

Studienarbeit

Visualisierung der Energie- und
Kommunikationsdaten einer auf
PowerTOSSIM basierenden
Simulation eines Sensornetzwerkes

Christian Czekay

14. April 2008

Betreuer: Dipl.-Inf. Timo Mika Gläßer
Lehrstuhl für Wissensmanagement in der Bioinformatik

Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die Studienarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen angefertigt habe.

Berlin, 14. April 2008

Inhaltsverzeichnis

1	Einleitung	7
1.1	Zielstellung	8
2	Verwandte Arbeiten	10
2.1	Simulatoren	10
2.1.1	ns-2	10
2.1.2	JiST/SWANS	10
2.1.3	PowerTOSSIM	11
2.1.4	Auswahl des verwendeten Simulators	12
2.2	Visualisierungswerkzeuge	13
3	Umsetzung	15
3.1	Simulationssetup	16
3.2	Simulationsdurchführung	18
3.2.1	Kompilierung	18
3.2.2	“Tracer”	18
3.3	Datenbankschema	22
3.3.1	ER Diagramm	22
3.4	Simulationsauswertung	24
3.4.1	Energiemodell	24
3.4.2	Vorverarbeitung der Energiedaten	28
3.4.3	Funknachrichten	29
4	Messungen	31
4.1	Simulator und “Tracer”	31
4.1.1	Messaufbau	31
4.1.2	Messergebnisse	33
4.1.3	Skalierbarkeit und Grenzen des Systems	34
4.1.4	Bewertung einzelner Implementationsaspekte	40
4.2	Vorverarbeitung der Energiedaten	43
4.2.1	Motivation	43
4.2.2	Zielstellung	44
4.2.3	Umsetzung	44
4.2.4	Skalierung des Verfahrens	46
4.3	Datenbankunabhängigkeit vs. Performance	47
4.4	Auswertung / Animation	48
5	Ausblick	50
5.1	Workflowverbesserungen	50
5.1.1	Integration des Simulationsstarts in die Browseroberfläche	50

5.1.2	Integration von existierenden Sensordatenaufzeichnungen	50
5.1.3	Dynamische Radiomodelle und mobile Knoten	50
5.2	Simulatorverbesserungen	51
5.2.1	Unterstützung für weitere Hardwareplattformen	51
5.3	Performance	51
5.3.1	Vorverarbeitung der Energiedaten	51
5.3.2	Auswertung	52
A	Nachrichtenklassenaufistung	55
B	Datenbankschema	57
C	Rohdaten der Messungen	61

Abbildungsverzeichnis

1	Schematische Übersicht der Simulationsumgebung	15
2	Anlegen eines Simulationsszenarios im “Viewer”: In der <i>GoogleMaps</i> -basierten Oberfläche kann der Nutzer Knoten platzieren sowie ihre Funkreichweite einstellen und visualisieren lassen.	17
3	ER Diagramm	23
4	Statische Simulationsvisualisierung im “Viewer”: Der Nutzer kann sich den Energieverbrauch der Knoten und ihre Funkaktivität in einem frei wählbaren Zeitfenster sowohl in Textform als auch grafisch darstellen lassen.	25
5	Animierte Simulationsvisualisierung im “Viewer”: Neben der statischen Visualisierung kann die Entwicklung von Energieverbrauch und Funkaktivität der Knoten auch in einer dynamischen Echtzeitanimation angezeigt werden.	26
6	Simulationsdauer in Abhängigkeit der simulierten Zeit	35
7	Simulationsdauer in Abhängigkeit der simulierten Knoten	36
8	Simulationsdauer pro simuliertem Knoten	36
9	Simulationsdauer in Abhängigkeit der erzeugten Simulationsergebnisse	37
10	Datenbanksimulationsergebnisse pro simulierter Sekunde	38
11	Datenbankgröße in MB pro simulierter Sekunde	38
12	Datenbanksimulationsergebnisse pro Knoten	39
13	Datenbankgröße in MB pro Knoten	39
14	Verarbeitungszeit in Abhängigkeit der zu verarbeitenden Simulationsergebnisse	47

Tabellenverzeichnis

1	Energieverbrauch der <i>Mica2</i> -Plattform	27
2	Messergebnisse zusammengefasst pro Messreihe (Zeiten)	41
3	Messergebnisse zusammengefasst pro Messreihe (Ereignisse)	41
4	Messung der Vorverarbeitung der Energiedaten	46
5	Messung der Geschwindigkeit des Schreibens von Sensorinputdaten in die Datenbank durch den “Viewer”	48
6	Auflistung aller definierten Ereignisklassen (aus <code>/opt/tinyos-1.x/tos/platform/pc/GuiMsg.h</code>)	55
7	Unterkategorien der <code>DebugMsgEvent</code> -Klasse (aus <code>/opt/tinyos-1.x/tos/types/dbg_modes.h</code>)	56
8	Übersicht über verwendete Einheiten im Datenbankschema	60
9	Daten der mit <code>SenseToRfm</code> durchgeführten Messreihen (alle Zeiten in Sekunden)	66

10	Aufschlüsselung der Simulationsereignisse nach Kategorien (in absoluten Zahlen)	67
11	Aufschlüsselung der Simulationsereignisse nach Kategorien in % (relativ zu den Simulationsereignissen vom Simulator) . . .	68
12	Aufschlüsselung der Simulationsereignisse nach Kategorien in % (relativ zu den Simulationsereignissen in der Datenbank) .	69

Listings

1	Datenbankschema	57
---	---------------------------	----

1 Einleitung

Die rasanten Fortschritte in der Hardwarefertigung in den letzten Jahren / Jahrzehnten haben den drahtlos kommunizierenden Sensornetzwerken einen weiten Raum an Einsatzmöglichkeiten eröffnet. Sie können z.B. zur großflächigen Datenerfassung oder als ad-hoc Kommunikationsinfrastruktur eingesetzt werden. Von Vorteil sind dabei ihre höhere Flexibilität und der geringere Preis gegenüber kabelgebundenen Lösungen.

Aufgrund der breiten Spanne an Einsatzmöglichkeiten gibt es viele sehr unterschiedliche Anforderungsprofile für Sensornetzwerke. Aber während die funktionalen Anforderungen stark variieren, ist praktisch allen Einsatzszenarien die nicht-funktionale Anforderung der Energieeffizienz der auf den Sensorknoten laufenden Algorithmen gemein. Es liegt in der Natur von Sensorknoten, dass sie eine autonome und sehr begrenzte Energieversorgung besitzen. Betrachtet man beispielsweise als Szenario das Ausbringen von Sensorknoten als Frühwarn- und Ersatzkommunikationsmedium im Katastrophenfall (z.B. Erdbeben[21, 8, 5]), so ist klar, dass man sich zur Energieversorgung der Knoten nicht auf das normale Stromnetz verlassen kann, da es im Ernstfall meist zu stark beschädigt wird und nicht zur Verfügung steht. Die Verwendung von Sensornetzen als Ersatz für die ausgefallene konventionelle Kommunikationsinfrastruktur macht natürlich nur Sinn, wenn garantiert werden kann, dass die Knoten auch wirklich zur Verfügung stehen und nicht aufgrund ineffizienter Energienutzung nach kurzer Zeit wegen Energiemangel ausfallen. Schließlich macht es die verteilte Natur solcher Netzwerke und die Tatsache, dass sie mitunter an sehr schwer zugänglichen Orten eingesetzt werden, schwierig bis unmöglich, die Energiequelle der einzelnen Knoten manuell zu ersetzen.

Wenn also Kommunikations- und Routingprotokolle für Sensorknoten erprobt und evaluiert werden, ist es unbedingt nötig, den Aspekt der Energieeffizienz in Betracht zu ziehen.

Da die Software von einmal im Einsatzgebiet installierten Sensorknoten nur unter großem Aufwand oder gar nicht aktualisiert werden kann, ist es zwingend nötig, die verwendeten Algorithmen bereits vor dem Einsatz auf Korrektheit, Robustheit und Energieeffizienz zu testen. Aus verschiedenen Gründen kann oder will man diese Tests aber nicht auf physischen Knoten durchführen. Dies liegt an den Anschaffungskosten für ausreichend große Mengen an Knoten, daran, dass die Knoten aufgrund ihrer schieren Menge nur schwer wartbar sind und schließlich an der Schwierigkeit, Experimente kontrolliert und reproduzierbar durchzuführen und alle für eine sinnvolle Auswertung nötigen Daten zu sammeln. Durch den Einsatz von Simulationen wird es möglich, diese Probleme zu vermeiden und Algorithmen in genau definierten Szenarien zu testen, zu vergleichen und ihre Funktionsweise zu verstehen. Eine entsprechende Simulationsumgebung muss einerseits die Realität möglichst detailgetreu nachbilden, um relevante Aussagen zu liefern,

andererseits darf die Simulationstiefe auch nicht zu hoch sein, da es sonst unmöglich wird, eine ausreichend große Menge von Knoten in vertretbarer Zeit zu simulieren.

1.1 Zielstellung

Zum Durchführen von Simulationen müssen Simulationsszenarien erstellt und anschließend die Simulationsergebnisse ausgewertet werden. Für erstere Aufgabe ist üblicherweise das manuelle Editieren teilweise kryptischer Konfigurationsdateien der verwendeten Simulationssoftware nötig. Das Auswerten geschieht normalerweise durch die Analyse der Tracedateien des Simulators, die sämtliche low-level Ereignisse auflisten. Wenn das manuell gemacht wird, ist dies insbesondere bei größeren Mengen von Knoten und nicht-trivialen Simulationszeiträumen fehleranfällig und mühselig. Um globale Fragen z.B. nach der Effizienz des Routingprotokolls oder der durchschnittlichen Auslastung der Knoten zu beantworten, werden meist on-the-fly erstellte Skripte verwendet, was zusätzlichen Aufwand erzeugt. Zwar wurden in den letzten Jahren viele Werkzeuge entwickelt, die diese Arbeiten vereinfachen, aber insbesondere die grafische Auswertung und der Aspekt der Energieeffizienz wurden meist nur unzureichend berücksichtigt.

Das Ziel meiner Studienarbeit ist es, ein Web- und Datenbankgestütztes Werkzeug bereitzustellen, welches den Nutzer bei diesen Aufgaben unterstützt. Die Nutzung erfolgt über eine Webbrowser-basierte Oberfläche und ist somit unabhängig vom Ort und Rechnersystem des Experimentators. Das Platzieren von Sensorknoten beim Erstellen eines Simulationsszenarios wird durch Integration von *GoogleMaps* stark vereinfacht. Aufgrund der Datenbank-gestützten Datenverarbeitung ist auch die Verwendung von existierenden Datenbanken, die Aufzeichnungen von Sensorpositionen und -messungen von realen Ereignissen enthalten, für Referenzexperimente denkbar.

Eine Visualisierung in Echtzeit direkt während des Simulationslaufs ist nicht vorgesehen, da sich gezeigt hat, dass der Simulator dafür deutlich zu langsam ist, sobald man nicht-triviale Knotenmengen simulieren will. Die kompletten Tracedaten der Simulation werden stattdessen zentral in der Datenbank gespeichert.

Besonderer Wert wird auf die grafische Visualisierung der Simulationsergebnisse gelegt. Zentraler Aspekt dabei ist, die verbleibende Energie der Knoten und ihren Energieverbrauch sowie die Kommunikationsaktivität zwischen den Knoten darzustellen. Es soll möglich sein, die Energieeffizienz des simulierten Protokolls aus der Visualisierung direkt abzuschätzen. Dafür ist geplant, dass man sich für beliebig wählbare Zeitintervalle Energie- und Radiostatistiken anzeigen lassen kann und außerdem von einem beliebigen Simulationszeitpunkt aus die Entwicklung der Kommunikationsaktivität und des Energieverbrauchs jedes Knotens in Echtzeit mitverfolgen kann.

Die Datenerfassung soll möglichst flexibel sein, damit der Nutzer sie bei Bedarf um weitere Simulationsdaten ergänzen kann, welche er dann mit eigenen Werkzeugen auswertet.

Das Ziel ist es, die Bewertung von Kommunikationsprotokollen hinsichtlich ihrer Energieeffizienz zu vereinfachen, sodass es leichter wird, Protokolle unter dem Energieeffizienzaspekt zu optimieren oder neu zu entwickeln.

2 Verwandte Arbeiten

2.1 Simulatoren

Zur Simulation von Sensornetzwerken wurden verschiedene “discrete event” Simulatoren entwickelt. Im Folgenden werden einige vorgestellt und ihre Eignung für die geplante Simulationsumgebung überprüft.

2.1.1 ns-2

ns-2 (NetzwerkSimulator Version 2)[13] ist ein Netzwerksimulator für die Simulation von TCP und diversen Routing- und Multicastprotokollen in sowohl drahtgebundenen als auch drahtlosen Netzen. Er ist einer der meist verbreiteten Simulatoren und bringt grundlegende Unterstützung für die Analyse des Energieverbrauchs eines Kommunikationsprotokolls mit. *ns-2* verfügt über ein Energiemodell, welches es erlaubt, die Startenergie eines Knotens und den Energieverbrauch im Idle- und Sleepmodus, beim Empfangen und Senden von Paketen und beim Erfassen von Sensordaten zu spezifizieren. Simulationsabläufe werden in *ns-2* mithilfe von *OTcl* (MIT Object Tcl) Skripten beschrieben. Deren Entwicklung ist nicht ganz einfach und es ist nicht möglich den Simulationscode auf realen Sensorknoten einzusetzen. Man müsste die simulierten Algorithmen stattdessen für den Sensorknoten komplett neu implementieren und Änderungen am Simulationscode manuell in die Sensorknotenversion einpflegen, ein aufwendiger und fehleranfälliger Prozess. Außerdem simuliert *ns-2* üblicherweise nur auf Paketebene und nicht darunter, eine Vereinfachung, die je nach Einsatzszenario die Validität der Ergebnisse fraglich machen kann. Um nicht bereits unterstützte Protokolle zu simulieren, muss der in C++ geschriebenen *ns-2* entsprechend erweitert werden, was nicht unerhebliche Einarbeitungszeit und Aufwand bedeutet.

2.1.2 JiST/SWANS

JiST (Java in Simulation Time)[3] ist eine Java-basierter Netzwerksimulationsplattform. Eine Simulation besteht aus Entitäten, die keinerlei Zustandsinformation teilen dürfen und nur über Nachrichten/Ereignisse miteinander kommunizieren. Die zu simulierenden Algorithmen werden in normalem Javacode geschrieben, welcher zu Javabytecode kompiliert wird. Ein *JiST* spezifischer “rewriter” modifiziert dann diesen Bytecode, um Simulationssemantik einzubetten. Dabei wird beispielsweise jeder Methodenaufruf an einer Entität in das Auslösen eines Ereignisses umgewandelt, der Methodenaufruf also vom Ausführen der Methode (welches erst geschieht, wenn die Simulation den entsprechenden Ereigniszeitpunkt erreicht hat) entkoppelt. Der modifizierte Bytecode wird dann in einer ganz normalen JVM zusammen mit dem *JiST* Simulationskernel ausgeführt. Da die Entitäten keine Zustandsinformation teilen, kann die Simulation an den Entitätsgrenzen zu

Parallelisierungszwecken partitioniert werden. Verschiedene Entitäten können sich dabei an unterschiedlichen Punkten in der Simulationszeit befinden. Diese optimistische Parallelisierung ist möglich, da der “rewriter” auch Checkpointing- und Rollbackfunktionalität in den Bytecode einbettet. Es ist sogar möglich die Simulation auf verschiedene Computer zu verteilen, um den Parallelisierungsgrad zu erhöhen. Durch die Verwendung von Java stehen außerdem sämtliche Vorteile der Javawelt, wie JIT und garbage-collection zur Verfügung. Im Vergleich zu *ns-2* ist *JiST* deutlich schneller und skalierbarer. *SWANS* (Scalable Wireless Ad hoc Network Simulator)[2] ist ein auf der *JiST*-Plattform aufsetzender Netzwerksimulator. Die verschiedenen Module eines Netzwerkstacks werden durch *JiST*-Entitäten abgebildet. Es gibt unter anderem Komponenten für IEEE 802.11b, TCP, UDP und verschiedene Routingprotokolle. Außerdem ist es möglich, bestehende Javanetzwerkanwendungen im simulierten Netzwerk zu betreiben. Die von diesen Anwendungen verwendeten Socketklassen werden vom “rewriter” durch äquivalente Implementationen ersetzt, die den simulierten Netzwerkstack verwenden.

JiST und *SWANS* verfügen über keinerlei Energiemodell, aber es gibt eine Arbeit[19], die ein Energiemodell für drahtlose Kommunikation hinzufügt, welches zwischen den vier Zuständen Senden, Empfangen, Sleep und Idle unterscheidet.

2.1.3 PowerTOSSIM

PowerTOSSIM[17] ist eine Erweiterung des Simulators *TOSSIM*[12], bei der die Fähigkeit, den Energieverbrauch einzelner Sensorknotenkomponenten akkurat zu messen, ergänzt wurde. Das von *PowerTOSSIM* verwendete Energiemodell erfasst den Energieverbrauch für die Kategorien CPU, Funk, LED, EEPROM, Sensor und ADC. Der CPU Energieverbrauch teilt sich auf in eine Grundlast und einen lastabhängigen Verbrauch. Die Höhe der Grundlast hängt davon ab, ob sich die CPU im normalen oder einem der fünf Energiesparmodi befindet und der lastabhängige Anteil besteht aus dem zusätzlichen Verbrauch, den jeder CPU Zyklus verursacht, in dem die CPU tatsächlich aktiv ist. Die einzelnen simulierten Hardwarekomponenten wurden so modifiziert, dass sie energierelevante Zustandsänderungen, wie das An- oder Ausschalten einer der LEDs oder die Änderung der Sendestärke des Funkmoduls, in den Simulationstraces protokollieren. Durch Auswertung dieser Daten kann nach der Simulation der Energieverbrauch eines jeden Knoten ermittelt werden. Durch den Einsatz von Codetransformationstechniken wird außerdem die Abschätzung der vom simulierten Programm verbrauchten CPU-Zyklen ermöglicht (siehe dazu Abschnitt 3.2.1). Außerdem beinhaltet *PowerTOSSIM* exakte Daten über den Energieverbrauch der Komponenten von auf der *Mica2*-Plattform basierenden Knoten, was Simulationen mit praxisrelevanten Ergebnissen ermöglicht.

