zur Identifikation der JDWP-Kommandomenge.
- **command:** Nummer zur Identifikation des JDWP-Kommandos innerhalb der JDWP-Kommandomenge
- **data:** Die eigentlichen Daten eines Paketes. Die Bedeutung der Daten ist vom JDWP-Kommando abhängig.

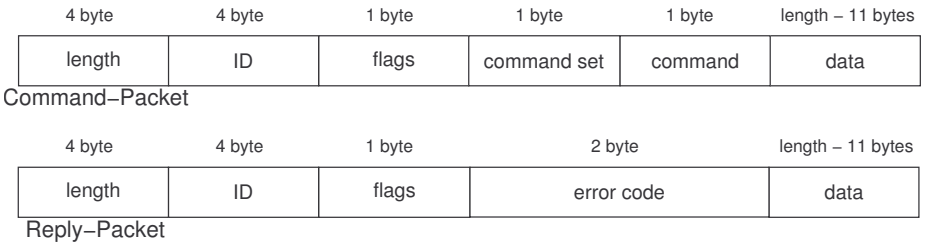

Abbildung B.1: JDWP-Packet-Formats

Die Antwort auf einen JDWP-Request wird mittels JDWP-Reply-Paket gesendet:

- **length:** Die Länge des kompletten Paketes einschließlich dem Feld length.
- **ID:** Eine eindeutige ID. Ist Identisch zur ID des zugehörigen JDWP-Request-Paketes.
- **flags:** flags = 0x80 : identifiziert ein JDWP-Reply-Paket. Andere Werte für flags haben bisher keine Bedeutung.
- **error code:** Nummer eines während der Verarbeitung des Requests aufgetretenen Fehlers.
- **data:** Die eigentlichen Daten eines Paketes. Die Bedeutung der Daten ist vom JDWP-Kommando des JDWP-Requests abhängig.

Die Größe der Header ist bei JDWP-Request-Packets und JDWP-Reply-Packets identisch, was eine Auswertung der JDWP-Pakete erleichtert.

B.2 Im ADP implementierte JDWP-Kommandomengen und Kommandos

Im JDWP sind Kommandos gruppiert. Folgende Kommandomengen und Kommandos sind im ADP implementiert:

JDWP	
Kommandomenge (commandSet)	Kommando (command)
VirtualMachine (1)	Version (1) ClassesBySignature (2) AllClasses (3) AllThreads (4) TopLevelThreadGroups (5) Dispose (6) IDSizes (7) Suspend (8) Resume (9) Capabilities (12) ClassPaths (13) DisposeObjects (14) CapabilitiesNew (17)
ReferenceType (2)	Signature (1) ClassLoader (2) Modifiers (3) Fields (4) Methods (5) GetValues (6) SourceFile (7) NestedTypes (8) Status (9) Interfaces (10)
ClassType (3)	Superclass (1) SetValues (2)
Method (6)	LineTable (1) VariableTable (2)
ObjectReference (9)	ReferenceType (1) GetValues (2) SetValues (3) InvokeMethod (6)
StringReference (10)	Value (1)
ThreadReference (11)	Name (1) Suspend (2) Resume (3) Status (4) ThreadGroup (5) Frames (6) FrameCount (7)
ThreadGroupReference (12)	Name (1) Parent (2) Children (3)
ArrayReference (13)	Length (1) GetValue (2)
EventRequest (15)	Set (1) Clear (2)
StackFrame (16)	GetValue (1) ThisObject (3)

Der Name einer Gruppe deutet hierbei auf den Kontext der enthaltenen Kommandos hin. Die Nummern in den Klammern sind die Kommandomengennummern (command set), welche jede Gruppe eindeutig identifizieren.

