Moduldokumentation

(MOD)

(TIT10AID, SWE I Praxisprojekt 2011/2012)

Projekt:	Multicastor 2.0
Auftraggeber:	*Andreas Stuckert und Markus Rentschler*
Auftragnehmer:	***Team 3*** *Michael Kern* *Pascal Schuhmann* *Roman Scharton* *Manuel Eisenhofer* *Tobias Michelchen (Ersteller der ModDoc)*

Modul

PcapListener

Inhaltsverzeichnis

1. History

Version	Datum	Autor(en)	Kommentare
1.0	19.04.2012	T. Michelchen	Dokument angelegt, Kapitel Anforderungen
1.1	20.04.2012	T. Michelchen	Anforderungen überarbeitet, Kapitel Analyse erstellt, Kapitel Design erstellt
1.2	23.04.2012	T. Michelchen	UML-Diagramm in Kapitel Design eingefügt, Kapitel Implementierung erstellt, Komponententestplan angefangen
1.3	26.04.2012	T. Michelchen	Tests durchgeführt, Ergebnisse in Kapitel Komponententest eingetragen
1.4	29.04.2012	T.Michelchen	Zusammenfassung und Ausblick, Allgemeine Überarbeitung, Erste fertige Version
2.0	06.05.2012	T.Michelchen	Abschließende Überarbeitungen. JavaDoc generiert und eingefügt. Finale Version zur Abgabe.

2. Scope

The Module Documentation (MOD) describes the architecture, the interfaces and the main features of the module. It also describes the module/component test including the results. It can also serve as a programming or integration manual for the module. If there are some risks related to the module itself, they shall be noted and commented within this document.

Die Moduldokumentation beschreibt die Architektur, die Schnittstellen und die Hauptmerkmale des Moduls. Außerdem werden die Modul bzw. Komponententests einschließlich der Ergebnisse beschrieben und dokumentiert. Die MOD dient bei Bedarf auch als Programmier- oder Integrationshandbuch für das Modul. Wenn bestimmte Risiken direkt mit der Verwendung des Moduls verknüpft sind, so sind sie in diesem Dokument zu benennen und zu kommentieren.

3. Definitionen

3.1 Definitionen

Netzwerkinterface Schnittstelle, die einem Computer oder einer Netzwerkkomponente Zugang zu einem Rechnernetz ermöglicht.

Jitter (engl. für „Fluktuation" oder „Schwankung") - Das zeitliche Taktzittern (also Unregelmäßigkeiten) bzw. bei der Übertragung von Digitalsignalen, eine leichte Genauigkeitsschwankung im Übertragungstakt.

API (englisch *application programming interface*) Bibliothek, mit deren Hilfe man standardisierte Probleme in der Softwareentwicklung einfach lösen kann

jNetPcap API, welche unterschiedliche Ressourcen für die Arbeit mit Netzwerkprotokollen zur Verfügung stellt. Diese Ressourcen ermöglichen das Versenden, Empfangen und Dekodieren von Paketen der unterschiedlichen Netzwerkprotokolle.

LAN (Local Area Network) - Ein lokal betriebenes Rechnernetz.

4. Anforderungen

4.1 Benutzersicht

Das Modul PcapListener beschäftigt sich mit dem Capturen bzw. „Abfangen" MMRP-relevanter Pakete vom Netzwerkinterface und der Weiterleitung dieser Pakete an die zuständigen Verarbeitungsklassen.
Dabei werden zwei Arten von Paketen unterschieden. MMRP-Steuerungspakete, die zum Re- und Deregistrieren von Multicast-Pfaden genutzt werden und Multicast-Ethernet-Pakete. Dies sind die eigentlichen Datenpakete, die vom Sender an alle Mitglieder der Multicast-Gruppe versendet werden.