TOSSIM ist wie *ns-2* ein “discrete event network simulator” auf Open-

Source Basis. Im Gegensatz zu *ns-2* ist *TOSSIM* kein allgemeiner Netzwerksimulator, sondern ein spezialisierter *TinyOS*-Simulator. *TinyOS*[9] ist ein in *nesC* (network embedded systems C) geschriebenes Betriebssystem für eingebettete drahtlose Systeme wie z.B. Sensorknoten. *nesC* ist eine für *TinyOS* entwickelte C-Erweiterung, die ein komponentenbasiertes Programmieren ermöglicht. *TinyOS* Anwendungen werden ebenfalls in *nesC* geschrieben und gemeinsam mit dem Betriebssystemcode zu einem Binary kompiliert, welches dann auf reale Knoten aufgespielt werden kann.

TOSSIM kann *TinyOS*-Anwendungen direkt simulieren, was im Umkehrschluss bedeutet, dass man in der Simulation entwickelten Code ohne Änderungen auf den Knoten einsetzen kann. *TOSSIM* simuliert dabei nicht aufwändig die einzelnen Ausführungsschritte der Knotenhardware. Stattdessen wird der Anwendungs Quellcode zu einem nativ auf der Simulationsmaschine lauffähigem Programm kompiliert, was es erlaubt, die Simulation auf Hunderte von Knoten zu skalieren. Dies wird durch das komponentenbasierte Design von *TinyOS* ermöglicht, denn es müssen für die Simulation lediglich einige wenige hardwarenahe Komponenten, die beispielsweise den Zugriff auf die Sensoren oder das Funkmodul regeln, ersetzt werden. Der Großteil des *TinyOS* Quellcodes und der gesamte Anwendungs Quellcode kann ohne Modifikation übersetzt werden. Durch Modifikationen am *nesC* Compiler *ncc* wurde zudem erreicht, dass jede Variable im *nesC*-Code durch ein Array ersetzt wird, auf welches die simulierten Knoten mit ihrer jeweiligen Knotennummer als Index zugreifen, wodurch jeder simulierte Knoten einen separaten Bereich im Adressraum des Simulatorprozesses erhält. Da die Speicherausstattung der Knoten im Verhältnis zu der des Simulationsrechners meist sehr klein ist, wird der Arbeitsspeicherbedarf der Simulation nur bei extrem großen Knotenmengen zu einem begrenzenden Faktor.

Das Simulationsmodell von *TOSSIM* ist ereignisbasiert und benutzt eine globale Ereigniswarteschlange. Dadurch ist *TOSSIM* strikt singlethreaded und nicht parallelisierbar.

Bei der Simulation der Funkübertragung wird nicht die Ausbreitung der Funkwellen selbst simuliert, sondern die Übertragung von einzelnen Bits. Jedes gesendete Bit wird mit einer gewissen Wahrscheinlichkeit bei der Übertragung invertiert und diese Wahrscheinlichkeit kann für jedes Paar von Knoten einzeln spezifiziert werden. Ist sie für ein Knotenpaar nicht spezifiziert, so können diese Knoten sich nicht hören und auch nicht durch Interferenz stören. Die Signalstärke ist für jedes Bit identisch und gleichzeitig gesendete Bits werden Oder-verknüpft empfangen.

2.1.4 Auswahl des verwendeten Simulators

Für *ns-2* spricht seine hohe Verbreitung und für *JiST/SWANS* seine Skalierbarkeit. Aber beide sind "nur" Netzwerksimulatoren, während *PowerTOSSIM* eine konkrete Hardwareplattform simuliert. Dies ermöglicht eine deut-

lich realitätsnähere Abbildung des Energieverbrauchs. Während beispielsweise das *SWANS* Energiemodell nur die genannten vier abstrakten Zustände der Funkkomponente beinhaltet, umfasst das *PowerTOSSIM* Energiemodell für die *Mica2*-Plattform die zehn möglichen Sendestärkenstufen dieser Hardware und bildet auch den Energieverbrauch beim Zugriff auf EEPROM, LEDs und Sensoren ab. Schließlich kann *PowerTOSSIM* den Energieverbrauch der CPU relativ exakt abschätzen, was reinen Netzwerksimulatoren prinzipiell unmöglich ist, da sie weder den letztendlich auf den realen Knoten zum Einsatz kommenden Code, noch die konkrete Hardware, auf der er laufen wird, kennen. Da ich in meiner Arbeit besonderen Wert auf die Energieeffizienz lege, habe ich mich für den Einsatz von *PowerTOSSIM* entschieden.

2.2 Visualisierungswerkzeuge

Die Visualisierung von Simulatorergebnissen ist keine neue Aufgabe und es gibt bereits diverse Tools für diese Aufgabe.

ns-2 selbst kommt bereits mit dem Visualisierungstool *nam*, dem “Network AniMator” [6]. *nam* erzeugt eine 2D Darstellung der Knotenpositionen und der Sende- und Empfangsevents. *nam* war ursprünglich dazu gedacht, Traces von drahtgebundenen Netzwerksimulationen von *ns-2* zu visualisieren, kann inzwischen aber auch das neuere Format von drahtlosen Traces darstellen. Allerdings sind die Visualisierungsfähigkeiten in diesem Bereich noch recht begrenzt, insbesondere gibt es keine Möglichkeit, energierelevante Werte zu visualisieren.

ad-hockey [14] wurde im Rahmen des *Monarch*-Projekts [4] entwickelt, welches *ns-2* um die Fähigkeiten zur Simulation von drahtlosen Netzwerken erweiterte. Allerdings ist das letzte Release von 1999 und enthält keinerlei Unterstützung für die Visualisierung von Energieparametern.

Ein weiteres *ns-2* Visualisierungswerkzeug ist *iNSpect* [11]. Es konzentriert sich darauf, die Knotenpositionen und -bewegungen sowie Paketrouten darzustellen. Außerdem ist es möglich ein *ns-2* “mobility-file”, welches die Knotenbewegungen im Simulationsverlauf definiert, von *ns-2* unabhängig zu visualisieren, um zu verifizieren, dass es dem gewünschten Verhalten entspricht. Auch *iNSpect* visualisiert keine energierelevanten Werte.

Mit *Huginn* [16] gibt es ein weiteres *ns-2* Visualisierungstool. Es visualisiert drahtlose *ns-2* Traces in 3D. Der Nutzer kann über einen GUI-Dialog flexibel bestimmen, wie bestimmte Simulationsereignisse oder Statistikwerte visualisiert werden sollen. Außerdem kann man zu beliebigen Punkten in der Simulationszeit springen und es ist (im Gegensatz zu z.B. *iNSpect*) nicht nötig, das gesamte Tracefile vor der Visualisierung einzulesen, was Speicher und Zeit spart. Allerdings ist das Setup des Programms und der benötigten Bibliotheken überaus kompliziert und fehleranfällig und auch bei *Huginn* liegt der Fokus lediglich darauf, Position und Kommunikationsaktivität der

Knoten darzustellen.

TinyViz ist ein allgemeiner Visualisierer für *TOSSIM* und *PowerTOSSIM* bringt ein *TinyViz*-Plugin mit, welches den Energieverbrauch der Knoten darstellt. Es ist aber recht einfach gehalten und hat eher Demonstrationscharakter.

3 Umsetzung

Die Simulationsumgebung besteht aus vier Komponenten:

- “Viewer”: Eine auf Basis des Ruby-basierten Webapplikationsframeworks *Ruby On Rails* [15] geschriebene Webapplikation, in der der Nutzer Simulationsszenarien erstellen und später die Ergebnisse visualisieren kann.
- Datenbank: In der PostgreSQL-Datenbank werden die Inputs der Simulation und ihre Ergebnisse gespeichert.
- “Tracer”: Ein Java-Programm, das den Simulationsprozess steuert, ihn mit Eingabedaten aus der Datenbank füttert und die Simulationstraces zur späteren Verarbeitung dorthin zurückschreibt.
- *PowerTOSSIM* als Simulator

“Viewer”, Datenbank und “Tracer” können auf verschiedenen Rechnern liegen. Das System, auf dem sich “Tracer” und Simulator befinden, sollte sinnvollerweise über eine komplette *TinyOS/PowerTOSSIM* Umgebung verfügen, um die zu simulierenden Programme kompilieren zu können.

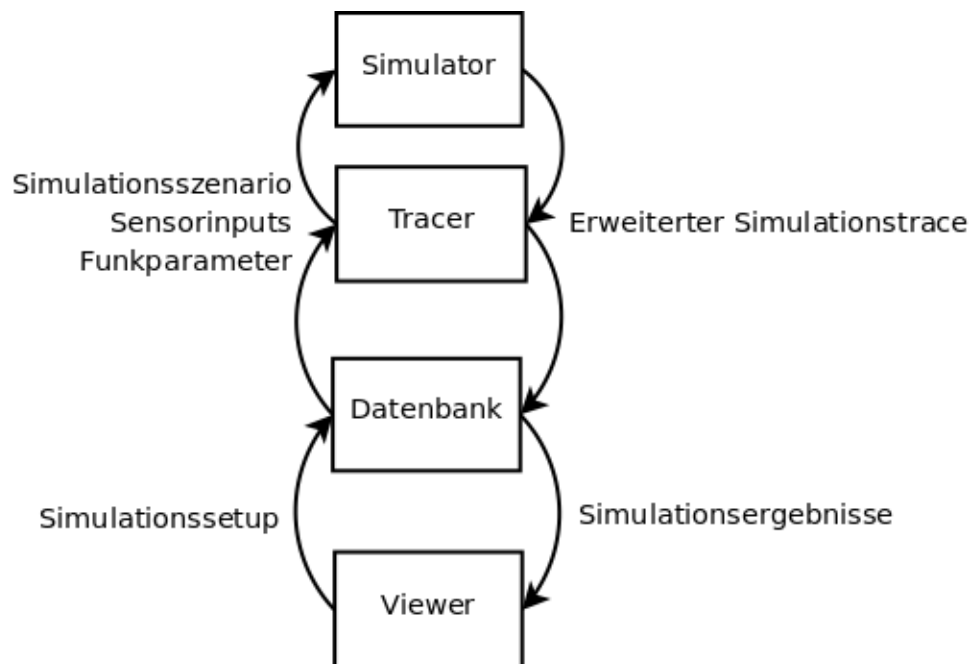


Abbildung 1: Schematische Übersicht der Simulationsumgebung

Als Entwicklungssystem kam ein PC mit *Intel Core2 6600* Doppelkern CPU mit 2.4 GHz und 4 GB RAM zum Einsatz. Der Datenbank standen 180

GB von einer 200 GB großen reiserfs Partition zur Verfügung. Das System lief unter einem 64-Bit *Gentoo Linux* und es kamen Rails-1.2.6, Ruby-1.8.6, Java-1.5 und PostgreSQL-8.0.15 zum Einsatz. Der “Viewer” und die Datenbank liefen direkt auf dem System, aber der “Tracer” lief in einer 32-Bit *VMware*-Instanz und kommunizierte über eine virtuelle Netzwerkschnittstelle mit der Datenbank. Der *VMware*-Instanz stand die volle Prozessorkapazität und 512 MB RAM zur Verfügung. Sie lief unter *XubuntuTOS-1.0*[23], einer auf der *Ubuntu*-Variante *Xubuntu* basierende Linuxdistribution mit einer vorinstallierten *TinyOS*-Umgebung. Das als Grundlage dienende *Ubuntu* befand sich in Version 6.10 (*edgy*) und es kam ebenfalls Java-1.5 zum Einsatz. Der Einsatz von *VMWare* resultierte aus der Schwierigkeit, eine funktionierende *TinyOS*-Umgebung aufzusetzen. Die vorhandenen Anleitungen sind alleamt veraltet, zueinander widersprüchlich und funktionieren nie vollständig. Aus diesem Grund habe ich mich schließlich für die Verwendung von *XubuntuTOS* entschieden und da dafür kein dedizierter Rechner zur Verfügung stand, musste es virtualisiert betrieben werden. Wenn das Programm einmal produktiv benutzt wird, sollte man *XubuntuTOS* nach Möglichkeit nativ laufen lassen, denn Messungen zeigen, dass durch den Wegfall der Virtualisierung die Simulation um weit über 100% beschleunigt wird.

3.1 Simulationssetup

In Abbildung 2 ist das Interface zum Platzieren der Knoten zu sehen. Durch die Integration mit *GoogleMaps* wird eine realitätsbezogene Platzierung sehr erleichtert. Anschließend werden die Positionen der Knoten als Länge/Breite Werte in der Datenbank gespeichert. Außerdem kann die als Idealkreis modellierte Reichweite der Funkübertragung spezifiziert und als Platzierungshilfe visualisiert werden. Dann können für jedes Szenario Sensoren und entsprechende Inputdaten definiert werden. Ein Sensor besteht aus einem Namen und einer ADC-Portnummer, die das simulierte Programm ansprechen muss, um die aktuellen Daten des entsprechenden Sensors zu lesen. Ein Sensor ist immer genau einem Knoten zugeordnet. Da das *TOSSIM*-Design davon ausgeht, dass alle Knoten identisch sind, definiert man Sensoren aber nicht pro Knoten, sondern pro Szenario und diese werden dann automatisch zu jedem Knoten des Szenarios hinzugefügt. Da die Generierung von Sensorinputs extrem situationsabhängig und kaum verallgemeinerbar ist, gibt es dafür keine Unterstützung. Der Nutzer muss die entsprechenden Daten selbst erzeugen und kann sie dann über den “Viewer” in die Datenbank laden, so dass sie der Simulation später zur Verfügung stehen. Das verwendete Format ist so einfach wie möglich gehalten: Pro Zeile wird ein Sensorwert definiert in der Form “Knoten-ID:Sensor ADC-Port:Sensorinput Zeitstempel:Sensorinput Wert”. Ein Sensorwert gilt ab dem definierten Zeitpunkt solange, bis er durch einen neuen Wert überschrieben wird. Die Simulation der Funkübertragung kann sich auf die Entfernung der Knoten zueinander



Update Show

name

map settings

description ([show](#))

center: latitude longitude

radio range cm

zoom

Show radio range for reliable transmission

width px height px

notes

#	name	latitude	longitude	
0	<input type="text" value="mote000"/>	<input type="text" value="52.502332"/>	<input type="text" value="13.583275"/>	<input checked="" type="checkbox"/>
1	<input type="text" value="mote001"/>	<input type="text" value="52.503063"/>	<input type="text" value="13.583736"/>	<input checked="" type="checkbox"/>
2	<input type="text" value="mote002"/>	<input type="text" value="52.503311"/>	<input type="text" value="13.581086"/>	<input checked="" type="checkbox"/>
3	<input type="text" value="mote003"/>	<input type="text" value="52.503194"/>	<input type="text" value="13.582438"/>	<input checked="" type="checkbox"/>
4	<input type="text" value="mote004"/>	<input type="text" value="52.502012"/>	<input type="text" value="13.585099"/>	<input checked="" type="checkbox"/>

Abbildung 2: Anlegen eines Simulationsszenarios im "Viewer": In der *GoogleMaps*-basierten Oberfläche kann der Nutzer Knoten platzieren sowie ihre Funkreichweite einstellen und visualisieren lassen.

stützen, aber der Anwender kann wenn nötig auch eine eigene Netztopologie definieren (für Details siehe Abschnitt 3.2.2). Als letzte Kategorie von Eingabedaten bleiben noch Bewegungsangaben für die Knoten. Mein Programm ist momentan auf stationäre Knoten beschränkt, aber Knotenmobilität lässt sich prinzipiell hinzufügen, falls dies einmal benötigt werden sollte.

3.2 Simulationsdurchführung

Im folgenden beschreibe ich zunächst, wie man ein *TinyOS* Programm kompilieren muss, um es in einer Simulation nutzen zu können. Dann gehe ich näher auf den “Tracer” ein: welche Parameter er unterstützt, wie er mit dem Simulationsprozess interagiert und welche Probleme bei der Verarbeitung der Simulationsdaten im “Tracer” zu lösen sind.

3.2.1 Kompilierung

Das Kompilieren eines *TinyOS* Programms in eine ausführbare *PowerTOSSIM* Simulation geschieht (sofern der Programmator die üblichen *TinyOS* Konventionen befolgt hat) mit dem Befehl *make pc*. Um aber auch das CPU-Profilieren zu aktivieren, ist folgender Befehl nötig

```
OPTFLAGS="-g -O0" /opt/tinyos-1.x/tools/scripts/PowerTOSSIM/compile.pl.
```

Dieses von den *PowerTOSSIM*-Entwicklern bereitgestellte Skript analysiert den Quellcode zunächst und zerlegt ihn in “basic blocks”. Dann kompiliert es das Programm in ein Binary für die Knotenhardware, welches daraufhin analysiert wird, welche Assemblerbefehle in welchem “basic block” erscheinen und wie viele CPU-Zyklen diese zur Ausführung benötigen. Anschließend wird der Programmcode mit Zählern für jeden “basic block” instrumentiert und in ein Simulatorbinary übersetzt. Während der Simulation kann dann gezählt werden, wie oft jeder “basic block” durchlaufen wird und diese Anzahlen werden mit den CPU-Zyklen multipliziert, die der jeweilige Block auf dem Knoten benötigen würde. Dadurch erhält man eine gute Abschätzung des CPU-Verbrauchs des Programms. Weitere Details und Analysen zur Exaktheit des Verfahrens kann man in [17] nachlesen.

3.2.2 “Tracer”

Der “Tracer” ist ein Javaprogramm, welches manuell von der Kommandozeile aus gestartet werden muss. Eine Integration in das Browserinterface des “Viewers” ist aber denkbar (siehe Abschnitt 5).