Anhang C

Vorbereitung einer Debugsession

C.1 Die benötigten Programme

Die an einer Debugkommunikation beteiligten Programme und ihre Kommunikationskanäle untereinander sieht man in Abbildung C.1. Zum Debuggen müssen folgende Programme installiert sein:

- **Linux-BIOS:** Das Linux-BIOS bietet eine Abstraktion von der Hardware. Es stellt eine Schnittstelle bereit um auf die Funktionen der Hardware zuzugreifen. Die Lunix-ACVM benutzt die Funktionalität des Linux-BIOS.
- **Linux-ACVM:** Die Linux-ACVM ist die JVM des AmbiComp-Projektes.
- **OTTO:** Um die Kommunikation mit der ACVM von deren Aufenthaltsort unabhängig zu machen, wird der OTTO benötigt. Alle Kommunikation zwischen der ACVM und dem ADP läuft über den OTTO. Falls nötig ändert er auch die Protokolle der ISO/OSI-Schichten 1-6, um eine ACVM auf einer AICU zu erreichen.
- **ADP:** Der Ambicomp-Debug-Proxy ist für die Kommunikation zwischen Java-Debugger und ACVM zuständig. Zum Verbindungsaufbau kommuniziert er auch mit dem OTTO.
- **Java-Debugger:** Ein Java-Debugger der Remote-Debugging mittels JDWP unterstützt.

Dies sind alle Programme, die zum Starten und Debuggen eines BLOBs benötigt werden. Um auch BLOBs erzeugen zu können, wird noch folgendes benötigt:

- **Transcoder:** Der Transcoder wandelt Java-Class-Files in BLOBs. Java-Programme können von der ACVM nur in der Form eines BLOBs verarbeitet werden.
- **AmbiCompAPI_mini oder AmbiCompAPI_CLDC:** Beide Bibliotheken stellen die grundlegenden Java Klassen bereit. Eine von Beiden wird deshalb immer benötigt.

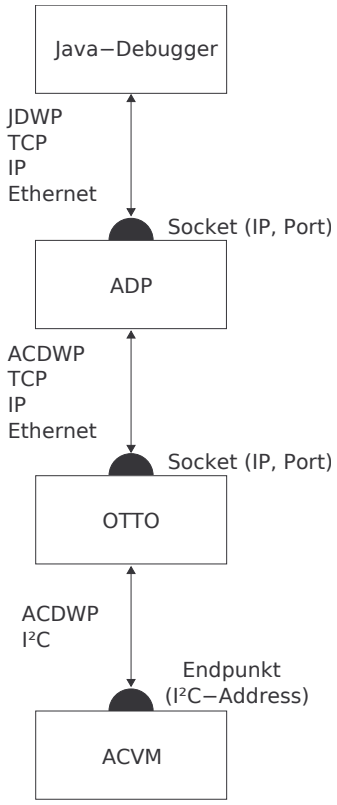

Abbildung C.1: ADP-Kommunikation

Nach der Installation aller erwähnten Programme, ist das Schreiben, Ausführen und Debuggen von eigenen Java-Programmen auf der ACVM möglich.

C.2 BLOBs erstellen (Transcodieren)

Um ein Java-Programm auf einer ACVM debuggen zu können, werden die Java-Class-Files des Programms und alle benötigten Java-Class-Files der Bibliothek mittels Transcoder in einen BLOBs gewandelt. Die entsprechenden Befehle für den Transcoder lauten:

- Debug-BLOB für den ADP: *java -cp transcoder.jar transcoder.Transcoder -o <output file> –debuginfo –codeinfo –classpath <jars (libraries)> <class files>*
- ACVM-BLOB für die ACVM: *java -cp transcoder.jar transcoder.Transcoder -o <output file> –classpath <jars (libraries)> <class files>*

Die Debugoptionen *debuginfo* und *codeinfo* veranlassen den Transcoder zusätzliche zum Debuggen benötigte Information in den BLOB zu schreiben. Ein BLOB der ohne diese Optionen beim Transcodieren entsteht, ist nicht für den ADP geeignet. Die ACVM kann auch Debug-BLOBs verarbeiten. Da der Debug-BLOB jedoch mehr Speicherplatz benötigt, ist dies nicht sinnvoll.

C.3 ACVM und ADP mit jeweiligem BLOB starten

Vor dem Start der ACVM muss zuerst der OTTO gestartet werden.

Um die ACVM bei ihrer Abarbeitung des Programms zu beobachten, kann zusätzlich das Programm *acmonitor* gestartet werden. Für das Programmdebugging mit Hilfe des ADP, wird der *acmonitor* aber nicht benötigt.

