Clusterung von Sensorknoten zur Waldbranddektion

Robert Oehlmann, Jan Winkelmann
Hamburg University of Technology
Institute of Communication Networks

I. Einleitung

DIESER Bericht dokumentiert ein Protokoll zur Bildung von Clustern aus Sensorknoten um Strom zu sparen. Innerhalb der Cluster wird kein Routing benötigt, und jedes Mitglied des Clusters kann die Verwaltungsaufgaben übernehmen.

Wir beginnen mit einer weiterführenden Beschreibung des Anwendungsszenarios und den getroffenen Annahmen in Abschnitt II. Darauf folgt Abschnitt III mit der Spezifikation der funktionalen Anforderungen. Abschnitt IV beschäftigt behandelt das entwickelte Protokoll. In Abschnitt V wird die Implementierung der Simulation erläutert und anschließend in Abschnitt VI mit Bezug auf die funktionalen Anforderungen analysiert.

Als letztes werden in Abschnit VII Ansätze für zukünftige Arbeit erwähnt, und in Abschnitt VIII ein Fazit gezogen.

II. Setting

Dieser Abschnitt beschreibt zuerst das Anwendungsgebiet, und danach die Annahmen die bei dem Design des Protokolles gemacht wurden. Während des Entwerfens des Protokolles, wurde nicht drauf geachtet das Anwendungszenario realistisch zu gestalten. Weder das Anwendungsszenarion noch Annahmen beruhen auf recherchierten Fakten.

A. Anwendungsgebiet

Um Frühwarnsysteme für Waldbrände zu verbessern könnten z.B. kleine Sensoren eingesetzt werden, welche sich untereinander vernetzen um so große Flächen messen zu können. In unserem Szenario seien Sensoren gegeben, die neben dem Messen von Metriken die für die Waldbranderkennung nötig sind, auch über zwei Kommunikationsmethoden verfügen. Eine Kommunikationsmöglichkeit zur Verständigung mit anderen Sensoren in der unmittelbaren Umgebung, wie zum Beispiel W-Lan, und eine zum Senden der Messergebnisse an eine Senke, wie zum Beispiel GPRS.

Ziel dieses Projektes es nun, Sensorknoten in *Cluster* zu organisieren. Cluster benuzten Kurzstrecken-Kommunikation um dem Sensoren zu ermöglichen Batterien zu sparen, indem die Messdaten bei manchen Sensoren gesammelt werden, und so nicht jeder Sensor seine Daten einzeln zu der Senke senden muss. Der Sensor der die Messdaten zur Senke sendet bezeichnen wir als *Clusterhead*.

B. Annahmen

Die technischen Voraussetzungen für unseren Ansatz sind die Folgenden:

- Alle Sensorknoten sind baugleich, d.h. jeder Sensorknoten kann mit anderen Sensorknoten und der Senke kommunizieren.
- Keine zwei Sensorknoten versuchen gleichzeitig einem Cluster beizutreten. Dies wird Momentan dadurch erreicht, dass alle sich Sensorknoten nacheinander aktivieren.
- Sensorknoten bewegen sich nicht.
- Keine Ausfälle von einzelnen Verbindungen. Ein Sensorknoten erreicht entweder alle Knoten seines Clusters, oder keine.
- Keine temporären Ausfälle. Ein Sensorknoten fällt entweder nicht oder komplett aus.
- Wenn ein Sensorknoten ausfällt, so merken es alle Sensorknoten im Cluster gleichzeitig und mit Sicherheit, aber nicht zwingend instantan.

Zusätzlich nehmen wir an, dass ein unterliegendes Protokoll existiert, das folgendes garantiert:

- Garantierte Zustellung von Nachrichten
- Zustellung der Nachrichten in der richtigen Reihenfolge

Ein Beispiel hierfür wäre ein TCP/IP Stack.

III. Funktionale Anforderungen

1) **Implementation des beschriebenen Protokoll**
   - *Beschreibung:*
     Das Protokoll bietet die Hauptfunktionalität für das Anwendungszenario. Eine vollständige und fehlerfreie Implementation ist daher *kritisch*.
   - *Siehe:*
     Abschnitt IV für die Spezifikation des Protokolls.

- *Abnahmekriterium:*
  Dierse Anforderung gelte als erfüllt, wenn die Implementation folgendes leistet: das Bilden von Verbünden, das Behandeln von Ausfällen von Sensorknoten und die Rotation von Verbundsleitern anhand des genannten Protokolls.

