

Distributed Debugging for Wireless Sensor Networks



Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühlingssemester 2011

| | |
|-----------------|--------------------------------|
| Autoren: | Patrick Dünser / Amon Grünbaum |
| Betreuer: | Prof. Oliver Augenstein |
| Projektpartner: | IBM Research GmbH |
| Experte: | Dr. Peter Rost |
| Gegenleser: | Thomas Letsch |

Inhaltsverzeichnis

| | | |
|------------|--|-----------|
| I | Allgemeine Informationen | 1 |
| 1 | Abstract | 1 |
| 1.1 | Ausgangslage | 1 |
| 1.2 | Vorgehen/Technologie | 1 |
| 1.3 | Ergebnis | 1 |
| II | Projektdokumentation | 2 |
| 2 | Einführung | 2 |
| 2.1 | Was sind Sensornetzwerke? | 2 |
| 2.2 | Beispiele von Sensornetz Anwendungen | 2 |
| 2.3 | IBM Mote Runner kurz erklärt | 3 |
| 2.4 | Übersicht über IBM Mote Runner | 3 |
| 2.4.1 | Browser Dashboard | 4 |
| 2.4.2 | Eclipse Plugin | 4 |
| 2.4.3 | Weitere Mote Runner Teile | 5 |
| 2.5 | Sensornetzwerke programmieren | 5 |
| 2.6 | Radionachrichten in einem Sensornetzwerk | 5 |
| 3 | Aufgabenstellung | 6 |
| 3.1 | Aufgabenstellung und Ziel der Bachelorarbeit | 6 |
| 3.1.1 | Eclipse Timeline Plugin | 6 |
| 3.1.2 | Protokollanalyse | 7 |
| 3.2 | Schnittstellen | 7 |
| 3.3 | Performanceanforderungen | 8 |
| 4 | Ergebnisse | 9 |
| 4.1 | Gesamtarchitektur | 9 |
| 4.2 | Ergebnisse MRPacketHistory | 10 |
| 4.3 | Ergebnisse MRFrameFilter | 13 |
| 4.3.1 | Generelle Übersicht | 13 |
| 4.3.2 | Erweiterungspunkt für andere Events | 14 |
| 4.3.3 | Dissection Übersicht | 14 |
| 4.3.4 | Wie schreibe ich einen eigenen Dissector | 16 |
| III | Projektmanagement | 17 |
| 5 | Projektorganisation | 17 |
| 5.1 | Autoren | 17 |
| 5.2 | Betreuer | 17 |
| 5.3 | IBM Mote Runner Team | 17 |
| 6 | Vorgehensmodell | 18 |
| 6.1 | Soll- / Ist-Vergleich | 18 |
| 6.2 | Zeiterfassung | 18 |
| 6.3 | Meetingstruktur | 18 |
| IV | Arbeitspakete | 19 |

| | | |
|-----------|---|-----------|
| 7 | Meilensteine | 19 |
| 7.1 | Übersicht | 19 |
| 7.2 | 1. MS (18. März 2011) | 20 |
| 7.2.1 | GUI | 20 |
| 7.2.2 | Protokoll | 21 |
| 7.3 | 2. MS (15. April 2011) | 22 |
| 7.3.1 | GUI | 22 |
| 7.3.2 | Protokoll | 24 |
| 7.4 | 3. MS (13. Mai 2011) | 25 |
| 7.4.1 | GUI | 25 |
| 7.4.2 | Protokoll | 26 |
| 7.5 | 4. MS Final Version (10. Juni 2011) | 26 |
| 7.5.1 | GUI | 27 |
| 7.5.2 | Protokoll | 27 |
| V | Abschluss | 29 |
| 8 | Future Work | 29 |
| 8.1 | Weitere Anwendungsbereiche | 30 |
| 9 | Lessons Learned | 31 |
| 9.1 | Erfahrungsbericht Amon Grünbaum | 31 |
| 9.2 | Erfahrungsbericht Patrick Dünser | 31 |
| VI | Anhang | 32 |
| A | User Manual MR-Timeline Perspective | 32 |
| A.1 | Overview | 32 |
| A.2 | Plugins | 33 |
| A.2.1 | MR-HistoryView | 33 |
| A.2.2 | MR-HeatMap | 36 |
| A.2.3 | MR-FilterDetail | 37 |
| A.2.4 | Configuration | 37 |
| B | Tutorial: „How to write a dissector“ | 40 |
| B.1 | General Information | 40 |
| B.2 | Tutorial: “Hello World” protocol dissector | 40 |
| B.2.1 | Create the dissector project using the wizard | 40 |
| B.2.2 | Implement the dissector | 42 |
| B.2.3 | Export the new dissector and configure the dissector tree. | 44 |
| B.3 | Different more complex caseses of dissectors | 45 |
| B.3.1 | How to test a dissector | 45 |
| B.3.2 | Different result types for the results of a dissection. | 46 |
| B.3.3 | How to define more then one color for a dissector | 48 |
| B.3.4 | Change the way how children will run inside of your dissector | 49 |
| B.3.5 | Configure the dissectors tree | 49 |
| B.4 | Troubleshooting Ant build | 51 |
| B.5 | Where do I find Additional Informations | 51 |
| B.5.1 | Source code | 51 |
| B.5.2 | Javadoc | 51 |
| C | Installation der Software | 52 |
| C.1 | Installation Mote Runner | 52 |
| C.2 | Ausführen eines Testprogrammes | 52 |
| C.3 | Quellcode | 53 |

| | |
|--|-----------|
| D Usability Test - Dissector API | 54 |
| D.0.1 Was wurde getestet | 54 |
| D.0.2 Wer hat getestet | 54 |
| D.0.3 Wie wurde getestet | 54 |
| D.0.4 Einschränkungen | 54 |
| D.0.5 Resultat | 54 |
| D.0.6 Auswertung: | 55 |
| E Verzeichnisse | 56 |
| F Inhalt der CD | 61 |
| G Erklärung über die eigenständige Arbeit | 62 |

Teil I

Allgemeine Informationen

1 Abstract

1.1 Ausgangslage

Heutzutage sind immer mehr Produkte mit Sensoren ausgestattet, welche Umgebungsinformationen messen und nahtlos in das Informationsnetz integriert sind. Die einzelnen Sensoren können untereinander Daten austauschen und so Synergien nutzen. Dies wird als "Internet of Things" bezeichnet. Sensornetzwerke sind jedoch kompliziert zu programmieren und zu testen, was den technologischen Fortschritt verlangsamt.

IBM hat mit Mote Runner eine Plattform entwickelt, die es erlaubt wireless Sensoren mittels einer Hochsprache (Java / C#) zu programmieren und zu debuggen. Mit der dazugehörigen Simulationsumgebung ist es bedeutend einfacher Sensornetze zu testen. Um das Debugging komfortabler zu machen, war es unsere Aufgabe die Kommunikation der verschiedenen Sensoren graphisch darzustellen.

1.2 Vorgehen/Technologie

Die Schwierigkeit beim Debuggen grosser Sensornetzwerken besteht darin, versteckte Fehler, wie Protokollverletzungen und Deadlock, zu finden. Grund dafür ist, dass die Sensoren nebenläufig und zum Teil von einander abhängig arbeiten. Unsere Visualisierung sollte deshalb die Radionachrichten zwischen den Sensoren in einem überschaubaren Raster darstellen. Um verschiedene Probleme der Sensornetzwerke anzugehen, haben wir Werkzeuge entwickelt, um Abstände zwischen Nachrichten zu messen und Details der Netzwerkprotokolle anzuzeigen. Für die Realisierung eignete sich ein Eclipse Plugin, da bereits andere Teile von Mote Runner darauf aufsetzen. Des Weiteren musste eine Lösung gefunden werden, um auch eigene beziehungsweise nicht integrierte Protokolle grafisch darzustellen.

1.3 Ergebnis

Entstanden ist ein Plugin, das aus mehreren Views besteht. Zum einen werden auf einer fortlaufenden Timeline die gesendeten Pakete der Sensoren visualisiert und nach Bedarf auch deren Empfänger aufgezeigt. Zum zweiten werden nach dem Markieren eines Pakets zusätzliche Infos über den Inhalt des Pakets in einer separaten Ansicht aufgelistet. Zum dritten werden die Sensoren entsprechend ihrer räumlichen Position und Kommunikationspfaden dargestellt.

Die empfangenen Radionachrichten werden analysiert. Je nach Format der Radionachricht wird diese anders dargestellt. Der Parsebaum kann vom Benutzer konfiguriert werden und falls noch kein Parser für sein Netzwerkprotokoll vorhanden ist, kann der Benutzer selber das API für seinen Protokollparser implementieren. Dieser kann in die Applikation eingehängt werden. Mittels eines Wizards wird der Einstieg zum Erstellen des eigenen Parsers erleichtert.

Teil II

Projektdokumentation

2 Einführung

Um Sie mit dem Thema Sensornetzwerke vertraut zu machen und an unsere Aufgabenstellung heranzuführen, haben wir eine grobe Übersicht über den Themenbereich zusammen gestellt.

2.1 Was sind Sensornetzwerke?

Unter einem Sensornetzwerk versteht man ein Netz von einzelnen Sensoren, die untereinander kommunizieren und zusammenarbeiten. Ein Sensor ist ein kleiner Computer, der verschiedene Messgeräte besitzt, um etwa Temperatur, Luftfeuchtigkeit und Helligkeit in seiner unmittelbaren Umgebung zu messen. Es gibt Sensor in allen Grössen und Formen, die mit den unterschiedlichsten Sensoren ausgerüstet sind. Dass ein einzelner Sensor etwas misst, gibt es schon lange und ist nichts spezielles. Das Spezielle an einem Sensornetzwerk ist, dass die verschiedenen Sensoren sich gegenseitig kennen, sich untereinander organisieren und Daten austauschen können. Dies geschieht ohne zentralen Server über den die Kommunikation läuft. Durch die Zusammenarbeit der einzelnen Sensoren kann ein Sensornetzwerk wesentlich komplexere Aufgaben übernehmen als ein einzelner Sensor. Sensornetze sind oft auch unter dem Namen Smart Dust bekannt. [1]



Abbildung 1: Sensor: MEMSIC Iris Mote

2.2 Beispiele von Sensornetz Anwendungen

Die Möglichkeiten für den Einsatz von Sensornetzwerken sind unbegrenzt und mit dem technologischen Fortschritt nehmen sie laufend zu. Bereits heute werden in der Gebäudetechnik teilweise Sensornetzwerke gebraucht.

Beispielsweise kann mit dem Einsatz eines Sensornetzwerks in einem Serverraum viel Strom gespart werden. Das Sensornetzwerk überwacht dabei die Temperatur und die Luftströme im Raum und optimiert mit den gesammelten Informationen die Klimaanlage laufend.

In Zukunft sind auch viel weiterführende Systeme vorstellbar. Stellen Sie sich vor Ihr Wecker sendet eine halbe Stunde vor dem Alarm automatisch der Heizung im Badezimmer ein Signal, damit diese sich erwärmt und in 30 Minuten warm ist. Die Heizung misst die aktuelle Temperatur und errechnet, wie lange sie braucht, um das Badezimmer bis zum Aufstehen optimal zu beheizen. Um eine bessere Verteilung der Stromlast zu erreichen, stellt sich die Waschmaschine, während der Zeit in der Sie aufstehen, automatisch aus, da die Heizung voll läuft und sich auch die Kaffeemaschine gerade eingeschaltet hat. Dieses Beispiel lässt sich endlos weiterführen.[2] Viele der möglichen Technologien dieser Zukunftsszenarien basieren auf der Idee, dass verschiedene Sensoren in alle Geräte eingebaut werden und die Geräte untereinander Informationen austauschen können. Diese Ideen der grossen Sensornetzwerke sind auch unter dem Begriff "Internet of Things" zusammengefasst.

Ein wesentliches Problem bei diesen Zukunftsszenarien ist die Komplexität der Programmierung und die fehlende Möglichkeit einfache Programme zu schreiben und effizient zu testen. Genau hier will IBM mit Mote Runner ansetzen und eine Komplettlösung für Simulation, Testing und Programmierung von Sensornetzwerken bieten.

2.3 IBM Mote Runner kurz erklärt

Das Ziel des Mote Runner Projekts ist es, eine effiziente und einfach zu bedienende Entwicklungsumgebung für Programmierer von Sensornetzwerken zu erstellen. Die bessere Nutzung der Ressourcen der Sensoren und die einfachere Programmierung sollen Sensornetzwerken zum Durchbruch verhelfen und viele neue Anwendungsgebiete für Sensornetze öffnen. Mote Runner besteht aus einer virtuellen Maschine, die speziell optimiert ist, um auf Sensoren mit wenig Ressourcen zu laufen. Der Programmierer muss die spezifischen Details der Hardware nicht kennen und kann mit einer objektorientierten Programmiersprache (Java oder C#) Programme für die Sensoren schreiben. Die Programme können gut getestet werden, da eine Simulationsumgebung und verschiedene Hilfsmittel zum Debuggen von Sensornetzwerken mitgeliefert werden. [3]

2.4 Übersicht über IBM Mote Runner

In Abbildung 2 ist eine technische Übersicht über Mote Runner zu sehen. Wichtig dabei ist, dass es für jeden unterstützten Sensortyp (Mote) eine Hardwareabstraktionsschicht (hal) von Mote Runner gibt. Dieser hal kann eine echte oder eine simulierte Hardware abstrahieren. Der simulierte hal Layer läuft im Saguaro Prozess. Auf dem hal Layer setzt das eigentliche Mote Runner Betriebssystem (os) auf. Dieses Betriebssystem und alle darauf aufbauenden Teile wissen nie, ob sie auf einer echten oder einer simulierten hal laufen. Dies hat den Vorteil, dass der gleiche Code verwendet werden kann.

Ein wesentlicher Teil des os ist die virtuelle Maschine, in der die Benutzerprogramme (apps) laufen. Auch die apps des Benutzers merken nicht, ob sie auf einem echten oder einem simulierten Mote laufen.

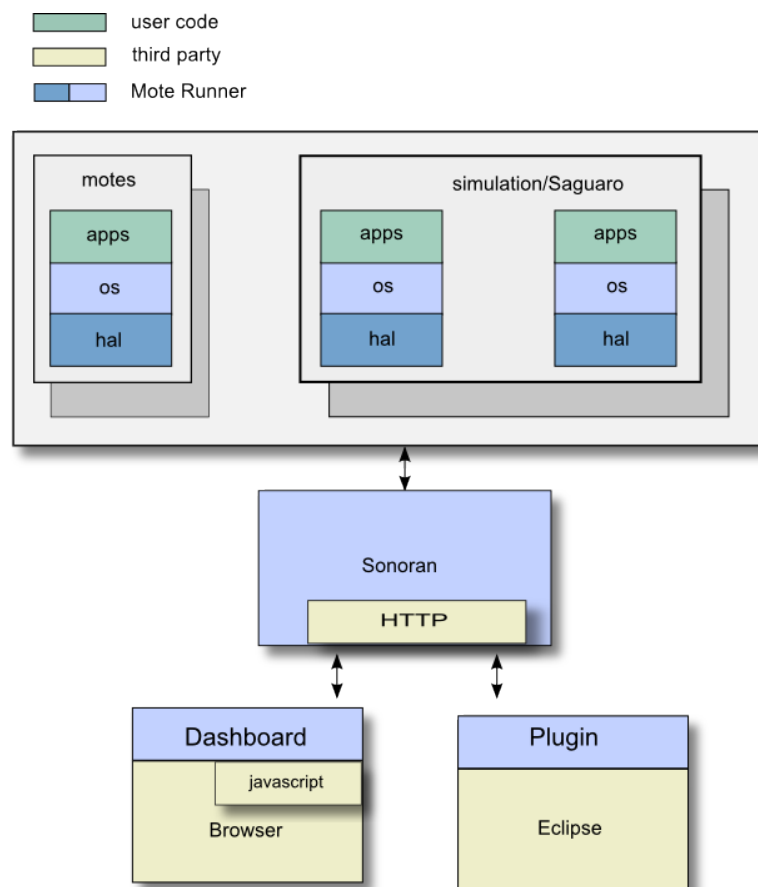


Abbildung 2: Vereinfachte Übersicht IBM Mote Runner

Neben den Motes gibt es den Sonoran Prozess. Der Sonoran implementiert den Edge Server. Der Edge Server ist die Schnittstelle zum simulierten oder realen Mote-Netzwerk. Der Sonoran kommuniziert dabei mit dem Edge Mote, der mit den anderen Motes kommuniziert. Sonoran ist eine Middleware-Plattform, die mit Scripten erweiterbar ist und dazu genutzt wird, um mit dem Mote-Netzwerk zu kommunizieren. Sonoran kann um Management-, Debugging- oder Kommunikationstools erweitert werden. Der Browser und das Plugin für Eclipse wird nachfolgend genauer beschrieben.

2.4.1 Browser Dashboard

Das Dashboard ist ein Client von Sonoran und kommuniziert über die HTTP Schnittstelle. Das Dashboard visualisiert die simulierten und die echten Motes. Es ist grundsätzlich eine visuelle Darstellung vom Sonoran Prozess und erlaubt fast alle Funktionen davon. Sofern ein Sonoran-Prozess gestartet ist, kann über die URL `http://localhost:5000` das Dashboard aufgerufen werden (siehe Abbildung 3).

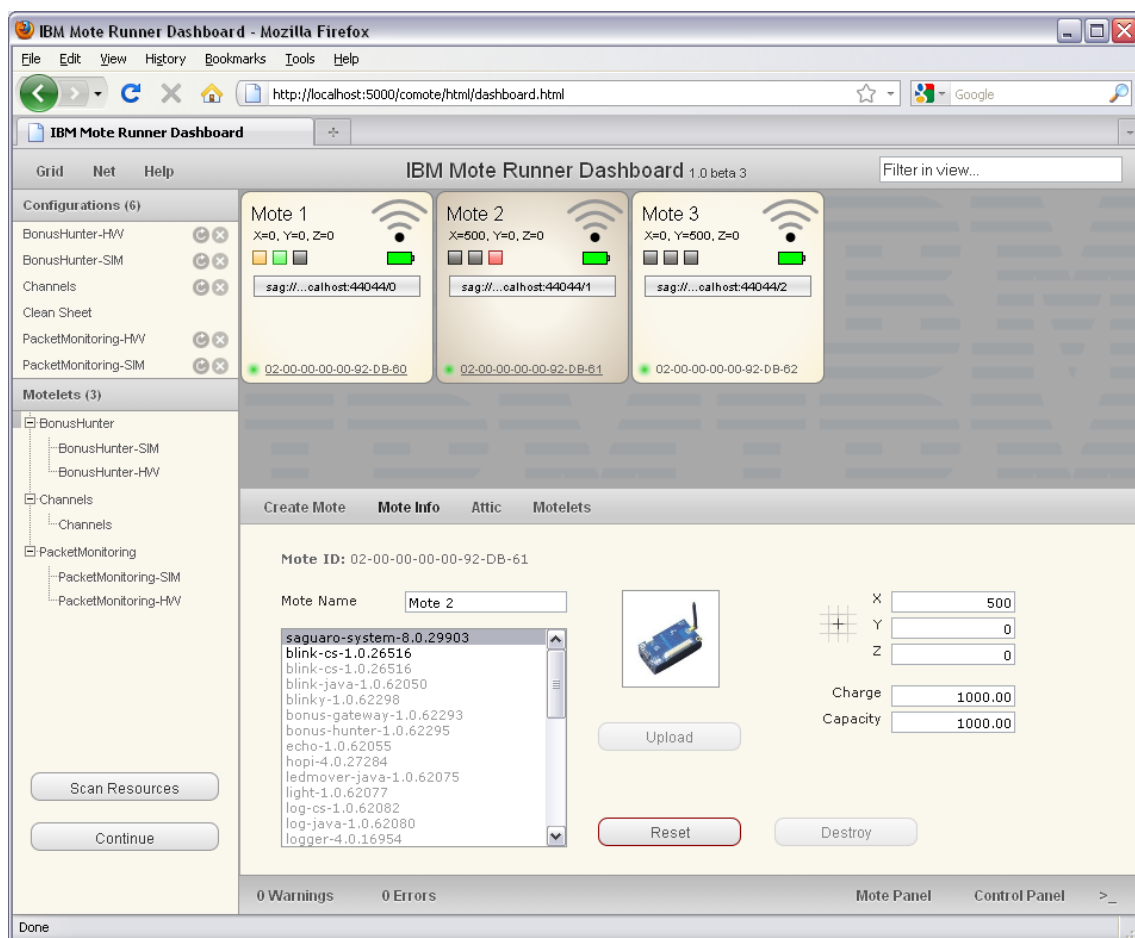


Abbildung 3: Mote Runner Dashboard

2.4.2 Eclipse Plugin

IBM hat für Eclipse verschiedene Plugins geschrieben, welche es erlauben Motes mit einer höheren Programmiersprache (Java, C#) zu programmieren. Die Applikationen werden kompiliert und in einen portablen Bytecode-Darstellung konvertiert. Neben den Programmierhilfen ist es möglich mit Eclipse auf den Sonoran Prozess zuzugreifen (analog zum Dashboard) und die Programme auf den simulierten Motes zu debuggen. Dabei kann man beispielsweise Breakpoints auf einzelne Motes, oder Events in der Simulation setzen und so die Sensornetze gut testen. Beim Debuggen

wird dabei jeweils die komplette Simulation im Sonoran Prozess angehalten. Dieses Debugging funktioniert jedoch nur bei simulierten Motes.

2.4.3 Weitere Mote Runner Teile

Diese Übersicht wurde auf Basis der Mote Runner Help Dokumentation [4] beschrieben. Dies ist keine vollständige Dokumentation. Es wurden lediglich die Teile beschrieben, die wichtig sind, um unsere Arbeit zu verstehen. Für eine ausführlichere Dokumentation schauen Sie sich bitte die Mote Runner Dokumentation an ([4]).

2.5 Sensornetzwerke programmieren

Will man nun eine eigene Sensornetz-Anwendung programmieren, hat man mit Mote Runner eine Vielfalt an Tools zur Verfügung, die einem dabei helfen, sein Projekt zu realisieren. Besonders die Simulation hilft verschiedene Problemfälle genau durchzuspielen und so ein Sensornetz viel robuster zu gestalten. In der Simulation können sogar komplexe Fehler, wie zufälliger Paketverlust oder ein Clock-Drift eines Sensors simuliert werden.

2.6 Radionachrichten in einem Sensornetzwerk

Wie schon mehrfach erwähnt, ist die Kommunikation zwischen den einzelnen Sensoren ein zentrales Element der Sensornetzwerke. Die Motes müssen sich immer wieder abgleichen und untereinander Daten austauschen. Da die Sensoren zudem auch sehr energieoptimiert funktionieren sollten, braucht es ausgeklügelte Protokolle zur Synchronisation der Sensoren. Sonst kann es schnell vorkommen, dass ein Sensor sein Radiomodul gar nicht aktiviert hat, im Moment, wo ein anderer Sensor seine Daten sendet. Andererseits dürfen die Sensoren auch nicht ständig eingeschaltet sein, da sonst die Batterie nicht lange hält und der Sensor schnell nutzlos wird.

Für spezifische Anwendungsfälle kommt es auch immer wieder vor, dass ein Entwickler ein eigenes Protokoll definiert, das genau für seinen Fall optimal ausgerichtet ist. Das Debugging und Testing solcher eigener Protokolle ist sehr schwer, da kleinste Fehler in der zeitlichen Abfolge von Paketen zu unvorhergesehenen Nebenerscheinungen führen kann. Mit der Simulationsumgebung von Mote Runner können solche Protokolle bereits gut simuliert und auch debugged werden. Momentan fehlt jedoch noch eine geeignete Form der Aufzeichnung der Radionachrichten in einem Format, das den Protokollentwickler bei seiner Arbeit optimal unterstützt. Genau hier soll unsere Bachelorarbeit anknüpfen.