Als Nächstes wird die ACVM und dann der ADP mit folgenden Shell-Aufrufen gestartet:

- ACVM: *acvm –blob-file <ACVM-BLOB file>*
- ADP: java -cp ADP.jar ADP <Debug-BLOB file> <access port for the debugger> <OTTO ip> <OTTO port> <aicu>,<sm>,<Endpunkt>

Der ADP verbindet sich automatisch mit dem OTTO und von dort weiter zu der in den Parametern angegebenen ACVM. Danach wartet er auf eine eingehende Verbindung auf dem Port des Debuggers.

C.4 Debugger einstellen und Debugsession starten

Der letzte Schritt zum Start der Debugsession ist die Einstellung des Debuggers. Bei Eclipse kann dies durch das Erzeugen einer neuen *debug-configuration* (Run->Debug configurations...) geschehen. Im Dialog erstellt man einen neuen *Remote Java Applications*-Launcher. Dem Launcher teilt man die IP-Adresse und die Port-Nummer des ADP sowie das zu debuggende Projekt mit.

Durch das Starten des Launchers beginnt die Debugsession. Der Debugger verbindet sich mit dem ADP und beginnt Informationen über das zu debuggende Programm abzufragen. Zusätzlich registriert er sich für Ereignisse und meldet Breakpoints an.

Nach der initialisierenden Kommunikation sendet er dem ADP das Startkommando (resume) und das JAVA-Programm auf der ACVM beginnt.

Anhang D

Squawk zum Debuggen nutzen

In den meisten Fällen genügt zur Bestimmung der Funktionalität von JDWP-Kommandos die Spezifikation [2]. Wenn einem dies jedoch nicht ausreicht, so kann man die Implementierung von JDWP-Kommandos in Squawk anschauen. Eine weitere Möglichkeit ist das Durchführen einer Squawk-Debugsession unter Verwendung des Squawk-Debug-Proxy (SDP). Wie man Squawk zu diesem Zweck verwendet, wird im Folgenden erklärt.

D.1 Installation von Squawk

Squawk kann von der Website des Squawk-Projektes heruntergeladen werden [5].

Vor der Installation muss man zwei Umgebungsvariablen setzen. Die folgenden Befehle setzt die benötigten Umgebungsvariablen:

- **export JAVA_HOME=<Verzeichnis des JDK>**
- **export LD_LIBRARY_PATH=<Verzeichnis des JDK>/jre/lib/i386/server: <Verzeichnis des JDK>/jre/lib/i386**

Nach dem Entpacken der heruntergeladenen Datei compiliert man Squawk mit folgenden Befehlen, welchen man im Squawk-Hauptverzeichnis ausführt:

- **d copyphoneme**
- **d && d -prod -mac -o2 rom -metadata cldc imp debugger**

D.2 Suite-Files erstellen und ausführen

In allen Projektverzeichnissen von Projekten aus denen ein Suite-File erstellt werden soll, muss falls nicht vorhanden das Verzeichnis *res* erstellt werden. Wenn das Verzeichnis nicht existiert, kann es bei der Erstellung von Suite-Files zu Fehlern kommen.

Nun können Suite-Files von eigenen Java-Programmen erzeugt werden. Folgender Befehl erstellt ein Suite-File aus dem als Parameter angegebenen Java-Projekt:

- **d user-suite <Projektpfad des Java-Programms>**

Um ein Suite-Files auf der Squawk-VM zu starten, kann man folgenden Befehl verwenden:

- **squawk -suitepath:<Projektpfad des Java-Programms>:. -suite:<Suite-File des Java-Programms> <Main-Class>**

D.3 Debugging mit Squawk

Um Programme auch debuggen zu können, muss sowohl der Squawk-Debug-Proxy als auch der Squawk-Debug-Agent gestartet sein:

- **squawk -suitepath:<Projektpfad des Java-Programms>:. -suite:<Suite-File des Java-Programms> com.sun.squawk.debugger.sda.SDA <Main-Class>**
- **d sdproxy -log:debug -cp:<Projektpfad zu den Class-Files des Java-Programms>:squawk_classes.jar -logFile:<Logdatei>**

Der erste Befehl startet den Squawk-Debug-Agent (SDA), welcher wiederum die Main-Methode in der Main-Class des zuvor erstellte Suite-File startet. Der SDA wartet vor der Ausführung des Java-Programms auf die Verbindung zum Squawk-Debug-Proxy (SDP) und das Startsignal des Java-Debuggers.