2) **Kapselung der Komponenten**
   - *Beschreibung:*
     Einzelne Systemkomponenten haben auf realistische Weise voneinander gekapselt sein. Dies bezieht sich insbesondere, aber nicht aussschließlich auf den Motes zur Verfügung stehenden Informationen, wie Lage, Informationen über andere Sensorknoten, sowie Zustellung der Kommunikationen unter dem Motes durch einen von dem Sensorknoten unabhänigen Mechanismus.
   - *Problembeschreibung:*
     Um eine möglichst realistische Simulation zu sein, und um als echter Proof-Of-Concept zu gelten, müssen die simulierten Komponenten voneinander entkoppelt sein, also über Schnittstellen kommunizieren, welche auch in einem Anwendungszenarion bestehen würden.
   - *Abnahmekriterium:*
     Zur Erfüllung dieser Anforderung sind nötig: Kapselung der Mote-Objekte untereinander, Kapselung der Mote-Objekte von der Simulationsumgebung und ein Mechanismus zum Nachrichtenversenden.

3) **Graphische Oberfläche**
   - *Beschreibung:*
     Die Implementation hat eine graphische Benuzteroberfläche zur Verfügung zu stellen, welche die Simulation visualisiert. Die Bedienung der Oberfläche hat ein sinnvolles Subset der Gesamtfunkionalität zu implementieren.
   - *Problembeschreibung:*
     Um die Simulation zu visualisieren ist eine graphische Oberfläche unentbehrlich. Sie dient zur demonstration des Proof-Of-Concepts. Das Hinzufügen und Entfernen von Motes erlaubt das Demonstrieren des vollen Umfanges der Funktionalität.
   - *Abnahmekriterium:*
     Die Implementation muss eine graphische Benuzteroberfläche haben, welche die simulierte Umgebung, samt Sensoreknoten darstellt. Gebildete Cluster müssen farblich markiert sein. Zusätzlich soll man Sensorknoten hinzufügen und entfernen können.

IV. Protokoll

Dieser Abschnitt beschäftigt sich mit den entwickelten Algorithmen für das bilden von Clustern, die Ausfallsicherheit, und das Rotieren von Clusterheads.

A. Clusterbildung

Die Grundlage des Bildes von Clustern ist das Finden von vollständigen Sub-Graphen. Die Knoten des Graphen sind die Sensorknoten, und die Kanten die Verbindungen zu anderen Sensorknoten in Reichweite. Dadurch reduziert sich die Clusterbildung auf das Cliquenproblem, ist aber aufgrund der getroffenen Annahmen nicht in NP.

Sensorknoten ordnen sich nach der Aktivierung einem Cluster zu, und verlassen diesen nicht, bis der Cluster sich auflöst. Es werden keine Optimierungen an bestehenden Clustern vorgenommen. Siehe Bild IV-A für die Finite-State-Machine des Protokolls.

Cluster werden durch folgendes Protokol gebildet:

1) Falls sich ein Sensor aktiviert, so sendet sie zuerst eine Nachricht die nach vorhandenen Clustern sucht. Findest der Sensor keine vorhandenen Cluster, so bildet er selber einen.
2) Alle schon vorhandenen Clusterheads antworten auf die Anfragen von neuen Knotens. Diese Antwort enthällt einen Identifikator des Clusters und die Anzahl der Sensoren in dem Cluster.
3) Der neue Sensor speichert alle Antworten der vorhanden Cluster, ordnet sie nach Größe und versucht der Reihe nach einem der Cluster beizutreten, beginnend mit dem Kleinsten.
4) Der erste Schritt zum Beitreten eines Clusters, das Senden eine Nachricht, auf die alle Mitglieder des Clusters mit ihrer Id antworten.
5) Nach dem Ablaufen eines Timeouts, sendet der neue Sensor die Ids aller empfangenen Sensoren an den Clusterhead. Dies stellt sicher, dass der neue Sensor alle schon vorhandenen Mitglieder erreichen kann.
6) Falls die Nachricht des neuen Sensors alle Ids des aktuellen Clusters entahlten, so sendet der Clusterhead dem neuen Sensor eine Nachricht mit der Bestätigung, dass er neue Sensor dem Cluster beigetreten ist. Zusätzlich ordnet der Clusterhead dem neuen Sensor einen Slot zu. Dieser Slot wird nötig, falls der Clusterhead ausfällt.
7) Falls die Nachricht des neuen Sensors nicht alle Ids enthalten sollte, so sendet der Server eine Ablehnung und der Client versucht dem nächst größeren Cluster beizutreten.