Parameter Die wichtigsten Parameter sind die ID des zu simulierenden Szenarios, der Pfad zum zuvor erstellten Simulatorbinary und die gewünschte Simulationsdauer. Es können wenn nötig Parameter direkt an den Simulator durchgereicht werden und um die Wiederholbarkeit von Simulationen sicherzustellen, kann optional der “random seed” des Zufallsgenerators des

Simulators angegeben werden. *PowerTOSSIM* erlaubt es, die Ausgabe von Tracedaten nach vorgegebenen Kategorien an- oder abzuschalten. Die für die Energie- und Funkauswertung nötigen Kategorien werden immer aktiviert und alle anderen sind aus Performancegründen normalerweise deaktiviert. Aber der Anwender kann auf Wunsch zusätzliche Meldungen aktivieren, um diese gegebenenfalls mit eigenen Programmen auszuwerten. Daneben gibt es noch zwei Parameter, die sich auf das Radiomodell und die Sensoreninputs beziehen und auf die ich in den entsprechenden folgenden Paragraphen näher eingehen werde. Schließlich kann der Nutzer eine Liste mit zu ladenden Plugins angeben. Plugins erlauben es, den "Tracer" auf einfache Weise zu erweitern, und z.B. Simulationsereignisse direkt während der Simulation zu verarbeiten und mit dem Simulator zu interagieren.

Simulationsinteraktion Der "Tracer" startet den Simulatorprozess, liest die nötigen Eingabedaten aus der Datenbank und schreibt die Tracedaten in die Datenbank zurück. Um mit dem (*Power*)*TOSSIM*-Simulator zu interagieren, muss ein Programm zwei TCP-Verbindungen zum Simulator aufbauen. Über die Ereignisverbindung schickt der Simulator Nachrichten über sämtliche Simulationsereignisse und über die Kommandoverbindung können Anweisungen an den Simulator geschickt werden. Jedes Ereignis vom Simulator muss vom Steuerungsprogramm bestätigt werden und solange bis die Bestätigung eintrifft, pausiert die Simulation. Dies ist nötig, damit das Steuerungsprogramm rechtzeitig auf Ereignisse, wie z.B. das Lesen von Sensordaten durch das simulierte Programm, reagieren kann. Es bedeutet aber auch, dass bei jedem einzelnen Simulationsereignis Netzwerkverkehr und, sofern nur eine CPU zur Verfügung steht, ein Kontextwechsel zum Steuerungsprogramm stattfinden, wodurch ein nicht zu vernachlässigender Overhead entsteht.

Um die vom Nutzer definierten Sensorinputdaten der Simulation zugänglich zu machen, sucht der "Tracer" in den empfangenen Ereignissen nach Sensordatenanfragen. Es liest aus der Datenbank den für den entsprechenden Knoten, ADC-Port und Zeitpunkt definierten Sensorwert und schickt das entsprechende Kommando zum Setzen des Sensorwertes an den Simulator. Dort bleibt dieser Wert solange bestehen, bis er durch ein neues Kommando überschrieben wird. Da das Sensorlesen-Ereignis erst nach Senden des Schreibkommandos bestätigt wird, die Simulation also solange nicht voranschreitet, ist garantiert, dass das Schreibkommando niemals zu spät kommen kann. Ist kein Inputwert definiert, wird der Sensorinput auf Null gesetzt. Der Wert 65535 ist als Sensorinput ungültig, da er den Simulator veranlasst Zufallswerte als Sensorinput zu verwenden. Er wird deshalb automatisch zu 65534 geändert. Es findet weder im "Tracer" noch im Simulator selbst ein Überprüfung statt, ob die Sensorwerte in einem für die Knotenhardware gültigen Bereich liegen, da dieser davon abhängt, wie breit der

ADC der gerade simulierten Hardware ist. Es ist Aufgabe des Nutzers sinnvolle Sensorinputdaten zu generieren. Während bei den nutzergenerierten Sensordaten in der Datenbank Zeit in Sekunden angegeben wird, wird im Simulator Zeit grundsätzlich in CPU-Ticks seit Simulationsstart gemessen. Die für die Umrechnung nötige CPU-Frequenz ist auf 4 MHz hartkodiert. Dies ist beispielsweise für die *Mica2*-Plattform falsch, da diese mit 7.3 MHz arbeitet. Leider sind die 4 MHz in *PowerTOSSIM* selbst fest verdrahtet, so dass der “Tracer” gezwungen ist, diesen Wert ebenfalls zu verwenden. Sollte *PowerTOSSIM* einmal entsprechend flexibilisiert werden, ist es ein leichtes den “Tracer” entsprechend anzupassen. Normalerweise wird beim Umrechnen der Zeiten Sekunde Null auf den Start der Simulation gemappt, aber um die potentielle Integration mit bestehenden Sensordaten zu ermöglichen, kann der Nutzer einen entsprechenden Offset angeben.

Radiomodell *TOSSIM* simuliert nicht die Ausbreitung von Funkwellen, sondern abstrahiert die Nachrichtenübermittlung als Folge von gesendeten Bits, von denen jedes mit einer gewissen Wahrscheinlichkeit unbeschadet beim Empfänger ankommt oder invertiert empfangen wird. Die Signalstärke ist für alle Bits identisch und simultan gesendete Bits werden Oder-verknüpft empfangen. *TOSSIM* unterstützt zwei Radiomodelle: “Simple” und “Lossy”. In ersterem kann jeder Knoten alle anderen hören und alle Bits werden ohne Fehler übertragen. Der “Tracer” verwendet das “Lossy” Modell. In ihm befinden sich alle Knoten in einem gerichteten Graphen, dem “Lossgraph”, dessen Kanten mit der jeweiligen Bitfehlerwahrscheinlichkeit beschriftet sind. Existiert zwischen zwei Knoten keinerlei Kante, können sie sich nicht hören und auch nicht durch Interferenz stören.

Der Simulator liest den “Lossgraph” zu Beginn der Simulation aus einer Datei, in der pro Zeile eine Kante im Format “KnotenId:KnotenId:Bitfehlerwahrscheinlichkeit” definiert ist. Es ist möglich, ihn zur Simulationslaufzeit durch entsprechende Kommandos zu modifizieren. Mit entsprechenden “Tracer”-Plugins könnte der Nutzer komplexe dynamische Radiomodelle erstellen, aber momentan ist das Radiomodell in Ermangelung solcher Plugins rein statisch.

Die Datei, die den “Lossgraph” definiert, kann auf verschiedene Weise erzeugt werden. Die erste Möglichkeit ist, sie aus den Distanzen der Knoten zueinander zu berechnen. Um aus den Breiten- und Längenangaben der Knoten ihre Distanz zu errechnen, gibt es verschiedene Möglichkeiten[1], z.B. den “Sphärischen Kosinussatz”[10], die Haversine-Formel[18] und die Vincenty-Formel[20]. Um die höchstmögliche Genauigkeit zu erhalten, habe ich die Vincenty-Formel verwendet. Für die Umrechnung von der Knotendistanz in eine Bitfehlerwahrscheinlichkeit, muss eine hardwarespezifische Javaklasse sorgen. *PowerTOSSIM* bringt eine solche für die *Mica2*-Plattform mit. Ausgehend von Messungen der *PowerTOSSIM* Autoren definiert sie für ver-

schiedene Distanzen Fehlerverteilungen und ermittelt aus diesen per Zufalls-generator die konkrete Bitfehlerwahrscheinlichkeit. Außerdem definiert sie eine “Maximale Interferenz”-Distanz. Die vom Nutzer im Simulationssetup definierte maximale Funkreichweite wird auf diesen Wert skaliert und der sich ergebende Skalierungsfaktor wird auf alle Knotendistanzen angewendet. Der Nutzer kann bei Bedarf eigene Klassen bereitstellen und für die Umrechnung verwenden lassen. Die zweite Möglichkeit, den “Lossgraph” zu spezifizieren, ist, eine bei einer vorherigen Simulation erzeugte Datei zu verwenden. Dies ist insbesondere für die exakte Wiederholbarkeit von Simulationen nötig. Schließlich kann der Nutzer auch über den “Viewer” eine selbstgenerierte Datei in die Datenbank laden und diese verwenden lassen.

Nachrichtenklassen Die Ereignisklasse *DebugMsgEvent* beinhaltet (fast) alles was normalerweise im Tracefile einer Simulation erscheinen würde, entspricht also den Ausgaben, die der Simulator auf die Standardausgabe schreibt. Sie beinhaltet diverse Unterkategorien, die man durch Setzen der Umgebungsvariablen `DBG` (oder durch Senden des entsprechenden Kommandos) einzeln an- und abschalten kann. Daneben gibt es diverse andere Ereignisklassen, deren Inhalt nicht im Traceoutput des Simulators erscheint. Jedes Ereignis hat einen Zeitstempel (CPU-Ticks seit Simulationsstart), eine Knoten-ID und weitere klassenspezifische Parameter.

Der einzige zusätzliche Parameter der Klasse *DebugMsgEvent* ist der String “message”. Im Tracefile erscheint nur die Knoten-ID und dieser “message” Parameter, der Zeitstempel ist dort nicht zu sehen. Aus diesem Grund ist der Zeitstempel bei manchen *DebugMsgEvent* Unterkategorien im “message” Parameter wiederholt. Auch die Knoten-ID wird dort manchmal wiederholt und z.B. bei den “CPU_CYCLES” Ereignissen der “power” Unterkategorie ist die generische Knoten-ID falsch und die im “message” Parameter genannte richtig. Bei den Ereignissen der “boot” Unterkategorie, die mitteilen, für welchen Zeitpunkt das Booten eines Knoten gescheduled wurde, ist die generische Knoten-ID ebenfalls falsch, aber im “message” Parameter ist keine Knoten-ID genannt, so dass diese Nachrichten keinem Knoten zugeordnet werden können. Das Format der Debugnachrichten folgt keinen allgemeinen Regeln. Wie schon gesagt, enthalten manche die Knoten-ID oder den Zeitstempel und manche nicht. Der Zeitstempel selbst ist manchmal in CPU-Ticks und manchmal in Sekunden angegeben. Manche tragen den Namen ihrer Unterkategorie als Präfix und manche nicht. Bei den Ereignissen der “am” Unterkategorie, die über versandte Radionachrichten informieren, erscheinen im Tracefile zwei Zeilen: eine der Form “1: Sending message: fff, 4” und in der zweiten stehen per Tabulator eingerückt die einzelnen Bytes der Nachricht in Hexdarstellung. Der “Tracer” erhält anstelle der zweiten Zeile eine Vielzahl von *DebugMsgEvent*-Ereignissen, wobei jedes genau ein Byte der Nachricht enthält und das letzte komplett leer ist. All diese Unregel-

mäßigkeiten machen eine automatische Verarbeitung der Ereignisse in der *DebugMsgEvent* Klasse schwer und zum Teil unmöglich. Außerdem gibt es neben dem Quellcode keinerlei Dokumentation über Syntax und Semantik der einzelnen Debugnachrichten. Aus diesen Gründen gebe ich in Tabelle 6 und 7 nur einen Überblick über die vorhandenen Nachrichtenklassen und erläutere lediglich die Klassen, die mein Programm verwendet, detailliert an entsprechender Stelle.

3.3 Datenbankschema

3.3.1 ER Diagramm

Aus dem ER-Schema in Abbildung 3 lässt sich leicht das Datenbankschema ableiten, welches in Listing 1 im Anhang aufgeführt ist.

Hervorzuheben ist die Tabelle “tossim_messages”, in der sämtliche Ereignisklassen gespeichert werden. Die Abbildung der Hierarchie der Nachrichten geschieht mittels “single table inheritance”. Das heißt, sie enthält sämtliche Attribute aller Nachrichtenklassen und das Attribut “type” gibt an, zu welcher Klasse das jeweilige Tupel gehört. Die OR-Mapping Komponente von *Ruby On Rails* unterstützt diese Verhalten automatisch. Momentan sind lediglich die Klassen *TossimMessage*, *RadioMessageSentMessage*, *DebugMessage*, *PowerMessage* und *BootMessage* implementiert, aber weitere können leicht hinzugefügt werden, sobald Bedarf dafür besteht. Für eine korrekte Verarbeitung der Nachrichten im “Viewer” ist es wichtig, die Reihenfolge, in der der Simulator sie erzeugt hat, rekonstruieren zu können. Das “timestamp” Attribut reicht dafür nicht aus, da oft mehrere Nachrichten zum selben Simulationszeitpunkt eintreffen. Aus diesem Grund verwendet der “Tracer” das Attribut “subtimestamp”, um Nachrichten mit identischem Zeitstempel durchzunummerieren und eine eindeutige Sortierung zu gewährleisten.

Eine weitere Besonderheit des Schemas ist, dass das Attribut “mote” in den Tabellen “tossim_messages” und “energy_aggregates” nicht als Fremdschlüssel auf die “motes” Tabelle genutzt werden kann. Denn es handelt sich um die vom Simulator vergebene Knoten-ID, nicht um die Datenbank-ID des Knotens. Man könnte diese IDs bei der Verarbeitung im “Tracer” konvertieren. Aber da die Knoten-ID auch in den Debugnachrichten selbst enthalten ist, würde man riskieren, einige Vorkommen der Knoten-ID beim Konvertieren zu übersehen, wodurch ein Gemisch von ID-Arten entstünde. Und man würde damit Details über die Art der Speicherung der Daten Bestandteil der Simulationsdaten werden lassen, was ihre mögliche Verarbeitung durch Dritte sehr erschweren würde. Aus diesen Gründen halte ich das momentane Vorgehen insgesamt für vorteilhafter, auch wenn dies bedeutet, dass das Datenbankschema die existierende Beziehung zwischen der Knoten-Entität und den “TOSSIM Message” und “EnergyAggregate” Entitäten nicht wiedergeben kann.

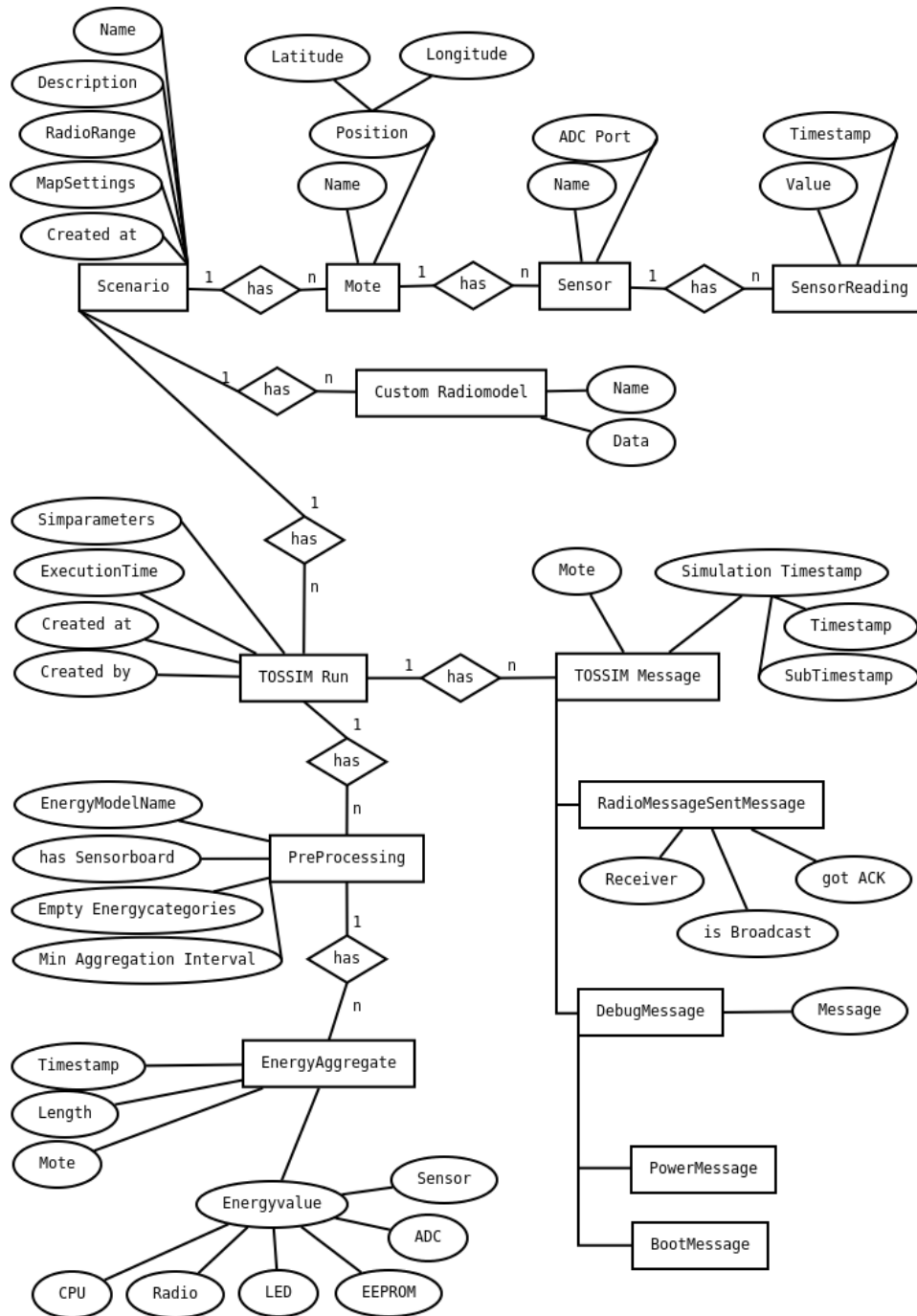


Abbildung 3: ER Diagramm

3.4 Simulationsauswertung

Nach Abschluss der Simulation kann der Nutzer sich die Ergebnisse im “Viewer” darstellen lassen. Zur komfortablen Auswertung werden der Energieverbrauch und die Funkaktivität der Knoten sowohl in textueller als auch in grafischer Form dargestellt. Er kann das zu betrachtende Zeitfenster per Slider auswählen, woraufhin die Darstellung entsprechend angepasst wird. Die Energie eines Knoten wird in der Grafik als Balken dargestellt, wobei die für die drahtlose Kommunikation verbrauchte Energie blau, die von der CPU verbrauchte Energie rot und die verbleibende Energie grün dargestellt werden. Die drahtlose Kommunikation wird durch Linien zwischen den Knoten dargestellt. Es kann gewählt werden zwischen der Anzeige der insgesamt gesendeten, der erfolgreich gesendeten und der nicht erfolgreich gesendeten Nachrichten. Die Dicke der Linien spiegelt die Anzahl der Nachrichten wieder und ihre Skalierung richtet sich immer nach der maximalen Zahl der zwischen zwei beliebigen Knoten verschickten Nachrichten im jeweils betrachteten Zeitfenster. Die mögliche Asymmetrie im Nachrichtenaustausch zwischen zwei Knoten wird abgebildet, indem die von Knoten A ausgehende Hälfte der Linie zwischen A und B die von A nach B gesendeten Nachrichten repräsentiert und die von Knoten B ausgehende Hälfte die von B nach A gesendeten Nachrichten. Ein Beispiel dieser Darstellung ist in Abbildung 4 zu sehen.