4.2 Kontext der Anwendung

Dieses Modul stellt eine zentrale Komponente für die MMRP-Funktionalität des Multicastors dar.
Der PcapListener leitet die MMRP-Steuerungs-Pakete an die Multicast-Sender bzw. Receiver-Klassen weiter. Diese Klassen analysieren dann die einkommenden MMRP-Pakete und reagieren mit entsprechenden Events darauf.
Die Ethernet-Pakete werden dagegen nur an die Multicast-Receiver-Klassen weitergeleitet, welche die Pakete dann an das PaketAnalyser-Modul übergeben. Dieser analysiert die einkommenden Pakete nach verschiedenen Kriterien (Angekommene Pakete, Verlorene Pakete, Jitter) und wertet diese aus.
Der PcapListener übernimmt also eine Vorselektierung der einkommenden Pakete nach MMRP-Steuerungspaketen und Ethernet-Paketen, anschließend untersucht er die Zieladresse des Paketes und verteilt dieses dann entsprechend weiter.

4.3 Anforderungen

Das Modul deckt teilweise die in [1] spezifizierten Anforderungen /LF10/ und /LF130/ ab. Außerdem muss das Modul so gestaltet werden, dass es in der Lage ist die Pakete beim Empfangen in Echtzeit zu verarbeiten und entsprechend weiterleiten zu können.

5. Analyse

5.1 Voruntersuchungen

Da in der bereits bestehenden Anwendung Multicastor 1.0 eine API verwendet wurde, die lediglich auf Layer 3 (IP-Ebene) Pakete senden und empfangen konnte, musste für das MMRP-Protokoll, welches auf Layer 2 (Ethernet-Ebene) operiert, eine andere Lösung entwickelt werden.
Dazu wurde vom Auftraggeber die jNetPcap-API zur Nutzung empfohlen. Hierzu musste im Voraus getestet werden, ob die gestellten Anforderungen (Senden und Empfangen von MMRP- und Ethernet-Paketen) mit Hilfe dieser API erfüllt werden konnten.
Bereits nach einer kurzen Einarbeitungsphase in die jNetPcap-API konnte festgestellt werden, dass Sie die Anforderungen erfüllt und zum Versenden und Empfangen von Paketen über Layer 2 verwendet werden kann.

5.2 Systemanalyse (bestehende Architektur)

Da in der bestehenden Anwendung Multicastor 1.0 keine MMRP-Funktionalität implementiert war und das Modul auf Basis einer völlig anderen API aufsetzt wurde, konnte die Entwicklung des Modules ohne eine umfassende Analyse der bestehenden Systemarchitektur entwickelt werden.

5.3 Problemstellung

Zentrales Problem beim Empfangen der Pakete, ist die schnelle Verarbeitung dieser zu gewährleisten. Denn während der Verarbeitungszeit eines Paketes stauen sich weitere ankommende Pakete in der „Capture"-Methode der jNetPcap-API an. Ist der Puffer dieser Methode voll, so werden Pakete verworfen. Jede Vergrößerung der Verzögerungszeit erhöht also das Risiko des Paketverlustes und kann außerdem dazu führen, dass nicht mehr rechtzeitig auf MMRP-Steuerungspakete reagiert werden kann und damit das Tool nicht korrekt arbeitet. Das PcapListener-Modul ist der „Flaschenhals" der Anwendung, da hier alle einkommenden Pakete zwingend durchgeleitet werden müssen.

6. Design

Um eine kurze Verarbeitungszeit der Pakete zu gewährleisten ist es notwendig die Verarbeitungsschritte im PcapListener so kurz wie möglich zu halten. Dazu nimmt der PcapListener nur eine kurze Unterscheidung der Pakete nach MMRP- und Ethernet-Paketen vor und gibt diese dann direkt an die Controller der Sender- bzw. Receiver weiter.