Der zweite Befehl startet den Squawk-Debug-Proxy (SDP). Die Parameter bestimmen, wo sich die Java-Class-Files des zu debuggenden Java-Programms befinden. Der Parameter *log* regelt das Debuglevel. Die Ausgabe kann mit dem Parameter *logfile* in eine Datei umgeleitet werden.

Anhang E

ACDWP-Nachrichten

Die ACDWP-Nachrichten unterteilen sich in die drei Typen: *Request*, *Response* und *Event*.

Alle 3 Typen halten sich an das grundlegende ACDWP-Paketformat, wie es in Abbildung 5.7 zu sehen ist. Ein Paket besteht immer aus den Feldern: *length*, *type* und *data*. Der Inhalt des Feldes *data* ist von der Nachricht abhängig.

Der Inhalt des Feldes *data* wird im Folgenden zuerst für alle Request- und Response-Nachrichten erklärt. Danach folgen die Event-Nachrichten.

E.1 Request & Response

VERSION

Dieses Kommando wird verwendet, um die Versionsnummer der ACVM abzufragen.

VERSION_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	001	Typ der Nachricht
Paketdaten:		

VERSION_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	101	Typ der Nachricht
Paketdaten:		
versionNumber	[int]	Versionsnummer der ACVM

THREAD_RUNNING_LIST

Dieses Kommando fragt die laufenden Threads in der ACVM ab.

THREAD_RUNNING_LIST_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	002	Typ der Nachricht
Paketdaten:		

THREAD_RUNNING_LIST_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2 + <Anzahl Threads>	Größe des Paketes in Integer
type	102	Typ der Nachricht
Paketdaten:		
Das folgende Feld wiederholt sich *length-2* mal		
threadID	[int]	ID eines Threads in der ACVM

THREAD_BLOCKED_LIST

Dieses Kommando fragt die blockierten Threads in der ACVM ab.

THREAD_BLOCKED_LIST_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	003	Typ der Nachricht
Paketdaten:		

THREAD_BLOCKED_LIST_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2 + <Anzahl Threads>	Größe des Paketes in Integer
type	103	Typ der Nachricht
Paketdaten:		
Das folgende Feld wiederholt sich *length-2* mal		
threadID	[int]	ID eines Threads in der ACVM

VM_SUSPEND

Dieses Kommando suspendiert alle Threads in der ACVM. Die Threads werden nicht beendet.

VM_SUSPEND_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	004	Typ der Nachricht
Paketdaten:		

VM_SUSPEND_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	104	Typ der Nachricht
Paketdaten:		

VM_RESUME

Dieses Kommando startet alle laufbereiten Threads in der ACVM.

VM_RESUME_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	005	Typ der Nachricht
Paketdaten:		

VM_RESUME_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	105	Typ der Nachricht
Paketdaten:		

STATIC_FIELD_GET

Dieses Kommando fragt den Inhalt eines statischen Feldes in einer Klasse ab. Der Inhalt kann ein primitiver Datentyp oder eine Referenz sein.

STATIC_FIELD_GET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	006	Typ der Nachricht
Paketdaten:		
classID	[int]	ID einer Klasse
slot	[int]	Position des statischen Feldes in der Klasse

STATIC_FIELD_GET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	106	Typ der Nachricht
Paketdaten:		
value	[int]	Inhalt der statischen Variable

STATIC_FIELD_SET

Dieses Kommando schreibt einen Wert in ein statisches Feld in einer Klasse. Der Inhalt kann ein primitiver Datentyp oder eine Referenz sein.

STATIC_FIELD_SET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	007	Typ der Nachricht
Paketdaten:		
classID	[int]	ID einer Klasse
slot	[int]	Position des statischen Feldes in der Klasse
value	[int]	neuer Inhalt der statischen Variable

STATIC_FIELD_SET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	107	Typ der Nachricht
Paketdaten:		

DYNAMIC_FIELD_GET

Dieses Kommando fragt den Inhalt eines Feldes in einem Objekt ab. Der Inhalt kann ein primitive Datentyp oder eine Referenz sein.