3 Aufgabenstellung

3.1 Aufgabenstellung und Ziel der Bachelorarbeit

Ziel dieser Bachelorarbeit ist es herauszufinden wie man effizient und effektiv die Radiokommunikation zwischen verschiedenen Motes eines Netzwerks darstellen kann. Dies sollte aufzeigen, wie es möglich ist, eine Vielzahl von Motes die regelmässig Nachrichten senden, in einer übersichtlich und performanten Art und Weise darzustellen. Des Weiteren dient diese Arbeit dazu, in einem späteren Schritt Debugging-Szenarien erstellen zu können und damit unsere Lösung zu optimieren. Die Bachelorarbeit muss als Eclipse-Plugin realisiert werden.

Da es bei der Programmierung von Sensornetzwerken oft vorkommt, dass eigene Protokolle geschrieben werden, soll es zusätzlich für jeden Entwickler möglich sein, Informationen über sein Protokoll einzuspielen, damit danach in der Darstellung der Kommunikation auch die eigenen Protokolle angezeigt werden.

3.1.1 Eclipse Timeline Plugin

Im Plugin werden alle vorhandenen Motes aufgelistet. Für jeden Mote werden alle gesendeten Radiosignale erfasst und das Plugin visualisiert den Zeitpunkt des Sendens sowie den Protokoll-Typ. Die Visualisierung dient als Real-Time-View und als History-View. Das Hauptaugenmerk liegt allerdings auf der History-View. Des Weiteren sollte es in dieser Ansicht möglich sein, die gesendeten Pakete im Detail, d.h. mit allen Informationen, welche durch die Protokollanalyse vorhanden sind, anzusehen.

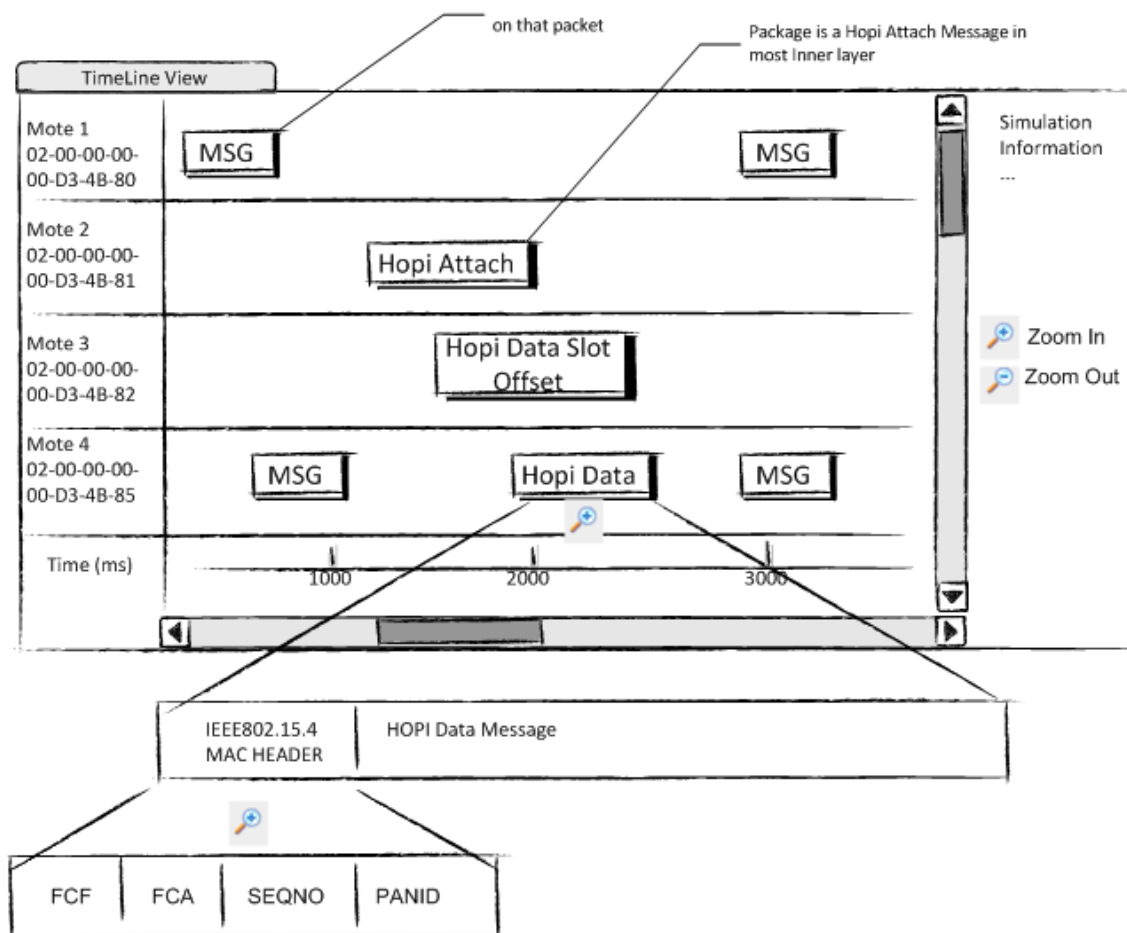


Abbildung 4: Paper Prototype 1

In den Abbildungen 4 und 5 sind die Anforderungen des Plugin aufgezeigt. Die ankommenden

Pakete sollen in geeigneter Form dargestellt werden. Die Detailtiefe der Pakete kann mittels rein- und rauszoomen bestimmt werden. Zudem sollen alle Informationen, die über das Paket vorhanden sind, dargestellt werden. Dies kann gegebenenfalls in einem zweiten Fenster geschehen. Das Plugin dient nur als Anzeigefunktion und kann somit die Simulation nicht verändern. Es muss im Plugin eine geeignete Möglichkeit gefunden werden, wie angezeigt wird, welche Motes die gesendeten Pakete empfangen haben. Mit einem Timemarker soll ein Teil der Timeline ausgewählt werden können, um die Zeit zwischen verschiedenen Ereignissen genau messen zu können.

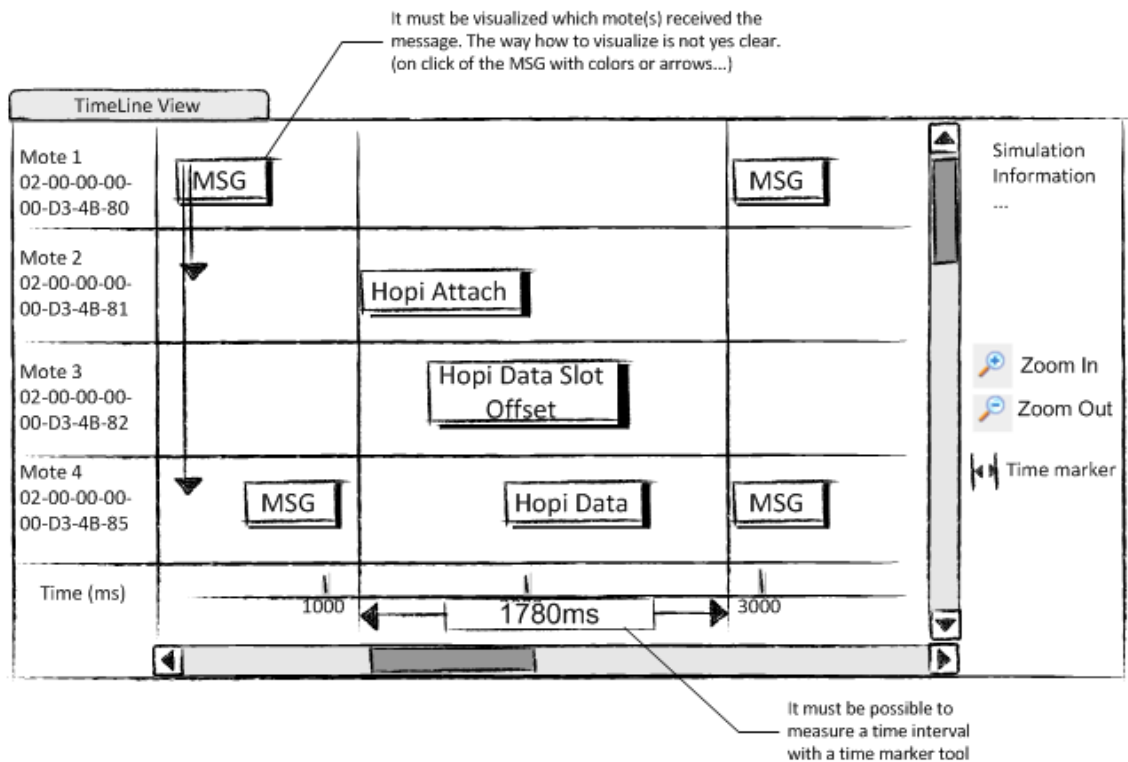


Abbildung 5: Paper Prototype 2

3.1.2 Protokollanalyse

In diesem Teil der Arbeit geht es darum, die Rohdaten, welche zwischen den Motes ausgetauscht werden, zu analysieren und mit Filterregeln eine hierarchische Struktur in die Pakete zu bringen. Die Mote Runner Simulation sendet die Rohdaten direkt an dieses Modul. Anhand von vordefinierten Regeln und Filtern muss das Modul herausfinden, welche Daten im Paket sind und was für verschiedene Header vorhanden sind. Dies muss in eine geeignete hierarchischen Struktur gebracht werden. Danach werden die bearbeiteten Daten dem Eclipse-Plugin übergeben, welches diese Informationen übersichtlich darstellt. Es muss eine einfach zu bedienende Möglichkeit geben, Filter für neue Protokolle zu schreiben.

3.2 Schnittstellen

Für die Implementierung des Timeline Plugin gibt es bereits fixe APIs, welche wir verwenden können. Der Sonoran-Server, auf dem die Simulation läuft, kann mit JavaScript Befehlen bedient werden. Es gibt einen Befehl, mit welchem man alle Radioevents anzeigen lassen kann. Mit diesen Radioevents wird unser Plugin gefüttert. Zusätzlich gibt es bereits ein Eclipse Plugin, welches die Sonoran-API fast vollständig implementiert. Für den Zugriff auf die Daten werden wir diese in Java geschriebene API verwenden. Für das GUI sollten, wenn immer möglich, bestehende Eclipse-Klassen erweitert werden, damit das Plugin sauber in Eclipse integriert ist.

3.3 Performanceanforderungen

Unsere Software sollte auch bei grossen Simulationen von bis zu hundert Motes, welche alle 2.5 Sekunden eine Nachricht schicken noch funktionsfähig sein. Momentan ist von Seiten der IBM noch nicht getestet worden, mit wie vielen Motes die Mote Runner Simulation funktioniert und wo der Flaschenhals ist. Deshalb kann keine klare Aussage gemacht werden, mit wie vielen Daten unser Plugin funktionieren muss. Wichtig ist, dass nicht unser Plugin der Flaschenhals ist.

4 Ergebnisse

In diesem Kapitel wird ein Überblick gegeben, was wir während dieser Bachelorarbeit erreicht haben. Eine ausführliche Beschreibung der einzelnen Funktionen und die Bedienungsanleitung unserer Software sind im Anhang verfügbar (Anhang A und B). Die Benutzerdokumentation ist in Englisch verfasst, da dieser Teil der IBM als eigenständiges Dokument abgegeben wird und dieses in Englisch gewünscht worden ist. Ebenfalls ein wichtiger Teil unseres Ergebnis ist unser Code, da dieser von der IBM weiterentwickelt wird. Deshalb haben wir auch konsequent für alle Packages, Klassen und für alle wichtigen Methoden eine Codedokumentation mit Javadoc erstellt.

Die einzelnen Teile unserer Arbeit wurden in Arbeitspaketen erarbeitet. Verschiedene Designentscheide, sowie einzelne Beschreibungen, wie wir vorgegangen sind, sind im Teil IV "Arbeitspakete" beschrieben.

4.1 Gesamtarchitektur

Unsere Software besteht aus zwei Eclipse Plugins. Das MRPacketHistory Plugin beinhaltet praktisch alle Komponenten des GUIs. Dies sind hauptsächlich verschiedene Views, die der Benutzer einblenden kann. Ebenfalls sind alle Methoden zur Kommunikation zwischen den verschiedenen Views in diesem Plugin vorhanden.

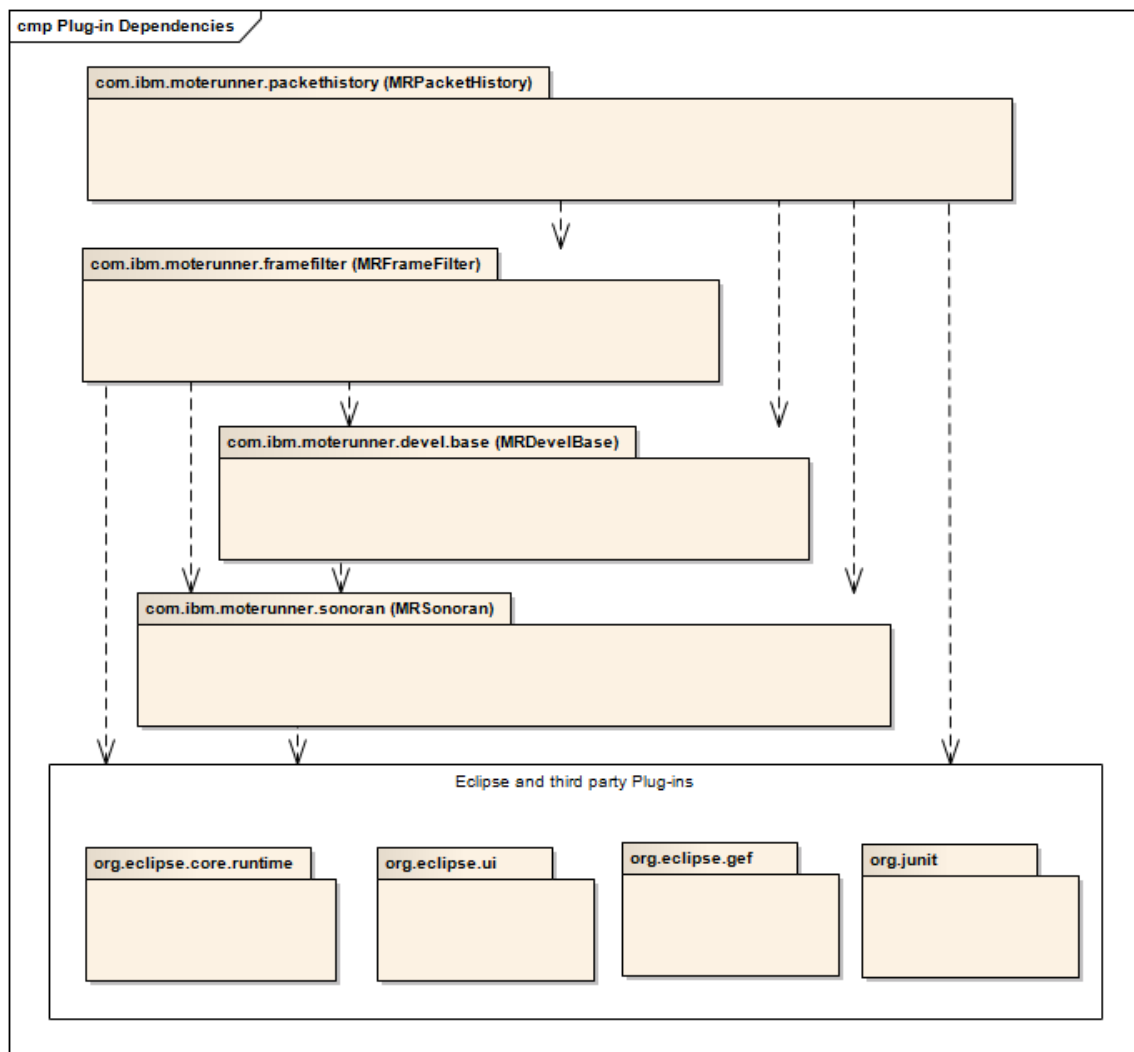


Abbildung 6: Abhängigkeiten zwischen den verschiedenen Plugins

Im MRFrameFilter Plugin werden die Daten von Sonoran aufbereitet und verwaltet. Die ge-

samte Analyse der Radionachrichten und das Parsen des Inhalts findet hier statt. Ein paar wenige Dialogfenster die Einfluss auf den Parsebaum nehmen, sind ebenfalls in diesem Plugin definiert. Das MRPacketHistory Plugin kann direkt auf das Model im MRFrameFilter Plugin zugreifen und alle nötigen Daten davon auslesen. Das MRFrameFilter Plugin kennt das MRPacketHistory Plugin nicht. Die Kommunikation in diese Richtung funktioniert ausschliesslich über Events, auf die sich das MRPacketHistory Plugin beim MRFrameFilter Plugin registrieren muss. Für die Anbindung an Sonoran greift der FrameFilter auf das Basisplugin von Sonoran zu und registriert sich dort für gewisse Events. Alle Plugins benötigen und erweitern verschiedene Teile des Eclipse CoreSystems (siehe Abbildung 6). Der Grund für die Unterteilung in zwei verschiedene Plugins ist einerseits die saubere Trennung von Model und View. Andererseits können so Teile des MRFrameFilter Plugins auch ausserhalb von Eclipse als Library benutzt werden.

Die Abhängigkeit vom MRPacketHistory Plugin zum MRSonoran Plugin und dem MRDevel-Base Plugin bestehen nur, da gewisse Datenstrukturen, wie zum Beispiel die Mote Klasse, hier definiert sind. Die Instanzen der Motes holt sich die MRPacketHistory jedoch im MRFrameFilter Plugin.

4.2 Ergebnisse MRPacketHistory

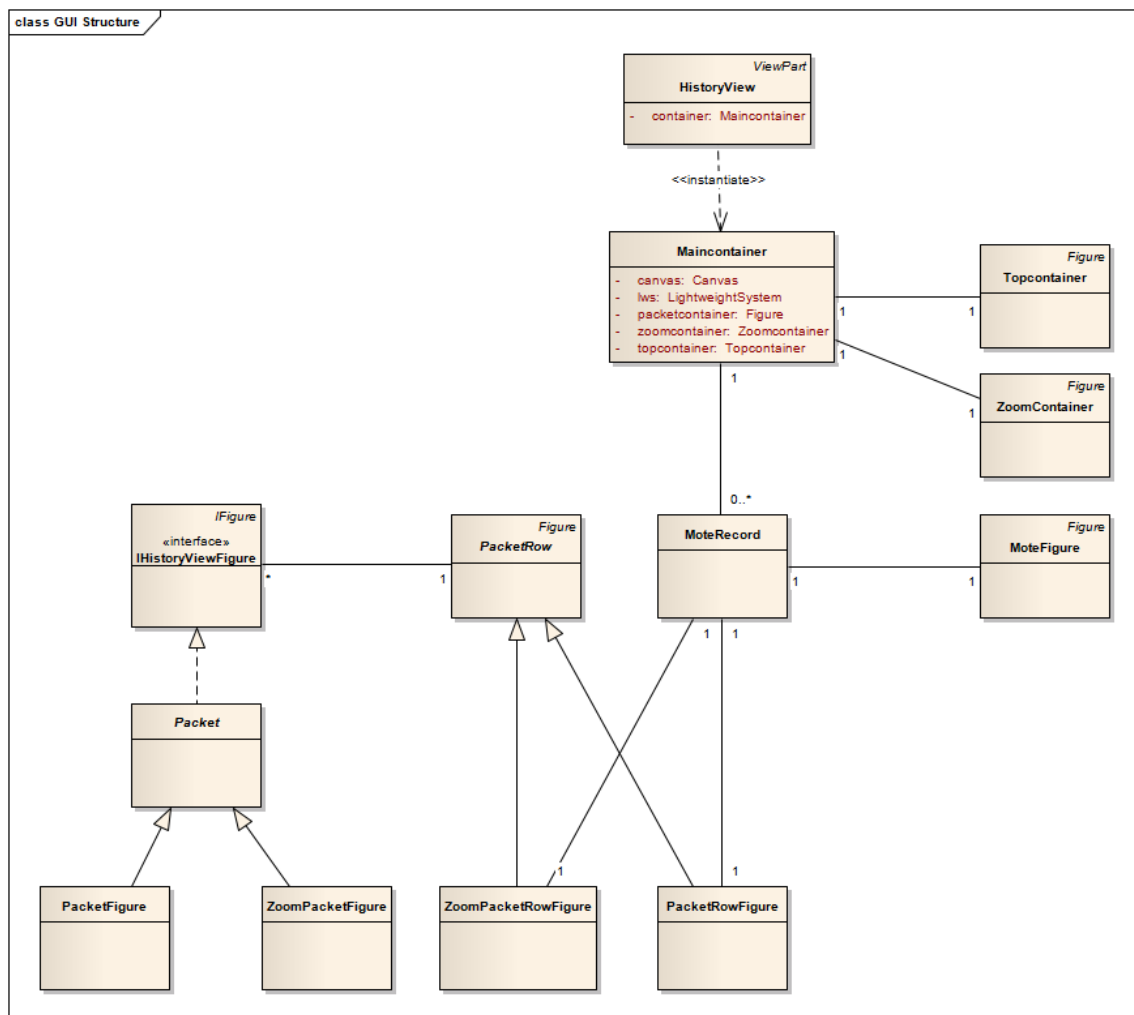


Abbildung 7: Grobe Übersicht der HistoryView-Struktur

In Abbildung 7 werden grob die Zusammenhänge in der GUI-Struktur der HistoryView aufgezeigt. Beim Start des Plugin wird die Klasse HistoryView aufgerufen, diese erstellt nur die Klasse

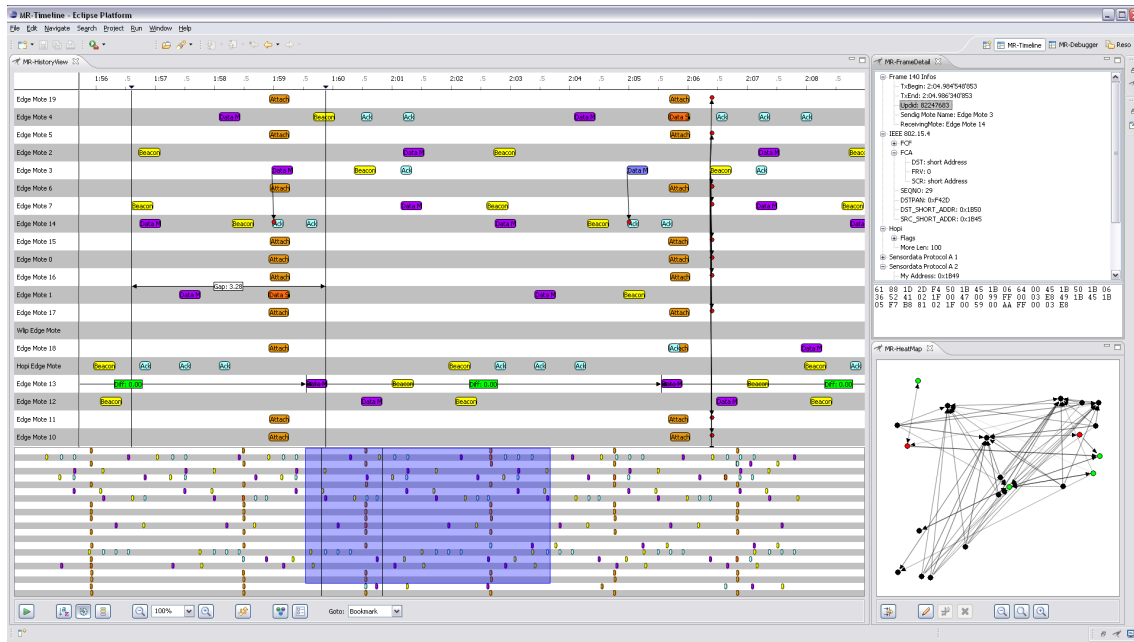


Abbildung 9: Eclipse Perspektive im Mote Runner Projekt

Wir haben drei übersichtliche, intuitiv zu bedienende Views programmiert, welche die Radiokommunikation zwischen Motes visuell darstellen. Diese Views sind eine grosse Hilfe beim Debuggen von mehreren Motes, da mit ihrer Hilfe Protokollverletzungen erkannt werden können und sich bei einem Deadlock die vorangegangenen Nachrichten analysieren lassen. Es lässt sich auch der Drift bei periodischen Signalen erkennen, welcher zu Störungen führen kann. Mit den Views kann man erkennen, was für ein Protokoll mit welchem Inhalt zu welchem Zeitpunkt von einem Mote gesendet wurde. Nachfolgend finden Sie eine kurze Beschreibung zu den einzelnen Views. Diese Beschreibung erklärt nicht alle Funktionen. Eine Erklärung aller Funktionen und deren Bedienung kann im Anhang A nachgelesen werden.