6.1 Architektur

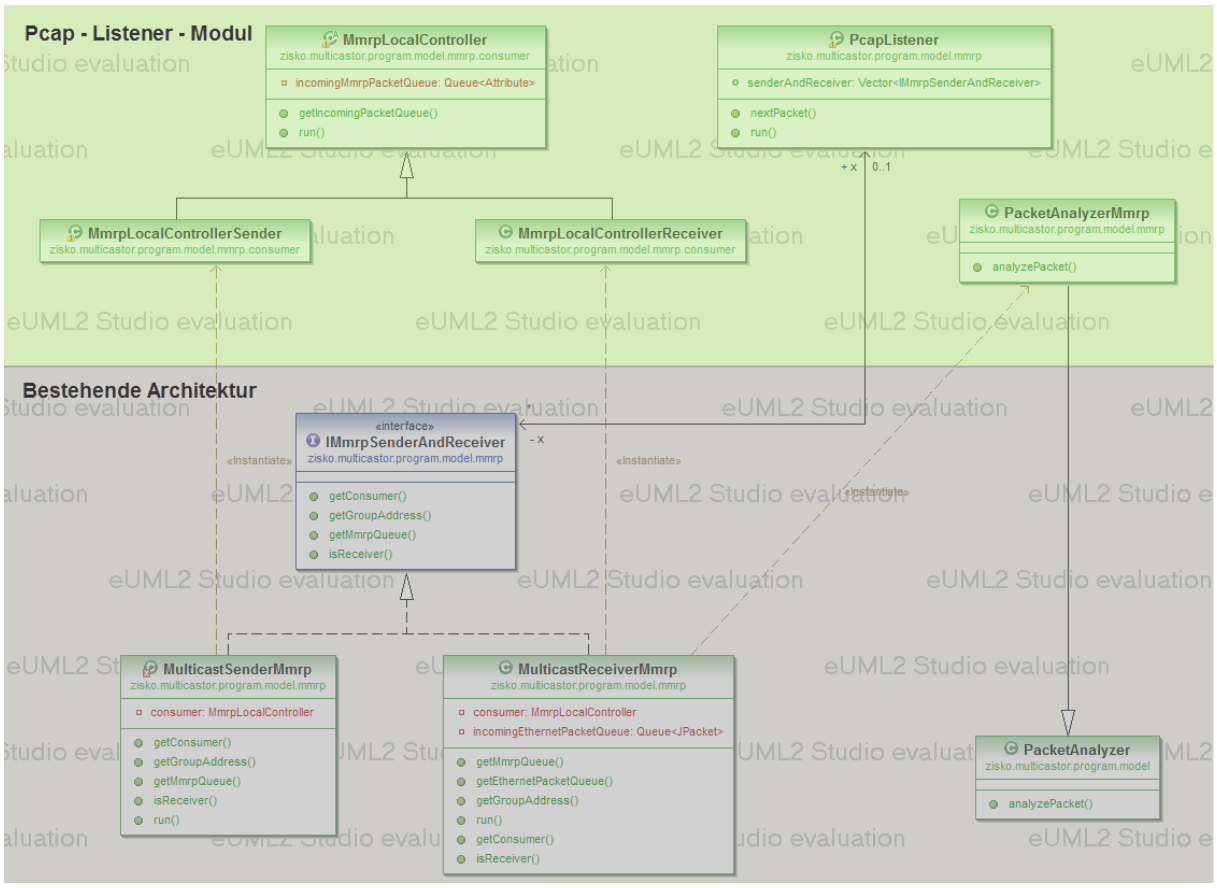

Abbildung 1 : Architektur PcapListener-Modul (Vereinfachte Darstellung)

Wie in der Grafik zu sehen erhält jeder MMRP-Receiver bzw. Sender einen eigenen Controller-Thread (*MmrpLocalController*). Dieser Controller hat die Aufgabe MMRP-Pakete zu analysieren und mit entsprechenden Events zu reagieren (näheres zu den Events siehe [2]). Dazu hat jeder Controller eine Thread-Sichere Queue, in der Pakete abgelegt werden können.