DYNAMIC_FIELD_GET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	008	Typ der Nachricht
Paketdaten:		
referenceID	[int]	ID einer Klasse
slot	[int]	Position des Feldes in der Klasse des Objektes

DYNAMIC_FIELD_GET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	108	Typ der Nachricht
Paketdaten:		
value	[int]	Inhalt der Variable

DYNAMIC_FIELD_SET

Dieses Kommando schreibt einen Wert in ein Feldes eines Objektes. Der Inhalt kann ein primitive Datentyp oder eine Referenz sein.

DYNAMIC_FIELD_SET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	009	Typ der Nachricht
Paketdaten:		
referenceID	[int]	ID einer Klasse
slot	[int]	Position des Feldes in der Klasse des Objektes
value	[int]	neuer Inhalt der Variable

DYNAMIC_FIELD_SET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	109	Typ der Nachricht
Paketdaten:		

OBJECT_TYPE

Dieses Kommando fragt die ACVM nach dem Typ einer Referenz.

OBJECT_TYPE_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	010	Typ der Nachricht
Paketdaten:		
referenceID	[int]	ID einer Klasse

OBJECT_TYPE_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	110	Typ der Nachricht
Paketdaten:		
empty	[2 byte]	ungenutzt
dimension	[4 bit]	Dimension der Referenz
classID	[12 bit]	ID des Typs der Referenz

THREAD_SUSPEND

Dieses Kommando suspendiert einen Thread. Er wird nicht beendet.

THREAD_SUSPEND_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	011	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID einen Threads

THREAD_SUSPEND_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	111	Typ der Nachricht
Paketdaten:		

THREAD_RESUME

Dieses Kommando startet einen Thread.

THREAD_RESUME_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	012	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID einen Threads

THREAD_RESUME_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	112	Typ der Nachricht
Paketdaten:		

THREAD_STATUS

Dieses Kommando fragt den Status eines Threads ab.

THREAD_STATUS_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	013	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID einen Threads

THREAD_STATUS_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	113	Typ der Nachricht
Paketdaten:		
empty	[3 bytes]	ungenutzt
threadSuspendStatus	[1 bit]	gibt an, ob der thread suspended ist
threadStatus	[31 bit]	Status des Threads

FRAME

Dieses Kommando fragt alle Frames eines Threads ab. Der Thread muss suspended sein.

FRAME_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	014	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID einen Threads

FRAME_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	114	Typ der Nachricht
Paketdaten:		
frameOffset	[2 bytes]	relative Startadresse zum Framecontainer
programCounter	[2 byte]	Position an der der Thread im BLOB steht

ARRAY_LENGTH

Dieses Kommando fragt die Länge eines Arrays ab.

ARRAY_LENGTH_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	015	Typ der Nachricht
Paketdaten:		
referenceID	[int]	Referenz des Arrays

ARRAY_LENGTH_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	115	Typ der Nachricht
Paketdaten:		
length	[int]	Anzahl der Elemente in einem Array

ARRAY_ELEMENT_GET

Dieses Kommando fragt den Wert einer Position in einem Array ab.

ARRAY_ELEMENT_GET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	016	Typ der Nachricht
Paketdaten:		
referenceID	[int]	Referenz des Arrays
index	[int]	Position im Array

ARRAY_ELEMENT_GET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	116	Typ der Nachricht
Paketdaten:		
value	[int]	Wert

ARRAY_ELEMENT_SET

Dieses Kommando setzt den Wert einer Position in einem Array.

ARRAY_ELEMENT_SET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	5	Größe des Paketes in Integer
type	017	Typ der Nachricht
Paketdaten:		
referenceID	[int]	Referenz des Arrays
index	[int]	Position im Array
value	[int]	neuer Wert

ARRAY_ELEMENT_GET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	117	Typ der Nachricht
Paketdaten:		

BREAKPOINT_SET

Dieses Kommando setzt einen Breakpoint.

BREAKPOINT_SET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	018	Typ der Nachricht
Paketdaten:		
blobID	[int]	Nummer des BLOBs
bytecodeOffset	[int]	Position im BLOB

BREAKPOINT_SET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	118	Typ der Nachricht
Paketdaten:		
eventNumber	[int]	den Breakpoint identifizierende Nummer

BREAKPOINT_CLEAR

Dieses Kommando entfernt einen Breakpoint.