Bild 2 ist ein UML-Sequenzdiagram welches einen erfolgreichen Clusterbeitritt mit einem weiterem Sensorknoten illustiert.

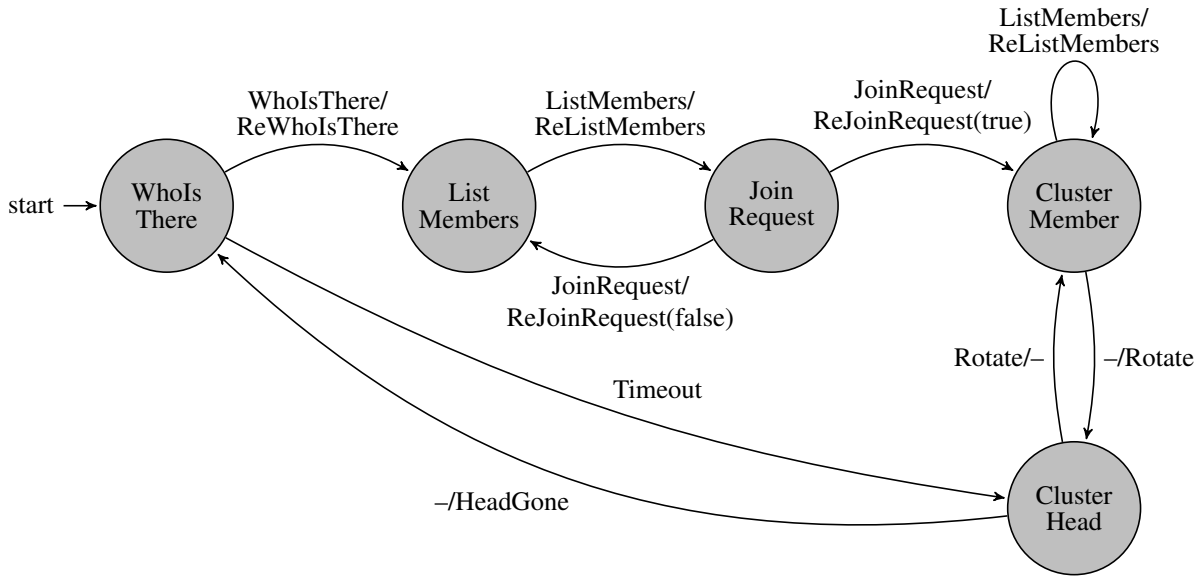

Fig. 1. State Machine des Cluster Protokolls

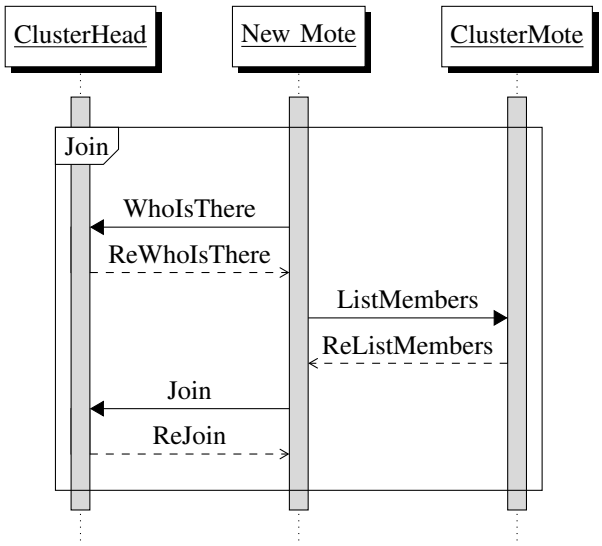

Fig. 2. Sequenzdiagram für einen erfolgreichen Clusterbeitritt

B. Ausfallsicherheit

Aufgrund der erwähnten Annahmen (TCP/IP, keine Bewegung der Sensorknoten, etc) gibt es nur zwei Ausfallszenarien die betrachtet werden müssen, der Ausfall des Clusterheads, sowie der Ausfall eines Sensorknotens der normales Mitglied in einem Cluster ist. Falls ein normaler Sensorknoten ausfallen sollte, so muss der zugehörige Clusterhead dies merken und die Liste der zum Cluster zugehörigen Sensorknoten aktualisieren. Da Sensorkntoen die dem Cluster betreten wollen Kontakt zu allen Sensorknoten nachweisen müssen, die der Cluster als zum Cluster zugehörig gespeichert hat, würde der unbemerkte Ausfall eines Sensorknotens dazu führen, dass keine neuen Knoten beitreten können. In einem realen Szenario erkennt der Clusterhead das Ausfallen eines normalen Knotens an dem Fehler von gesendeten Sensordaten. In der nie vorgestellten Simulation wird durch das Löschen direkt beim Server ein Löschevent.