MR-HistoryView Hier werden die Pakete auf einer fortlaufenden Timeline platziert. Man hat die Möglichkeit zu sehen, welche Motes eine gesendete Nachricht empfangen haben. Des Weiteren kann die Zeit zwischen zwei oder mehreren Markierungen in Erfahrung gebracht werden und man sieht, ob periodische Nachrichten tatsächlich periodisch gesendet werden.

MR-FrameDetail In dieser Ansicht wird der Inhalt eines Pakets als Baum dargestellt. Der erste Knoten des Baumes enthält allgemeine Informationen über das Paket. Die weiteren Knoten beinhalten Informationen über die einzelnen Protokolle. Je weiter unten der Knoten ist, umso tiefer ist man im Paket.

MR-HeatMap Diese View zeigt die Motes, welche durch Punkte repräsentiert werden, in ihrer geographischen Position. Wenn ein Mote eine Nachricht sendet, wird eine Verbindung zu allen empfangenen Motes erstellt. Sofern keine weitere Kommunikation zwischen diesen zwei Motes stattfindet, wird die Verbindung nach und nach schwächer, bis sie zum Schluss ganz verschwindet. Wenn ein Mote eine Nachricht sendet, wird der Punkt grün, wenn sie eine Nachricht empfängt rot. In dieser View kann man Motes filtern, sodass nur noch die ausgewählten in der History View ersichtlich sind.

4.3 Ergebnisse MRFrameFilter

4.3.1 Generelle Übersicht

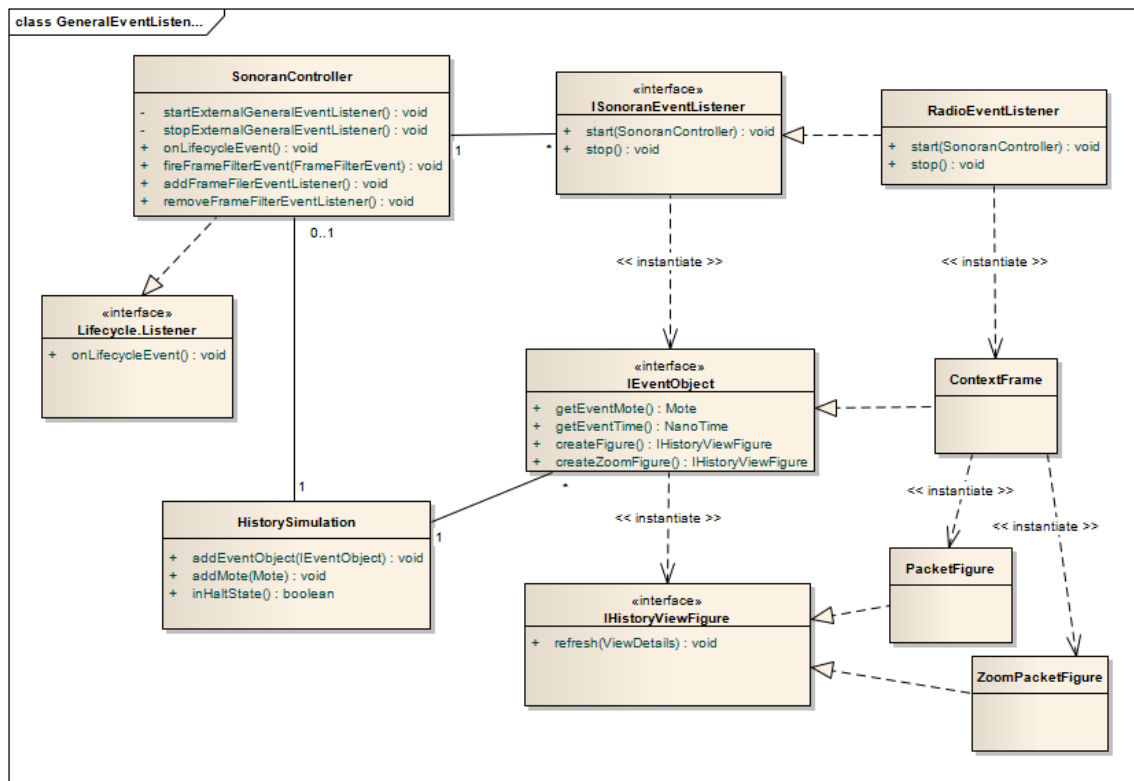


Abbildung 10: Klassen Diagramm, General Events

Die wichtigste Klasse ist der **SonoranController**. Dieser hat vier Hauptaufgaben. Zum ersten implementiert es den **Sonoran Lifecycle.Listener** und ist so immer up-to-date, ob eine Mote Runner Simulation mit Eclipse verbunden ist oder nicht. Zum zweiten ist der **SonoranController** Eventproducer für alle Views, die sich auf FrameFilter Events registrieren möchten (Die Views sind nicht in Abbildung 10 aufgezeigt). Zum dritten startet und stoppt er alle registrierten **ISONoranEventListener**, wenn eine neue Recording Session gestartet wird, und als viertes erstellt er für jede Recording Session eine **HistorySimulation**. In der **HistorySimulation** Klasse werden alle Events einer Simulation abgespeichert.

Alle Views können über den **SonoranController** auf die aktuelle **HistorySimulation** zugreifen und die darin enthaltenen Modeldaten auslesen. Wichtig ist, zu verstehen, dass in einer **HistorySimulation** nicht alle Daten seit dem Start der Sonoran Simulation enthalten sind, sondern nur alle Daten, seit der Benutzer sich über "Recording" in eine Simulation eingehängt hat. Wenn die Simulation schon länger lief, sind die Daten davor nicht gespeichert.

Sobald eine Simulation gestartet wird, startet der **SonoranController** alle registrierten **ISONoranEventListener**. Diese Event Listener bauen sich eine eigene Verbindung zu Sonoran auf und hören auf einen spezifischen Eventtyp. (Als Beispiel ist der **RadioEventListener** in Abbildung 10 angegeben, der auf alle **RadioUpdateEvents** von Sonoran hört). Immer wenn sie einen Event erhalten, erstellen sie ein **IEventObject** und hängen dies der **HistorySimulation** an. Die **HistorySimulation** ruft **fireFrameFilterEvent()** auf dem **SonoranController** auf und informiert alle registrierten Views darüber, dass ein neues **IEventObject** vorhanden ist. Die registrierten Views können mit den Methoden **createFigure()** und **createZoomFigure()** des **IEventObjects** eine graphische Repräsentation für das **IEventObject** erzeugen und dieses anzeigen.

4.3.2 Erweiterungspunkt für andere Events

Hauptsächlich wird die HistoryView für das Anzeigen von Radionachrichten genutzt. Je nach Problemstellung kann es jedoch auch wichtig sein, andere Events von Sonoran in der HistoryView anzuzeigen. Um in Zukunft andere Eventtypen anzuzeigen, haben wir mit dem `ISonoranEventListener`, dem `IEventObject` und dem `IHistoryViewFigure` drei Interfaces erstellt mit deren Implementation man einfach neue Eventtypen definieren und hinzufügen kann. Nachfolgend wird das Prinzip am Beispiel der `MotePositionEvents`, die bereits implementiert sind, aufgezeigt.

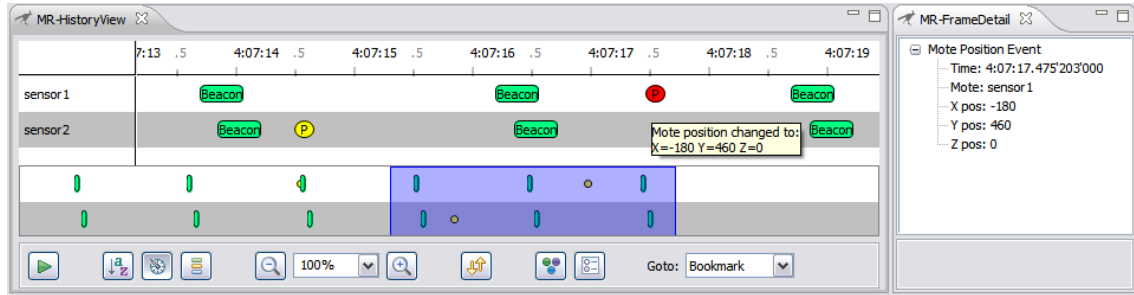


Abbildung 11: Mote Position Events

In Abbildung 11 sieht man, wie die `MotePositionEvents` im GUI angezeigt werden. Dafür musste folgendes implementiert werden.

MotePositionChangeListener implementiert `ISonoranEventListener`. In dieser Klasse wird eine Verbindung zum Sonoran Client erstellt, der auf diese Events hört. Wenn so ein Event empfangen wird, erstellt diese Klasse ein `MotePositionChangeObject`, welche `IEventObject` implementiert und fügt dieses der `HistorySimulation` hinzu. Diese Klasse muss man im `SonoranController` in der Methode `createExternalListenerList()` registrieren, abgesehen davon muss keine bestehende Klasse verändert werden.

MotePositionChangeObject implementiert `IEventObject`. In dieser Klasse ist der Event gekapselt. Jeder Event muss definitiv einem Mote und einem Zeitpunkt zugeordnet werden können. Dadurch kann es auf der View richtig positioniert werden. Mit den Methoden `createFigure()` und der `createZoomFigure()` kann ein `IHistoryViewFigure` erzeugt werden, welches den Event auf der View oder im Zoomcontainer anzeigt. Wenn man den Event im Zoomcontainer nicht anzeigen will, muss die entsprechende Methode null zurück geben.

MotePositionFigure implementiert `IHistoryViewFigure`. Um bei einem Klick auf das Figure etwas in der FrameDetail View anzuzeigen, kann man in der Methode `getDataOnClick()` eine `LinkedHashMap<String, Object>` zurück geben. Diese wird als Baum angezeigt. Die Verschachtelung erhält man, wenn als Value wieder eine `LinkedHashMap<String, Object>` mitgegeben wird.

4.3.3 Dissection Übersicht

Hauptsächlich werden im GUI Radioevents angezeigt. Diese durchlaufen vor ihrer Anzeige den Dissection Tree, in dem versucht wird herauszufinden, was für ein Protokoll die Nachricht implementiert. Folgendes Klassendiagramm (Abbildung 12) zeigt wie der Dissection Tree aufgebaut ist und von wem er verwendet wird.

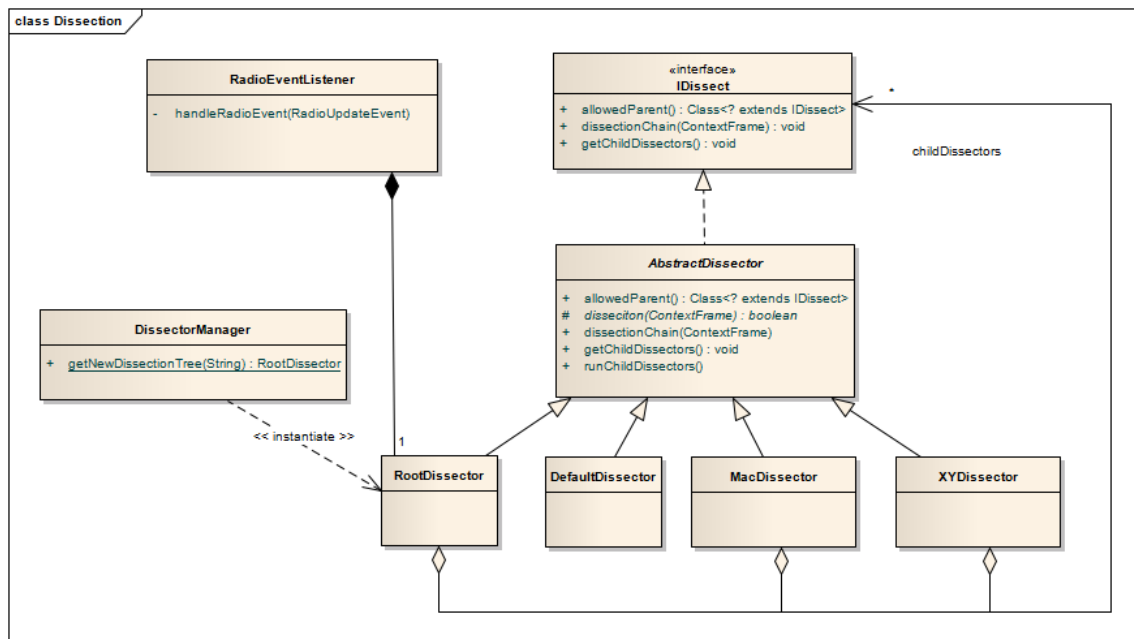


Abbildung 12: Klassen Diagramm, Dissection

Der **DissectorManager** ist eine statische Klasse, die für die Instanzierung und Verwaltung der Dissector Trees verantwortlich ist. Bei der Instanzierung des **RadioEventListener**s ruft dieser den **DissectorManager** auf, um Zugriff auf einen Dissector Tree zu erhalten. Der ParseTree wird vom **DissectorManager** anhand der Einstellungen auf der dazugehörigen Eclipse Preference Page aufgebaut. Alles ist so ausgerichtet, dass es später einfach möglich sein wird, verschiedene Parsebäume in einer Simulation zu haben. Dies könnte nötig werden, wenn beispielsweise für verschiedene Motes unterschiedliche Parsebäume gebraucht werden oder je nach Programm, das auf einem Mote läuft, ein unterschiedlicher Parsebaum benötigt wird.

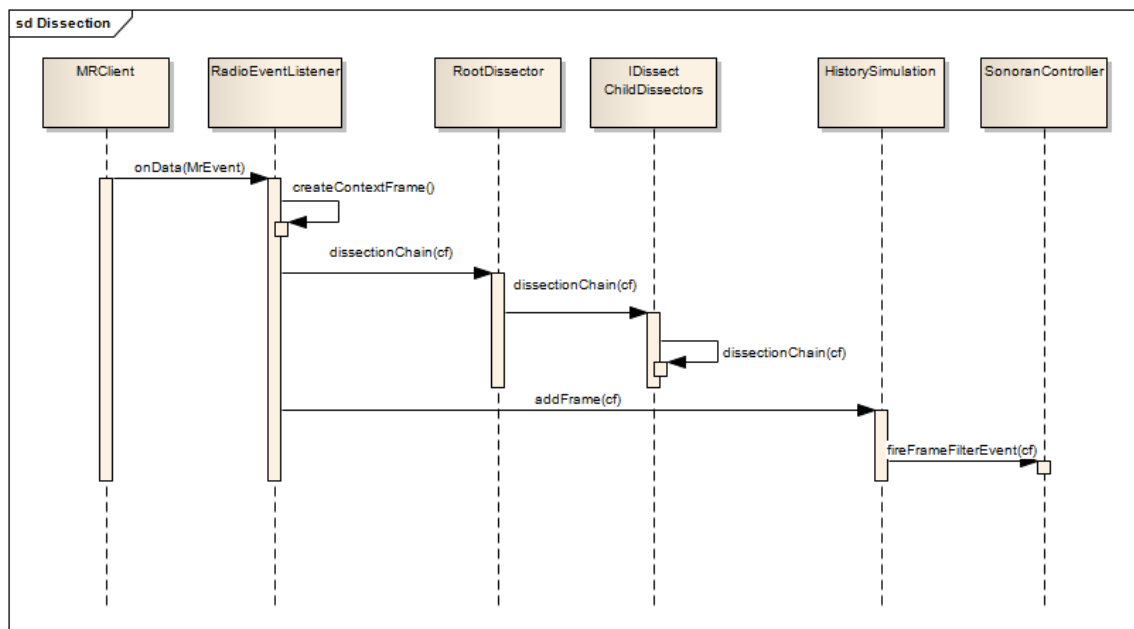


Abbildung 13: Sequenz Diagramm, Dissection

Wenn nun eine neue Radionachricht eintrifft (Abbildung 13), wird vom Sonoran Client der registrierte **RadioEventListener** aufgerufen. Dieser erstellt ein **ContextFrame**, wo später alle Resultate

der Dissection abgespeichert werden. Danach gibt er das `ContextFrame` seinem `RootDissector`. Jeder Dissector versucht nun, einen Teil der Nachricht zu parsen und wenn er einen Teil der Nachricht erfolgreich geparkt hat, leitet er diese den unter ihm eingehängten Dissectoren weiter. Dies geschieht so lange, bis die Nachricht komplett geparkt ist. Sobald die Nachricht fertig analysiert wurde, wird sie in der Simulation hinzugefügt. Die Simulation informiert über den `SonoranController` alle registrierten Views, dass ein neues `IEventObject` vorhanden ist.

4.3.4 Wie schreibe ich einen eigenen Dissector

Damit unsere Applikation fortlaufend von jedem Benutzer erweitert werden kann, haben wir ein API entwickelt, das jedem Benutzer ermöglicht für ein Protokoll einen eigenen Dissector zu programmieren. Dieser eigene Dissector kann während dem Laufenden Programm eingebunden werden. Um dies für den Benutzer möglichst einfach zu gestalten, haben wir einen Project Wizard erstellt, der die optimale Struktur für ein Dissector Projekt anlegt und auch das Skelett der eigenen Dissector Klasse erzeugt.

Die detaillierte Beschreibung des Interfaces und des Wizards wurde als Tutorial verfasst. Dieses ist im Anhang B zu finden.

Teil III

Projektmanagement

5 Projektorganisation

5.1 Autoren

Nachfolgend die beiden Autoren dieser Bachelorarbeit:



(a) Amon Grünbaum



(b) Patrick Dünser

Abbildung 14: Autoren

5.2 Betreuer

Von Seite der HSR wurden wir durch Professor Oliver Augenstein betreut.

5.3 IBM Mote Runner Team

Michael Bentsch ist der Manager des Teams. Thorsten Kramp hat die Projektleitung für Mote Runner. Marcus Oestreicher ist für die Entwicklung des Eclipse Plugins und Thomas Eirich für die Protokollentwicklung zuständig. Um dieses Kernteam gibt es zahlreiche Bachelor- und Master-Studenten sowie Doktoranden, welche das Projekt mit- und weiterentwickeln.

6 Vorgehensmodell

Beim Vorgehensmodell entschlossen wir uns für eine Mischung von Scrum und Extreme Programming. Zu Beginn teilten wir das gesamte Projekt in zwei Aufgabenbereiche auf. Der eine Teil umfasste das GUI, der andere die Aufarbeitung der Paketinformationen, sowie das Erstellen der API. Wir beide hatten je die Hauptverantwortung für einen dieser Teile. Danach erstellten wir einen groben Plan, in dem jeder von uns seinen Aufgabenbereich in vier Sprints aufteilte. Nach einem abgeschlossenen Sprint schauten wir, ob die Anforderungen des nachfolgenden Sprints noch dieselben sind wie zu Beginn. Falls sich diese geändert hatten, passten wir den Sprint entsprechend an.

Die Vorgaben unseres Industriepartners waren sehr vage. Das IBM Team hatte zwar viele Ideen, aber sie wussten nicht genau, was ihnen wichtig war und auf welche Aufgaben wir den Fokus richten sollten. Deshalb war unser Ziel, möglichst schnell einen ausführbaren Prototypen zu haben, welcher als Ideenpool, Diskussionsgrundlage und "Playground für das Mote Runner Team" diente. Der Prototyp wurde laufend den Wünschen und Anforderungen des Entwicklungsteam angepasst, erweitert und verbessert. Diese iterative Arbeitsweise ist bei IBM Research gebräuchlich, da viele Ideen und Anforderungen erst nach dem Erproben des Produkts entstehen. Darum haben wir von Beginn an sehr viel programmiert und Prototypen erstellt.

6.1 Soll- / Ist-Vergleich

Der Grobplan, den wir zu Beginn erstellt hatten, war sehr stabil. Das heißt wir mussten nichts an den Daten unserer Meilensteine ändern und waren, mit kleinen Abweichungen immer im Zeitplan. Die anfangs geplanten Arbeitspakete waren nur eine Bestandsaufnahme, welche laufend den neuen Anforderungen angepasst wurde. Weniger wichtige Arbeitspakete wurden deshalb gestrichen, dafür kamen Neue dazu. Ein strikter Soll-/Ist-Vergleich ist deshalb nicht möglich.

6.2 Zeiterfassung

Wir führten keine Zeiterfassung, da wir beide 100% am Projekt arbeiteten. Im Schnitt arbeiteten jeder etwa 40 Stunden pro Woche. Die geforderten 360 Stunden für die ETCS Punkte wurden bei weitem übertroffen.

6.3 Meetingstruktur

Wir führten während dem ganzen Projekt wöchentlich ein Meeting mit Prof. Oliver Augenstein. Bei diesen Meetings ging es einerseits darum, den aktuellen Stand zu präsentieren und andererseits wurden auch immer aktuelle Probleme gelöst, sowie das weitere Vorgehen besprochen.

Mit dem Team der IBM waren wir in ständigem Kontakt. Mindestens nach jedem Meilenstein haben wir ihnen die aktuelle Version unserer Plugins demonstriert und auf ihren SVN geladen, damit sie sie selber testen konnten.

Vor dem Start, in der Mitte und am Ende unseres Projekts gab es ein Gesamtmeeting, bei dem sowohl die Betreuer der IBM als auch Prof. Oliver Augenstein teilgenommen haben.

Teil IV

Arbeitspakete

7 Meilensteine

7.1 Übersicht

In der untenstehenden Tabelle sind die Daten für unsere Meilensteine angegeben. Für jeden Meilenstein gab es verschiedene Arbeitspakete, welche im GUI und im Protokollbereich erarbeitet wurden. In den nachfolgenden Kapitel werden alle Arbeitspakete kurz beschrieben.

| Meilenstein | Datum | GUI | Protokoll |
|-------------|-----------|--|--|
| 1. MS | 18. März | <p>1. Prototyp:</p> <ul style="list-style-type: none">• Eingehende Pakete werden erfasst und beim entsprechenden Mote dargestellt• Pakete werden im korrekten "Zeitmastab" aufgezeichnet | <p>1. Prototyp</p> <ul style="list-style-type: none">• Anbindung an Sonoran (Filterung der Events, Eventlistener)• Parsen des MAC Headers• Ausgeben des Default Payloads• Zusammenführen der beiden Prototypen |
| 2. MS | 15. April | <p>2. Prototyp</p> <ul style="list-style-type: none">• Zoomen funktioniert inkl. Zoomfenster• Markieren von Zeitabschnitt funktioniert und periodische Signale werden erkannt (Drifts erkennen) | <p>2. Prototyp</p> <ul style="list-style-type: none">• Implementierung Hopi Protokoll• Interface für Protokollentwickler definieren (verschiedene Hooks / AbstractFilterBase und Interface)• Loadmechanismus für eingene Protokollerweiterungen (Eclipse Extension Points) |

| | | | |
|-------|----------|--|---|
| 3. MS | 13. Mai | 3. Prototyp <ul style="list-style-type: none"> • HeatMap -> Aktionen (senden / empfangen) von Motes werden durch blinken dargestellt • Sortier Algorithmus (wenn ein Mote selektiert wird, werden Motes die viel mit diesem kommunizieren, in dessen Nähe "gezogen") • Protokolle Farbe geben • Abstrakte Klasse für andere anzuzeigende Events | 3. Prototyp <ul style="list-style-type: none"> • Eigene View um Filter zu selektieren / Anzeigeeinstellungen zu machen • Filterreihenfolge einstellen • Extension Point für generelle Events (nicht Radio) |
| 4. MS | 10. Juni | Final Version <ul style="list-style-type: none"> • HeatMap Gruppen auswählen (Filtern) und in der Timeline anzeigen. • Drag & Drop von Motes • Input von IBMer, so weit möglich, umgesetzt | Finale Version <ul style="list-style-type: none"> • Global dissector cache • Wizard zum erstellen eines Dissectors • Bugfixing/Testing |

Tabelle 2: Meilenstein Übersicht

7.2 1. MS (18. März 2011)

Im ersten Meilenstein wurden die Grundlagen von Eclipse Plugins erlernt, verschiedene Möglichkeiten für die grafische Darstellung ausprobiert und analysiert und die grundlegende Struktur des Projekts erstellt. Eine Erste Version des GUI's zeigt alle einkommenden Radionachrichten nach Zeit und Motes geordnet an. Nachfolgend sind die einzelnen Arbeitspakete des Meilensteins genauer beschrieben.