Neben dieser statischen Darstellung, kann der Nutzer sich außerdem eine dynamische Echtzeitanimation der Entwicklung von Energieverbrauch und Funkaktivität innerhalb des gewählten Zeitfensters anzeigen lassen. Dies ist in Abbildung 5 dargestellt.

3.4.1 Energiemodell

PowerTOSSIMs Energiemodell erfasst den Energieverbrauch der Knoten in den Kategorien CPU, Funk, LED, EEPROM, Sensor und ADC. Der Energieverbrauch der CPU besteht dabei aus einer Grundlast und einem lastabhängigen Anteil. Die Höhe der Grundlast hängt davon ab, ob sich die CPU im normalen oder einem der fünf Energiesparmodi befindet. Sie fällt immer an, unabhängig davon ob die CPU aktiv Befehle ausführt oder nicht. Der lastabhängige Anteil besteht aus dem zusätzlichen Verbrauch, den jeder CPU Zyklus verursacht, in dem die CPU tatsächlich aktiv ist. Die Sensor- und ADC-Kategorien wurden von den *PowerTOSSIM*-Autoren anfangs als relevant angesehen. Sie stellten aber später fest, dass das an die Knoten anschließbare Sensorboard einen konstanten Energieverbrauch hat und das Auslesen der Sensoren und des ADC bei der *Mica2*-Plattform keine zusätzliche Energie kostet. Momentan existieren diese Kategorien also nur der prinzipiellen Vollständigkeit halber. In Tabelle 1 sind die konkreten Energieverbrauchswerte der Komponenten der *Mica2*-Plattform aufgelistet.

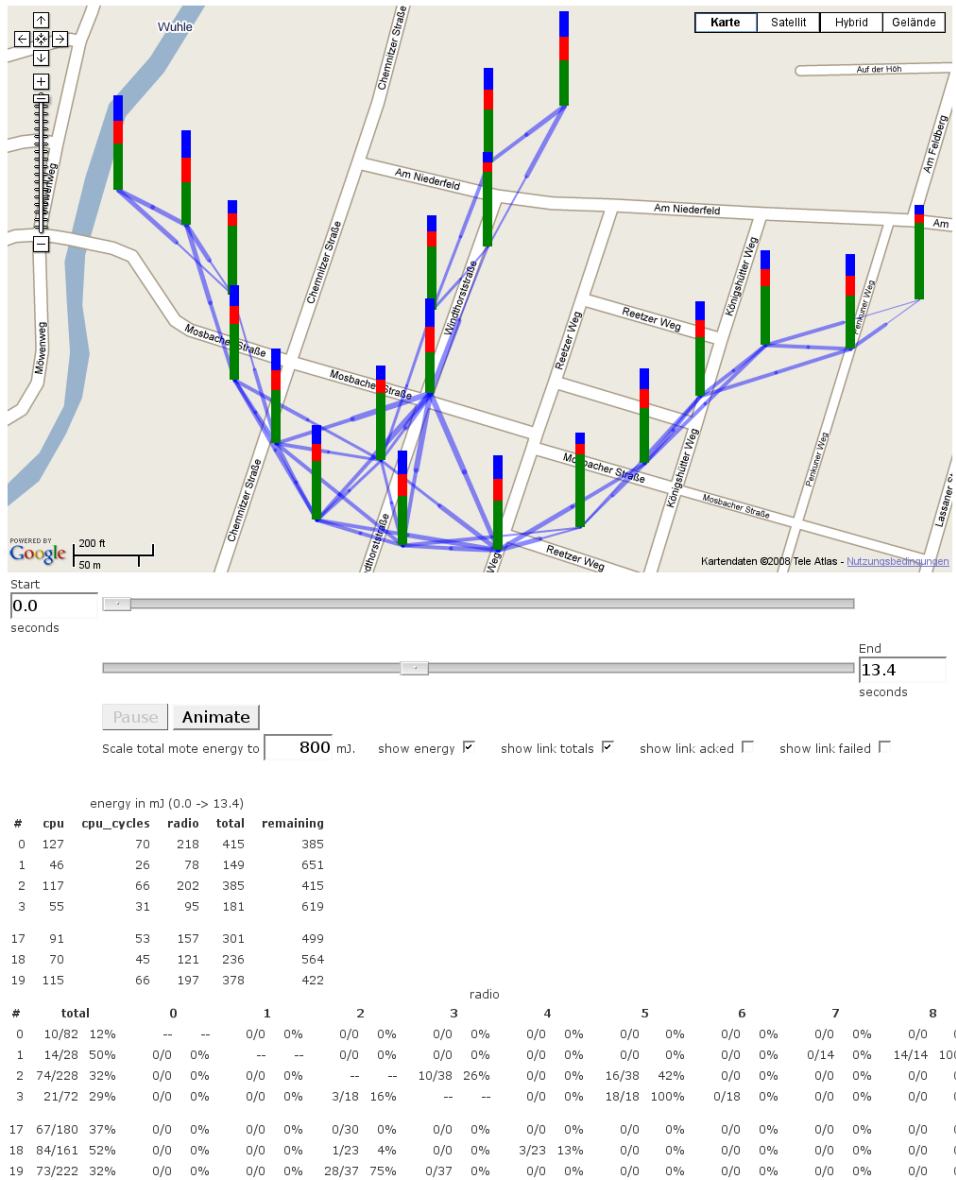


Abbildung 4: Statische Simulationsvisualisierung im "Viewer": Der Nutzer kann sich den Energieverbrauch der Knoten und ihre Funkaktivität in einem frei wählbaren Zeitfenster sowohl in Textform als auch grafisch darstellen lassen.

Komponente	Modus	Leistung in mW
CPU		
Grundlast	idle	12,390
	ADC noise reduction	3,000
	extended standby	0,669
	standby	0,648
	power save	0,330
	power down	0,309
lastabhängiger Teil	pro CPU-Zyklus	26,790
Funk	aus	0
	empfangen	21,09
	senden mit Sendestärke	
	00	11,16
	01	15,63
	03	16,11
	06	19,41
	09	21,15
	0F	25,41
	60	34,71
	80	41,31
	C0	52,11
	FF	64,44
LED	an	6,60
	aus	0
EEPROM	lesen	18,72
	schreiben	55,20
Sensoren	Sensorboard eingesteckt	2,07
	lesen	0
ADC	Zugriff	0

Tabelle 1: Energieverbrauch der *Mica2*-Plattform

Während eines *PowerTOSSIM* Laufs werden energieverbrauchsrelevante Zustandsänderungen als Ereignisse der “power” Unterkategorie der Klasse *DebugMsgEvent* an den “Tracer” übermittelt. Beispiele sind Beginn und Ende von EEPROM-Zugriffen oder der Wechsel des Funkmoduls zwischen Sende- und Empfangsmodus. Außerdem wird alle 0,2 Sekunden für alle Knoten die Anzahl der verbrauchten CPU-Zyklen ausgegeben. Der “Tracer” speichert diese Nachrichten in der Datenbank. Nach der Simulation kann dann aus diesen Daten mithilfe separat definierter konkreter Verbrauchszahlen für die einzelnen Hardwarekomponenten, dem eigentlichen Energiemodell, der Energieverbrauch eines jeden Knoten berechnet werden. Da die Daten über den Knotenzustand unabhängig vom Energiemodell gespeichert werden, kann man es theoretisch leicht austauschen und vorhandene Simulationen mit neuen Energiemodellen auswerten, ohne die Simulation zeitaufwändig erneut durchführen zu müssen. In der Praxis wird dies dadurch eingeschränkt, dass sowohl die Zählung der verbrauchten CPU-Zyklen, als auch die Umrechnung der Knotendistanzen in Bitfehlerwahrscheinlichkeiten im Radiomodell hardware-spezifisch sind, die Simulationsdaten also doch implizit Informationen über die Knotenhardware beinhalten. Ein Nachteil an diesem Verfahren ist, dass der Simulator zur Laufzeit nichts über die verbleibende Energie eines Knotens weiß. In der Simulation können Knoten also niemals wegen Energiemangels ausfallen.

Um den Energieverbrauch der Knoten wirklich 100%ig exakt berechnen zu können, muss man neben den “power” auch die “boot” Debugnachrichten auswerten. Die Knoten werden nämlich nicht alle gleichzeitig zu Beginn der Simulation gestartet, sondern per Zufallsgenerator innerhalb der ersten zehn Sekunden gebootet. Das zu *PowerTOSSIM* gehörende Auswertungsskript betrachtet einen Knoten als aktiv, sobald es die erste “power” Nachricht für ihn sieht. Die verbrauchten CPU-Zyklen werden aber alle 0,2 Sekunden für alle Knoten ausgegeben, unabhängig davon, ob diese bereits gebootet wurden oder nicht. Und leider kommt es manchmal vor, dass für noch inaktive Knoten bereits eine geringe Anzahl von verbrauchten CPU-Zyklen angegeben wird, meist 46,5. Dadurch werden diese Knoten zu früh als eingeschaltet betrachtet und sie verbrauchen bereits Energie für die CPU-Grundlast und das Funkmodul, obwohl sie eigentlich noch ausgeschaltet sind. Für längere Simulationszeiträume sind die resultierenden Abweichungen sicherlich zu vernachlässigen, aber um auch bei kurzen Simulationszeiträumen exakte Werte zu erhalten, muss man die “boot” Debugnachrichten auswerten und dadurch den wahren Einschaltzeitpunkt eines Knoten ermitteln.

3.4.2 Vorverarbeitung der Energiedaten

Um die Simulationsergebnisse mit akzeptabler Performance darstellen zu können, ist es notwendig, die “power” Debugnachrichten vorzuvorbereiten und zu aggregieren (siehe Abschnitt 4.2). Dabei werden alle “power” Nach-

richten verarbeitet und pro Knoten der Energieverbrauch pro Zeitintervall für jede Energiekategorie gespeichert. Die minimale Länge dieses Zeitintervalls ist momentan eine Sekunde, aber der Nutzer kann diesen Wert ändern, falls er z.B. bei sehr langen Simulationen die Geschwindigkeit der Animation erhöhen will und keine sekundengenaue Auswertung braucht. Nach Ablauf des minimalen Zeitintervalls wird gewartet, bis eine "power" Nachricht eintrifft, die den Stromverbrauch des Knotens ändert. Wird diese Änderung nicht noch im selben Augenblick wieder rückgängig gemacht (dies geschieht beispielsweise bei Lesezugriffen auf den ADC, die im Simulator verzögerungsfrei sofort ein Ergebnis zurückliefern), so wird das aktuelle Energieaggregat abgeschlossen und ein neues begonnen. Auf diese Weise wird verhindert, dass für Zeiten, in denen am Knoten keinerlei Aktivität stattfindet, eine große Zahl leerer und nutzloser Aggregate erzeugt werden. Um Rundungsfehler zu vermeiden, werden die in mJ vorliegenden Energiewerte in μJ gespeichert.

Ein Problem bilden dabei die "CPU_CYCLES" Ereignisse. Diese werden vom Simulator alle 0,2 Sekunden ausgegeben, unabhängig davon, ob am Knoten Aktivität herrscht oder nicht. Deshalb werden diese Ereignisse beim Festlegen der Aggregatsgrenzen ignoriert. Das führt möglicherweise dazu, dass der Energieverbrauch durch die CPU zu sehr gleichmäßig wird. Aber die CPU-Aktivität sollte in Perioden, in denen ansonsten keinerlei Ereignisse am Knoten stattfinden, relativ gleichmäßig sein.

3.4.3 Funknachrichten

Der Versand von Funknachrichten erfolgt bei der *Mica2*-Plattform, indem ein Startsymbol gefolgt von Nachrichtenkopf und -körper gesendet werden. Hat der Empfänger die Nachricht korrekt empfangen, sendet er sofort nach Empfang eine Bestätigung. Für jede versandte Nachricht wird vom Simulator ein "RadioMessageSentEvent" erzeugt, das unter anderem Informationen über Sender, Empfänger und darüber, ob eine Bestätigung empfangen wurde, enthält. Diese Daten werden in der Datenbank für die spätere Auswertung gespeichert.

Ein Problem bilden Broadcasts, denn die Bestätigung einer Funknachricht enthält keinerlei Informationen darüber, welcher Knoten die Nachricht bestätigt. Das heißt, dass ein Broadcast dann als bestätigt gilt, wenn wenigstens ein Empfänger eine Bestätigung sendet. Um die Genauigkeit der Auswertung an dieser Stelle zu verbessern, kann die "am" Unterkategorie der *DebugMsgEvent* Ereignisklasse ausgewertet werden. Hier teilen Knoten jede gesendete und empfangene Funknachricht mit. Unter Zuhilfenahme des "Lossgraph" lässt sich dadurch feststellen, ob ein Knoten einen an ihn gesendeten Broadcast korrekt empfangen hat oder nicht. Dafür wird jede Nachricht, die ein Knoten versendet, bei allen potentiellen Empfängern vermerkt. Empfängt ein Knoten nun eine Funknachricht, wird diese mit allen Nachrichten verglichen, die momentan an ihn unterwegs sind, und die entsprechende

Nachricht wird geeignet markiert. Hat der Sender die Nachrichtenübermittlung abgeschlossen, wird wiederum bei allen potentiellen Empfängern überprüft, ob diese Nachricht inzwischen als empfangen markiert wurde. Das zugehörige “RadioMessageSentEvent” wird dann für jeden einzelnen Empfangsknoten dupliziert in der Datenbank gespeichert, jeweils mit der Information, ob der betreffende Knoten die Nachricht korrekt empfangen hat oder nicht.

Da zwischen zwei Knoten zu jedem Zeitpunkt maximal eine Nachricht im Transit sein kann, funktioniert dieses Verfahren exakt genug. Es ist zwar theoretisch vorstellbar, dass zwei Knoten zu (beinahe) gleichen Zeitpunkten die exakt gleiche Nachricht an einen dritten Knoten schicken, wodurch die Zuordnung von empfangener zu gesendeter Nachricht nicht mehr korrekt funktionieren würde. Solche Fälle sollten aber extrem selten sein und sie sind völlig ausgeschlossen, wenn beim verwendeten Kommunikationsprotokoll die Knoten-ID des Senders Bestandteil der Nachricht ist, denn dann ist es nicht mehr möglich, dass zwei verschiedene Knoten identische Nachrichten verschicken. Letztlich bleibt noch das Problem, dass ein Empfänger eine Nachricht korrekt empfängt, aber seine Bestätigung verloren geht und Sender und Empfänger somit unterschiedlicher Meinung über den Erfolg der Transmission sind. Mein Algorithmus würde die Übertragung in einem solchen Fall als erfolgreich betrachten, obwohl sie eigentlich “halb-erfolgreich” war. Aber ein solcher dritter Zustand ist in Auswertungen normalerweise nicht vorgesehen und deshalb kann dieses grundsätzliche Problem auch nicht vom “Tracer” gelöst werden.

Unglücklicherweise erzeugt der Simulator für jedes verschickte Byte ein separates “am” Debugereignis, wodurch das beschriebene Vorgehen die Gesamtzahl an generierten Ereignissen um etwa den Faktor fünf ansteigen lässt und sich die Ausführungszeit der Simulation mehr als verdoppelt. Alternativ kann man die “crc” Unterkategorie auswerten, in der die Prüfsummen von versandten und empfangenen Paketen mitgeteilt werden. Das Vorgehen ist dasselbe, nur wird statt dem Inhalt der Nachrichten nur noch ihr CRC-Wert verglichen. Dies ist potentiell ungenauer, vermindert aber auch die Simulatorperformance in weit geringerem Maße. Letztendlich muss der Nutzer entscheiden, ob der Performancegewinn die potentielle Ungenauigkeit rechtfertigt.

4 Messungen

Zur Evaluierung der geschaffenen Simulationsumgebung werden Simulator und “Tracer” auf ihre Skalierbarkeit hin untersucht und verschiedene Implementationsvarianten für den “Tracer” hinsichtlich ihrer Auswirkung auf die Performance des “Tracers” bewertet. Außerdem wird auf die Notwendigkeit zur Vorverarbeitung der Energiedaten (siehe Abschnitt 3.4.2) eingegangen, es werden Implementationsmöglichkeiten abgewogen und die Skalierbarkeit der momentanen Implementierung untersucht. Abschließend wird näher auf die Möglichkeiten zur Umsetzung der grafischen Visualisierung eingegangen und es werden die Performanceauswirkungen der verschiedenen Alternativen insbesondere im Hinblick auf die Echtzeitanimation erläutert.

4.1 Simulator und “Tracer”

Um die praktische Verwendbarkeit der Simulationsumgebung bewerten zu können, wird untersucht, wie gut Simulator und “Tracer” mit der Größe und Komplexität der Simulation skalieren und welche Grenzen sich für die mögliche Größe der Simulation aufgrund der zur Simulationsdurchführung benötigten Zeit und der Menge der entstehenden Daten ergeben. Dazu wurden Simulationen mit verschiedenen Knotenmengen und Simulationslängen durchgeführt und ihre zeitliche Performance analysiert.

Um den durch den “Tracer” verursachten Overhead beurteilen und die Vorzüge bzw. Kosten einiger Implementationsaspekte abwägen zu können, wurden dieselben Simulationen auch mit modifizierten Versionen des “Tracers” durchgeführt und deren zeitliches Verhalten verglichen.

4.1.1 Messaufbau

Die Messungen wurden für verschiedene Szenarien mit 5, 10, 20, 50 und 100 Knoten und Simulationszeiträumen von 10, 60, 300, 600 und 1800 Sekunden durchgeführt.