Sobald der *PcapListener* ein MMRP-Steuerungspaket empfängt greift er über das Interface *IMmrpSenderAndReceiver* auf die Queue´s der jeweiligen LocalController-Threads zu und legt das Paket dort ab.
Eine ähnliche Vorgehensweise gilt für die Multicast-Ethernet-Pakete. Da nur die *Receiver*-Klassen Ethernet-Pakete empfangen, besitzen diese eine separate Ethernet-Queue, in der nur Ethernet-Pakete abgelegt werden. Sobald ein Paket in dieser Queue abgelegt wird, gibt der Receiver dieses an den globalen *PaketAnalyser* weiter, der die Pakete analysiert

und nach den Kriterien „Angekommene Pakete", „Verlorene Pakete" und „Jitter" für die Ausgabe auf der Oberfläche aufbereitet.
Da der bestehende PacketAnalyzer die Ethernet-Packete aufgrund eines unterschiedlichen Aufbau´s nicht verarbeiten konnte, war es notwendig die Verarbeitung in einer eigenen Klasse (*PacketAnalyzerMmrp*) abzuwickeln. Diese Klasse erbt von der Klasse *PacketAnalyzer* und überschreibt die *analyzePacket*-Methode, welche für die Verarbeitung des Paketes notwendig ist. Da die Methoden zur Berechnung des Jitter´s, angekommener und verlorener Pakete gleich geblieben sind, wurden die nicht überschrieben und konnten so einfach wiederverwendet werden.
Alle Aufgaben „Capturen von Paketen" (*PcapListener*), „Auswerten von MMRP-Steuerungspaketen" (*MmrpLocalController*) und „Auswerten von Multicast-Ethernet-Paketen" (*PacketAnalyzerMmrp*) laufen in separaten Threads ab, was einerseits eine hochgradige Parallelisierung und damit einen Performance-Vorteil ermöglicht, aber andererseits einen Thread-sicheren Zugriff auf Ressourcen erfordert.

7. Risiken

7.1 Risiko 1 – jNetPcap – API enthält Fehler

Beschreibung:
Die jNetPcap – API ist eine 3rd party Software, also ein fertiger Baustein, auf dessen Grundlage das Modul „PcapListener" aufgebaut wird. Leider ist es bei solchen Fertigkomponenten meist so, dass man nicht weiß, was im inneren eines solchen Bausteins passiert. D.h. man verlässt sich hier darauf, dass alles so funktioniert, wie man es sich vorstellt. Allerdings kann es natürlich genauso gut sein, dass die API Schwächen oder sogar Fehler enthält.

Entdeckbarkeit:
Mittel-Hoch (je nachdem, wie offensichtlich der Fehler ist)

Wahrscheinlichkeit:
Gering (Die jNetPcap-API scheint ein auf den ersten Blick sehr seriöses Open-Source-Projekt zu sein, welches sich nun mittlerweile schon in der Version 1.4 befindet. Außerdem gibt es eine „Production/Stable"-Version (1.3.0), was darauf schließen lässt, dass es sich hier nicht um eine Version handelt, in der viele Bugs enthalten sind. Die User-Rezessionen im Internet und den Foren waren außerdem durchweg positiv und der Support scheint auch noch sehr aktiv zu sein. Insgesamt ist die Wahrscheinlichkeit einer fehlerbehafteten Version als gering einzuschätzen).

Auswirkungen:
Sollte die jNetPcap-API wirklich Fehler enthalten, könnten sich die Auswirkungen im unterschiedlichsten Ausmaß bewegen. Fehler die gleich zu Beginn der Implementierung oder sogar davor entdeckt werden, würden dazu führen, dass auf eine Ausweich-API zurückgegriffen werden müsste. Dies würde die Modulentwicklungszeit um bis zu eine Woche verlängern. Sollte ein essentieller Fehler allerdings erst spät in der Implementierungs- oder der Testphase erkannt werden, würde dies dazu führen, dass im gesamten Modul sehr viel an der bereits bestehenden Implementierung geändert werden müsste, was zu einer erheblichen Verlängerung der Modulentwicklungszeit führen würde und auch die Gesamt-Projektlaufzeit beeinflussen würde.