7.2.1 GUI

Eingehende Pakete werden erfasst und beim entsprechenden Mote dargestellt Die eingehenden Pakete werden, mittels Anbindung an Sonoran, als Events empfangen und im GUI dargestellt. Da Eclipse SWT Komponenten verwendet waren wir bei der Wahl der Bibliothek für die Oberfläche eingeschränkt. Nach zahlreichen Tests mit verschiedenen Grafikbibliotheken wie SWT, JFace, GEF und Draw2d entschlossen wir uns für Draw2d, weil Draw2d eine Zeichnungsfläche (Canvas) zur Verfügung stellt, auf der man leicht Freiformen zeichnen kann. Draw2d ist ein Layout und Rendering Toolkit Konstrukt, das auf SWT aufsetzt. Es ist als Teil von GEF entstanden, kann aber auch unabhängig von diesem verwendet werden.[5]

Pakete werden im korrekten "Zeitmassstab" aufgezeichnet Die Pakete werden in der korrekten Reihenfolge und mit exaktem Abstand zueinander dargestellt. Zudem gibt es einen Header, welcher die Zeitachse beinhaltet, die Sekunden in einem fixen Abstand darstellt. Horizontales und vertikales Scrollen funktioniert. Wenn die Scrollbar am rechten Rand ist, wird der Realtime-Modus aktiviert. Das heisst die Scrollbar bleibt am rechte Rand um so die aktuell eingehenden Pakete in Echtzeit anzuzeigen.

7.2.2 Protokoll

Anbindung an Sonoran Die Anbindung an Sonoran ist bereits in den Eclipse Projekten MR-Sonoran und MRDevelBase vorhanden. Ziel in diesem Arbeitsschritt ist es, sich beim bestehenden Client zu registrieren und von ihm alle Events von Sonoran zu erhalten. Das Verbinden zu Sonoran soll über die bestehende View MR-Info geschehen und soll nicht selbst implementiert werden. Die Mote Runner Simulationsumgebung (Sonoran) hat verschiedene Zustände. Je nachdem in welchem Zustand sich die Simulation befindet, muss das HistoryView Plugin verschiedene Informationen Anzeigen und auf verschiedene Events reagieren. In der nachfolgenden Abbildung 15 sind die verschiedenen Zustandsabfragen, die Anzeige in der HistoryView, sowie die Events visualisiert:

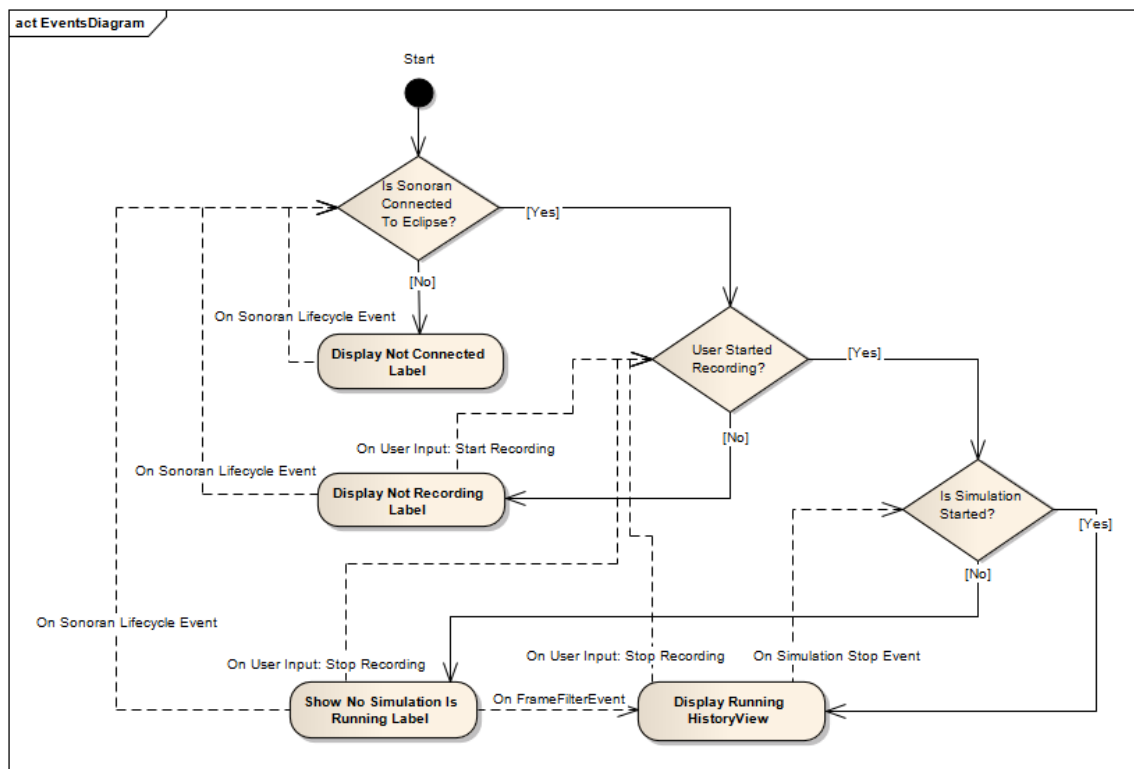


Abbildung 15: Events und Zustände von Mote Runner in Verbindung mit der History View

Da das Aufzeichnen aller Events in der HistoryView sehr rechenintensiv ist und der Benutzer dies eventuell nicht bei jeder Debug Session im Eclipse machen will, muss der User das Recording in der HistoryView manuell starten und stoppen. Wenn Eclipse mit dem Sonoran Simulationsprozess verbunden ist, heisst dies noch nicht, dass auch eine Simulation am laufen ist. Deshalb braucht es auch eine Abfrage ob eine Simulation läuft. Wie die Mote Runner Komponenten zusammenspielen wurde in Kapitel 2.4 beschrieben.

Parsen des MAC Headers Die meisten Sensornetzwerke benutzen für die Radiokommunikation Nachrichten mit einem IEEE 802.15.4 MAC Header. Der erste Dissector soll Radionachrichten von diesem Protokoll korrekt aufparsen können.

Ausgeben des Default Payloads Wenn kein passender Dissector für das ganze Paket oder einen Teil davon vorhanden ist, soll der Payload des Pakets trotzdem angezeigt werden. Dafür soll ein Default Dissector geschrieben werden.

Zusammenführen der beiden Prototypen (Gesamtarchitektur) Das GUI und das Protokollplugin, welche in zwei verschiedenen Eclipse Plugin Projekten erstellt sind, sollen miteinander

kommunizieren. Das GUI soll sich beim Protokollplugin auf Events anmelden können und auch mit geeigneten Methoden auf die Daten im Protokollprojekt zugreifen können.

In diesem Arbeitspaket ging es darum die Views des einen Plugins mit den Daten aus dem anderen Plugin zu füttern, so dass der gesamte Datenfluss von der Simulation bis zur Anzeige für den User funktioniert. Die Gesamtarchitektur wie sie hier erarbeitet wurde, ist bereits im Kapitel 4.1 dokumentiert.

7.3 2. MS (15. April 2011)

In diesem Meilenstein sind erste Werkzeuge für den Benutzer geschaffen worden. Zudem wurde die Architektur für das Dissector API ausgearbeitet und die einkommenden Nachrichten werden analysiert. Erste Protokollinformationen werden im GUI angezeigt.

7.3.1 GUI

Zoomen funktioniert inkl. Zoomfenster Man kann horizontal zoomen um mehr oder weniger Details über Pakete zu sehen. Es gibt verschiedene “Zoomstufen”, welche zum einen die Details der Informationen auf den Paketen, zum anderen die Genauigkeit der Zeitachse bestimmen. Des Weiteren gibt es unterhalb der HistoryView ein Zoomfenster, welches den packetcontainer miniaturisiert darstellt. Somit gibt das Zoomfenster einen Überblick und hilft beim Orientieren. In diesem Zoomfenster kann man mittels eines Rechtecks bestimmen, welcher Bereich in der HistoryView angezeigt werden soll. Durch Grössenänderung des Rechtecks kann man ebenfalls den Zoom bestimmen. (siehe Abbildung 16)

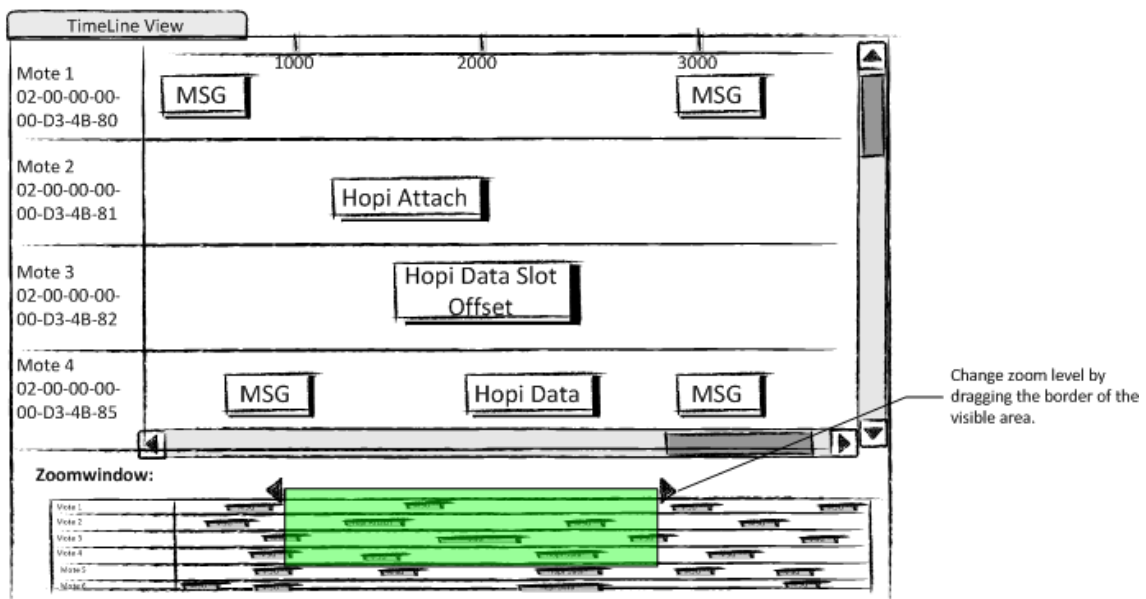


Abbildung 16: PaperPrototype Zoom Window

Zuerst wollten wir für das Zoomfenster die Klassen `Thumbnail` und `ScrollableThumbnail` verwenden. Diese Klassen machen von einem bestehenden Figure ein Screenshot, also erstellen ein Bild, und stellen dieses verkleinert dar. Allerdings wird beim skalieren des Thumbnails immer die X- und Y-Koordinate angepasst. Wir wollten aber nur, dass die X-Achse skaliert. Darum haben wir die relevanten Methoden der Klasse überschrieben. Als es danach immer noch nicht funktionierte, besprachen wir das Problem mit unserem Dozenten. Im Gespräch mit ihm, stellte sich heraus, dass in den Tiefen von Java Bilder nur mit einem Faktor für X- und Y-Koordinate skaliert wird. So mussten wir für das Zoomfenster ein komplett neues Figure erstellen. Dies war einer der aufwändigsten Arbeiten bei unserem Projekt.

Wenn die Pakete den Rand des Zoomfenster erreicht hatten, musste dieses wieder verkleinert und gegebenenfalls auch alle Pakete verkleinert werden. Dies war nötig, damit der Selektor den

korrekten Ausschnitt des Hauptfensters darstellt. Zudem muss beim Selektor immer wieder dessen Grösse angepasst werden. Des Weiteren hatten wir sehr lange Probleme mit der Positionierung des Selector nach dem Zoomen. Die Grösse des Selector war die gewünschte aber die Position war erst korrekt, als man ihn um ein paar Pixel bewegt hatte. Der Grund für dieses Phänomen war, dass der `UpdateManager` des `Draw2d` selbst entscheidet, wann er die invalidierten Figures neu zeichnet.

Dies war für uns fatal, da wir beispielsweise oft mit `setSize()` die grösse der View neu gesetzt haben. Wenn kurz darauf der `Selector` sich der View anpassen wollte und dazu die `getSize()` Methode der View aufgerufen hatte wurde teilweise nicht der zuvor gesetzte Wert zurück gegeben, sondern noch den Alten. Erst nachdem der `UpdateManager` lief wurde die neue Grösse zurück gegeben. Damit synchronisierte sich der `Selector` immer zu einem veralteten Zustand.

Diese Problem haben wir mit zwei verschiedenen Ansätzen gelöst. Zum einen benutzen wir, wenn möglich, für Berechnungen neuer Grössen und Positionen, Variablen. Zum anderen, falls wir doch auf Objekte zugreifen müssen, um deren Grösse oder Position zu erhalten, stossen wir den `UpdateManger` von Hand an.

Markieren von Zeitabschnitt funktioniert und periodische Signale werden erkannt (Drifts erkennen) Diese Aufgabe besteht aus zwei Teilaufgaben:

1. Es gibt ein Tool mit dem man zwei oder mehrere Markierungen setzen kann. Jeweils zwischen zwei Markierungen wird der Zeitabstand angezeigt. Wenn möglich wird in der Nähe des Anfangs eines Pakets ein “Magnet” geschaltet, so dass die Markierung an den Anfang des Pakets springt.
2. Es gibt eine Funktion, mit welcher man überprüfen kann, ob ein periodisches Signal auch wirklich periodisch gesendet wird. Unterschiede in der Periodizität werden mit einer Farbe markiert. Zusätzlich sollte die Toleranz vom Benutzer eingestellt werden können. (siehe Abbildung 17)

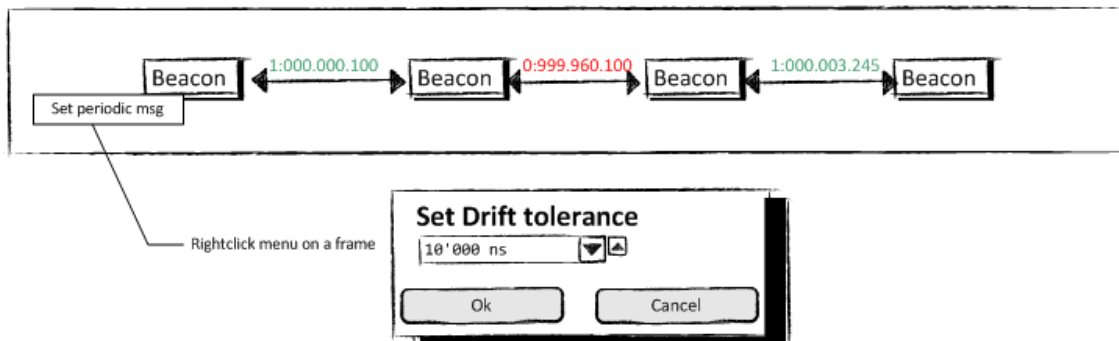


Abbildung 17: Paperprototype Driftanzeige

Beim Usability-Test mit dem Mote Runner Team kam die Anforderung, dass für die Markierungen sowie für Pakete noch Lesezeichen erstellt werden sollte. Zu diesen Lesezeichen sollte man immer wieder zurück springen können und damit besser zwischen verschiedene Problemstellen navigieren. Das Speichern dieser Lesezeichen war auch kein Problem. Was aber Schwierigkeiten bereitete, war, dass es in `Draw2d` keine Combobox zur Verfügung stellt. Zuerst versuchten wir die Combobox, sprich deren Design und Verhalten, mittels Label und Button selbst zu erstellen. Dies war sehr umständlich, und so beschlossen wir die gesamte Button Leiste mit einem SWT-Komposit zu realisieren, auf dem wir eine SWT Combobox platzieren konnten.

Der Drift-Dialog wird mittels SHIFT plus Mausklick geöffnet. Wir haben das so gelöst, da `Draw2d` auch kein Kontextmenü bietet. Darum mussten wir uns entscheiden, das Kontextmenü mit den zur Verfügung stehenden Komponenten zu basteln, was einen Zeitaufwand von ca. 2 Tagen gewesen wäre, oder eine andere Lösung zu finden. Nach Rücksprache mit den IBM Mitarbeitern wurde entschieden, dass die Zeit zu wertvoll ist, um was eigenes zu implementieren und wir entschieden uns gemeinsam für den SWT-Dialog.

7.3.2 Protokoll

Implementierung Hopi Protokoll Als zweites Protokoll soll das Hopi Protokoll implementiert werden. Dieses Protokoll ist ein gutes Beispiel für ein Sensornetzprotokoll, da es verschiedene Pakettypen enthält, die viele wichtige Eigenschaften von Sensornetzwerken besitzen.

- Beacon Message: Dies ist eine periodische Nachricht, bei der es möglich sein muss, die Periodizität festzustellen und zu überprüfen.
- Attach Message: eine Message, um sich bei einem anderen Knoten (also einem anderen Sensor im Netz) zu registrieren.
- Data Message: In einer Data Message können Datenpakete von verschiedenen Knoten enthalten sein. Beim Parsen muss eine solche Verschachtelung korrekt sichtbar sein.

Das Hopi Protokoll basiert auf dem bereits erstellten IEEE 802.15.4 Mac Header. Es soll also nur aufgerufen werden, falls ein korrekter Header geparkt wurde.

Interface für Protokollentwickler definieren Die Gemeinsamkeiten zwischen den bereits implementierten Protokollen müssen in einem Interface und/oder einer abstrakten Basisklasse zusammengefasst werden und es braucht eine einfache Möglichkeit für den Protokollentwickler einen neuen Parser zu erstellen. Es muss möglich sein zu definieren welche Parser vor und nach diesem Parser greifen können und die bestehenden Parser dürfen dabei nicht verändert werden. Dieses Interface wird sich im Laufe des Projekts sicher noch verändern und den neuen Anforderungen angepasst.

Da wir vor diesem Projekt noch nie einen Parser für ein Protokoll geschrieben haben, mussten wir uns zuerst einmal in dieses Thema einarbeiten. Ein guter Startpunkt dafür, war zu schauen wie Wireshark seine Pakete parst und wie man dort selber einen Dissector schreiben kann [6]. Das Parsen der Protokolle und alle Herausforderungen, die es dabei gibt, hatte Wireshark genau gleich wie wir sie haben. Besonders die Art, wie Protokolle ein Infofeld des darunterliegenden Protokolls erweitern können und wie Resultate des Parsens gespeichert werden, haben wir sehr ähnlich wie Wireshark gemacht. Das Interface wurde gründlich mit verschiedenen Personen in der IBM besprochen.

Wie in Eclipse üblich haben wir auch für die dissector Extension ein Interface (`IDissect`) definiert und eine abstrakte Basisklasse (`AbstractDissector`) erstellt. Überall im Code wird das Interface verlangt. Wenn jemand will, kann er einen Dissector also komplett selber schreiben. In fast allen Fällen macht es jedoch mehr Sinn, die `AbstractDissector` Klasse zu erweitern.

Loadmechanismus für eigene Protokollerweiterungen (Eclipse Extension Points) Für jeden Protokollentwickler muss es möglich sein, das Interface für neue Protokollfilter zu implementieren und ohne Eclipse neu zu kompilieren seine Erweiterung zu nutzen. Eine Möglichkeit dafür ist das Eclipse Extension Point Interface zu nutzen, das auch das Plugin von uns verwendet, um im Eclipse angezeigt zu werden. Da die Eclipseplugin Entwicklung relativ komplex ist, soll dem User mit einem Wizard so viel wie möglich abgenommen werden. Die Implementierung dieses Wizards ist jedoch optional in diesem Meilenstein.

Analyse: Anfangs gingen wir davon aus, dass ein externer Dissector mittels einem XML Dokument beschrieben werden könnte und der Benutzer sich den Dissector nur zusammen-klicken kann. Dies erwies sich jedoch sehr schnell als falscher Ansatz, da es viel zu viele Ausnahmen und Möglichkeiten beim Parsen gibt. Die zweite Idee war einen eigenen Eclipse Extension point zu definieren, damit ein Benutzer ein Plugin schreiben kann, das unser Plugin erweitert. Dies wäre sicher eine saubere Lösung gewesen, jedoch hat diese ein paar grobe Nachteile. Der Benutzer muss dafür das Eclipse “Plug-in Development Environment” (PDE) installiert haben und kann sein Plugin immer nur testen in dem er ein neues Eclipse startet. Änderungen am eigenen Dissector wären somit sehr kompliziert und unhandlich.

Lösung: Schlussendlich sind wir auf folgende Lösung gekommen: Der Benutzer, der einen Dissector schreiben will, erstellt den Dissector als normales Java Projekt und exportiert dieses in ein

JAR File. In den Settings der HistoryView gibt er dieses JAR an und der Dissector wird on the fly ins Eclipse nachgeladen. Dieses Nachladen funktioniert indem der ClassLoader des HistoryView Plugin mit einem `URLClassLoader` um dieses JAR erweitert wird. Dieser Weg ist für uns ein wenig komplizierter, jedoch ist es für den Benutzer sehr angenehm, da er für neue Versionen seines Dissectors einfach sein JAR File überschreiben kann. Er benötigt so nicht eine spezielle Infrastruktur und muss sein Eclipse nicht neu starten.

7.4 3. MS (13. Mai 2011)

In diesem Meilenstein wurde das GUI um die Heat Map erweitert und den bisher fixen Parsebaum für den Benutzer konfigurierbar gemacht. Zudem wurde im GUI ein Abstraktionslevel für die Radioevents eingefügt, damit später auch andere Events dargestellt werden können.