Als zu simulierendes Programm wurde das *TOSSIM* beiliegende Beispielprogramm “SenseToRfm” gewählt. Es installiert einen Timer, der alle 250ms veranlasst, dass der Photosensor ausgelesen und der gemessene Wert per Broadcast verschickt wird. Es beinhaltet also genau die zwei Grundfunktionen eines “echten” Programms, das periodische Auslesen der Sensoren und das drahtlose Verschicken von Sensordaten, und ist deshalb gut geeignet realitätsnahe Messwerte zu produzieren, mit denen man verschiedene Implementationsvarianten bewerten und die Skalierbarkeit des Gesamtsystems ermitteln kann. Man sollte aber beim Betrachten der Daten bedenken, dass “echte” Programme vermutlich wesentlich komplexere Algorithmen abarbeiten werden, weshalb die hier ermittelten absoluten Messwerte als etwas optimistisch angesehen werden sollten.

Alle Messungen wurden auf dem in Abschnitt 3 beschriebenen System durchgeführt. Der PC wurde während der Messungen nicht anderweitig genutzt und es waren insbesondere der *cron-daemon* und die dynamische Anpassung der CPU-Frequenz abgeschaltet, um eine Verfälschung der Messwerte zu vermeiden. Außerdem stellte sich zu Beginn der Messungen heraus, dass es nicht nötig ist, den “Tracer” und den *PowerTOSSIM*-Simulator innerhalb der *VMWare*-Instanz, in der sie installiert sind, zu betreiben. Stattdessen wurde das Dateisystem der *VMWare*-Instanz per *NFS* im eigentlichen PC gemountet, was es erlaubte, “Tracer” und Simulator außerhalb der Virtualisierung mit deutlich höherer Performance zu betreiben.

Bei allen Messungen wurde der Zufallsgenerator des Simulators mit demselben Wert initialisiert. Außerdem wurde bei allen Messungen zu einem bestimmten Szenario immer derselbe “Lossgraph” verwendet. Dies garantiert, dass die Simulationen vergleich- und wiederholbar sind.

Beim Bewerten der kleineren Simulationszeiträume sollte man berücksichtigen, dass die Knoten vom Simulator zu zufälligen Zeitpunkten innerhalb der ersten zehn Sekunden der Simulation gebootet werden. Insbesondere die Messungen über zehn Sekunden weichen deshalb z.B. bei der Anzahl der Simulationsereignisse pro Knoten und Zeiteinheit deutlich von den anderen Messungen ab. Bei den größeren Simulationszeiträumen fällt diese Anfangsphase dann kaum noch ins Gewicht.

Die verschiedenen Szenarien sind nicht direkt miteinander vergleichbar, da es schwierig ist, die Knoten jeweils mit identischem Verteilungsmuster zu platzieren. Bei den größeren Knotenzahlen liegen die Knoten eher dichter gedrängt, weshalb die Anzahl der von einem Knoten per Funk erreichbaren Knoten dort im Mittel größer ist. Deshalb sollte man beim Vergleich von Daten verschiedener Szenarien die Anzahl von Simulationsereignissen pro Knoten und Zeiteinheit als Skalierungsfaktor benutzen.

Um Optimierungsmöglichkeiten und -grenzen einzelner Bestandteile des “Tracers” aufzuzeigen, wurden die Messungen für verschiedene Ausbaustufen durchgeführt:

- “no tracer”: Zunächst wurde nur der reine *PowerTOSSIM*-Simulator gemessen. Diese Daten bilden die natürliche Grenze für Komplexität und Umfang der möglichen Simulationen.
- “nop tracer”: Hier wurde ein “Tracer” gemessen, der alle Simulationsereignisse einfach nur sofort bestätigt und nichts weiter tut. Aus diesen Daten ergeben sich die Kosten für die notwendige TCP-gestützte Interaktion mit dem Simulator.
- “ro tracer”: Der “Tracer” verarbeitete die Simulationsdaten hier zwar noch nicht, aber er fütterte die Simulation schon mit den in der Datenbank definierten Sensorinputs. Dies zeigt z.B., ob es sich lohnen würde,

den momentan verwendeten, relativ einfachen Cachingalgorithmus zum Einlesen der Sensordaten noch zu verbessern.

- “no brc tracer”: Diese “Tracer” Version entsprach im Wesentlichen der “current” Variante. Lediglich das in Abschnitt 3.4.3 beschriebene Analysieren von Broadcasts auf Basis von “crc” Ereignissen war nicht aktiv, um zu ermitteln, wie viel Performance dieses Feature kostet.
- “current”: Dann kam die aktuelle “Tracer” Version zum Einsatz, welche die Sensorinputs in die Simulation einspeist, Broadcasts auf Basis von “crc” Ereignissen analysiert und sämtliche Simulationsereignisse in die Datenbank speichert. Das Schreiben der Simulationsdaten in die Datenbank erfolgt asynchron in einem zweiten Thread, wodurch der Hauptthread bei der Interaktion mit dem Simulator entlastet wird und Simulationsergebnisse schneller bestätigen kann. Außerdem werden die Daten nicht einzeln in die Datenbank geschrieben, sondern gesammelt per Batchinsert zur Datenbank geschickt.

Um die Grenzen und Skalierungsfähigkeit des Systems besser bestimmen zu können, wurde dieser Messschritt zusätzlich mit einem Szenario mit 150 Knoten und außerdem mit einer weiteren Simulationszeit von 3600 Sekunden durchgeführt.

- “am brc tracer”: In der “current” Version wurde der “crc” basierte Broadcastanalytiker durch die auch in Abschnitt 3.4.3 beschriebene “am” basierte Variante ersetzt, um diese zwei Implementationsmöglichkeiten miteinander vergleichen zu können.
- “sync write tracer”: In dieser Variante schreibt der Hauptthread jedes Simulationsereignis einzeln in die Datenbank. Man kann also den Performancegewinn ermitteln, den die Verwendung von Batchinserts und eines zweiten Threads in der “current” Variante erzielt.
- “vmware”: Der “current” “Tracer” wurde schließlich innerhalb der *VMWare*-Instanz ausgeführt, um zu ermitteln, wie teuer die Virtualisierung ist.

4.1.2 Messergebnisse

Rohdaten Aufgrund des Umfangs der Rohmessdaten befinden sich diese im Anhang in Tabelle 9.

Filterung von “adc”, “crc” und “am” Debugnachrichten Zum Erfüllen seiner Funktion muss der “Tracer” zwingend einige Unterkategorien der Ereignisklasse *DebugMsgEvent* aktivieren. “boot” und “power” werden für die anschließende Auswertung gebraucht. Aber “adc” wird nur benötigt, um den

Simulator mit den Sensorinputdaten versorgen zu können (siehe Abschnitt 3.2.2), und “crc” bzw. “am” wird nur benötigt, um vom Knoten gesendete Broadcasts den einzelnen Empfängern zuordnen zu können (siehe Abschnitt 3.4.3). Um Festplattenplatz und Verarbeitungszeit zu sparen, werden diese Nachrichten nach ihrer “Tracer” internen Verarbeitung aus dem Datenstrom ausgefiltert und nicht in die Datenbank geschrieben, es sei denn, der Nutzer hat diese Unterkategorien beim Starten des “Tracers” explizit aktiviert. Aufgrund der in Abschnitt 3.2.2 beschriebenen Probleme beim Klassifizieren von Debugnachrichten kann nicht ausgeschlossen werden, dass manche Nachrichten durch den Filter rutschen, aber in den getesteten Szenarien funktionierten sie einwandfrei.

Wie man aus den “current” Messdaten in Tabelle 11 sehen kann, bilden die “adc” Nachrichten im Mittel 5% und die “crc” Nachrichten 18% der von der Simulation insgesamt erzeugten Ereignisse. Ihr Ausfiltern lohnt sich also durchaus und reduziert die Zahl der in die Datenbank geschriebenen Daten spürbar. Was man an den Messdaten außerdem sieht, ist, dass 0,005% der in die Datenbank geschriebenen *DebugMsgEvent* Ereignisse nicht klassifiziert werden konnten und deshalb den allgemeinen Typ “DebugMessage” bekamen. In den untersuchten Fällen sind dies immer Nachrichten der “boot” Unterkategorie gewesen, die z.B. die Form “PHOTO initialized.” haben. Aufgrund ihrer geringen Anzahl lohnt es sich nicht, sie gesondert zu filtern.

4.1.3 Skalierbarkeit und Grenzen des Systems

Im Folgenden wird basierend auf den Daten der “current” Messreihe die Skalierbarkeit der Simulation in Bezug auf die Länge der simulierten Zeit, die Anzahl der simulierten Knoten und die Anzahl an verarbeiteten Simulationseignissen betrachtet. Anschließend wird jeweils näher auf die Grenzen des Systems eingegangen, die sich aus der für die Simulation benötigten Zeit, der zu verarbeitenden Zahl von Simulationseignissen und des benötigten Festplattenplatzes ergeben.

Simulationsdauer In Abbildung 6 ist zu sehen, dass die Simulationsdauer für jedes der sechs Szenarien linear mit der Menge der simulierten Zeit skaliert. Der konkrete Skalierungsfaktor hängt natürlich von der Anzahl der Knoten ab. Bei 20 Knoten ist die Simulation mit 0,6 Sekunden pro simulierter Sekunde noch schneller als Echtzeit, bei 50 Knoten benötigt man bereits zwei Sekunden für jede simulierte Sekunde und bei 150 Knoten 6,5 Sekunden. Je nach Geduld des Nutzers ist die maximal mögliche Simulationslänge dementsprechend begrenzt. Auf eine Woche simulierter Zeit müsste man z.B. mit dem 150 Knoten Szenario bereits anderthalb Monate warten, was in der Praxis vermutlich nicht mehr akzeptabel ist.

Diese Aussagen hängen natürlich vom simulierten Programm ab. Das Verhalten des verwendeten Testprogramms ist über die Zeit konstant. Ver-

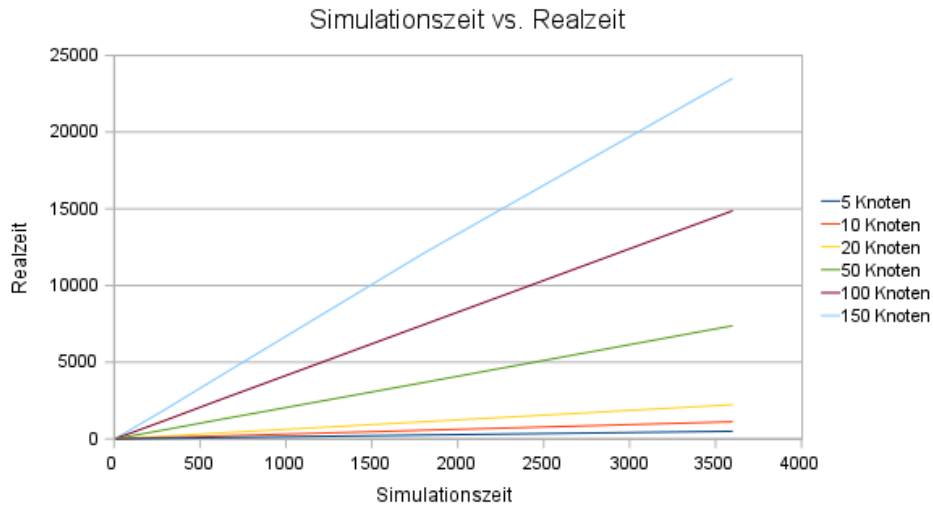


Abbildung 6: Simulationsdauer in Abhängigkeit der simulierten Zeit

wendet man ein Programm mit anderem zeitlichen Verhalten, bei dem beispielsweise das simulierte Sensordatentransportprotokoll so beschaffen ist, dass die Zahl der Transmissionen zwischen den Knoten quadratisch mit der simulierten Zeit wächst, dann wird auch die Simulationsdauer wenigstens quadratisch mit der Menge der simulierten Zeit wachsen.

In Abbildung 7 ist zu sehen, wie die Simulationsdauer für jede der sechs Simulationszeiten mit der Menge der simulierten Knoten skaliert. Auch wenn es auf den ersten Blick nach einer linearen Abhängigkeit aussieht, ist dies nicht der Fall, wie man in Abbildung 8 sieht, in dem für jede der sechs Simulationszeiten die Entwicklung der Simulationsdauer pro Knoten abgebildet ist. Der Zeitaufwand pro Knoten ist nicht konstant, sondern steigt mit zunehmender Knotenzahl. Für die meisten Simulationszeiten erhöht sich dieser Wert vom “5 Knoten” Szenario zum “150 Knoten” Szenario um den Faktor 1,5. Dies ist, wie schon weiter oben beschrieben, darauf zurückzuführen, dass die Knoten in den größeren Szenarien dichter liegen und dadurch die Broadcasts eines Knoten im Schnitt mehr andere Knoten erreichen.

Schließt man solche außerhalb des “Tracers” liegende Einflüsse aus, um die Zahlen von Simulation 36 (150 Knoten, 3600 Sekunden) linear hochzurechnen, so kann man, bei einer Begrenzung der Simulationsdauer auf eine Woche, die Knotenzahl auf über 3800 erhöhen.

Schließlich ist in Abbildung 9 die Entwicklung der Simulationsdauer in Abhängigkeit von den erzeugten Simulationsereignissen dargestellt. Neben den absoluten Werten ist auch die (geeignet skalierte) Ableitungsfunktion eingezeichnet. Zur besseren Visualisierung sind beide Achsen logarithmisch skaliert. Das Daten zeigen, dass die Simulationsdauer linear mit der Anzahl

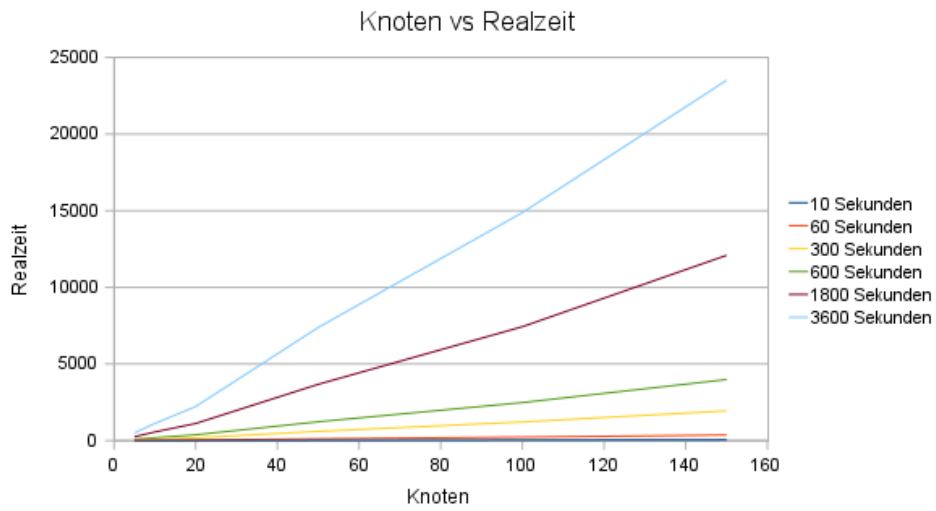


Abbildung 7: Simulationsdauer in Abhängigkeit der simulierten Knoten

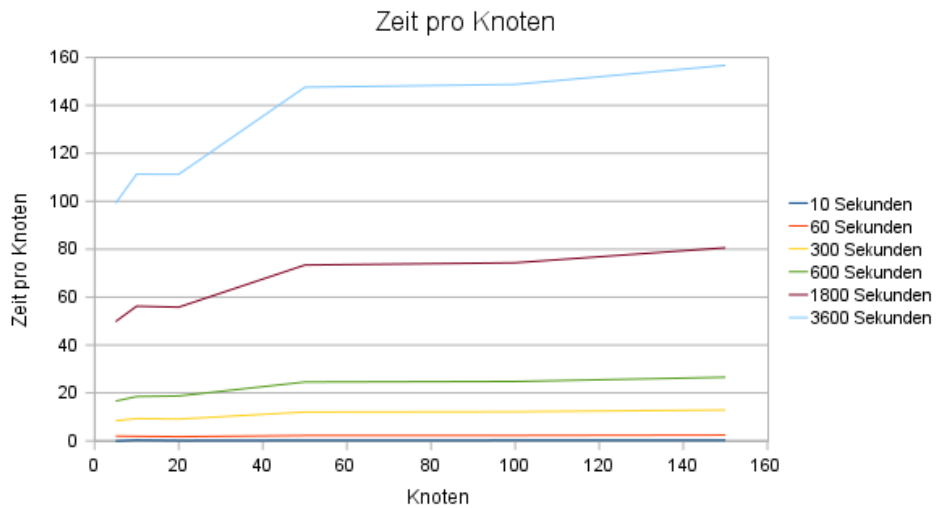


Abbildung 8: Simulationsdauer pro simuliertem Knoten

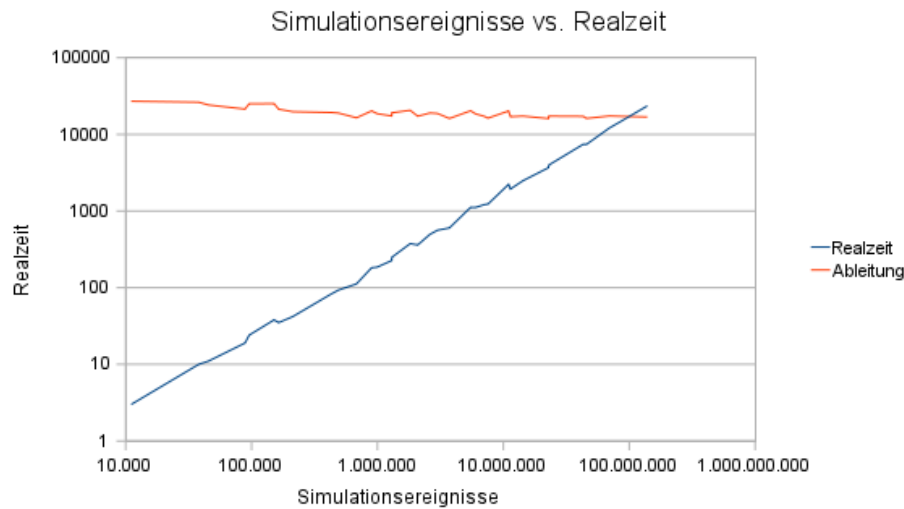


Abbildung 9: Simulationsdauer in Abhängigkeit der erzeugten Simulationsergebnisse

der Simulationsergebnisse skaliert.