Vermeidungsstrategie:
Es wird bereits in der Analyse-Phase mit der Einarbeitung in die jNetPcap-API begonnen. Außerdem werden Test´s durchgeführt, die prüfen, ob die API zumindest die grundlegende Funktionalität abdeckt und mögliche offensichtliche Fehler aufdeckt.

Notfallplan:
Ausweich auf eine andere API (z.B. Jpcap) zum Capturen und Versenden von Paketen.

8. Implementierung

8.1 Producer – Consumer – Konzept

Der PcapListener wurde im Zusammenspiel mit den *MmrpLocalController*-Klassen nach dem Producer-Consumer-Konzept entwickelt. Bei diesem Konzept teilen sich zwei Klassen eine gemeinsame Queue. Der Producer (*PcapListener*) legt immer wieder neue Objekte (Pakete) in die Queue, während der Consumer (*MmrpLocalController*) die Objekte aus dieser Queue herausnimmt und verarbeitet. Jeder Multicast-Sender bzw. Receiver hat eine solche Consumer-Klasse als Helfer-Klasse. Diese Klasse ist ein eigenständiger Thread, der nur dafür verantwortlich ist eingehende Pakete zu analysieren und mit entsprechenden Events zu reagieren. Zum besseren Verständnis soll die folgende Grafik dienen:

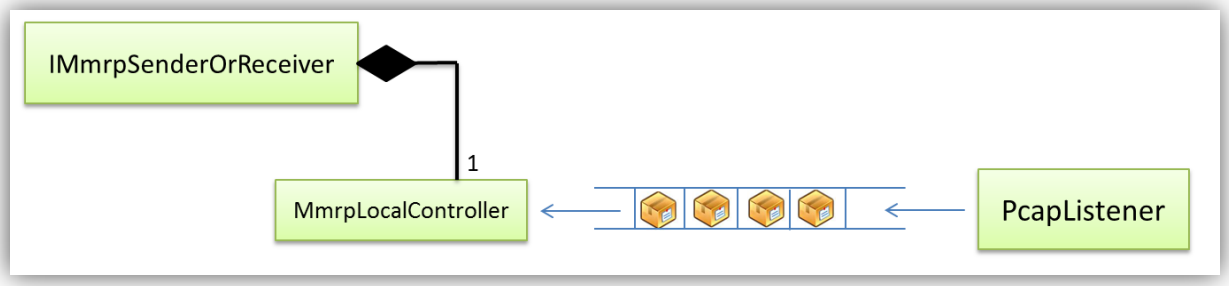

Abbildung 2 : Producer - Consumer – Konzept

Der PcapListener fängt alle Pakete vom Netzwerkinterface ab und legt diese in die jeweiligen Queue´s der *MmrpLocalController*. Diese nehmen wiederum Pakete aus der Queue raus und analysieren diese. Somit ist es möglich „gleichzeitig" Pakete durchgängig zu empfangen und zu verarbeiten.

8.2 Registrieren am PcapListener

Der PcapListener hält eine Liste aller Sender und Receiver, die im Moment aktiv sind. Sobald ein Sender bzw. Receiver gelöscht, bzw. nicht mehr aktiv ist, wird dieser aus der Liste des PcapListener´s entfernt.
Dies hat folgende Vorteile:

- Der PcapListener-Thread kann „schlafen", wenn keine Receiver bzw. Sender in der Liste sind (spart Systemressourcen)
- Der PcapListener muss nicht alle Sender bzw. Receiver prüfen, ob das eingegangene Paket für Sie bestimmt sein könnte, sondern hat eine sehr viel kleinere Liste zu verarbeiten (Performance-Vorteil)