7.4.1 GUI

HeatMap; Aktionen (senden / empfangen) von Motes werden durch blinken dargestellt Die HeatMap ist eine zusätzliche Eclipse-View. Auf einer Heat Map werden die Motes räumlich angeordnet und der Verkehr untereinander mit Pfeilen dargestellt. Bei häufiger Kommunikation untereinander sind die Pfeile schwarz. Wenn keine Kommunikation zwischen den Motes stattfindet, werden die Pfeile immer durchsichtiger bis sie schliesslich verschwinden. (Siehe Abbildung 18)

Zu Beginn kam die Idee die Pfeile zwischen den Motes in der Farbe des gesendeten Protokolls anzuzeigen. Diese Idee wurde jedoch aus folgenden Gründen wieder verworfen:

- Wenn verschiedene Protokolle in kurzer Zeit gesendet werden, können die einzelnen Farben der verschiedenen Protokoll nicht erkannt werden.
- Je nach Farbwahl die Protokolle haben, könnte man sie bei höherer Transparenz nicht mehr sehen.
- Eine Rücksprache mit IBM-Mitarbeitern hat ergeben, dass dieses Feature nicht nötig ist

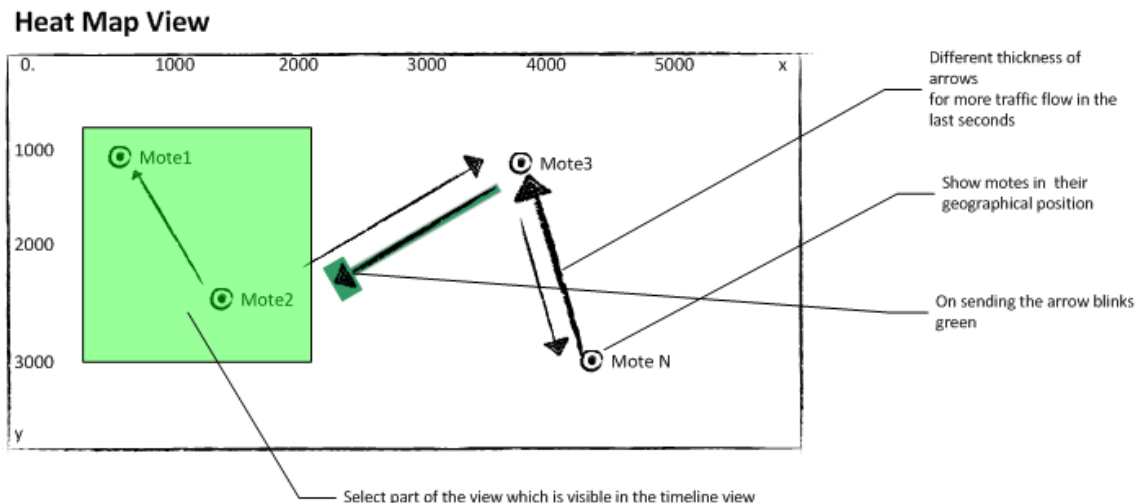


Abbildung 18: Paperprototype Heat Map

Sortier Algorithmus (wenn ein Mote selektiert wird, werden Motes die viel mit diesem kommunizieren, in dessen Nähe "gezogen") Implementierung eines Algorithmus, bei welchem nach Auswahl eines Motes die Motes, die viel mit dem ausgewählten Mote kommunizieren, in dessen Nähe gezogen werden. Der Anforderung des Algorithmus hat sich nach der ersten Implementation ein wenig geändert. Da die bis jetzt entwickelten Protokolle meist hierarchische

Strukturen aufweisen, war die neue Anforderung, dass solche Motes die häufig dem ausgewählten Mote Daten senden oberhalb und solche die viel Daten des ausgewählten Mote empfangen unterhalb sind. Solche die keine Datenaustausch mit dem ausgewählten Mote haben, werden am Ende angefügt. Damit lässt sich der Kommunikationsfluss darstellen. Meistens senden die Kinder den Eltern Nachrichten. Bei diesem Algorithmus wären somit die Kinder des ausgewählten Motes oberhalb und die Eltern unterhalb positioniert.

Protokolle Farbe geben Es gibt die Möglichkeit zu bestimmen, welche Protokolle man angezeigt haben möchte. Und man kann ebenfalls bestimmen, welche Farbe die Protokolle haben. Das Einstellen der Protokollfarben wurde in einem Zug mit den Anzeigeeinstellungen im Protokoll durchgeführt. Diese sind im Arbeitspaket “Eigene View um Filter zu selektieren / Anzeigeeinstellungen zu machen” beschrieben.

Abstrakte Klasse für andere anzuzeigende Events Es soll eine GUI Klasse geben, die einfach erweitert werden kann, um eigene Eventtypen im GUI anzuzeigen. Wir haben ein Interface `IHistoryViewFigure` erstellt, das alle Objekte, die in der HistoryView angezeigt werden sollen, implementieren müssen. Dieses Interface erweitert das `IFigure` Interface, welches alle `Draw2d` Objekte implementieren. Damit bei einem neuen Event nur das `FrameFilter` Plugin erweitert werden muss, haben wir dieses GUI Interface für die einfachere Erweiterbarkeit im `FrameFilter` Plugin definiert.

7.4.2 Protokoll

Eigene View um Filter zu selektieren / Anzeigeeinstellungen zu machen Im GUI soll es eine Property View oder ein eigenes Property Einstellungsfenster geben, in dem man selektieren kann, welche Filter in der aktuellen Session angewendet werden. In dieser Ansicht soll es auch gerade möglich sein, die Anzeigeeinstellungen (Bsp. Farbe der einzelnen Protokolltypen) für das GUI zu ändern.

Diese Funktion ist mit Eclipse Preference Pages implementiert. Eclipse bietet mit dem `PreferenceStore` eine einfache Möglichkeit, Einstellungen eines Plugins zu verwalten und zu persistieren. Alle vom Standard abweichenden Einstellungen werden in eine Datei im Workspace abgespeichert. Macht der Benutzer keine Änderung an der Standardeinstellungen, wird die Datei nicht angelegt. Dadurch wird die Speicherung spezifischer Einstellungen minimiert. [7]

Bei unserem Plugin kann der Benutzer verschiedene Farben für die verschiedenen Protokolle definieren. Weiter kann er JAR Files von externen Dissectoren angeben. Als letztes kann der Benutzer auch noch angeben, welche Dissectoren für das Parsen benutzt werden und in welcher Reihenfolge diese angewendet werden.

Filterreihenfolge einstellen Bei Filtern der gleichen Hierarchiestufe muss der Benutzer die Reihenfolge, wann welcher Filter angewendet wird, selber bestimmen können. Dies ist wichtig, da es sein kann, dass mehrere Filter mit dem Payload etwas anfangen können. Der erste Filter der den Payload brauchen kann, wird diesen als “analysiert” markieren, so dass spätere Filter diesen gar nicht mehr zu sehen bekommen.

Diese Funktion wurde in der Preference Pages zur Auswahl der Dissectoren und derer Reihenfolge implementiert. (siehe Kapitel “Eigene View um Filter zu selektieren”) Die Reihenfolge kann mittels drag & drop geändert werden.

Extension Point für generelle Events Für alle anderen Events ausser den Radio Events, die Sonoran produziert, soll es eine einfache Möglichkeit geben eine Extension zu schreiben, die diesen Event in der Timeline anzeigt.

Diese Funktion wurde bereits sehr genau in Kapitel 4.3.2 beschrieben.

7.5 4. MS Final Version (10. Juni 2011)

Beim Meilenstein 4 musste die finale Version fertig sein. Neben ein paar neuen Funktionen ging es hauptsächlich darum, die bereits erarbeiteten Funktionen zu perfektionieren und ausgiebig zu

testen.

7.5.1 GUI

HeatMap Gruppen auswählen (Filtern) und in der Timeline anzeigen. Auf der HeatMap kann ein Bereich mit einer bestimmten Anzahl Motes markiert werden. Diese Motes werden in der der HistoryView angezeigt. Auf diese Funktion sind wir besonders stolz, da dies sehr sauber implementiert ist und sehr gut funktioniert. Zum einen wird der Mote gelb eingefärbt, wenn sich die Maus über dem Mote befindet. Dabei spielt es keine Rolle, ob man über dem Mote auf der HistoryView oder auf der HeatMap ist. Zusätzlich wird der Mote wenn möglich in die Mitte der Anzeige platziert, sofern er nicht bereits im sichtbaren Bereich ist. Zum zweiten kann ein oder mehrere Motes per Mausklick selektiert werden. Diese werden blau eingefärbt. Zum Dritten können die bereits selektierten Motes mittels Filter-Knopf gefiltert werden, so dass nur noch diese angezeigt werden. Damit die beiden Views immer einen konsistenten Zustand haben, werden bei all diesen Aktionen Events verschickt, welche beide Views mittels Listener empfangen und entsprechend behandeln können.

Input von IBM Mitarbeitern, so weit möglich, umgesetzt Alle Inputs der Tester wurden nach Möglichkeiten umgesetzt. Während der ganzen Arbeit wurden aktuelle Versionen den IBM Mitarbeitern präsentiert und ihnen zum Testen gegeben. So erhielten wir fortlaufend Inputs, was noch besser oder anders gemacht werden kann beziehungsweise welches Feature doch nicht benötigt wird.

Drag & Drop von Motes Positionierung der Motes selber bestimmen durch drag & drop. Die "Drag & Drop"-Funktion wurde von verschiedenen Testern gewünscht. Deshalb war es uns wichtig, dass wir dieses Feature implementierten. Als Vorarbeit hatten wir die alphabetische Sortierung, sowie die Sortierung nach oben beschriebenen Algorithmus programmiert. So konnten wir größtenteils die vorhandenen Methoden für dieses Feature nutzen.

7.5.2 Protokoll

Global dissector cache Neben dem Hinzufügen von Pfaden zu externen Dissectoren, soll es auch einen Ordner geben, in dem in allen JAR-Files nach Dissectoren gesucht wird.

Wir haben im lib Verzeichnis von Mote Runner den Ordner java-dissectors hinzugefügt. Alle darin enthaltenen JAR Files werden bei jedem Start der HistoryView nach Dissectoren durchsucht. Der Ort dieses Verzeichnis eignet sich gut, da es bei der Installation von Mote Runner automatisch angelegt wird und da der Benutzer beim ersten Starten von den Mote Runner Eclipse Plugins bereits den Pfad zum Mote Runner Verzeichnis angeben muss. So ist diese Information bereits vorhanden. Für alle spezifisch angegebenen externen Files wird beim Starten immer eine Kopie des JAR Files erstellt, damit der User, wenn er an diesem JAR noch etwas ändert, immer das File neu builden kann. Alle Files in diesem globalen Ordner werden nicht kopiert. Wenn also eine Änderung an einem darin enthaltenen Dissector gemacht wird, muss Eclipse neu gestartet werden.

Wizard zum erstellen eines Dissectors Um das Erstellen eines eigenen Dissectors für ein Protokoll so einfach wie möglich zu gestalten, soll ein Wizard erstellt werden, der so viel wie möglich dem Protokollentwickler abnimmt.

Das Komplizierte an einem Dissector Projekt sind zu Beginn folgende Punkte:

- Es muss die java-aux-dissectors.jar Library dem Classpath hinzugefügt werden.
- Es muss die Klasse `AbstractDissector` erweitert werden.
- Es muss ein JAR File des Projekts erzeugt werden.
- Das JAR File muss in den HistoryView Settings eingebunden werden, damit der neue Dissector verwendet wird.

Alle diese Punkte sind im Wizard enthalten. Im neu erzeugten JavaProjekt ist im Classpath bereits das Library File hinzugefügt. Im Wizard können ein paar Startinformationen zum Protokoll angegeben werden und es wird ein Skelett der Dissector Klasse erzeugt. Für das Generieren des JAR Files des eigenen Dissectors wird ein Ant-Buildfile angelegt, mit dem das Projekt jederzeit gebuildet werden kann. Das vom AntBuilder erzeugte JAR File wird automatisch in die Liste der externen JAR-Files von der HistoryView aufgenommen und es öffnet sich ein Dialog in dem der Dissector dem Parsetree hinzugefügt werden kann.

Zudem erzeugt der Wizard eine JUnit4 Test Klasse in der einfach aufgezeigt wird, wie man seinen eigenen Dissector einfach testen kann. Mehr Informationen wie der Wizard und das Erzeugen eines eigenen Dissectors funktioniert, ist im Kapitel B „How to write a Dissector“ beschrieben.

Bugfixing/Testing Das Programm soll ausgiebig getestet werden und möglichst alle Fehler sollten beseitigt werden.

Das Testen ist natürlich in jedem Arbeitspaket bereits enthalten, wurde im letzten Meilenstein jedoch nochmals intensiver durchgeführt. In verschiedenen Bereichen wurden verschiedene Tests durchgeführt: Nachfolgend zwei Beispiele:

- Unit Tests:
 - Für das FrameFilter Projekt wurden verschiedene UnitTests erstellt, um die einzelnen Funktionen zu überprüfen.

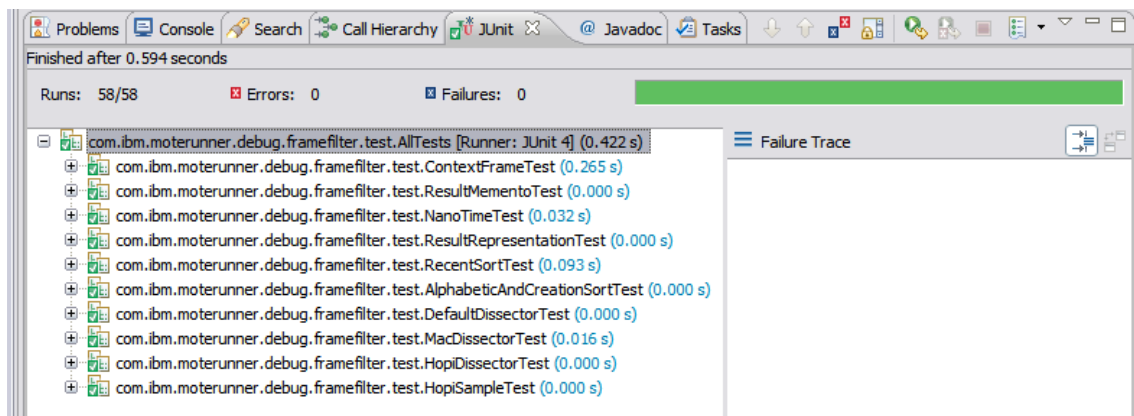


Abbildung 19: JUnit Tests des FrameFilter Plugin

- Usability Test Dissector API:
 - Um zu testen wie gut das Dissector API funktioniert, wurde es von einem anderen IBM Mitarbeiter getestet. Der detaillierte Usability Test ist im Anhang D beigelegt.

Teil V

Abschluss

8 Future Work

Während dem Projekt sind uns und den Personen, welche die Software getestet haben, viele neue Ideen gekommen, was zusätzlich noch gut wäre zu implementieren. IBM wollte mit unserer Arbeit eine möglichst breite Palette an Möglichkeiten und Funktionen ausprobieren, damit man ein Grundtool hat, welches man einsetzen kann um damit herausfinden, welche Funktionen noch perfektioniert und verfeinert werden müssen. Dass nicht alle Funktionen bis ins letzte Detail implementiert sind und immer neue Ideen dazu kamen, ist dabei logisch und auch beabsichtigt. Dieses Kapitel gibt einen kurzen Überblick über die nächsten Arbeiten die anstehen:

Dissectoren für die gängigen Protokolle implementieren. Für die Software sollten, für die am meisten verwendeten Protokolle, bereits Dissectoren vorhanden sein. Besonders für das WLIP Protokoll braucht es einen Dissector.

Weitere Events anzeigen. Wie in Kapitel 4.3.2 beschrieben, kann man einfach andere Events in der HistoryView anzeigen. Besonders die VMStart Events wären sinnvoll, da diese zeigen wann sich die VM auf dem Sensor startet. Je nachdem wie viele Events implementiert werden, muss es für den Benutzer auch möglich sein die Anzeige der Events zu filtern.

Mehrere Dissector Trees verwenden. Für verschiedene Anwendungsfälle könnte es sinnvoll sein verschiedene Protokoll Trees zu definieren. Dies wurde bereits vorbereitet, indem beim Erstellen eines Dissector Trees ein Profil angegeben werden muss. Momentan wird überall automatisch das Default Profil verwendet. Im gleichen Zug kann es auch sinnvoll werden in einer Simulation verschiedene Dissection Trees zu brauchen. Momentan hat der RadioEventListener einen Dissection Tree, mit dem er alle Nachrichten parst. Man könnte aber auch sehr einfach das Programm so umschreiben, dass es für jeden Mote oder für jeden Mote Typ einen eigenen Dissection Tree gibt.

Mehrere FrameDetail Tabs. Mehrere analysierte Paketinhalte in verschiedenen Tabs gleichzeitig anzeigen, um diese besser vergleichen zu können.

Similar Methode verbessern. Es gibt die Möglichkeit Pakete im GUI als periodisch zu markieren. Damit dies funktioniert, braucht es die similar Methode die zwei ContextFrames vergleicht und prüft ob diese ähnlich sind. Diese Methode ist momentan für alle Pakettypen gleich und entscheidet nicht immer richtig. So wird ein Hopi Data Paket mit Sensordaten von 3 Sensoren und ein Paket mit Sensordaten von nur 2 Sensoren nicht als similar angezeigt. Eine mögliche Verbesserung ist, dass die similar Methode von den Dissectoren überschrieben werden kann und diese damit ein protokollspezifisches Verhalten bekommt.

Mote Sortierung verbessern. Motes können nach verschiedenen Kriterien sortiert werden. Der kommunikation Sortier Algorithmus ist sinnvoll, jedoch ist seine Visualisierung noch nicht optimal. Zudem wäre es auch analog zur Similar Methode vorstellbar, dass ein Mote sortier Algorithmus von einem Netzwerkprotokoll abhängig ist und vom Protokollentwickler implementierbar ist.

Verhalten während dem Recording optimieren. Marker können nicht einwandfrei gedragged werden. Des Weiteren kann die Grösse des Selektors nicht angepasst werden.

Bookmarks als Zustand speichern. Statt bei den Bookmarks nur den Zeitpunkt zu speichern den ganzen Zustand speichern. Das heisst Position, Zoomfaktor, markiertes Packet, Verbindungen, etc. werden als Zustand im Bookmark gespeichert und wieder hergestellt, wenn man zu diesem Bookmark springt.

Korrelation zwischen verschiedenen Paketen. Für ein Protokoll sollte man definieren können, welche Pakete beispielsweise bei einem Verbindungsaufbau zusammengehören und diese geeignet

visuell darstellen. Der einfachste Fall der Korrelation von Paketen ist das Zusammenfügen von fragmentierten Paketen.

8.1 Weitere Anwendungsbereiche

Ein Sensornetzwerk kann man mit zahlreichen Prozessen, auf denen verteilte Software läuft vergleichen. Darum kann das von uns entwickelte Eclipse Plugin als Grundlage dienen, um eine Software zu schreiben, welche das Debugging von Multithread-Anwendungen erleichtert. Die Visualisierung von einzelnen Threads, bei denen die Kommunikation untereinander in einer Timeline angezeigt wird, gibt es für Eclipse nicht. Deshalb wäre unser Eclipse Plugin ein guter Startpunkt.

9 Lessons Learned

Wenn wir in einem nächsten Projekt wieder ein GUI entwickeln müssten, würden wir genauer abklären, ob es brauchbare Tools gibt, welche das Testen des GUI übernehmen. Erstens ging für das Testen des GUIs viel Zeit verloren und zweitens wurden nicht immer alle Fälle berücksichtigt.

9.1 Erfahrungsbericht Amon Grünbaum

Zufrieden schaue ich auf ein intensives und lehrreiches Semester zurück. Wir konnten ein wirklich spannendes Thema bearbeiten, bei welchem es nur so von interessanten Herausforderungen wimmelte. Ich habe sehr viel über Sensornetzwerke gelernt und bin gespannt was mit dieser Technologie in Zukunft alles noch erreicht wird. Das Einarbeiten in Eclipse war anfangs sehr mühsam, da nicht alles auf den ersten Blick gut verständlich ist. Je länger ich mich aber mit Eclipse beschäftigt habe, desto sinnvoller finde ich die Art und Weise wie die Erweiterbarkeit mit den Extension Points und dem Pluginmechanismus gelöst ist.

Sehr angenehm war auch die Arbeit in der IBM Research. Wir waren in einem sehr internationalen Umfeld von Anfang an gut integriert und wurden immer gefordert aber auch gefördert. Bei Problemen nahm sich immer jemand Zeit und versuchte, gemeinsam mit uns, eine Lösung zu erarbeiten.

Die Teamarbeit mit Patrick Dünser klappte ebenfalls bestens. Obwohl jeder einen klar definierten Bereich hatte, für den er verantwortlich war, haben wir uns stets unterstützt und alle Entscheidungen gemeinsam besprochen.

Die Struktur mit wöchentlichen Meetings mit Prof. Oliver Augenstein war gut. So konnten wir ihn immer über den aktuellen Stand unserer Arbeit informieren und aktuelle Probleme besprechen. Wir konnten sehr viel von der Erfahrung von Oliver profitieren, da er alle unsere Fragen sehr genau und überlegt beantwortete.

Mit unserem Resultat bin ich sehr zufrieden. Ich finde, wir haben ein umfangreiches Programm von hoher Qualität erzeugt. Auch die ersten Rückmeldungen von Benutzern und von den Mote Runner Entwicklern zeigen, dass wir etwas Brauchbares erzeugt haben. Besonders stolz bin ich auf die einfache Erweiterbarkeit. Dadurch sind dem Programm keine engen Grenzen gesetzt, da es die Benutzer für ihre spezifische Anwendung erweitern können.

Ich möchte mich bei allen Beteiligten für die spannende, lehrreiche und angenehme Zusammenarbeit bedanken.

9.2 Erfahrungsbericht Patrick Dünser

Eines vorweg: Es war ein intensives, anstrengendes aber auch spannendes und lehrreiches Semester. Die Arbeit hat sehr viel Spass gemacht, darum war ich die ganze Zeit motiviert, auch wenn es ab und zu einmal nicht so ganz funktionierte, wie ich es gerne wollte. Die GUI-Programmierung trieb mich so manches Mal zur Verzweiflung. Ich schätze mich glücklich, dass wir unsere Bachelorarbeit bei einer so renommierten Firma wie der IBM Research machen durften. Ich habe sehr viel gelernt, nicht nur in informatik-technischer Hinsicht, sondern auch persönlich.

Es ist schön zu sehen, dass unsere Arbeit bei der täglichen Arbeit mit Mote Runner genutzt und geschätzt wird. Dies ist ein Beweis dafür, dass wir eine gute, brauchbare Arbeit geleistet haben. Nun hoffe ich, dass unser Plugin beim nächsten Release bereits Bestandteil von Mote Runner ist.

Die Teamarbeit mit Amon Grünbaum war sehr gut. Wir verstanden uns während des ganzen Projekts super. Die Arbeitsaufteilung klappte einwandfrei und wir ergänzten uns sehr gut. Es war sicherlich von Vorteil, dass wir bereits die Semesterarbeit zusammen gemacht haben. So wussten wir, wie der andere arbeitet und konnten unsere Stärken voll ausschöpfen. Die Meetings mit Prof. Oliver Augenstein waren immer sehr produktiv. Er lehrte uns, wie wir uns gegenüber einem Kunden verhalten und das wir öfters eine Kundensicht auf unsere Software und Dokumentation einnehmen sollten. Auch die Zusammenarbeit mit dem Mote Runner Team war lustig und lehrreich. Das IBM-Team war immer sehr hilfsbereit, aber auch sehr kritisch. So waren wir ständig gefordert, eine saubere und durchdachte Arbeit zu präsentieren und schliesslich abzugeben.

A.2 Plugins

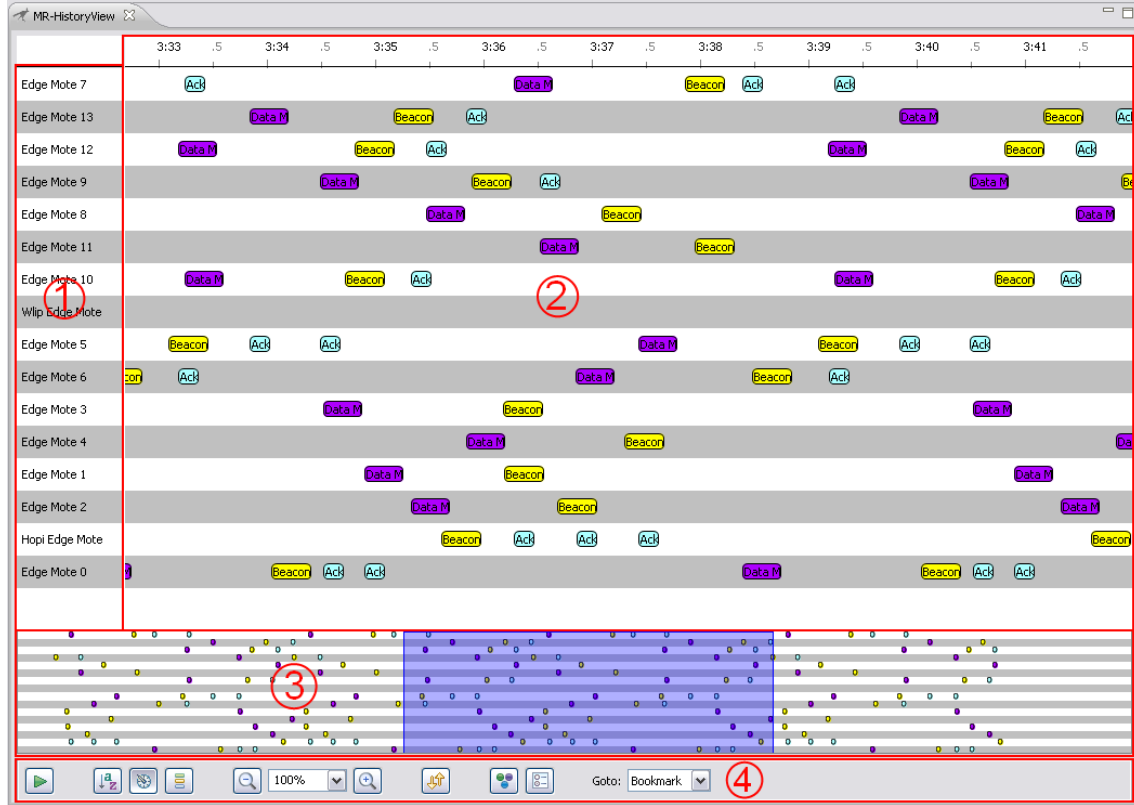


Figure 21: MR-HistoryView

A.2.1 MR-HistoryView

This is the main view and shown in the screenshot above. On the left side all motes are listed which are executed in the Mote Runner-simulation. The top panel shows the timeline which expands every second by default. The packet container (see 2. Packet Container) in the middle shows all sent packets at the given time. At the bottom is an overview of the packet container. It shows every sent packet and it can be used to scroll the packet container or jump directly to a specific place/time. At the bottom of the overview is a button toolbar (see 3. Zoom Container).