Simulationsdatenmenge In den Abbildungen 10 und 11 kann man sehen, dass die Anzahl der in die Datenbank geschriebenen Simulationsergebnisse bzw. die Größe der Datenbank linear von der Simulationszeit abhängen. Die Abweichung bei den Simulationszeiten von 10 und 60 Sekunden liegen, wie schon erwähnt, daran, dass die Knoten über die ersten zehn Sekunden verteilt booten. Bei den größeren Simulationszeiten fällt diese Anfangsphase nicht mehr ins Gewicht und der lineare Zusammenhang tritt deutlich hervor.

Beim 150 Knoten Szenario entstehen pro simulierter Sekunde 7 MB an Daten in Form von 35.000 Simulationsergebnissen. Der verfügbare Festplattenplatz begrenzt entsprechend die Größe der möglichen Simulationen und die Datenmenge hat auch Auswirkungen auf die zur weiteren Verarbeitung der Daten nötige Zeit. Würde man mit diesem Szenario beispielsweise eine Woche simulieren, entstünden 4134 GB an Daten in Form von über 21 Milliarden Ereignissen. Dies übersteigt die Kapazitäten eines heute üblichen Rechners deutlich und es ist auch klar, dass die weitere Verarbeitung solcher Datenberge sehr zeitaufwändig sein wird.

In den Abbildungen 12 und 13 ist der Zusammenhang zwischen der simulierten Knotenzahl und den Simulationsergebnissen bzw. der Datenbankgröße dargestellt. Genau wie bei der Simulationsdauer ist auch dieser Zusammenhang nicht linear und zwar aus analogen Gründen.

Um die Grenzen des Systems abzuschätzen, muss man wieder einen linearen Zusammenhang annehmen. Geht man dann von den Daten von Simula-

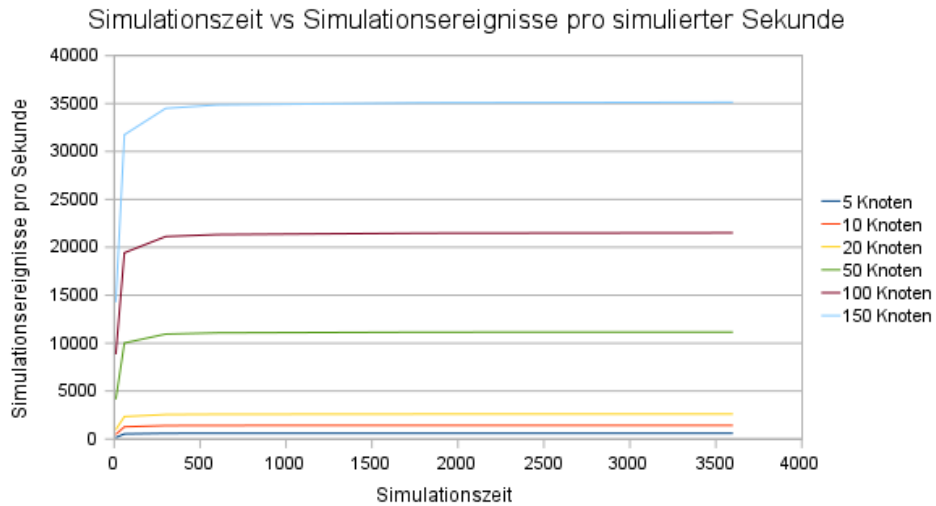


Abbildung 10: Datenbanksimulationsereignisse pro simulierter Sekunde

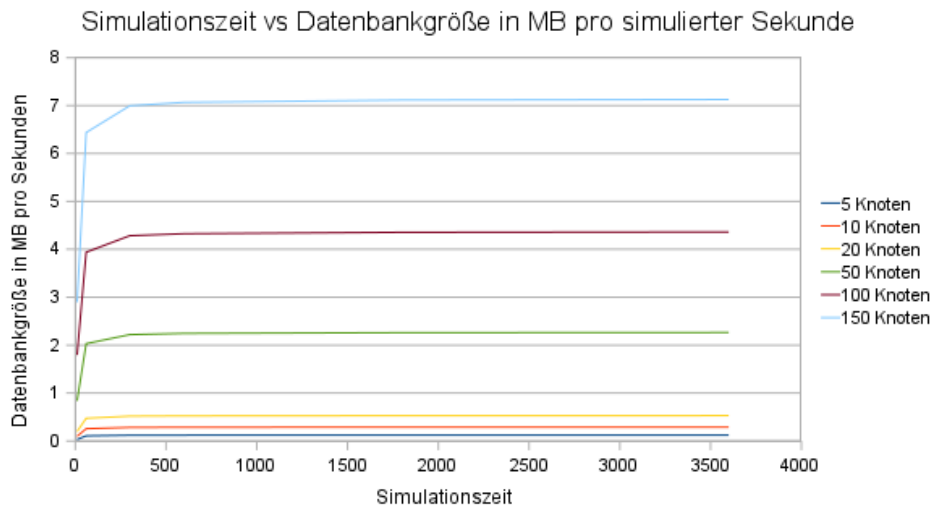


Abbildung 11: Datenbankgröße in MB pro simulierter Sekunde

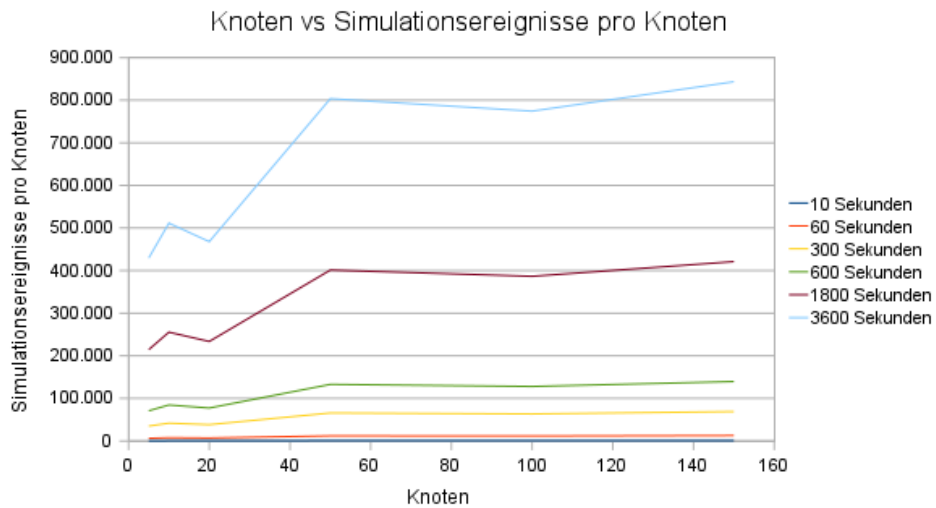


Abbildung 12: Datenbanksimulationsereignisse pro Knoten

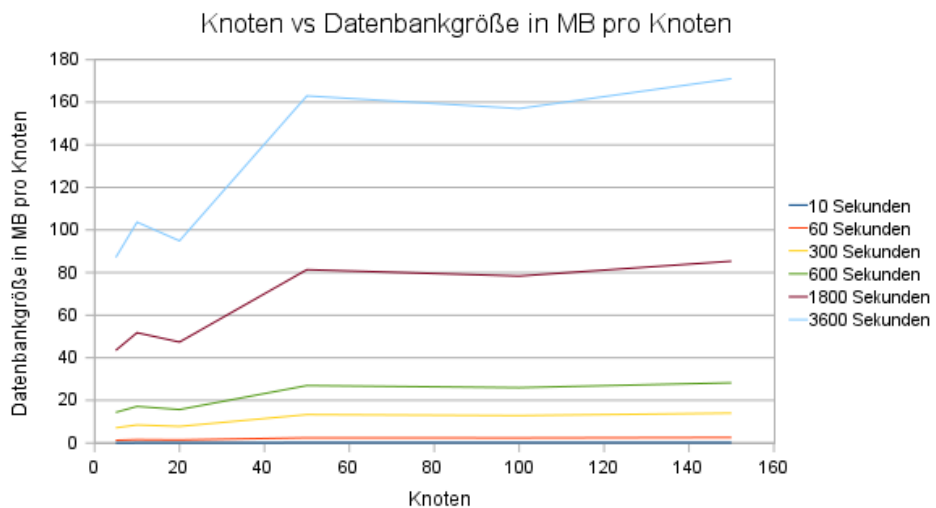


Abbildung 13: Datenbankgröße in MB pro Knoten

tion 36 (150 Knoten, 3600 Sekunden) aus und begrenzt die Datenbankgröße auf 100 GB so lassen sich knapp 600 Knoten simulieren.

Rechnerressourcen Darüber wie die Simulationsdauer mit der Menge an Hauptspeicher oder der Zahl und der Geschwindigkeit der CPUs skaliert, können die Messungen keine Aussagen geben, da sie ja alle auf demselben Rechner durchgeführt wurden.

Es lässt sich nur festhalten, dass die 4 GB Hauptspeicher bei keiner Messung auch nur annähernd ausgenutzt wurden. Dies ist auch nicht verwunderlich, da die zu simulierenden Speicherressourcen eines Knotens im Verhältnis zu denen einer typischen Workstation verschwindend gering sind, weshalb der Hauptspeicher wohl nie ein limitierender Faktor sein wird.

Die Festplattengeschwindigkeit dürfte ebenfalls keinen Einfluss auf die Simulation haben, da die während der Simulation anfallende Datenmenge im Verhältnis zur Simulationsdauer viel zu gering ist. Bei Simulation 36 (150 Knoten, 3600 Sekunden) musste beispielsweise etwas mehr als 1 MB/s in die Datenbank geschrieben werden, was weit unter den Möglichkeiten heutiger Festplatten liegt. Die Bandbreite der Verbindung zwischen Simulator und “Tracer” dürfte aus ähnlichen Gründen ebenfalls unkritisch sein. Dies gilt aber nur, solange man immer nur eine Simulation auf einmal laufen lässt. Liefen genügend Simulationen parallel in die Datenbank schreiben, käme es irgendwann zu Performanceengpässen.

Die Latenz der Verbindung zwischen Simulator und “Tracer” bzw. “Tracer” und Datenbank kann hingegen entscheidenden Einfluss auf die Performance des Gesamtsystems haben. Bei Zwischentests während der Systementwicklung waren der “Tracer” und die Datenbank zwischenzeitlich durch eine WLAN-Verbindung getrennt und die Performance brach daraufhin um mehr als den Faktor fünf ein.

Man kann davon ausgehen, dass die Simulationsperformance linear mit der Prozessorgeschwindigkeit skaliert. Der Simulator selbst ist komplett single-threaded, profitiert also nicht von zusätzlichen CPUs. Im Gesamtsystem kann eine zweite CPU aber durchaus von Nutzen sein, da der “Tracer” und die Datenbank dann den Simulator nicht behindern. Während der gesamten Messreihen waren die zwei CPUs aber niemals beide völlig ausgelastet, weshalb zu vermuten ist, dass weitere CPUs keinen entscheidenden Vorteil mehr bringen würden.

4.1.4 Bewertung einzelner Implementationsaspekte

Um die verschiedenen Messreihen miteinander vergleichen zu können, wurden die Daten der einzelnen Simulationen pro Messreihe zusammenaddiert und in den Tabellen 2 und 3 dargestellt.

Für die “no tracer”, “nop tracer” und “ro tracer” Messreihen fehlen die Anzahlen der Simulationsergebnisse. Für erstere könnte man zwar die Zeilen in

Messreihe	WallTime in		prozentualer Vergleich	
	Sekunden	Minuten		
no tracer	17.676	294,6	100	
nop tracer	18.870	314,5	106,8	
ro tracer	18.909	315,2	107,0	
no brc tracer	18.862	314,4		94,5
current	19.955	332,6	112,9	100
am brc tracer	39.456	657,6		197,7
sync write tracer	55.729	928,8		279,3
vmware	59.226	987,1		296,8

Tabelle 2: Messergebnisse zusammengefasst pro Messreihe (Zeiten)

Messreihe	Simulationsereignisse			Festplatten- speicher	
	vom Simulator	in Datenbank	pro Knoten- sekunde	in MB	in GB
no tracer	0	0	0	5.200	5
nop tracer	115.830.539	0	0	0	0
ro tracer	115.821.865	0	0	0	0
no brc tracer	95.042.670	88.806.944	173,3	18.432	18
current	115.821.865	102.333.114	199,7	20.751	20
am brc tracer	517.462.670	102.333.113	199,7	20.480	20
sync write tracer	115.821.865	102.333.113	199,7	20.480	20
vmware	115.821.865	102.333.113	199,7	20.480	20

Tabelle 3: Messergebnisse zusammengefasst pro Messreihe (Ereignisse)

den Tracedateien zählen, aber da dort die *RadioMessageSentEvent* Kategorie fehlt, wären keine sinnvollen Vergleiche mit anderen Messreihen möglich. Für die beiden letzteren Messreihen existieren diese Daten per Definition nicht, da bei diesen Messreihen keine Verarbeitung der Simulationsdaten stattfand. Um die Vergleichbarkeit der “current” Messreihe zu gewährleisten, wurden die zusätzlichen Simulationen mit 150 Knoten und 3600 Sekunden hier nicht mitgezählt.

Wie zu erwarten, sind die vom Simulator erzeugten Ereignisse bei der “no brc tracer” Reihe niedriger als bei den anderen, da hier die “crc” Ereignisse fehlen und bei der “am” Reihe höher, da hier “am” statt “crc” Ereignisse verwendet wurden. Ebenfalls zu erwarten war, dass die in die Datenbank geschriebene Zahl von Ereignissen bei “no brc tracer” geringer ist, da die Ereignisse für Broadcasttransmissionen hier nicht pro Empfänger dupliziert wurden. Dementsprechend ist auch der Festplattenplatz leicht geringer, während diese Werte bei den anderen Reihen identisch sind. (Warum die “current” Reihe ein Ereignis mehr hat als die “am brc tracer” und “sync write tracer” Reihen, ist mir unerklärlich. Vermutlich gab es während Simulation 16 (50 Knoten, 300 Sekunden) irgendeinen Glitch.)

Sensordateninput Die Ausführungszeit für die “nop tracer” und die “ro tracer” Reihe sind beinahe identisch. Das Lesen der Sensordaten aus der Datenbank und ihr Eingeben in die Simulation verlangsamt den “Tracer” also nur in vernachlässigbarem Maße und der bisherige Algorithmus, der lediglich den aktuellen Sensorwert und die Dauer seiner Gültigkeit cached, ist bereits völlig ausreichend.

Broadcastanalyse Beim Vergleich der verschiedenen Möglichkeiten die korrekte Übertragung von Broadcasts genauer zu analysieren, zeigt sich, dass die “am” basierte Variante beinahe doppelt so lange braucht, wie die momentan benutzte “crc” Variante, was hauptsächlich daran liegt, dass sie fünfmal so viele Simulationsergebnisse erzeugt. Angesichts dieser Werte ist es meiner Meinung völlig gerechtfertigt, die potentiell größere Ungenauigkeit der “crc” Variante in Kauf zu nehmen. Diese Analyse allerdings völlig wegzulassen, würde lediglich einen Gewinn von 5,5% bringen und lohnt sich deshalb nicht.

Asynchrones stapelweißes Schreiben in die Datenbank Durch den Einsatz eines separaten Threads für das Schreiben der Simulationsergebnisse in die Datenbank, kann der Hauptthread schneller auf das nächste Simulationsereignis reagieren und es schneller bestätigen. Das stapelweise Ausführen der Insert-Anweisungen entlastet die Datenbank und reduziert den Overhead gegenüber dem separaten Ausführen eines jeden Inserts. Der dadurch

erreichte Performancegewinn von über 60% rechtfertigt den zusätzlichen Implementationsaufwand völlig.

Virtualisierung Der Einsatz des “Tracers” innerhalb einer *VMWare*-Instanz verursacht einen erheblichen Performanceverlust von beinahe 200% und sollte deshalb offensichtlich vermieden werden. Das weiter oben beschriebene Verfahren dafür veranlasst den Simulator zwar zu der Warnung “getenv JNI library not found. Env.getenv will not work”, aber die am Ende in die Datenbank gelangenden Daten sind 100% identisch zu denen, die beim virtualisierten Betrieb entstehen, die Simulation wird also in keiner Weise beeinträchtigt oder verfälscht und die erlangten Messdaten büßen nichts von ihrer Aussagekraft ein.

Aktuelle Implementation Die momentane Implementation ist nur 13% langsamer als der Standalone betriebene Simulator. Angesichts der umfangreichen Datenverarbeitung, die der “Tracer” durchführen muss, halte ich das für einen sehr guten Wert. Insbesondere da die Hälfte dieses Wertes auf die Art und Weise der Interaktion zwischen “Tracer” und Simulator über eine TCP-Verbindung entfällt, bei der die Simulation nach jedem einzelnen Ereignis pausiert, bis es vom “Tracer” bestätigt wurde. Dieser Performanceverlust gegenüber dem Standalone-Simulator ist durch die Architektur von *TOSSIM* erzwungen und kann vom “Tracer” nicht vermieden werden.

4.2 Vorverarbeitung der Energiedaten

4.2.1 Motivation

Ursprünglich war vorgesehen, in der Auswertungsphase direkt mit den Simulationsrohdaten in der Datenbank zu arbeiten. Es wurde aber schnell klar, dass dies aus Performancegründen völlig unmöglich ist. Warum dies so ist, sieht man, wenn man sich z.B. die Messdaten von Simulation 36 (150 Knoten, 3600 s) ansieht. Dort fallen etwa 25 GB an Simulationsdaten in Form von über 126 Millionen Simulationsereignissen an, davon 105 Millionen “power” Debugnachrichten. Es dürfte leicht einzusehen sein, dass es unmöglich ist, solche Mengen aus der Datenbank zu lesen, durch ein ORM zu pumpen und schließlich zu parsen und zu verarbeiten und dabei auch nur ein Minimum an Interaktivität zu erzielen.

Es ist also zwingend nötig, die Daten vorzuverarbeiten und das Datenvolumen dabei deutlich zu reduzieren. Auf diesem Weg muss die zeitintensive Verarbeitung der Rohdaten nur einmal durchgeführt werden, und die Auswertungsphase kann deutlich schneller zu Werke gehen. Aufgrund der deutlich größeren relativen Anzahl der “power” Debugnachrichten gegenüber der *RadioMessageSentEvents* Kategorie hatte deren Vorverarbeitung Priorität.