8.3 Verwendung der bisherigen PaketAnalyser-Klasse

Die Klasse PacketAnalyser wurde bereits schon im Multicastor 1.0 zum Analysieren der Multicast-Pakete von IGMP und MLD verwendet. Aufgrund des ähnlichen Aufbaues der Multicast-Pakete von MMRP (siehe [3]) war es möglich die Logik der bereits vorhandenen PaketAnalyser-Klasse zu übernehmen. Lediglich die Länge der Pakete unterscheidet sich zwischen MMRP und IGMP- bzw. MLD-Multicast-Paketen. Deshalb wurde eine Klasse PaketAnalyserMmrp erstellt, die vom bisherigen PaketAnalyser erbt und die Methode zum Bestimmen der Länge des Paketes überschreibt.

8.4 MmrpLocalController

Bei der Erstellung der MMRP-Controller-Klassen wurde eine abstrakte Oberklasse *MmrpLocalController* erstellt, die die Grundlogik enthält, die jeder MMRP-Controller beinhaltet. Dazu zählt z.B. das jeder MMRP-Controller eine Queue hat, oder dass jeder MMRP-Controller die Funktion analyseMessage() zum Analysieren eines MMRP-Steuerungspaketes besitzt.
Anschließend wurden von der abstrakten Klasse *MmrpLocalController* die Klassen *MmrpLocalControllerSender* und *MmrpLocalControllerReceiver* abgeleitet (Vererbung). In dieses wurden dann die Methoden, wie z.B. die analyseMessage()-Methode Überschrieben (Polymorphismus).
Dadurch bieten sich mehrere Vorteile. Einerseits wird der geschriebene Code einfacher zu Warten, da zentrale Logik sich nur an einer Stelle befindet und nicht in mehreren Klassen verteilt ist und andererseits wird die Logik des Code´s einfacher, da keine Unterscheidungen nach Sender oder Receiver mit einer Abfrage vorgenommen werden muss, sondern die gleichen Methoden aufgerufen werden, nur mit einer unterschiedlichen Implementierung.

9. Komponententest

9.1 Testvorgehen

Bei der Erstellung der Testfälle wurde versucht möglichst wenige Testfälle zu erstellen, die aber einen sehr großen Teil der Funktionalität des Moduls abdecken. Dazu wurde das Verfahren der Äquivalenzklassenbildung angewendet, bei dem jeweils nur ein Repräsentant pro Äquivalenzklasse getestet wird. Dadurch ist es möglich mit „wenigen" Tests einen großen Bereich der Anforderungen sehr schnell abzudecken.

9.2 Komponententestplan

Test No	Feature ID	Test Specification (Description or TCS)
1	PcapListener - 001	PcapListener für ein beliebiges Netzwerkinterface initialisieren. Der PcapListener geht in den „wait()"-Abschnitt der Run-Methode.
2	PcapListener - 002	PcapListener für ein beliebiges Netzwerkinterface initialisieren. Einen Sender zum PcapListener hinzufügen und aktivieren. Der PcapListener geht in den „loop()"-Abschnitt der Run-Methode.
3	PcapListener - 003	Zwei Rechnern via LAN direkt miteinander verbinden. Rechner 1 (id: R1), Rechner 2 (id: R2). Auf R1 einen PcapListener für das Ethernet-Netzwerkinterface initialisieren. Hinzufügen eines Receivers (Multicast-Adresse: 01:34:56:78:9A:BC) zum PcapListener. Von R2 aus ein Ethernet-Paket an die Multicast-Adresse senden. Der MMRP-Controller von R1 muss das Paket in der Queue haben.
4	PcapListener - 004	PcapListener – 003 mit MMRP statt Ethernet-Paketen.
5	PcapListener - 005	Zwei Rechnern via LAN direkt miteinander verbinden. Rechner 1 (id: R1), Rechner 2 (id: R2). Auf R1 einen PcapListener für das Ethernet-Netzwerkinterface initialisieren. Hinzufügen von 3 Receivern (Multicast-Adresse: 01:34:56:78:9A:BC) zum PcapListener. Von R2 aus ein Ethernet-Paket an die Multicast-Adresse senden. Alle MMRP-Controller von R1 müssen das Paket in der Queue haben.
6	PcapListener - 66	PcapListener – 005 mit MMRP statt Ethernet-Paketen.