1. Mote List

All motes appear in the mote list. By default the sort order is as the motes have been created. If the mouse is over a mote, a tooltip shows further information about the mote. By clicking on the mote, it is selected. By clicking on another mote, more than one mote can be selected. By clicking a second time on a selected mote, it is deselected. If the mouse is over a mote, it will be emphasized. Both of these events are also shown on the Heat Map (Chapter A.2.2). It is possible to drag & drop the motes to list them in a different order.

2. Packet Container

All sent packets are drawn here at a given time. The container expands every second. If the mouse pointer is moved on the timeline, a marker is shown, which helps to orientate on the timeline. With a mouse click on the timeline the marker is drawn. If the marker is closed-to a packet, the marker will attach to it. With the CTRL key this behaviour can be suppressed. With a second marker, the gap between the two markers will be shown in seconds (figure 22a). With clicking on a marker with SHIFT, a dialog box will open and you can add a marker as a bookmark. If you click on a packet two things happen. First, a connection with a red dot to all receiving motes will be

shown. The red dot symbolizes the receiving time (figure 22b). Secondly, the Frame Detail view will show some further information about the packet content. If you press the modifier key **SHIFT** and click on a packet, a dialog box will open which offers some options. The different options are explained in the paragraph Dialog Box below.

There are many options to scroll the packet container. One option for horizontal scrolling is to press **CTRL** and use the scrollwheel on the mouse. To scroll vertically press the **ALT** key and use the scrollwheel on the mouse. Another option to scroll in both directions is to press the mouse and drag the container. The last option is to use the selector in the zoom container (see 3. Zoom Container).

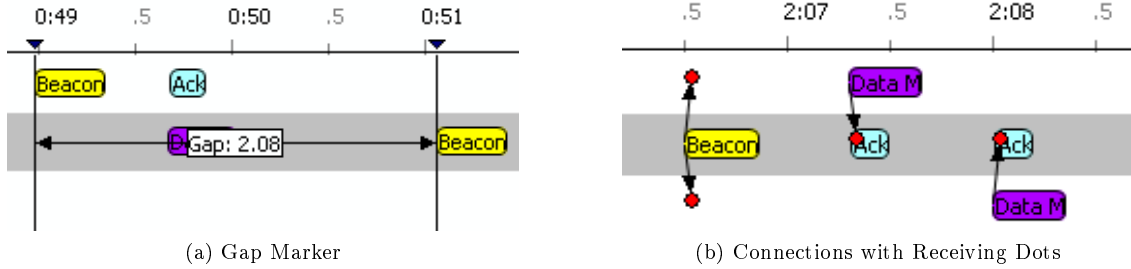


Figure 22: MR-HistoryView tools

Dialog Box

- In the drift tab (figure 24a) the expected period time between two packets can be set. Also a drift tolerance can be set. After pressing OK, it shows the difference between the specified period and the current period. This happens for all similar packets on a mote. If the difference equals zero or is within the specified tolerance the box is green otherwise it is red.



Figure 23: Period Marker

- In the bookmark tab (figure 24b) a bookmark for a time marker or packet is set. The bookmark requires a unique name. The added bookmark is shown in the combo box at the bottom. If the bookmark already exists it can be removed here.
- In the communication tab (figure 24c) two bookmarks can be chosen between which the communication is shown.

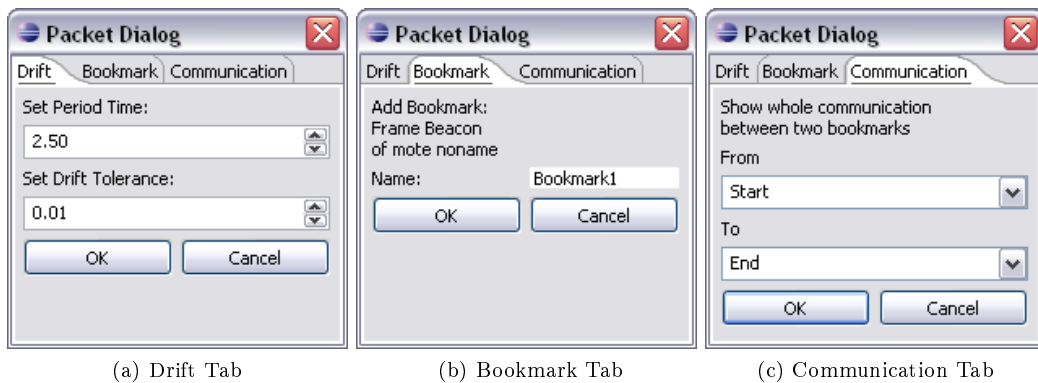










Figure 24: Dialog Box

3. Zoom Container

The zoom container shows a miniature of the packet container. To resize the zoom container, the mouse has to be top of the container. The blue rectangle, called "selector", represents the visible part of the packet container. It corresponds with the current visible part in the packet container. The selector can be used to jump to a certain point by clicking on the mouse. By dragging the selector the packet container is scrolled. If SHIFT is pressed the selector is fixed either horizontally or vertically. It depends on the first mouse movement after pressing SHIFT. Now the scrollwheel can be used to zoom in and out. It is also possible to use the arrow keys, Home/End and PageUp/PageDown to scroll the packet container. The zoom level can be set by resizing the selector at the left or the right side.

4. Bottom Container

Explanation of the buttons from the bottom container.

| | |
|---|---|
|   | Starts / stops the recording |
|  | <p>Mote sort orders:</p> <ul style="list-style-type: none"> • Alphabetical Order • Creation Order (default) • Recent Communication Order. <p>The "recent communication order" reacts to pressing SHIFT and clicking on a mote. This sorts the motes as follows: The motes which sent to the selected motes more packets than they receive from are on top of the selected mote. The motes which receive more packets than they sent to are underneath of the selected mote. The motes which did not communicate with the selected mote are at the bottom</p> |
|  <input data-bbox="355 1272 467 1317" type="text" value="100%"/>  | The buttons allow for zooming in and out. But zoom factor can be selected in the combo box. The zoom factor can also be typed into the combo box. For another options how to zoom see 2. Packet container and 3. Zoom Container. |
|  | If this button is selected the whole communication between all motes is shown in the packet container. (Attention: The view will be slowed down) |
|   | These two buttons change the preferences of the protocol. The first button shows the color preference page. The second shows the whole protocol preference dialog box. For further information see chapter A.2.4 |
| <input data-bbox="316 1709 523 1753" type="text" value="Goto: Bookmark"/> | All bookmarks are listed in this combo box. Picking one leads to the packet container jumping to this bookmark. The "Start" and "End" bookmark are listed by default. See 2. Packet container, on how to define bookmarks. |

A.2.2 MR-HeatMap

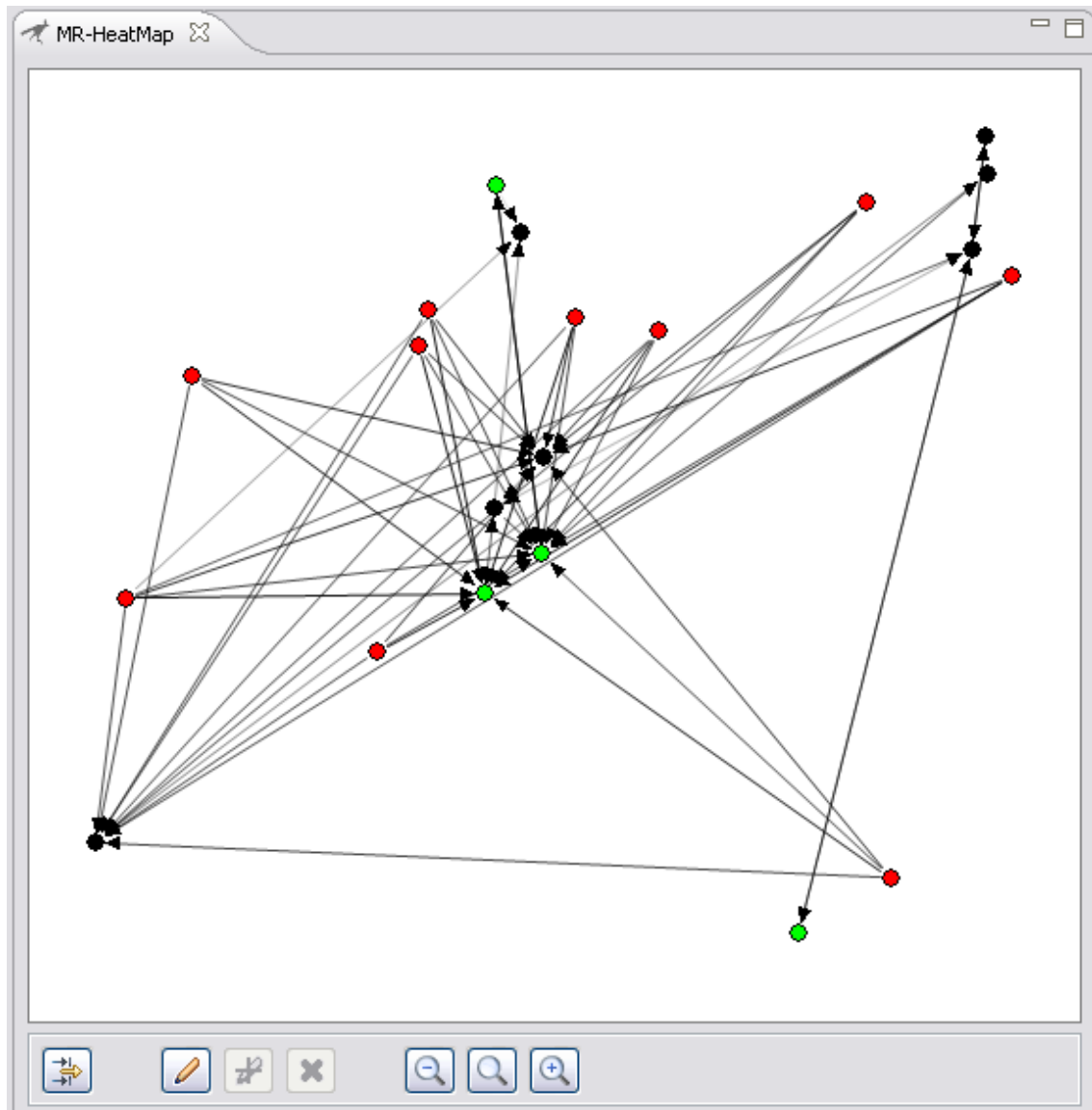





Figure 25: MR-HeatMap

The HeatMap shows the position of the mote based on their geographical position. Each mote is represented by a dot. Furthermore, it shows the communication between the motes which are indicated by arrows. The arrows fade out if no new packet is sent. Additionally the dots turns green if a mote sends data and turns red if it receives data. If the mouse is over a mote, then the same tooltip will be shown as the HistoryView does. Also, selecting and emphasizing motes works as in the HistoryView.

| | |
|---|--|
|  | If the filter button is pressed, all selected notes are shown in the HistoryView. The filter will be updated regardless of selecting or deselecting notes. |
|  | With the first button it is possible to mark one or more notes at once by drawing a line around the desired notes. The second button deselects one or more notes. The third button deselects all selected notes. |
|  | These buttons allow for zooming in and out. The middle one adjusts the view so that all notes are shown on the view. |

A.2.3 MR-FilterDetail

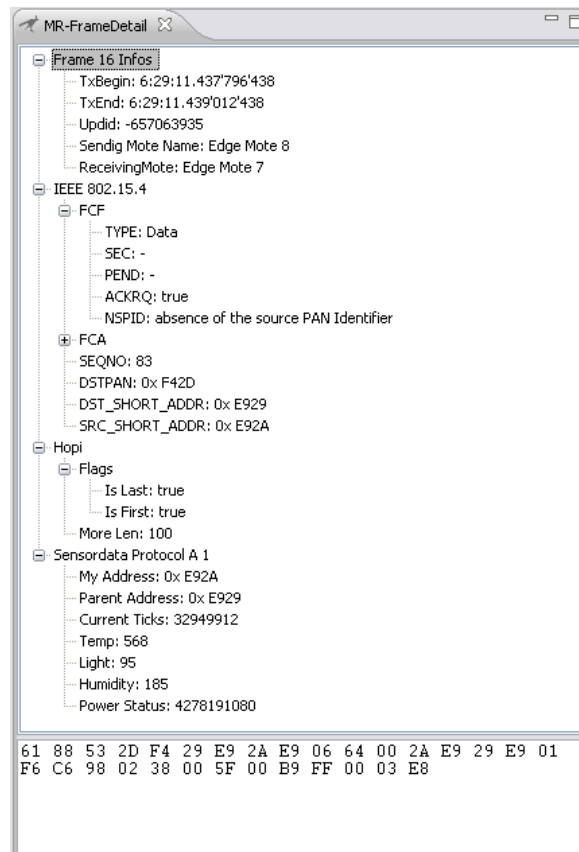


Figure 26: MR-FrameDetail

This view details the content of the selected packet. The first node shows general information about the packet. All other nodes represent the information of a protocol in the protocol stack which has sent or received the selected packet. At the bottom a window shows the whole content of the packet as hexadecimal values. If you select a node the regarding hex values will be highlighted. If the hex values are not precise enough, it shows a hex value as binary and highlights the bit(s).

A.2.4 Configuration

The HistoryView configuration panels are used to change different settings of the analyzed protocols and the protocol tree. The configuration panels are selected by the button in the HistoryView or

by opening the general Eclipse preferences at “Window -> Preferences -> Mote Runner Debugger -> Mote Runner History View”.

External Dissector Configuration For every protocol to be analyzed the HistoryView needs a dissector that parses the content of the protocol. Dissectors are already available for the most common protocols. They are stored in the global dissector cache (moterunner/lib/java-dissectors). A dissector JAR file for a new dissector can be added the path to your dissector in this list (figure 27).

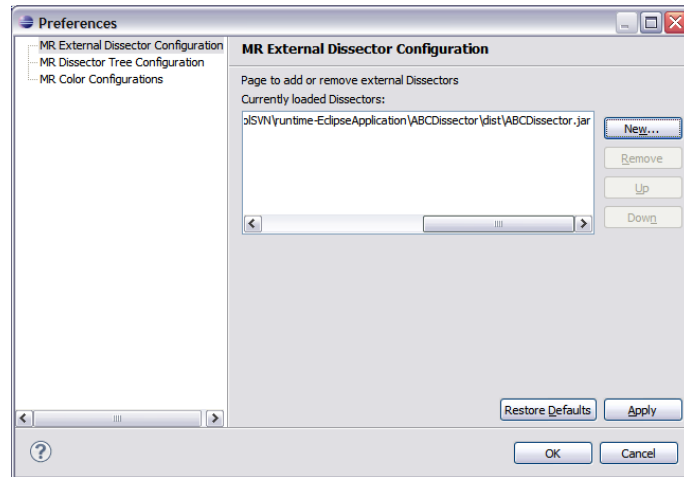


Figure 27: Configuration window to choose external dissectors

Process of the protocol dissection To analyze protocol data the eclipse plugin contains a dissection tree. Every radiomessage is passed to the root of the tree and the message is analyzed by its dissectors. Each dissector analyzes the message data and decides if the given message is dissected by him. An example (see figure 28): The HistoryView receives a "Hopi Data A" message.

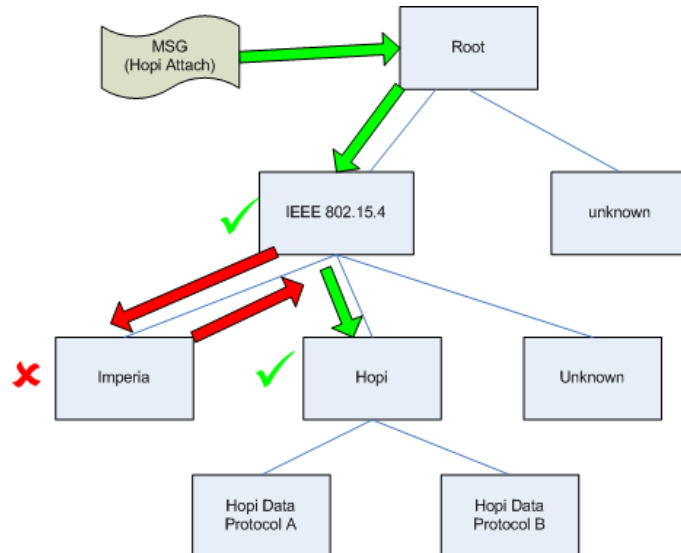


Figure 28: Example of a dissection

The raw Data message is passed to the root dissector which sends it to his first child, the IEEE 802.15.4 (MAC) dissector. This dissector parses the given message. As the message starts with a valid mac header the MAC dissector marks the header bytes as dissected and "if there are bytes

left" presents the message to his first child dissector, in our example the Imperia dissector. The imperia dissector then checks the remaining bytes and as they don't have the right structure he returns false. The MAC header then sends the message to the next child. The hopi dissector finds out that it is a hopi package and as the end of the message is reached he finishes the dissection. If no dissector can parse the message correctly, a default dissector consumes the rest of the message.

Dissection Tree Configuration In the dissection tree window different protocol dissectors can be enabled or disabled. As the dissection traverses the tree always in pre-order, it is required that the order of the tree can be changed. Using drag & drop the order of the dissectors at the same hierarchy level can be changed. The new protocol tree is active immediately after the return to the HistoryView.

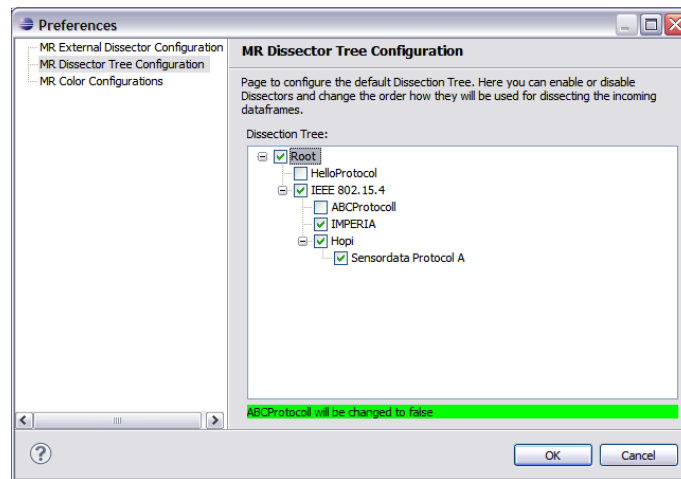


Figure 29: Configuration window to change the dissection tree

Color Configuration To better differentiate messages of different protocols in the HistoryView different colors are supported. Colors can be changed in the color configuration window.

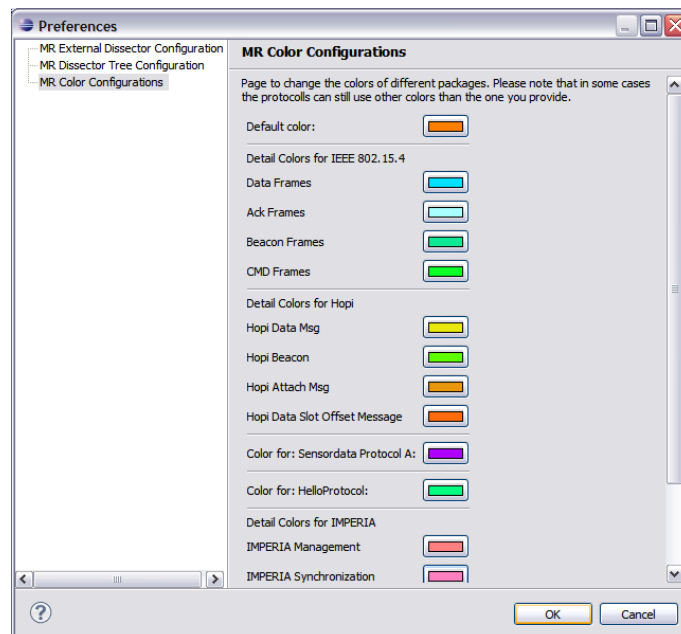


Figure 30: Configuration window to change colors of dissectors

B Tutorial: „How to write a dissector”

B.1 General Information

This document gives an overview how a dissector (=parser) can be implemented for a protocol and how this dissector can be used in the Mote Runner debug environment. The MR-HistoryView shows all radio frames that are transmitted in the connected sonoran simulation. The content of these radio frames is analyzed and parsed by dissectors. If there is no parser available for a protocol a dissector can be written to analyze your protocol content and view it in the HistoryView.

B.2 Tutorial: “Hello World” protocol dissector

This tutorial presents a simple HelloWorld dissector. The dissector checks if a radiomessage is 7 Bytes long and if it is, it parses this 7 Bytes and displays the results in the HistoryView. The "Hello" protocol has the following fields:

| | | |
|-------|-------|-------|
| 0-1 | 2 | 3 - 6 |
| Start | SEQNO | Rest |

To test the dissector the radiocount-java sample from the Radiocount `moterunner/doc/system/index.html#tutorials_tutRadio_0` tutorial can be used. The notes in this example exchange messages that adhere to this protocol.

B.2.1 Create the dissector project using the wizard

Start the Mote Runner Eclipse and create a new Java Project using the “Mote Runner Dissector Project Wizard” (figure 31).