4.2.2 Zielstellung

Diese Vorverarbeitung musste zwei Ziele erfüllen. Sie sollte die Verarbeitung der Energiedaten während der Auswertung im “Viewer” beschleunigen. Und es sollte dem Nutzer ermöglicht werden, bei der Auswertung der Simulation zu beliebigen Simulationszeitpunkten zu springen, ohne dass dazu sämtliche Daten vom Beginn der Simulation bis zum gewählten Zeitpunkt verarbeitet werden müssen.

4.2.3 Umsetzung

Eine Möglichkeit das zweite Ziel zu erreichen, ist Zustandscaching oder Zustandscheckpointing. *Huginn*[16] wendet ein solches Verfahren an, allerdings nicht in einem Vorverarbeitungsschritt sondern zur Laufzeit. Ein Nachteil einer solchen Lösung ist, dass die Zustandsinformationen recht komplex und umfangreich sind, denn sie beinhalten nicht nur, welche Komponenten bisher bereits wie viel Energie verbraucht haben, sondern auch z.B. welche LEDs gerade an- bzw. ausgeschaltet sind, in welchem Modus sich die CPU und das Funkmodul gerade befinden, ob gerade auf das EEPROM zugegriffen wird usw.. Außerdem verbessert dieses Vorgehen nicht die Performance der Echtzeitanimation, denn vom jeweils nächstliegenden Zustandscheckpoint müssen wieder sämtliche Simulationsdaten verarbeitet werden.

Als Alternative bietet es sich an, stattdessen die Energiedaten zu aggregieren. Man bildet Intervalle und berechnet für jedes Intervall, wie viel Energie die verschiedenen Komponenten eines Knoten in diesem Intervall verbraucht haben. Diese Informationen sind im Gegensatz zum kompletten Zustand eines Knoten einfach, kurz und kompakt. Die Verarbeitung der Energiedaten wird dadurch enorm vereinfacht und beschleunigt. Die Anzahl der zu verarbeitenden Datensätze verringert sich bei einer Intervalllänge von einer Sekunde typischerweise um mehr als den Faktor 100, aber bei größeren Intervallen auch um beliebig größere Faktoren. Außerdem müssen diese Werte in der Auswertungsphase nur noch zusammenaddiert werden, womit das komplette Parsen der Energienachrichtenstrings entfällt. Das Springen zu einem beliebigen Simulationszeitpunkt lässt sich mit diesem Vorgehen ebenfalls leicht umsetzen. Man ermittelt alle Intervalle, die den gewünschten Zeitpunkt überlappen und zählt ihre Energiewerte gewichtet danach, welcher Anteil des Intervalls hinter dem gewünschten Zeitpunkt, also im betrachteten Zeitfenster, liegt. Dann addiert man einfach alle folgenden Intervalle hinzu, bis man zum Ende des gewünschten Zeitfensters kommt, wo die Werte der das Ende überlappenden Intervalle wieder anteilig nach Überlappingsgrad gezählt werden. Natürlich handelt man sich Ungenauigkeiten dadurch ein, dass die verbrauchte Energie in den Grenzwerten als über das jeweilige Intervall gleichverteilt angenommen wird. Allerdings kann der Nutzer durch Wahl der Intervalllänge den Tradeoff zwischen Performancegewinn und mög-

lichen Ungenauigkeiten selbst bestimmen.

Als Nächstes ist zu entscheiden, ob man mit festen oder dynamischen Intervalllängen arbeiten will. Feste Intervalllängen sind in der Verarbeitung einfacher zu handhaben. Insbesondere das Auffinden der einen Zeitpunkt überlappenden Startintervalle ist deutlich einfacher durchzuführen, wenn die Intervallgrenzen vorhersagbar sind. Allerdings haben feste Intervalllängen den Nachteil, dass auch für Intervalle ohne jede Aktivität und damit ohne Energieverbrauch leere Intervalle gespeichert werden müssen. Es hängt natürlich stark von der konkreten Simulation ab, wie stark sich dieser Nachteil auswirkt, aber man sollte ihn in jedem Fall vermeiden. Eine Möglichkeit wäre, leere Intervalle einfach nicht in die Datenbank zu schreiben, also Lücken zwischen den Intervallen zuzulassen. Alternativ kann man mit einer festen minimalen Intervalllänge arbeiten und das Intervall erst am ersten nach der minimalen Intervalllänge auftretenden energieverbrauchsändernden Ereignis beenden. Die Intervallgrenzen sind dann nicht mehr willkürlich gewählt, sondern liegen an natürlichen Schnittpunkten, an denen sich die Größe des Energieverbrauchs wenigstens einer Knotenkomponente ohnehin ändert. Dies hilft, mögliche Rundungsfehler zu minimieren, und Perioden ohne Aktivität werden automatisch durch ein entsprechend langes Intervall abgedeckt. Schließlich erhöht sich durch dieses Vorgehen auch die effektive Intervalllänge, denn jedes Intervall wird ja über die minimale Länge hinaus bis zum nächsten Energieereignis für den Knoten verlängert. Da sich per Definition die Stromaufnahme der Knotenkomponenten in diesem Verlängerungszeitraum nicht ändert, verliert man durch diese Intervallverlängerung nicht an Genauigkeit in der Auswertung. Von Nachteil ist, dass das Bestimmen der einen Zeitpunkt überlappenden Intervalle mit dieser Lösung schwieriger ist und dass die jeweilige Intervalllänge in jedem Datensatz mitgespeichert werden muss. Letzteres ließe sich zwar vermeiden, wenn man die Intervalllänge zur Laufzeit aus dem Startzeitpunkt des folgenden Intervalls ermitteln würde, aber dadurch verkomplizierte sich die Verarbeitung unverhältnismäßig stark, da man Intervalle nicht mehr unabhängig voneinander verarbeiten könnte. Insgesamt bin ich der Ansicht, dass die beiden letztgenannten Nachteile hinsichtlich Implementationsaufwand und Datensatzgröße durch die erzielbaren Vorteile zur Laufzeit gerechtfertigt werden und habe mich deshalb für das beschriebene Vorgehen entschieden.

Schließlich stellt sich die Frage, ob man jede der sechs Energiekategorien in einem separaten Tupel speichert oder pro Intervall nur ein Tupel mit allen Energiewerten benutzt. Die erste Variante hat den scheinbaren Vorteil, dass wenn man während der Auswertung nur an bestimmten Energiekategorien, z.B. nur CPU, interessiert ist, man diese gezielt zugreifen kann, und die restlichen Daten nicht zwangsweise mit aus der Datenbank lesen muss, wie es bei der zweiten Variante der Fall ist. Allerdings sollte man bedenken, dass ein solches Energieaggregatstupel einen Overhead von 17 Byte hätte, nämlich 4 Byte für den Fremdschlüssel auf die Vorverarbeitungsentität, 8

Byte für den Startzeitpunkt des Intervalls, 4 Byte für die Intervalllänge, 4 Byte für die Knotennummer und 1 Byte für die Energiekategorie. Demgegenüber stünden nur 4 Byte Nutzdaten für den Energieverbrauchswert. In der zweiten Variante hätte man 16 Byte Overhead (das Byte für die Energiekategorie entfällt) und 24 Byte Nutzdaten. Die erste Variante bräuchte also 126 Byte pro Intervall und die zweite nur 40 Byte. Und sobald man auch nur zwei Energiekategorien in der Auswertung betrachten will, z.B. CPU und Funk, muss man mit der ersten Variante schon 42 Byte fetchen und zwei separate Objekte vom ORM erzeugen lassen, während man mit der zweiten Variante 40 Byte fetcht und nur ein Objekt vom ORM erzeugt wird. Die erste Variante hätte ihren Vorteil also bereits vollständig eingebüßt und wenn noch mehr Energiekategorien in die Auswertung genommen werden, wäre sie immer mehr im Nachteil. Aus diesem Grund halte ich es für sinnvoller pro Intervall nur ein Tupel zu erzeugen, welches alle sechs Energiewerte enthält.

4.2.4 Skalierung des Verfahrens

Knoten	Simulationszeit	Simulationsereignisse	Aggregationsintervall		
			1	5	60
5	10	1.545	0,16	0,21	0,14
10	10	4.749	0,48	0,50	0,49
20	10	9.693	1,08	1,17	1,12
50	10	41.306	4,59	4,43	4,46
100	10	88.396	10,77	10,54	10,70
150	10	142.751	17,91	17,53	17,16
50	60	600.841	62	62	63
150	60	1.904.661	224	220	220
50	300	3.282.349	349	350	351
150	300	10.348.788	1.321	1.263	1.264
5	600	354.263	34	35	35
10	600	845.090	81	81	80
20	600	1.546.028	149	148	148
50	600	6.640.503	740	716	707
100	600	12.797.042	1.567	1.500	1.481
150	600	20.911.527	5.973	5.889	5.833
50	1800	20.061.731	2.519	2.490	2.508
150	1800	63.145.538	11.945	11.445	11.234
50	3600	40.184.934	5.979	5.668	5.432
150	3600	126.509.168	18.256	17.116	16.311

Tabelle 4: Messung der Vorverarbeitung der Energiedaten

In Tabelle 4 sind die Messwerte zur beschriebenen Vorverarbeitung der Energiedaten dargestellt. Jede Messung wurde viermal durchgeführt. Die

jeweils erste Messung diente nur dazu, den Cache der Datenbank “vorzuwärmen”, damit die folgenden Messungen nicht durch Cacheeffekte verfälscht werden. Bei den dann folgenden drei Messungen wurden verschiedene Aggregationsintervalle benutzt. Bei einem Teil der Messungen sinkt die Verarbeitungszeit mit steigendem Aggregationsintervall, was dadurch erklärt werden kann, dass weniger Aggregationsdaten in die Datenbank geschrieben werden müssen. Allerdings ist dieser Effekt unterschiedlich stark ausgeprägt und fehlt bei einer Reihe von Messungen völlig. Die Länge des Aggregationsintervalls hat also keinen oder nur sehr geringen Einfluss auf die Verarbeitungszeit.

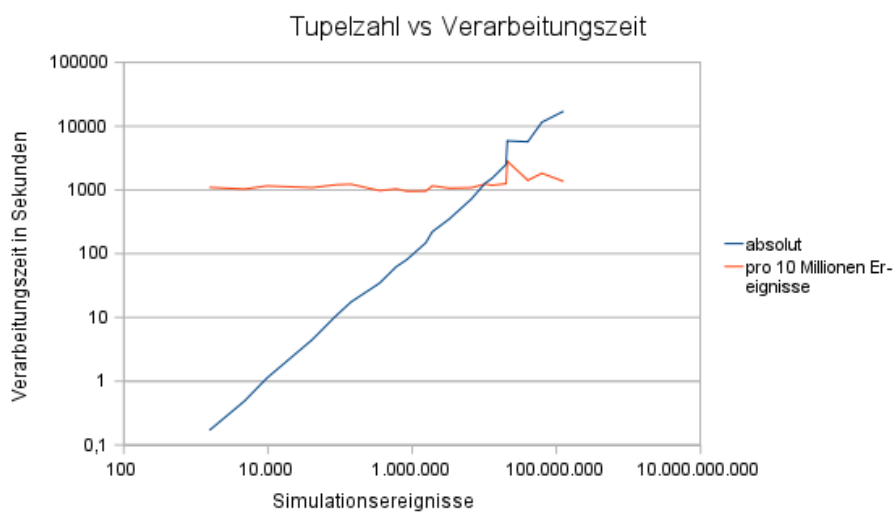


Abbildung 14: Verarbeitungszeit in Abhängigkeit der zu verarbeitenden Simulationereignisse

In Abbildung 14 ist die Entwicklung der durchschnittlichen Verarbeitungszeit in Abhängigkeit von der Zahl der zu verarbeitenden Daten dargestellt. Es gibt einige Fluktuationen in der Verarbeitungszeit und insbesondere Simulation 24 (150 Knoten, 600 Sekunden) zeigt eine deutliche Abweichung nach oben. Insgesamt aber kann man sehen, dass die Verarbeitungszeit grob linear von der Zahl der Simulationereignisse abhängt. Die Vorverarbeitung der Energiedaten skaliert also wie erwartet.

4.3 Datenbankunabhängigkeit vs. Performance

Sowohl der “Tracer” als auch der “Viewer” sind prinzipiell Datenbank agnostisch. Ersterer weil er die JDBC-Schnittstelle zur Kommunikation mit der Datenbank benutzt und letzterer durch die Nutzung der *Ruby On Rails* ORM-Komponente *ActiveRecord*. Im “Viewer” führt dies aber an einigen Stellen zu enormen Performanceengpässen. Und zwar immer dann wenn eine

große Zahl an Tupeln in die Datenbank zu schreiben ist, wie beim Einlesen von nutzergenerierten Sensorinputdaten und bei der Energienachrichtenaggregation. Denn für jedes Tupel wird vom ORM ein Objekt erstellt und dann einzeln per Insert in die Datenbank geschrieben. An diesen Stellen habe ich mich entschieden, die Daten unter Umgehung des ORM mittels des PostgreSQL COPY Befehls direkt in die Datenbank zu schreiben. Das vermeidet den Overhead des ORM und statt vieler kleiner Inserts führt man lediglich einige lange COPY Befehle aus, mit denen die Daten in einem Rutsch in der Datenbank landen. Da aber der COPY Befehl nicht zum SQL Standard gehört, sondern PostgreSQL spezifisch ist, verliert man dadurch die Datenbankunabhängigkeit. Zur Abwägung habe ich einige Messungen zum

Kno- ten	Sen- so- ren	Daten pro Sensor	Tupel	Zeit in Sekunden		Fak- tor	Zeit für 1k Inputdaten	
				COPY	ORM		COPY	ORM
66	5	501	165.330	8,91	364,5	41	0,054	2,20
66	5	200	66.000	3,75	147,2	39	0,057	2,23
66	3	200	39.600	1,72	86,9	51	0,043	2,20

Tabelle 5: Messung der Geschwindigkeit des Schreibens von Sensorinputdaten in die Datenbank durch den “Viewer”

Einlesen von Sensorinputdaten durch den “Viewer” in Tabelle 5 dargestellt. Dabei werden die als Datei vorliegenden Sensorinputs durch den “Viewer” in die Datenbank geschrieben, damit sie später dem “Tracer” zur Verfügung stehen. Einmal wurde dies mithilfe des COPY Befehls durchgeführt und einmal mit den normalen ORM Methoden von *ActiveRecord*. Die Messwerte zeigen deutlich, dass es mehr als vierzig Mal so lange dauert, wenn man über das ORM geht. Der Verlust der Datenbankunabhängigkeit an dieser und ähnlichen Stellen ist also durch den Performancegewinn mehr als gerechtfertigt.

4.4 Auswertung / Animation

Bei der Auswertung sendet der Browser per AJAX das vom Nutzer gewählte Zeitfenster an den “Viewer”, dieser berechnet die Energie- und Radio-statistiken und sendet entsprechenden XHTML- und Javascriptcode zurück an den Browser, der die Daten dann anzeigt. Bei der Durchführung einer Echtzeitanimation wiederholt sich dieses Verfahren einfach, bis das Ende des darzustellenden Zeitfensters erreicht ist.

Während die Darstellung der Statistiken in Textform kein Problem ist, bereitet die grafische Darstellung der Daten einige Schwierigkeiten. Eine Möglichkeit wäre es, auf pluginbasierte Techniken wie Javaapplets, Flash, ActiveX oder Silverlight zurückzugreifen. Weiterhin gibt es diverse Javascriptbibliotheken, die es ermöglichen im Browserfenster zu malen, aber diese basieren alle darauf, jedes Lienelement durch entsprechend platzierte <div>

Elemente oder passend verzerrte Grafiken darzustellen. Für ein paar einfache Linien funktioniert dies zufriedenstellend, aber wenn z.B. die Energiewerte von Hunderten Knoten und die Kommunikationswege zwischen ihnen visualisiert werden sollen, dann ist die Performance indiskutabel schlecht. Als Alternative bietet sich das `<canvas>` Element an. Es ist Teil der *WhatWG Web applications 1.0 specification* [22] auch bekannt als *HTML 5* und erlaubt es, im Browser per Javascript 2D-Zeichenoperationen auszuführen. Momentan wird es von Safari, Firefox und Opera unterstützt und das *ExplorerCanvas* [7] Projekt verspricht, es mittels einer Javascriptkompatibilitätsbibliothek auch im IE zu unterstützen. Letzteres habe ich aber nicht weiter untersucht, da dies aus meiner Sicht nur untergeordnete Priorität hat. Mithilfe des `<canvas>` Elements kann die grafische Darstellung wesentlich performanter durchgeführt werden. Letztendlich ist das Verfahren aber immer noch stark Javascript basiert und kann deshalb niemals die Performanz und Skalierbarkeit einer nativ implementierten Animation erreichen. Außerdem hängt die Leistung stark vom jeweiligen Browser und seiner Javascriptimplementation ab. Für die Integration mit *GoogleMaps* wird das `<canvas>` Element über der *GoogleMaps* Darstellung platziert und die als Längen- und Breitengrade vorliegenden Knotenkoordinaten werden mithilfe der *GoogleMaps*-API in die aktuellen Pixelkoordinaten umgerechnet. Ein wesentlicher Nachteil dieses Vorgehens ist, dass Mausklicks die *GoogleMap* nicht mehr erreichen, da ja das `<canvas>` Element darüber liegt. Der Nutzer kann die Karte also nicht verschieben oder zoomen. Während einer Echtzeitanimation halte ich diese Einschränkung für akzeptabel, aber während der statischen Auswertung eines gewählten Zeitfensters, muss die *GoogleMap* auf jeden Fall bedienbar bleiben. Aus diesem Grund wurde ein ausschließlich die *GoogleMaps*-API nutzendes Alternativverfahren entwickelt. Die Energiewerte eines Knoten werden durch dynamisch vom Server generierte Grafiken dargestellt, in deren URL die Parameter der entsprechenden Balkengrafik kodiert sind. Die Funkkommunikation zwischen den Knoten wird mit der von der *GoogleMaps*-API bereitgestellten *GPolyline* Klasse visualisiert. Im IE benutzt die *GoogleMaps*-API dafür dessen proprietäre VML (*Vector Markup Language*) Erweiterung, während in anderen Browsern vom Googleserver gerenderte Grafiken zum Einsatz kommen. Dadurch erreicht man eine vollständige Integration mit der *GoogleMap* und die im Vergleich zum `<canvas>` Element deutlich schlechtere Performance ist bei der statischen Visualisierung eher zu verschmerzen.