9.3 Komponententestreport

TestNo	Pass/Fail	If failed: Test Result	Date	Tester
1	**Passed**		**26.04.2012**	**T.Michelchen**
2	**Passed**		**26.04.2012**	**T.Michelchen**
3	**Passed**		**26.04.2012**	**T.Michelchen**
4	**Passed**		**26.04.2012**	**T.Michelchen**
5	**Passed**		**26.04.2012**	**T.Michelchen**
6	**Passed**		**26.04.2012**	**T.Michelchen**

10. Zusammenfassung und Ausblick

Das Modul PcapListener ist insgesamt ein sehr gut strukturiertes Modul, mit dem es theoretisch schnell möglich sein dürfte auch andere Arten von Paketen zu Empfangen und Weiterzuleiten. Hierzu müssten lediglich weitere LocalController-Klassen abgeleitet werden und die Filter-Methode des PcapListener´s umgeschrieben werden.
Auch die Übertragbarkeit der Lösung als Modul dürfte keine großen Probleme bereiten, da das Modul nur wenige Abhängigkeiten zu außenstehenden Klassen des Multicastor´s hat. Daher dürfte der Export des PcapListener´s und der LocalController-Klassen verbunden mit kleineren Änderungen hier schon ausreichen, um das Modul als solches zu übernehmen.
Allerdings sollte man bei der Übertragung in sicherheitskritische Projekte vorsichtig sein, da das Modul die jNetPcap-API verwendet. Der Entwickler hat keinen Einfluss darauf, welche Sicherheitslücken diese 3rd-Party-Software möglicherweise besitzt.

11. Anhang

11.1 JavaDoc zur PcapListener-Klasse

Class PcapListener

java.lang.Object
    dhbw.multicastor.program.model.mmrp.PcapListener

All Implemented Interfaces:

java.lang.Runnable, org.jnetpcap.packet.PcapPacketHandler<java.lang.Object>

```
public class PcapListener
extends java.lang.Object
implements org.jnetpcap.packet.PcapPacketHandler<java.lang.Object>, java.lang.Runnable
```

Filtert aus den einkommenden Packeten die relevanten MMRP-Steuerungs- und Ethernet-Pakete und legt diese in die Queues der MmrpLocalController ab

Field Detail

mmrp

```
private final dhbw.multicastor.program.model.mmrp.MMRP mmrp
```

Final initialisierter MMRP-Header. Wird benötigt, um in der nextPacket()-Methode MMRP-Pakete zu bestimmen.

dataHeader

```
private final dhbw.multicastor.program.model.mmrp.DataHeader dataHeader
```

Final initialisierter Data-Header. Wird benötigt, um in der nextPacket()-Methode "unsere" Ethernet-Pakete herauszufiltern. Diese Pakete besitzen einen eigenen Header, mit dessen Hilfe wir unsere Pakete wiedererkennen.

ethernetHeader

```
private final org.jnetpcap.protocol.lan.Ethernet ethernetHeader
```

Final initialisierter Ethernet-Header. Wird benötigt, um in der addEthernetPacket()-Methode die Zieladresse des Paketes zu bestimmen.

isActive

```
public boolean isActive
```

Sagt aus, ob der PcapListener aktiv ist oder nicht. (Pakete captured oder nicht).