Der Ausfall von einem Clusterhead wird durch das Fehlen von Bestätigungsnachrichten erkannt. Aufgrund unserer Annahme geschickt dies bei allen Knoten im Cluster gleichzeitig. Sobald ein Knoten den Ausfall des Clusterheads wahrnimmt, so wird der Cluster als tot angenommen. Der Knoten wartet nun eine bestimmte Zeit b und versucht dann einem neuen Cluster beizutreten. b berechnet sich durch die upper bound für das Beitreten eines Sensorknoten zu einem Cluster multipliziert mit dem “slot” welcher dem Knoten beim Beitreten des jetzt toten Clusters zugewiesen wurde. Der Slot ist ein streng monoton ansteigender, pro Cluster eindeutiger Integer. Somit wird sichergestellt, dass die Knoten alle nacheinander in der ursprünglichen Beitrittsreihenfolge versuchen einem neuen Cluster beizutreten.

C. Rotation der Clusterheads

Die Rotation des Clusterhead ist der Prozess welcher von dem aktuellen Clusterhead ausgeführt wird, und einem anderen Knoten die Führung des Clusters zuweist. Dieser Rotation kann ausgelöst werden von z.B. einem timeout oder dem Batteriestand. Beider Durchführung der Rotation wählst der Clusterhead der neuen Cluster und sickt alle relevanten Daten an ihn. Die gesendeten Daten sind: Der Slot der als nächstes vergeben wird, eine Liste der Ids aller Knoten in dem Cluster, sowie die Id des Clusters.

V. Implementierung

Um die dargelegten Algorithmen zu testen und zu demonstrieren, haben wir eine Simulation entwickelt. Im folgenden wird zunächst die gewählte Entwicklungsplattform vorgestellt und begründet, anschließend werden die Implementierung des Äthers, der Sensorknoten und die Spezifika der Simulations-Umgebung dargestellt.

A. JavaScript

Die Simulation ist in JavaScript implementiert; als Entwicklungsplattform haben wir Webbrowser mit aktueller JavaScript-Engine (z.B. Googles V8 und Mozillas JägerMonkey) und Unterstützung für 2D-Grafikdarstellung mithilfe des canvas-Tags gewählt.

Mit JavaScript als dynamisch typisierter, prototypenbasierter Skriptsprache lassen sich ohne große Deklarations-Overhead schnell sichtbare Ergebnisse produzieren. Mit dem neuen HTML5 Canvas Element steht eine einfach zu nutzende Zeichenfläche zur Verfügung und für alle gängigem Betriebssysteme gibt es Browser die die Simulation auführen können.

Obwohl JavaScript eine klassenlose Sprache ist, werde ich ähnliche Termini wie in anderen objektorientierten Sprachen (also auch den Terminus “Klasse”) verwenden.

B. Sensorknoten

Die simulierten Sensorknoten sind Instanze der Klasse “Mote”, deren Implementierung sich über die Dateien mote.js, mote_member.js und mote_head.js erstreckt.

Das in Abschnitt IV dargestellte Protokoll ist vollständig, aber zustandslos, in der Klasse Mote implememtiert.

Wird ein neuer simulierter Sensorknote von der Klasse Mote instanziiert, initialisiert der Konstruktur einige interne Variablen und registriert den neuen Sensorknoten beim Äther; anschließend kann dieser gestartet (“eingeschaltet”) werden.

C. Äther

Die simulierten Sensorknoten kommunizieren untereinander nur über Nachrichten. Den (virtuellen) Versandt der Nachrichten übernimmt der simulierte Äther, implementiert als Modul [1] “MoteList” in der Datei motelist.js.

Ähnlich dem Observer-Entwurfsmuster stellt MoteList eine Methode “register” bereit, die neuer Sensorknoten aufnimmt. Sie wird vom Mote-Konstruktur aufgerufen. Zudem gibt es eine Methode “send”, die den Versandt von Nachrichten anstößt.

Beim Versandt wird zunächst die Position des Absenders abgerufen, anschließend wird die Distanz zu allen anderen registrierten Sensorknoten berechnet. Ist die Distanz nicht größer als ein festgelegter Senderadius, wird die Nachricht zugestellt indem sie der “onRecv” Methode des jeweiligen Empfänger-Knotens übergeben wird. Der Absender erhält seine eigenen Nachrichten nicht.