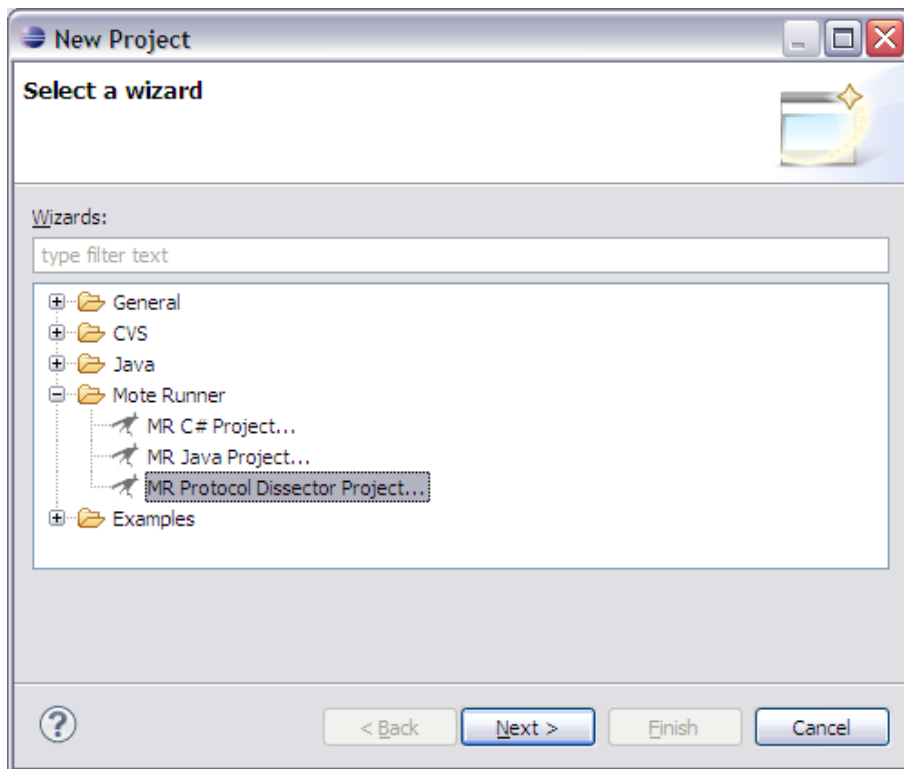


Figure 31: Select the MR Protocol Dissector Project

On page two a project name must be chosen. On the third page nothing has to be changed. On the fourth page some information about the dissector can be specified which is used to create the skeleton of the dissector. This tutorial expects an empty skeleton. The field `ParentDissector` specifies the dissector the new dissector runs after. As the sample dissector dissects the complete frame it runs after the `RootDissector`. The `RootDissector` is the root of the dissection tree and executes the `runDissection()` method of his childs. (See figure 32 for all settings we used in this tutorial). After that the wizard can be finished.

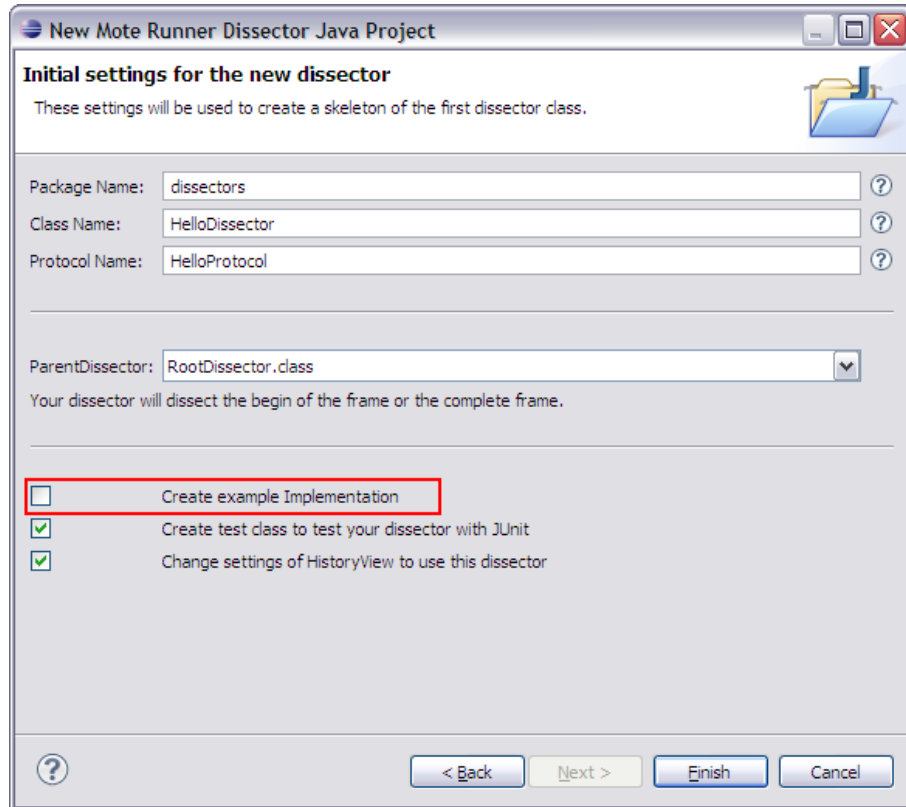


Figure 32: Configure initial Settings for the dissector

The project is now generated and the output JAR file is added to the settings of the MR-HistoryView. These settings popup and allow to enable the use of the new dissector. When a Simulation is started the dissection with the new protocol dissector is active. (See figure 33) It is important that the `HelloProtocol` dissector runs before the `IEEE 802.15.4 MAC Header` dissector. Otherwise the packets will be consumed by the `MacDissector` and the Hello dissection results cannot be seen.

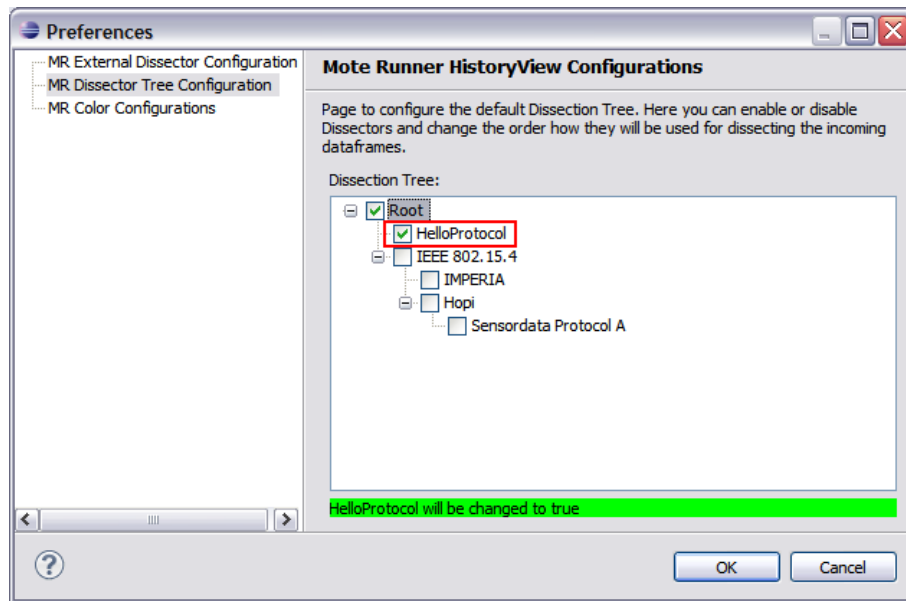


Figure 33: Configure the new dissection tree

The project generated a Java source file for your dissector. It also contains a helper library for compiling and testing the dissector. An ant build file is included which helps to build the JAR file of the dissector.

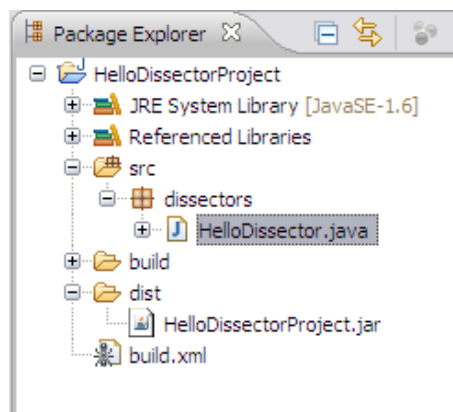


Figure 34: Structure of the generated Project

B.2.2 Implement the dissector

Open the created `HelloDissector.java` file. The first method to implement is the `initializeKeys` method. All the results a dissector creates are stored in a map in the `ContextFrame`. Every result in that map needs a unique key. As dissectors may occur in different parts of the dissection tree a `static final` keyword is not sufficient. It is therefore best practise to declare the keys as a `public String` variable and to initialize your keys by concat the `dissectorKey` with a text that represents the current result in the `initializeKeys` method. Note: The `initializeKeys` method saves execution time as the String concatenation is done once and not during the dissection.

The field `dissectorKey` is generated by the base class and contains the protocol name and all the protocol names of the parent dissectors.

The Hello dissector declares and initializes the three keys `StartData`, `SeqNr` and `RestData`.

```
public String keyStartData;  
public String keySeqNr;  
public String keyRestData;  
  
@Override  
protected void initializeKeys() {  
    keyStartData = dissectorKey+".StartData";  
    keySeqNr = dissectorKey+".SeqNr";  
    keyRestData = dissectorKey+".RestData";  
}
```

Figure 35: Code initializeKeys method

Now the real dissection in the `dissection` method is implemented. The method take a parameter of type `ContextFrame`. The context frame contains the bytes and all the results of the previous dissections. The steps taken by the dissector are:

- First check if the frame is 7 Bytes long. If not, this can not be a Hello frame and the method returns `false`!
- Get the first two bytes of the frame as an Integer. Hello frames are big endian so the “be” version of the method to get the data from the context frame is used. An offset must be provided to the frame which is relative to the first byte dissected with this dissector. At the end of this dissection, the field `dissectedBytesBegin` of the frame must be set to the new value so that child dissectors can start dissecting their part of the frame with an offset of 0.
- Save the dissected result in the result map of the frame. Provide a title, a value and how the result is represented. Add the offset and the dissected length of the result so the hex value is highlighted in the MR-FrameDetail.
- The two steps above have to be done three times for the different dissections.
- After the results have been added to the frame some general information is stored in the frame. First set the field `dissectecBytesBegin` to the new value. Then set the color of the frame to the one defined in the constructor. Any color can be set. But if storred in variable `color`, the user is able to change the color in the settings dialog. The color information can be overridden by any child dissector running after this dissector
- Set an info text on the frame. This info text is displayed in the HistoryView. Depending on the Zoom Level the short or the normal version or none of them will be displayed. Also the info text can be overridden by any child dissector.
- Return `true` to inform the base class that the dissection was successfull. If an exception is thrown at any time the dissection was probably not successfull and we return false.

```
@Override
public boolean dissection(ContextFrame f) {
    int offset=0;
    try {
        if(f.getLength()!=7){
            return false;
        }
        int start = f.getU2be(offset);
        f.setResult(keyStartData, "First_two_Bytes", start, offset, 2,
            ResultNode.REPRESENTATION_HEX);
        offset += 2;

        int seqNr = f.getU1(offset);
        f.setResult(keySeqNr, "Seq_Nr.", seqNr, offset, 1,
            ResultNode.REPRESENTATION_NR);
        offset += 1;

        long rest = f.getU4be(offset);
        f.setResult(keyRestData, "Rest_Data", rest, offset, 4,
            ResultNode.REPRESENTATION_HEX);
        offset += 4;

        //add the new dissected bytes to the contextframe
        f.dissectedBytesBegin+=offset;
        //set my color to the contextframe
        f.setRGB(color);
        //set my info
        f.setInfo("Hello_" + seqNr, String.valueOf(seqNr));
        return true;
    } catch (Exception e) {
        // an exception can be thrown when the end of the package
        // was reached
        return false;
    }
}
```

Figure 36: Code dissection method

B.2.3 Export the new dissector and configure the dissector tree.

To activate the dissector, it must be rebuild so that the MR-HistoryView takes the actual .JAR file. For that simply select the build.xml file and click right -> Run As -> 1 Ant Build (see figure 37).

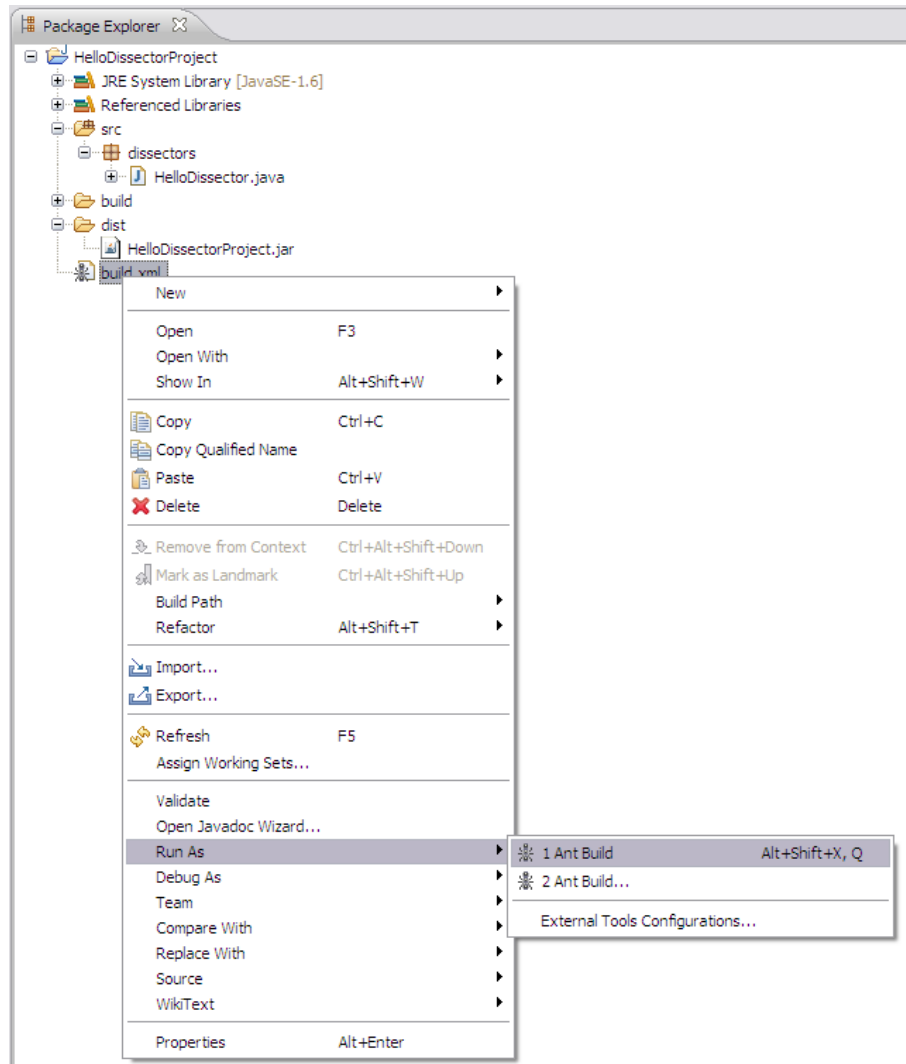


Figure 37: Export the dissector project as a JAR File using the Ant Buildfile

Figure 38 shows the MR-HistoryView and the MR-FrameDetail for the “radiocount-java” example running in the simulation. The MR-FrameDetail shows the packet data as dissected by the Hello-dissector.

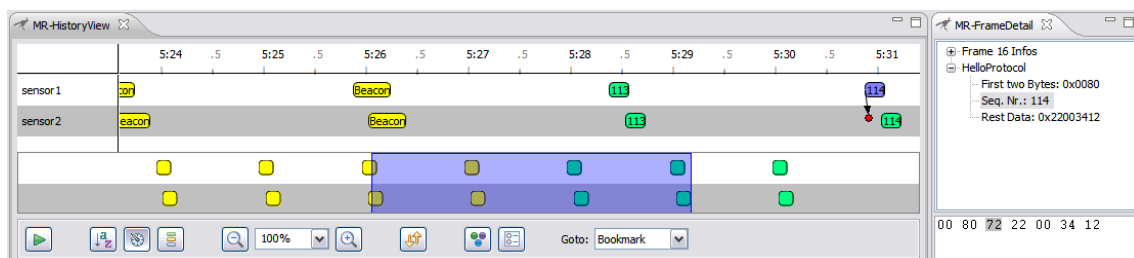


Figure 38: The HistoryView with a radiocount example and the HelloDissector

B.3 Different more complex caseses of dissectors

B.3.1 How to test a dissector

The easiest way to test a dissector is to enable the “Create test class...” option when creating the project with the wizard. This creates a test folder and a sample Class that shows how to test the

dissector. Remove the comments of the Sample class (Select the code and press “Ctrl + 7”) and add JUnit 4 to your build path to use the generated skeleton to test the dissector.

The `initializeDissectionTree()` method creates the dissection tree. As the `DissectorManager` is not available in the test method the dissection tree must be created manually.

In all the test methods create a `ContextFrame` and pass the data to dissect to it as a `String` (`HexRepresentation`) or as a `Byte Array`. Then call the `dissectionChain()` method of the root dissector and pass the `ContextFrame` to it. If the tree is correct the root dissector passes the frame to the dissector and the dissection method is called. Figure 39 shows a sample test method that checks if there is a sequence number `ResultNode` after the dissection has been performed.

```
@Test
public void testFrameDissection() {
    // Create a frame with the bytecode you want to dissect
    ContextFrame f = new ContextFrame("00805D22003412");

    // run the dissectionChain on that frame
    root.dissectionChain(f);

    // Print out the full result list of the ContextFrame.
    // not needed but helpful to see all dissected results.
    f.printResultsString();

    // Analyze results:
    ResultNode res1 = f.getResult(helloDissector.keySeqNr);
    assertNotNull("keySeqNr_should_be_dissected", res1);
    assertEquals("keySeqNr_of_this_frame_should_be_93", 93,
        res1.getLongValue());
}
```

Figure 39: Code test method

B.3.2 Different result types for the results of a dissection.

A protocol may contains data of different types. For any data dissected a `ResultNode` is added to the `ContextFrame`. The GUI then displays all results of a dissection in the MR-FrameDetail view when a packet is selected (see figure 40).

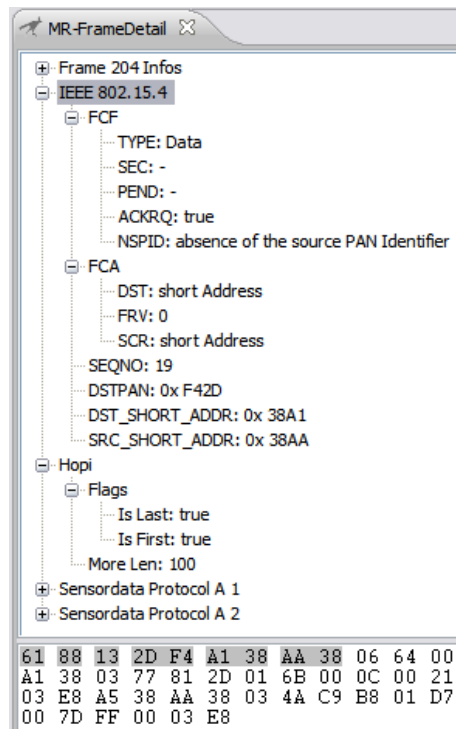


Figure 40: FrameDetails View: Results are displayed in different types

A `ResultNode` can specify how the number is displayed to the user. The following list contains the different result representations:

| Code: <code>ResultNode.REPRESENTATION...</code> | Display of 42 | Description |
|---|-----------------|--|
| <code>_NR</code> | "42" | Just returns the long value as string |
| <code>_HEX</code> | "0x2A" | Returns the hex value of the long. If the <code>ResultNode</code> represents 2 Bytes "0x002A" is displayed |
| <code>_BIN</code> | "0b00101010" | Returns the bin value of the long. If the <code>ResultNode</code> represents only 6 bit "0b101010" is displayed |
| <code>_NS</code> | "0.000'000'042" | Creates a <code>NanoTime</code> object with the given long and returns the <code>toString()</code> method of the <code>NanoTime</code> class. |
| <code>_DICTIONARY</code> | "Magic Number" | This is useful for flags. By calling <code>ResultNode.getDictioanry().put("fieldName", 42, "Magic Number");</code> you can add the value of a flag to the dictionary. If a <code>ResultNode</code> has this value the string from the dictionary is taken. |
| <code>_BOOLEAN</code> | "true" | Returns "false" when the result has the long value 0, otherwise it returns "true". |
| <code>_HIDDEN</code> | - | Results of the type hidden are not displayed in the GUI. |

| | | |
|--------------------|------------|--|
| <code>_OWN</code> | ex. “420” | The <code>IResultRepresentation</code> interface can be implemented which receives the long value that is stored in the <code>ResultNode</code> and returns a string containing anything. For example it can return the given long value multiplied by 10. |
| <code>_TEXT</code> | “42 magic” | can store your data also directly as a text in the result instead of a long value. This is the easiest way but needs the most storage and is more complicate for child dissectors if they want to check what this result contains. |

Table 3: List of all result representations

B.3.3 How to define more then one color for a dissector

If the dissector parses different subtypes of your protocol the user may change the color for every subtype individually. For that the `IProvideColorOptionsToUser` interface have to be implemnted. A good example for that is the `MacDissector` which provides the possibility to change the color for Beacon, Data, Ack and CMD messages.

The only method which must be implemented is the `provideColorOptionsToUser()` method. This method returns an array of color options (id,title and default color). The `MacDissector` implements this method as followed:

```
private static RGB colorAck = new RGB(170, 255, 255);
private static RGB colorData = new RGB(0, 224, 255);
private static RGB colorBeacon = new RGB(12, 232, 148);
private static RGB colorCMD = new RGB(13, 255, 39);

@Override
public ArrayList<ColorOption> provideColorOptionsToUser() {
    ArrayList<ColorOption> res = new ArrayList<ColorOption>();
    res.add(new ColorOption(
        Integer.toString(FCF_DATA), "Data_Frames", colorData));
    res.add(new ColorOption(
        Integer.toString(FCF_ACK), "Ack_Frames", colorAck));
    res.add(new ColorOption(
        Integer.toString(FCF_BEACON), "Beacon_Frames", colorBeacon));
    res.add(new ColorOption(
        Integer.toString(FCF_CMD), "CMD_Frames", colorCMD));
    return res;
}
```

Figure 41: `provideColorOptionsToUser` method of the `MacDissector`

The GUI now provides a selection dialog for each `ColorOption` returned.

The following code shows how the dissector gather the user-defined colors and overwrites his default colors with the configured ones.


```
private void initColors() {
    colorData=getCorrectColorFromStore(colorData,
        FilterVar.getVarForColorType(this,
            Integer.toString(FCF_DATA)));
    colorBeacon=getCorrectColorFromStore(colorBeacon,
        FilterVar.getVarForColorType(this,
            Integer.toString(FCF_BEACON)));
    colorCMD=getCorrectColorFromStore(colorCMD,
        FilterVar.getVarForColorType(this,
            Integer.toString(FCF_CMD)));
    colorAck=getCorrectColorFromStore(colorAck,
        FilterVar.getVarForColorType(this,
            Integer.toString(FCF_ACK)));
}
```

Figure 42: initialize your colors with the colors from the preference store

B.3.4 Change the way how children will run inside of your dissector

In case of a protocol where there are fixed data blocks inside of a package it might be necessary to have a dissector for each datablock in the package. In that case it is possible that the same dissector for a datablock can run twice at the same level. The unique dissector keys for a datablock are then not unique anymore in case the default strategie for defining dissector keys was used. For cases like this you have to overwrite the `runChildDissectors()` method. A good example is the Hopi Protocol implementation. (See Javadoc)

B.3.5 Configure the dissectors tree

Open the Eclipse Configuration “Window -> Preferences -> Mote Runner Debugger -> Mote Runner History View” and go to the first page “External Dissectors” to add or remove dissectors (Figure 43).