alive

```
public boolean alive
```

Sagt aus, ob die Pcap des PcapListeners geschlossen wurde oder noch aktiv ist.

pcap

```
public org.jnetpcap.Pcap pcap
```

Pcap, auf der der PcapListener horcht.

senderAndReceiver

```
public java.util.Vector<dhbw.multicastor.program.model.mmrp.
                                    IMmrpSenderAndReceiver> senderAndReceiver
```

Liste mit allen Sendern und Receivern, die Pakete von dieser Schnittstelle empfangen. Wird vom MMRP-Controller gefüllt und geleert. Wenn die Liste leer ist, schläft der PcapListener.

Constructor Detail

PcapListener

```
public PcapListener(org.jnetpcap.Pcap pcap)
```

Zentraler Konstruktor. Jeder PcapListener ist für eine Pcap, also ein Netzwerkinterface verantwortlich. Die PcapListener werden einmalig vom MMRPController initialisiert.

Parameters:

`pcap` - Pcap, von welcher Pakete gecaptured werden sollen.

Method Detail

nextPacket

```
public void nextPacket(org.jnetpcap.packet.PcapPacket packet,
              java.lang.Object o)
```

Methode vom PcapPacketHandlerInterface. Einkommende Pakete kommen dekodiert in dieser Methode an. Hier wird das Paket dann nach Ethernet- oder MMRP-Paket unterschieden und den entsprechenenden Methoden addEthernetPacket() bzw. analyzeEvents() übergeben.

Specified by:

`nextPacket` in interface `org.jnetpcap.packet.PcapPacketHandler<java.lang.Object>`

analyzeAttributes

```
private void analyzeAttributes()
                        throws java.lang.InterruptedException
```

Analysiert die Attribute in einem MMRP-Paket. Wenn in einem Attribut ein LeaveAll-Event enthalten ist, wird dieses an alle Sender und Receiver weitergeleitet, andernfalls wird das Attribut nur an die Sender und Receiver weitergeleitet, für dessen Multicast-Gruppe das Attribut bestimmt ist.

Throws:

`java.lang.InterruptedException`

notifyLeaveAll

```
private void notifyLeaveAll(dhbw.multicastor.program.model.mmrp.Attribute at)
```

Legt ein Attribut mit einem LeaveAll in die MmrpLocalController-Queue´s alle Sender und Receiver

Parameters:

`at` - Übergebenes Attribut mit LeaveAll-Event

forwardEvents

```
private void forwardEvents(dhbw.multicastor.program.model.mmrp.Attribute at)
```

Legt ein Attribut in alle MmrpLocalController-Queue´s, deren Sender und Receiver für dessen Multicast-Gruppe bestimmt sind

Parameters:

`at` - Übergegebenes Attribut

addEthernetPacket

```
private void addEthernetPacket(org.jnetpcap.packet.PcapPacket packet)
                        throws java.lang.InterruptedException
```

Leitet Multicast-Ethernet-Packete an alle Receiver weiter, die für die Multicast-Adresse registriert sind, für die Adresse das Ethernet-Packet bestimmt ist.

Parameters:

`packet` - Übergebenes Ethernet-Paket

Throws:

`java.lang.InterruptedException`

run

```
public void run()
```

Capture-Methode. Solange er aktiv ist, Captured der Thread jeweils ein Paket. Wenn er deaktiviert wird, schläft der Thread.

Specified by:

`run` in interface `java.lang.Runnable`

11.2 Literaturverzeichnis

[1] „TIT10AID_CRS_MultiCastor20_Team_3_2v1.doc".
[2] R. Scharton, „ModDoc_MMRP-Paketanalyse und Steuerung der Multicastströme.doc".
[3] R. Scharton, „ModDoc_MMRP-Paketdekodierung und -erzeugung.doc".
[4] „TIT10AID_STP_Multicastor_Team_3_0v1-2.doc".

11.3 Testfälle

Siehe [4].