D. Nachrichten

Bei den übergebene Nachrichten handelt es sich um JavaScript-Objekte. Sie sind nicht Instanzen einer bestimmten Klasse sondern nur des Basis-Typs Object, somit folgen sie auch keinem strikten Format.

Alle Nachrichten haben jedoch per Konvention eine Eigenschaft “type”, die Aufschluss über die Intention der Nachricht gibt (z.B. WHOISTHERE, JOINREQ, ROTATE) und somit auch die Kodierung der übrigen Eigenschaften impliziert.

Über eine Netzwerk könnten die Nachrichten JSON-kodiert verschickt werden, in der Simulation verzichten wir jedoch zugunsten der Performance auf das Marshalling und Unmarshalling beim Absender respektive Empfänger.

E. Zeit

Um die Geschwindigkeit der Simulation regulieren zu können sind alle relevanten Zeitfenster (also insbesondere Timeouts) an die globale Variable timeScale gekoppelt.

[1] http://yuiblog.com/blog/2007/06/12/module-pattern/

F. Grafische Oberfläche

Die grafische Oberfläche beschränkt sich im wesentlichen auf eine “Karte” in der alle Motes, möglichen Verbindungen, sowie Cluster verzeichnet sind.

Die Zeichenfunktion ist dabei im Modul MoteList implementiert, da nur darin Zugriff auf die Positionen der Sensorknoten möglich ist.

Die Sensorknoten sind als kleine Quadrate auf der Karte dargestellt, oberhalb eines Cluserheads wird jeweils die ID des Clusters angezeigt. Zwischen Sensorknoten die nicht mehr als einen Senderadius voneinander entfernt sind, ist eine blasse Linie eingezeichnet. Zwischen zwei Knoten die zu einem Cluster gehören, wird diese Linie farblich hervorgehoben.

Weitere Sensorknoten werden hinzugefügt, wenn man auf eine freie Fläche der Karte klickt. Klickt man einen vorhandenen Knoten an, wird dieser entfernt.

Die Rotation der Cluster-Heads wird durch Drücken der Taste ’r’ angestoßen. Die Taste ’a’ schaltet die automatische Rotation der Cluster-Heads in einem fixen Intervall ein oder aus.

VI. Analyse

Im folgenden wollen wir überprüfen, ob die Implementierung die in Abschnitt III formulierten Anforderungen erfüllt und die Performance der Simularion untersuchen.

A. Implementierung des Protokolls

Die Bildung von Clustern ist in den simulierte Sensorknoten gemäß der Spezifikation in Abschnitt IV implementiert. In der Simulation können Sensorknoten sowohl automatisch, als auch manuell platziert werden und die Knoten schließt sich dann zuverlässig einem Cluster an oder eröffnen ein neues.

Entsprechend der Spezifikationen erkennen Sensorknoten das Verschwinden anderer Knoten nicht selber, reagieren aber wie im Protokoll festgelegt, sobald sie über den Ausfall informiert werden.

Die Rotation des Clusterheads erfolgt bei Clustern mit mehreren Knoten ebenfalls nach Protokoll, jedoch wird (entsprechend der Spezifikation) nur nach externem Impuls rotiert und die Funktion des Clusterheads übernimmt ein zufällig ausgewählter, anderer Sensorknoten.

B. Kapselung der Komponenten

Eine grundlegende Kapselung der simulierten Sensorknoten untereinander besteht bereits darin, dass die Knoten jeweils als Instanzen der Klasse Mote realisiert sind.

Referenzen auf die Sensorknoten (also die Instanzen der Klasse Mote) werden nur innerhalb des MoteList Modules gespeichert. Die Liste der Sensorknoten ist im Modul-Entwurfsmuster der MoteList durch den Variablen-Skopus geschützt und nur Funktionen dieses Moduls können darauf zugreifen. Der direkte Zugriff zwischen Sensorknoten ist somit per Design ausgeschlossen.

Zusammen mit der Liste der Sensorknoten werden auch die Positionen der Knoten im Modul MoteList geschützt gespeichert. Damit sind alle relevanten Informationen der Simulationsumgebung vor den Sensorknoten abgekapselt.

Die Kommunikation der Sensorknoten untereinander läuft somit ausschließlich über den Nachrichtenversand, der im Äther (Modul MoteList) implementiert ist. Nachrichten werden unter Berücksichtigung eines Senderadius nur an ein Subset aller Sensorknoten übermittelt.