BREAKPOINT_CLEAR_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	019	Typ der Nachricht
Paketdaten:		
eventNumber	[int]	den Breakpoint identifizierende Nummer

BREAKPOINT_CLEAR_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	119	Typ der Nachricht
Paketdaten:		

BREAKPOINT_CLEAR_ALL

Dieses Kommando entfernt alle Breakpoints.

BREAKPOINT_CLEAR_ALL_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	020	Typ der Nachricht
Paketdaten:		

BREAKPOINT_CLEAR_ALL_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	120	Typ der Nachricht
Paketdaten:		

FRAME_SLOT_GET

Dieses Kommando liest eine lokale Variable im Frame.

FRAME_SLOT_GET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	021	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID eines Threads
frameOffset + slot-Number	[int]	frameOffset + slotNumber ergibt die absolute Position im Framecontainer

FRAME_SLOT_GET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	121	Typ der Nachricht
Paketdaten:		
value	[int]	Wert der lokalen Variable

FRAME_SLOT_SET

Dieses Kommando schreibt einen Wert in eine lokale Variable.

FRAME_SLOT_SET_REQUEST		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	5	Größe des Paketes in Integer
type	022	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID eines Threads
frameOffset + slot-Number	[int]	frameOffset + slotNumber ergibt die absolute Position im Framecontainer
value	[int]	neuer Wert der lokalen Variable

FRAME_SLOT_SET_RESPONSE		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	2	Größe des Paketes in Integer
type	122	Typ der Nachricht
Paketdaten:		

E.2 Event

Event-Nachrichten haben keine Antwort. Sie dienen zur Mitteilung von Ereignissen in der ACVM oder als Fehlerbenachrichtigung.

VM_START_EVENT

Mit dieses Event teilt die ACVM die fertige Initialisierung mit. Das Event wird vor dem Start des Main-Threads von der ACVM automatisch versendet.

VM_START_EVENT		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	3	Größe des Paketes in Integer
type	201	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID des Main-Threads

THREAD_STATE_CHANGE_EVENT

Mit diesem Event teilt die ACVM mit, dass sich der Zustand eines Threads auf der ACVM geändert hat.

THREAD_STATE_CHANGE_EVENT		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	202	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID des Main-Threads
empty	[3 byte]	ungenutzt
threadSuspendStatus	[1 bit]	Suspendstatus eines Threads
threadStatus	[31 bit]	Status eines Threads

BREAKPOINT_EVENT

Mit diesem Event teilt die ACVM mit, dass ein Thread bei einem Breakpoint angekommen ist.

BREAKPOINT_EVENT		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	203	Typ der Nachricht
Paketdaten:		
threadID	[int]	ID eines Threads
eventNumber	[int]	den Breakpoint identifizierende Nummer

ERROR_EVENT

Mit diesem Event teilt die ACVM mit, dass ein Fehler bei der Verarbeitung eines ACDWP-Kommandos in der ACVM aufgetreten ist.

ERROR_EVENT		
Feldname:	**Inhalt oder Typ:**	**Erklärung:**
Paketkopf:		
length	4	Größe des Paketes in Integer
type	204	Typ der Nachricht
Paketdaten:		
typeOfThe ErroneousPacket	[int]	Nachrichtentyp der fehlerverursachenden Nachricht
errorCode	[int]	Fehlercode

Literaturverzeichnis

[1] BeeCon GmbH. **AmbiComp (Ambient Computing)**. BeeCon GmbH, Albert-Nestler-Strasse 10, 76131 Karlsruhe.

[2] Sun Microsystems Inc. **Java Debug Wire Protocol**. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054.

[3] Sun Microsystems Inc. **Java SE 6 API**. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054.

[4] Sun Microsystems Inc. **KVM Debug Wire Protocol (KDWP)**. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054.

[5] Sun Microsystems Inc. **Squawk Java ME VM (JVM)**. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054.

[6] Sven Schlender and Thomas Fuhrmann. **Hardware-Datenblaetter**. BeeCon GmbH, Albert-Nestler-Strasse 10, 76131 Karlsruhe.

[7] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. **Java(TM) on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine**. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054.