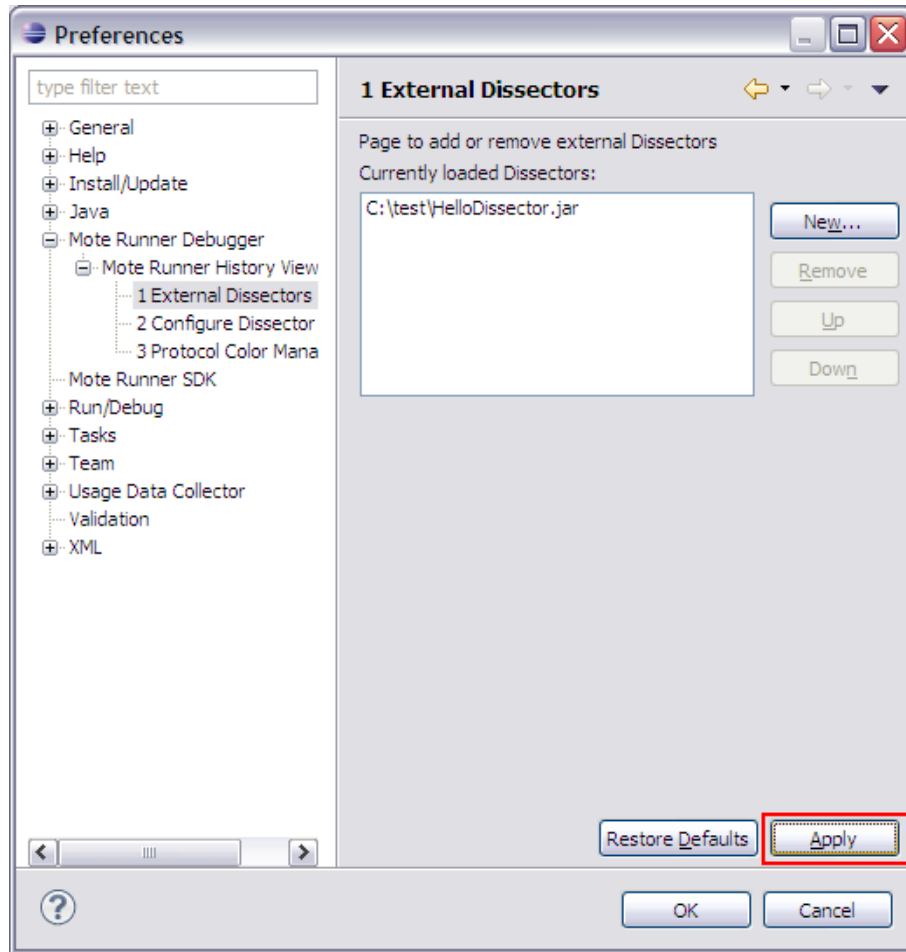


Figure 43: Load the new dissector

Next one can switch to “Configure Dissector Tree” to enable or disable the available dissectors. The order of dissectors can be modified using drag & drop. In case of the Hello-dissector it is important that it is on the top of the list, before the "IEEE 802.15.4" dissector.

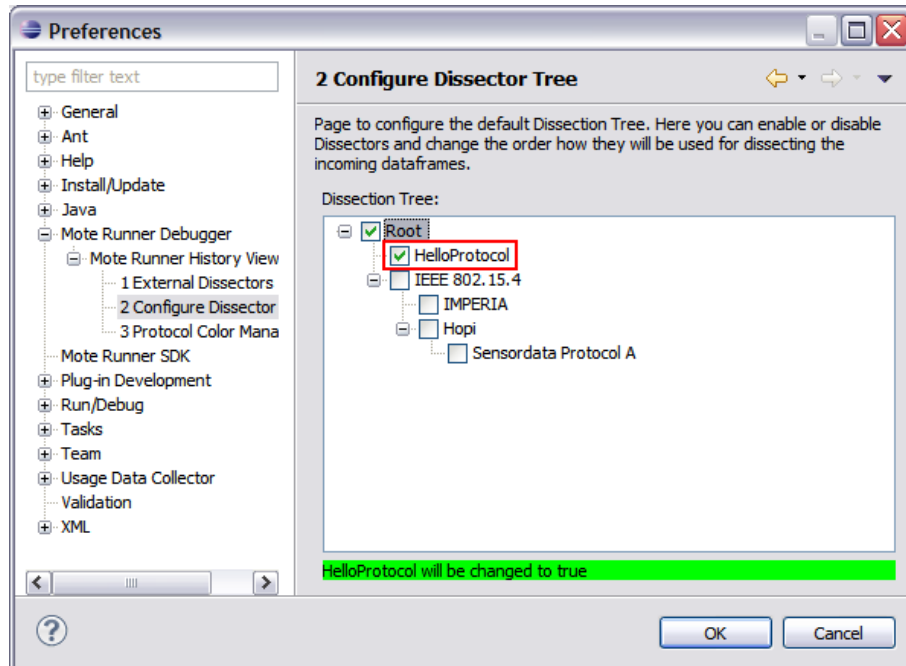


Figure 44: Configure the new dissection tree

B.4 Troubleshooting Ant build

If the `JAVA_HOME` environment variable does not point to the JDK Folder or is not available ANT builds are not possible.

By setting `JAVA_HOME` it should work again.

It is also possible to manually add the needed `tools.jar` file to the ant settings of Eclipse. The file is in the `JDK/lib` folder and must be added to “Window -> Preferences -> Ant -> Runtime -> Classpath -> Ant Home Entries -> Add External JARs”.

B.5 Where do I find Additional Informations

B.5.1 Source code

The source code of these examples can be found on the CD in the folder `06_Dissector_Tutorial_Files`.

B.5.2 Javadoc

JavaDoc for the `FrameFilter` Plugin is available on the CD in the folder `03_JavaDoc`.

C Installation der Software

C.1 Installation Mote Runner

Wenn Sie unser Plugin testen wollen, müssen Sie das komplette Mote Runner Software Paket in der Version Beta 4.1 oder höher installieren. Die Version 4.1 ist auf der beigelegten CD im Ordner 04_MoteRunnerSW als ZIP File vorhanden.

- Entpacken Sie das ZIP File und starten Sie die Setup Datei
- Installieren Sie Mote Runner gemäss den Informationen auf dem Bildschirm
- Nach der Installation muss noch das Mote Runner Plugin im Eclipse installiert werden
 - Sie können dazu ein Eclipse Helios (3.6.2) von der Eclipse Webseite herunterladen. Auf unserer CD ist im Ordner 05_SonstigeSW diese Eclipse Version bereitgestellt (Nur Windows 32bit)
- Installieren Sie das MoteRunner Eclipse Plugin gemäss der offiziellen Mote Runner Installationsanleitung (moterunner/doc/system/index.html#start_startFirstSteps_EclipseInstallation)
- Unsere Teile des Mote Runner Features sind über "Window -> Show View -> Other -> MoteRunner" zu finden oder man öffnet unsere Perspective mit "Window -> Open Perspective -> Other -> MR-Timeline"

C.2 Ausführen eines Testprogrammes

Um unsere View so schnell wie möglich zu testen, gehen Sie am besten wie nachfolgend beschrieben vor:

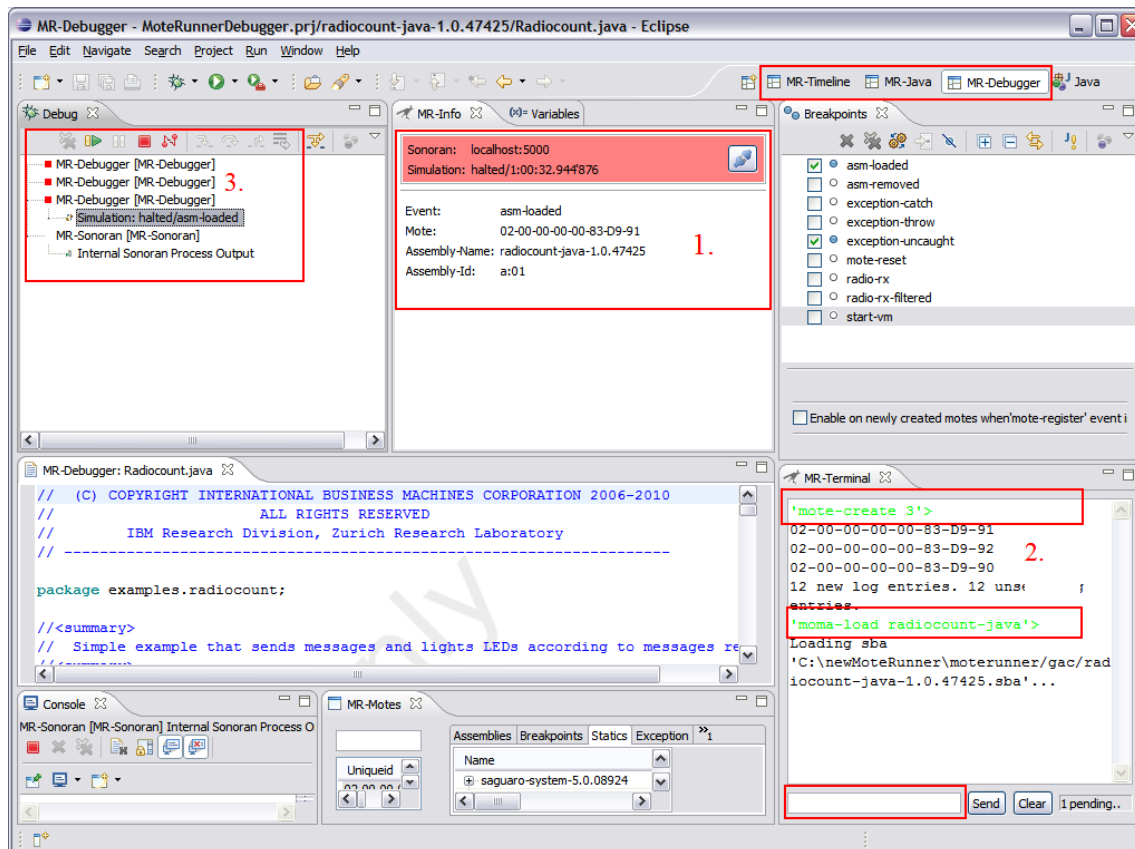


Abbildung 45: Mr-Debugger Perspektive. Starten von einer Simulation

- Öffnen Sie die MR-Debugger Perspektive
- Verbinden Sie Eclipse mit einem Sonoran Prozess (Da noch keiner läuft wird automatisch einer gestartet) (siehe Abbildung 45, Markierung 1)
- Erstellen Sie im MR-Terminal Fenster ein paar Sensoren mit dem Befehl „mote-create X” (siehe Abbildung 45, Markierung 2)
- Laden Sie auf die neu erstellten Motes das Programm Radiocount mit dem Befehl „moma-load radiocount-java” (siehe Abbildung 45, Markierung 2)
 - Dieses simple Programm sendet alle 2.5 Sekunden eine Nachricht an alle Motes in Reichweite. Alle Motes die eine Nachricht erhalten, ändern den Zustand ihrer LED's.
- Momentan ist beim Laden eines Programms ein Breakpoint gesetzt. Später wird dieser konfigurierbar sein, da dies für die MR-Timeline nicht sinnvoll ist. Um das Programm fortzusetzen, muss im Debugger noch „Resume” geklickt werden. (siehe Abbildung 45, Markierung 3)
- Jetzt läuft im Hintergrund die Simulation mit dem Radiocount-Beispiel. Nun kann auf die MR-Timeline Perspektive gewechselt werden und mit „Start Recording” eine Session in unserem Plugin gestartet werden.
- Eine Simulation kann auch über andere Kanäle gestartet werden (aus der Konsole oder dem Dashboard im Browser). Eine so gestartete Simulation muss zum Debuggen in Eclipse verbunden werden (siehe Abbildung 45, Markierung 1). Danach kann man gleich auf die MR-Timeline Perspektive wechseln und die HistoryView starten.
- Das komplexere Hopi Beispiel kann gestartet werden, indem man im Installationsverzeichnis in den Ordner „moterunner\examples\data-gathering” wechselt und dort mit dem Befehl „sonoran-i ./demo.mrsh” die Demo startet. In Eclipse kann man wieder auf die Simulation verbinden (siehe Abbildung 45, Markierung 1) und das Recording in der HistoryView starten.

C.3 Quellcode

Der Quellcode unserer Plugins ist auf der beigelegten CD im Ordner 02_SourceCode zu finden. Unsere zwei Plugins sind abhängig von anderen Mote Runner Plugins (siehe Abbildung 6 im Kapitel 4.1). Leider haben wir keine Erlaubnis bekommen den Quellcode der anderen Plugins weiter zu geben. Deshalb ist es für Externe nicht möglich die Unit Tests laufen zu lassen oder den Code zu compilieren. Für eine Demo des Quellcodes, melden Sie sich am besten bei uns.

D Usability Test - Dissector API

D.0.1 Was wurde getestet

Die funktionsweise des Dissector Interface und wie man selber einen Dissector für sein eigenes Protokoll schreibt.

D.0.2 Wer hat getestet

Es gibt eine Gruppe in der IBM, die Sensornetz Anwendungen erstellt. In diesem Team wird das Imperia Netzwerkprotokoll entwickelt. Jemand aus dem Team wurde angefragt, ob er einen Dissector für das Imperia Protokoll schreiben könnte.

D.0.3 Wie wurde getestet

Wir haben uns zu Beginn kurz getroffen und der Testperson die aktuelle Programmversion übergeben. Danach gaben wir eine kurze Einführung über unser Programm und ein paar Zusatzinformationen zum Erstellen des Wizards. Danach hat die Testperson den Dissector alleine implementiert.

D.0.4 Einschränkungen

Durch das Gespräch am Anfang könnte das Ergebnis leicht verzerrt sein, da die Testperson neben der Dokumentation auch noch eine mündliche Einführung erhalten hat. Dadurch kann nicht komplett sichergestellt werden, dass das Schreiben des Dissectors ohne Hilfe möglich ist.

D.0.5 Resultat

Nach einer Woche haben wir uns wieder getroffen, um zu besprechen, wie die Implementation geklappt hat und was implementiert wurde. Folgende Punkte wurden von der Testperson angesprochen:

- Das Erstellen eines eigenen Dissector hat gut geklappt. Insbesondere das einfache Builden und Integrieren des Dissectors waren für den Tester positiv. Auch die angebotene Möglichkeit eines JUnit-Tests hat ihm sehr gut gefallen. Alles in allem war die Implementation mit dem Interface gut durchführbar.
- Die Testperson hat einen Dissector für ihr Protokoll erstellt, der 3 verschiedene Subtypen enthält. Der Tester hatte nur wenig Zeit, darum wurde nicht das komplette Protokoll implementiert. Wenn sicher ist, dass die MR-Timeline Perspektive in der nächsten Version von Mote Runner verfügbar ist, wird der Dissector noch weiterentwickelt und zu Ende programmiert.
- Unklarheiten bei der Implementation
 - Warum braucht es die initializeKeys() Methode? Warum ist diese zwingend notwendig? -> Der Nutzen ist zu wenig klar und darum wurde die Methode nicht richtig eingesetzt.
 - Bei den ResultRepresentations wurde nicht verstanden, was mit REPRESENTATION_OWN gemeint war und wie diese zu benutzen ist.
 - Das Erstellen von verschiedenen Farben für verschiedene Typen von Paketen ist im Wizard nicht beschrieben, beziehungsweise es wird kein SampleCode dafür erzeugt.
- Positives bei der Implementation
 - Die Implementation des MAC und HopiDissector sind sehr nützlich als Referenz. -> Eventuell sollte darauf noch besser hingewiesen werden.
- Zusätzlich erwünschte features

- Eventuell will man auch ein Result abspeichern, dass dem Benutzer im GUI nicht angezeigt werden soll. Ein ResultType REPRESENTATION_HIDDEN wäre dazu hilfreich für Resultate, die nur für ChildDissectors relevant sind, jedoch nicht für den Benutzer.
- Mit der JUnit-Test-Funktion für eigene Pakete, kam die Idee, dass das DissectorPlugin auch ausserhalb von Eclipse für gewisse Logging Funktionen gut zu gebrauchen wäre. So könnte in Logfiles anstatt dem hex Wert eines Pakets, das aufgeparste Resultat angezeigt werden. Dies ist eigentlich jetzt schon möglich, wobei einige zusätzliche Methoden zum Darstellen der Resultate noch von Vorteil wären.
- Um Fehlerzustände im Protokoll zu dokumentieren und zu vergleichen wäre es sehr nützlich, wenn eine aufgenommene Session abgespeichert und später wieder angezeigt werden könnte.

D.0.6 Auswertung:

Der Wizard und das API sind bereits sinnvoll nutzbar und funktionieren. Hauptsächlich wurde das Tutorial angepasst und die Dokumentation überarbeitet. In Teilen des Codes wurden die Kommentare angepasst. Eine neue REPRESENTATION_HIDDEN wurde, wie gewünscht, implementiert.

E Verzeichnisse

Glossar

| Begriff | Beschreibung |
|-------------------------|---|
| Assembly | Ausführbares Programm. Kann auf einen simulierten oder realen Mote geladen werden. |
| Dissector | Ein Protokoll Parser, dieser Begriff wird auch bei Wireshark für die Parser genutzt (Dissector engl. Zergliederer). |
| Draw2d | Draw2d ist ein Layout und Rendering Werkzeugsatz, welches auf SWT aufbaut. |
| Edge Mote | Verbindungsmote, um eine Verbindung zwischen einem PC und einem Sensornetzwerk herzustellen. Ist über USB mit dem PC verbunden und kann per Radiokommunikation mit den anderen Motes im Netzwerk kommunizieren. |
| Edge Server | Verbindungsprozess auf der Serverseite (Sonoran). Hat eine Verbindung zum Edge Mote und bietet die Möglichkeit über das Internet auf die realen Motes zuzugreifen. |
| Figur / Figure | Eine Figur in Draw2d ist alles, was das Interface IFigure implementiert. Jede Figur ist ein Container für weitere Figuren. |
| Hopi | Multi Hop Sensornetz Protokoll. Sammelt Daten. |
| IEEE 802.15.4 | Wireless Standard für Datenübertragung in wireless Netzwerken. Definiert den MAC Header, der in Mote Runner verwendet wird. |
| Imperia | Data Gathering Protokoll für Sensornetze, welches von IBM Rüschlikon entwickelt wird. |
| JUnit | JUnit ist ein Framework zum Testen von Java-Programmen. |
| LightweightSystem | Ein LightweightSystem assoziiert ein Figure mit einem SWT Canvas. |
| Mote | Ein Mote ist ein Sensorknoten. Mote (engl. Stäubchen) weist auf die Winzigkeit von heutigen Sensoren hin. |
| Saguaro | Der Saguaro Prozess simuliert die Hardware der Motes in der Simulationsumgebung. |
| Sonoran | Der Sonoran Prozess ist das Management-Terminal von Mote Runner (siehe auch Edge Server). |
| Standard Widget Toolkit | Standard Widget Toolkit ist eine Bibliothek für die Erstellung graphischer Oberflächen mit Java. |

Abkürzungsverzeichnis

| Abkürzung | Ausgeschrieben |
|-----------|--|
| GUI | Graphical User Interface |
| HAL | Hardwareabstraktionsschicht (engl. Hardware Abstraction Layer) |
| IDE | Integrated Development Environment |
| JAR | Java ARchive Datei |
| LIP | LIInk Protocol |
| OS | Betriebssystem (engl. Operating System) |
| SWT | Standard Widget Toolkit |
| VM | Virtuelle Maschine |
| WLIP | Wireless LIP (Management Protokol für Wireless Motes) |

Literatur

- [1] Sensornetz allgemeine Beschreibung, Text,
URL: <http://de.wikipedia.org/wiki/Sensornetz>
(Stand 9.6.2011)
- [2] Internet of Things, Video von IBMResearchZurich, veröff. am 7.6.2010
URL: www.youtube.com/watch?v=sfEbMV295Kk
(Stand: 7.6.2011)
- [3] Use of Wireless Sensor Networks, Video von IBMSocialMedia, veröff. am 15.3.2010
URL: www.youtube.com/watch?v=M_7rYvQESsg
(Stand: 7.6.2011)
- [4] Mote Runner Help Dokumentation, im Installationsverzeichnis:
`moterunner/doc/system/index.html#start_startBirdsEye_0`
(Stand: Beta 4.1 14.06.2011)
- [5] Draw2d Einführung
URL: <http://wiki.fernuni-hagen.de/eclipse/index.php/Draw2d>
(Stand 09.06.2011)
- [6] Packet dissection, Wireshark Dokumentation
URL : http://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html
(Stand 7.6.2011)
- [7] SWT/JFace IN ACTION, Scarapino, Holder, Ng, Mihalkovic, veröff. 2005 Manning Publications (S.275)

Abbildungsverzeichnis

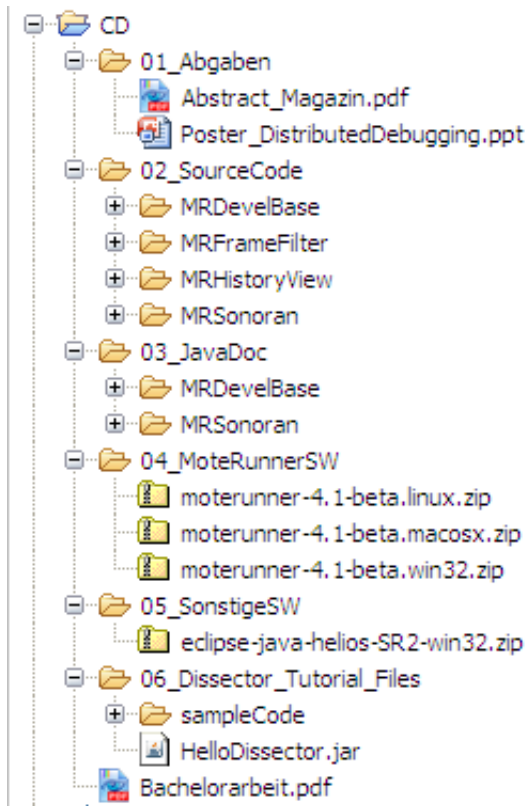
| | | |
|----|--|----|
| 1 | Sensor: MEMSIC Iris Mote | 2 |
| 2 | Vereinfachte Übersicht IBM Mote Runner | 3 |
| 3 | Mote Runner Dashboard | 4 |
| 4 | Paper Prototype 1 | 6 |
| 5 | Paper Prototype 2 | 7 |
| 6 | Abhängigkeiten zwischen den verschiedenen Plugins | 9 |
| 7 | Grobe Übersicht der HistoryView-Struktur | 10 |
| 8 | Container und Subpanels der HistoryView | 11 |
| 9 | Eclipse Perspektive im Mote Runner Projekt | 12 |
| 10 | Klassen Diagramm, General Events | 13 |
| 11 | Mote Position Events | 14 |
| 12 | Klassen Diagramm, Dissection | 15 |
| 13 | Sequenz Diagramm, Dissection | 15 |
| 14 | Autoren | 17 |
| 15 | Events und Zustände von Mote Runner in Verbindung mit der History View | 21 |
| 16 | PaperPrototype Zoom Window | 22 |
| 17 | Paperprototype Driftanzeige | 23 |
| 18 | Paperprototype Heat Map | 25 |
| 19 | JUnit Tests des FrameFilter Plugin | 28 |
| 20 | MR-Timeline Perspective | 32 |
| 21 | MR-HistoryView | 33 |
| 22 | MR-HistoryView tools | 34 |
| 23 | Period Marker | 34 |
| 24 | Dialog Box | 34 |
| 25 | MR-HeatMap | 36 |
| 26 | MR-FrameDetail | 37 |
| 27 | Configuration window to choose external dissectors | 38 |
| 28 | Example of a dissection | 38 |
| 29 | Configuration window to change the dissection tree | 39 |
| 30 | Configuration window to change colors of dissectors | 39 |
| 31 | Select the MR Protocol Disector Project | 40 |
| 32 | Configure initial Settings for the dissector | 41 |
| 33 | Configure the new dissection tree | 42 |
| 34 | Structure of the generated Project | 42 |
| 35 | Code initializeKeys method | 43 |
| 36 | Code dissection method | 44 |
| 37 | Export the dissector project as a JAR File using the Ant Buildfile | 45 |
| 38 | The HistoryView with a radiocount example and the HelloDissector | 45 |
| 39 | Code test method | 46 |
| 40 | FrameDetails View: Results are displayed in different types | 47 |
| 41 | provideColorOptionsToUser method of the MacDissector | 48 |
| 42 | initialize your colors with the colors from the preference store | 49 |
| 43 | Load the new dissector | 50 |
| 44 | Configure the new dissection tree | 51 |
| 45 | Mr-Debugger Perspektive, Starten von einer Simulation | 52 |

Tabellenverzeichnis

| | | |
|---|--|----|
| 2 | Meilenstein Übersicht | 20 |
| 3 | List of all result representations | 48 |

F Inhalt der CD

Die beigelegte CD enthält folgende Unterlagen:



G Erklärung über die eigenständige Arbeit

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.

Ort, Datum

Amon Grünbaum



Rüschlikon, 16.06.2011

Ort, Datum

Patrick Dünser



Rüschlikon, 16.06.2011