C. Graphische Oberfläche

Wie in den funktionalen Anforderungen gefordert, wird in der Kartenansicht die simulierte Umgebung mit den darin enthaltenen Sensorknoten dargestellt. Cluster werden durch farbige Linien zwischen allen Knoten des Clusters als zusammenhängende Einheit erkennbar gemacht.

Sensorknoten können hinzugefügt und entfernt werden.

D. Performance

Eine gute Performance war nicht Ziel dieser Simulation und ist im Ergebnis auch nur für Demonstrationszwecke ausreichend.

Probleme treten auf, wenn die Nachrichten-Verteilung nicht schnell genug erfolgen kann und Timeouts der Sensorknoten ablaufen, bevor die Nachrichten zugestellt wurden. Das führt meistens dazu, dass Sensorknoten nicht von vorhandenen Clustern erfahren oder nicht rechtzeitig alle Mitglieder eines Clusters auflisten können, somit der Beitritt zu vorhanden Clustern fehlschlägt und unnötig neue Cluster gegründet werden.

Wie viele Sensorknoten ohne verspätete Zustellung von Nachtichten bewältigen kann, hängt stark von der gewählten Simulationsgeschwindigkeit, sowie der der verwendeten JavaScript-Engine und natürlich der Rechenleistung eines CPU-Kerns ab. Ebenso beeinflusst die Verteilung der Sensorknoten und der Senderadius die Performance: Liegen die Sensorknoten stark

geballt oder ist der Senderadius groß gewählt, werden alle Nachrichten von vielen Knoten empfangen und das System kommt schneller an seine Grenzen, als bei weit verteilten Sensorknoten und kleinen Senderadien.

Bei den Standardeinstellungen (Karte mit 780x1248 Positionen, Senderadius von 15% der Kartenhöhe, 150 Sensorknoten) werden alleine 800 Nachrichten verschickt und rund 3000 bis 3300 Nachrichten empfangen.

Platziert man dagegen nur 30 Knoten so dicht beieinander, dass sie ein einzelnes Cluster bilden, werden dabei bereits 551 Nachrichten verschickt und 10.294 Nachrichten empfangen. Zwingt man dieses Cluster zur Neubildung, indem man den Clusterhead löscht, entstehen dabei innerhalb von 5 Sekunden nochmal 519 Nachrichten die 14.533 mal empfangen werden.

Die obenstehenden Versuche wurden auf einem Core2Duo Mobil-Prozessor mit 1,87GHz Kerngeschwindigkeit in Googles Browser Chrome mit der V8-JavaScript-Engine durchgeführt. Während der Clusterneubildung wurde ein Prozessorkern für 5 Sekunden voll ausgelastet.

Aufgrund der vielen Faktoren die die Performance beeinflussen soll hier auf eine weitere Analyse der Performance auf verschiedenen Systemem verzichtet werden.

Mit den Standardeinstellungen und in einem aktuellen Browser sollte es problemlos möglich sein, die Simulation zu testen.

VII. Future Work

Die zur Verfügung gestellte Implementierung des beschriebenen Protokolls erfüllt die funktionalen Anforderungen im vollen Maße. Um aber den Einsatz des Protokolles realistischer zu machen, ist noch Arbeit an den Annahmen aus Abschnitt II nötig. So wäre s möglich das Protokoll um die Möglichkeit zu erweitern mehreren Sensorknoten zu erlauben gleichzeitig einem Cluster beizutreten. Dies wäre zum Beispiel realisierbar, in dem der Clusterhead bei einem schon laufenden join-Vorgang blockiert, und der Sensorknoten wartet und danach erneut versucht dem Cluster beizutreten.

Zusätzlich müsste das Protokoll auf in der Realität genutze drahtlose Kommunikationprotokolle, die unter anderem kein Multicast unterstützen, angepasst werden.

VIII. Fazit

Dieser Bericht stellt ein Protokoll vor, welcher das Bilden von Sensorknoten-Clustern erlaubt. Es unterstützt neben dem Bilden von Clustern auch den Ausfall von Knoten, sowie die Rotation des Clusterheads. Bereitgestellt wird zusätzlich eine Implementation des Protokolls in Form einer Simulation. Umfang der Simulation ist neben einer graphischen Darstellung auch ein gekapseltes Simulationsumfelt, und die Möglichkeit Motes hinzuzufügen, oder zu entfernen.