5 Ausblick

5.1 Workflowverbesserungen

5.1.1 Integration des Simulationsstarts in die Browseroberfläche

Während der Kompilervorgang wohl immer “manuell” ausgeführt werden muss, wäre es sicher aus Nutzersicht wünschenswert, das Starten einer Simulation aus der Weboberfläche heraus initiieren zu können. Der “Viewer” müsste um ein Modul erweitert werden, welches die Simulationsparameter per Formular entgegennimmt, soweit möglich auf Korrektheit prüft und anschließend den “Tracer” startet. Der “Tracer” Prozess sollte unabhängig vom “Viewer” ausgeführt werden, so dass z.B. ein Webserverneustart nicht die laufenden Simulationen beendet. Die Startparameter und die PID des “Tracers” müssen in der Datenbank gespeichert werden, so dass der “Viewer” später alle laufenden Simulationen finden und für den Nutzer auflisten kann. Die Ausgaben des “Tracers” sollten in eine dedizierte Datei umgeleitet werden, deren Inhalt dem Nutzer im Falle eines Simulationsfehlers zum Zwecke der Fehlersuche angezeigt wird. Um eine Fortschrittsanzeige für die Simulationen zu realisieren, ist die einfachste Lösung den “Viewer” die Ausgabedatei des “Tracers” parsen zu lassen.

5.1.2 Integration von existierenden Sensordatenaufzeichnungen

Das verwendete Datenbankschema sollte einfach genug gehalten sein, um die Verwendung von existierenden Sensordatendatenbanken ohne größere Probleme zu ermöglichen. Man könnte in den “Tracer” und den “Viewer” eine Datenzugriffsindirektion einbauen, in der Tabellen- und Attributnamen, sowie nötige Einheitenumrechnungen flexibel konfiguriert werden können. Oder man schreibt ein Transformationsskript, welches die Daten in meine Tabellen einfügt. Auch eine auf Datenbankviews basierende Lösung ist vorstellbar. Welche Variante am geeignetsten ist, müsste an einem konkreten Beispiel evaluiert werden.

5.1.3 Dynamische Radiomodelle und mobile Knoten

Ein dynamisches Radiomodell ist meiner Meinung nach so simulationsspezifisch, dass man es nicht auf generische Weise im “Tracer” unterstützen kann. Das vorhandene Pluginsystem des “Tracers” gibt dem Nutzer bei Bedarf die Möglichkeit, es ohne großen Mehraufwand zu implementieren.

Mobile Knoten können natürlich auch über Plugins realisiert werden. Dies wäre dann aber im Grunde nur eine andere Form eines dynamischen Radiomodells, da die Knotenbewegungen nicht in der Visualisierung im “Viewer” erscheinen würden. Die Implementation von entsprechender Funktionalität direkt im “Tracer” und “Viewer” ist relativ einfach. Die Schwierigkeit besteht

darin, ein effizientes Format zum Beschreiben und ein gutes Nutzerinterface zum Spezifizieren der Knotenbewegungen zu finden.

5.2 Simulatorverbesserungen

5.2.1 Unterstützung für weitere Hardwareplattformen

PowerTOSSIM unterstützt momentan nur die *Mica2*-Plattform. Die Unterstützung von weiterer aktueller Knotenhardware ist sicherlich wünschenswert und bereits in meinem Programm vorgesehen. Die praktische Umsetzung dessen erfordert das Ermitteln der Energieverbrauchswerte für das Energiemodell, Funkmessungen als Basis für die Umrechnung von Knotendistanzen in Bitfehlerwahrscheinlichkeiten, das Ergänzen der entsprechenden hardware-spezifischen *TOSSIM*-Klassen und möglicherweise geringe Anpassungen in der Verarbeitung des Energiemodells selbst.

5.3 Performance

5.3.1 Vorverarbeitung der Energiedaten

Dieser Schritt ist momentan im “Viewer” implementiert und wird vom Nutzer nach Abschluss der Simulation angestoßen. Um ihn zu beschleunigen, sind verschiedene Maßnahmen vorstellbar.

Man könnte die Bearbeitung parallelisieren, indem man mit mehreren Threads arbeitet, von denen jeder jeweils eine Untermenge der Knoten verarbeitet. Allerdings wird dies dadurch erschwert, dass Rubys Threadingmodell mit “green threads” arbeitet und deshalb nicht mehrere Prozessoren nutzen kann. Man müsste also an Stelle von Threads zusätzliche Rubyprozesse starten und der Overhead des dann nötigen IPC könnte einen Teil des Performancegewinns wieder aufzehren.

Auch könnte eine Umgehung der ORM-Komponente *ActiveRecord* beim Einlesen der Daten Geschwindigkeitszuwächse bringen, da *ActiveRecord* nicht für die hochperformante Verarbeitung großer Objektmengen konzipiert ist.

Wenn man diesen Schritt schon im “Tracer” während der Simulation durchführen würde, so sollte dies wesentlich schneller vonstatten gehen, da Java performanter ist als Ruby und sich leichter parallel auf mehreren Prozessoren betreiben lässt. Außerdem müssten die eigentlichen Energienachrichten dann gar nicht mehr in der Datenbank gespeichert werden, sondern nur noch die Aggregate, deren Zahl um Größenordnungen kleiner ist, was Zeit und vor allem Speicherplatz sparen würde. Auch wäre es dadurch während der Simulation möglich, Knoten, die ihren Energievorrat aufgebraucht haben, abzuschalten und damit ihren Ausfall zu simulieren. Aber man verlöre die Möglichkeit, eine Simulation mit verschiedenen Energiemodellen auszuwerten, ohne die ganze Simulation erneut durchführen zu müssen. Außerdem

stunden die ursprünglichen Simulationsdaten nicht mehr für mögliche Validierungen oder alternative Auswertungen durch den Nutzer zur Verfügung.

5.3.2 Auswertung

Animation Die grafische Echtzeitanimation ist bei großen Knotenzahlen noch nicht performant genug. Dies ist zu einem gewissen Grad unvermeidbar, da die Animation mittels Javascript im Browser des Nutzers ablaufen muss und Javascript für solche Aufgaben nicht wirklich gedacht ist. Aber durch verstärkten Einsatz von Clipping ließen sich noch Verbesserungen erreichen für den Fall, dass der Nutzer durch entsprechendes Zoomen gerade nur einen Teil der Knoten auf der Karte sieht.

Statische Auswertung Die benötigte Zeit für die statische Auswertung wird bei großen Knotenzahlen von der Verarbeitung der Funknachrichten dominiert. Um dies zu beschleunigen, könnte man die *RadioMessageSentMessage* Ereignisse aus dem “single table inheritance” Schema der “tossim_messages” Tabelle herauslösen und in eine eigene Tabelle packen. Dann würden diese Daten wesentlich kompakter auf der Festplatte liegen, anstatt zwischen all den *PowerMessage* Ereignissen verstreut zu sein, was den Datenbankzugriff beschleunigen sollte.

Außerdem könnte das Umgehen der ORM-Komponente *ActiveRecord* auch hier zu erheblichen Performancesteigerungen führen.

Schließlich könnte man eine aggregierende Vorverarbeitung der Funknachrichten ähnlich wie bei den Energienachrichten in Erwägung ziehen. Deren Effektivität hängt aber stark vom Kommunikationsmuster der Knoten und der gewählten Länge des Aggregationsintervalls ab und sollte deshalb vorher gründlich evaluiert werden.

Literatur

- [1] Distanzen aus Längen- und Breitenangaben berechnen. <http://www.movable-type.co.uk/scripts/latlong.html>.
- [2] Rimon Barr. SWANS – scalable wireless ad hoc network simulator - user guide.
- [3] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. JiST: an efficient approach to simulation using virtual machines. *Softw, Pract. Exper*, 35(6):539–576, 2005.
- [4] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobile Computing and Networking*, pages 85–97, 1998.

-
- [5] SAFER Seismic eArly warning For EuRope. <http://www.saferproject.net/>.
 - [6] Deborah Estrin, Mark Handley, John S. Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with nam, the VINT network animator. *IEEE Computer*, 33(11):63–68, 2000.
 - [7] ExplorerCanvas. <http://excanvas.sourceforge.net/>.
 - [8] Joachim Fischer. Selbstorganisation im Wettlauf mit tödlichen Wellen - Start des interdisziplinären Graduiertenkollegs METRIK. *Humboldt-Spektrum*, pages 32–37, March 2006.
 - [9] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, pages 93–104, 2000.
 - [10] Sphärischer Kosinussatz. <http://mathworld.wolfram.com/SphericalTrigonometry.html>.
 - [11] Stuart Kurkowski, Tracy Camp, Neil Mushell, and Michael Colagrosso. A visualization and analysis tool for NS-2 wireless simulations: iNSpect. In *MASCOTS*, pages 503–506. IEEE Computer Society, 2005.
 - [12] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137, New York, November 5–7 2003. ACM Press.
 - [13] The ns-2 network simulator. <http://www.isi.edu/nsnam/ns/>.
 - [14] CMU Monarch Project. The CMU Monarch project’s ad-hockey visualization tool for ns scenario and trace files. *Carnegie Mellon University*, August 1998.
 - [15] Ruby On Rails. <http://www.rubyonrails.org/>.
 - [16] Björn Scheuermann, Holger Füller, Matthias Transier, Marcel Busse, Martin Mauve, and Wolfgang Effelsberg. Huginn: A 3D Visualizer for Wireless ns-2 Traces. In *Proc. of the 8th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM’05)*, pages 134–150, Motreal, Canada, October 2005.
 - [17] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In John A. Stankovic, Anish Arora, and

Ramesh Govindan, editors, *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, Baltimore, MD, USA, November 3-5, 2004*, pages 188–200. ACM, 2004.

- [18] Roger W. Sinnott. Virtues of the haversine. *Sky and Telescope*, page 159, August 1984.
- [19] Trishla Sutaria, Imad Mahgoub, Ali Humos, and Ahmed Badi. Implementation of an energy model for JiST/SWANS wireless network simulator. In *ICN*, page 24. IEEE Computer Society, 2007.
- [20] T. Vincenty. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, 23(176):88–93, April 1975.
- [21] Metrik Modellbasierte Entwicklung von Technologien für selbstorganisierende dezentrale Informationssysteme im Katastrophenmanagement (Interdisziplinäres Graduiertenkolleg am Institut für Informatik der Humboldt-Universität zu Berlin). <http://casablanca.informatik.hu-berlin.de/grk-wiki/>.
- [22] WHATWG. <http://www.whatwg.org/>.
- [23] XubunTOS. <http://toilers.mines.edu/Public/XubunTOS>.

A Nachrichtenklassenaufistung

Ereignisse	
DEBUGMSGEVENT	Debugnachricht eines Knoten
RADIOMSGSENTEVENT	Radionachricht wurde von Knoten gesendet
UARTMSGSENTEVENT	UART-Nachricht wurde von Knoten gesendet
ADCDATAREADYEVENT	ADC-Wert steht bereit
TOSSIMINITEVENT	Initinfo (Knotenanzahl) für TinyViz (siehe 2.2)
INTERRUPTEVENT	Nutzerveranlasste Simulationsunterbrechung
LEDEVENT	Zustandsänderung der LEDs
Kommandos	
TURNONMOTECOMMAND	Knoten anschalten
TURNOFFMOTECOMMAND	Knoten ausschalten
RADIOMSGSENDCOMMAND	Radionachricht an Knoten
UARTMSGSENDCOMMAND	UART-Nachricht an Knoten
SETLINKPROBCOMMAND	Bitfehlerwahrscheinlichkeit zwischen zwei Knoten setzen
SETADCPORVALUECOMMAND	ADC-Port Wert für Knoten setzen
INTERRUPTCOMMAND	Simulationsunterbrechung zu bestimmtem Zeitpunkt anfordern
SETRATECOMMAND	
SETDBGCOMMAND	Debug-Unterkategorien an-/ausschalten
VARIABLERESOLVECOMMAND	Adresse einer Variablen (Speicherregion) anfordern
VARIABLERESOLVERESPONSE	Liefert Adresse einer Variablen
VARIABLEREQUESTCOMMAND	Variable (Speicherregion) lesen
VARIABLEREQUESTRESPONSE	Liefert Wert einer Variablen
GETMOTECOUNTCOMMAND	Javaklasse fehlt,
GETMOTECOUNTRESPONSE	aber im Simulator implementiert
SETEVENTMASKCOMMAND	
BEGINBATCHCOMMAND	
ENDBATCHCOMMAND	

Tabelle 6: Auflistung aller definierten Ereignisklassen (aus /opt/tinyos-1.x/tos/platform/pc/GuiMsg.h)

Knotenkernfunktionen	
boot	Bootvorgang
clock	Hardwareuhr
task	Task enqueueing/dequeueing/running
sched	TinyOS scheduler
sensor	Sensozugriffe
led	Knoten-LEDs
crypto	Kryptografische Operationen (z.B. TinySec)
Netzwerk	
route	Routingsystem
am	ActiveMessages Senden/Empfangen
crc	CRC-Prüfung von ActiveMessages
packet	Paketlevel Senden/Empfangen
encode	Paket(de-)kodierung
radio	Low-level Radiooperationen (Bits & Bytes)
Sonstige Hardware	
logger	“Logger“-komponente
adc	Analog Digital Konverter
i2c	I2C Bus
uart	UART (serieller Port)
prog	Fernreprogrammierung
sounder	“sounder“-Sensor
time	Zeitgeber
power	Energieverbrauchsstatus
Simulator	
sim	TOSSIM Interna
queue	TOSSIM Ereigniswarteschlange
simradio	TOSSIM Radiomodelle
hardware	TOSSIM Hardwareemulation
simmem	TOSSIM Speicher anfordern/freigeben (zum Finden von Speicherlecks)
Anwendung	
usr1	Nutzerausgabemodus 1
usr2	Nutzerausgabemodus 2
usr3	Nutzerausgabemodus 3
temp	Temporäre Testausgaben
Sonstiges	
all	Alle Kategorien aktivieren
none	Alle Kategorien deaktivieren
error	Fehlerzustände

Tabelle 7: Unterkategorien der DebugMsgEvent-Klasse (aus /opt/tinyos-1.x/tos/types/dbg_modes.h)

B Datenbankschema

Da der “Viewer” für *Ruby On Rails* geschrieben wurde, folgt das Datenbankschema den dort gültigen Konventionen. Diese beinhalten unter anderem:

- Jede Tabelle erhält automatisch eine Spalte “id”, die Primärschlüssel ist (womit PostgreSQL automatisch einen Index auf dieser Spalte erstellt) und auto-inkrementiert wird.
- Spalten der Form “tablename_id” fungieren als Fremdschlüssel auf die “id”-Spalte der Tabelle “tablename”. Dies wird vom OR-Mapper umgesetzt, nicht in der Datenbank.
- Spalten mit Namen “created_at” oder “updated_at” werden automatisch mit Zeitstempeln gefüllt, wenn eine Zeile erzeugt oder verändert wird.
- Spalten mit Namen “tablename_count” agieren als Cache für die Anzahl von assoziierten Objekten in einer 1:n Beziehung. (Sie sind in Listing 1 nicht aufgeführt, da sie nur der Performancesteigerung dienen und ansonsten transparent sind.)

Listing 1: Datenbankschema

```
CREATE TABLE scenarios (  
  name character varying(30) NOT NULL,  
  description text,  
  radio_range integer NOT NULL,  
  map_center_latitude double precision NOT NULL,  
  map_center_longitude double precision NOT NULL,  
  map_zoom integer NOT NULL,  
  map_width integer NOT NULL,  
  map_height integer NOT NULL,  
  created_at timestamp without time zone NOT NULL  
);  
  
CREATE TABLE notes (  
  scenario_id integer NOT NULL,  
  name character varying(30),  
  latitude double precision NOT NULL,  
  longitude double precision NOT NULL  
);  
  
CREATE TABLE sensors (  
  note_id integer NOT NULL,  
  name character varying(30),  
  adc_port bigint NOT NULL  
);
```

```
CREATE TABLE sensor_readings (  
    sensor_id integer NOT NULL,  
    "timestamp" bigint NOT NULL,  
    value integer NOT NULL  
);  
  
CREATE TABLE custom_radio_models (  
    scenario_id integer NOT NULL,  
    name character varying(30) NOT NULL,  
    data text NOT NULL  
);  
  
CREATE TABLE tossim_runs (  
    scenario_id integer NOT NULL,  
    cmdline character varying(400) NOT NULL,  
    dbg character varying(255) NOT NULL,  
    simulation_duration integer NOT NULL,  
    simulation_starttime bigint NOT NULL,  
    cpu_frequency integer NOT NULL,  
    extra_tossim_params character varying(255) NOT NULL,  
    created_at timestamp without time zone NOT NULL,  
    created_by character varying(60) NOT NULL,  
    execution_time integer NOT NULL  
);  
  
CREATE TABLE tossim_messages (  
    tossim_run_id integer NOT NULL,  
    "type" character varying(255) NOT NULL,  
    mote integer NOT NULL,  
    "timestamp" bigint NOT NULL,  
    subtimestamp integer NOT NULL,  
    message character varying(255),  
    receiver integer,  
    is_broadcast boolean,  
    got_ack boolean  
);  
  
CREATE TABLE pre_processings (  
    tossim_run_id integer NOT NULL,  
    energy_model_name character varying(255) NOT NULL,  
    has_sensorboard boolean NOT NULL,  
    empty_energy_categories character varying(255) DEFAULT  
        '' ::character varying NOT NULL,  
    min_aggregation_interval integer NOT NULL  
);  
  
CREATE TABLE energy_aggregates (  
    pre_processing_id integer NOT NULL,
```

```
"timestamp" bigint NOT NULL,  
length integer NOT NULL,  
mote integer NOT NULL,  
cpu integer NOT NULL,  
radio integer NOT NULL,  
led integer NOT NULL,  
eeprom integer NOT NULL,  
adc integer NOT NULL,  
sensor integer NOT NULL  
);  
  
CREATE UNIQUE INDEX index_sensors_on_mote_id_and_adc_port ON  
sensors(mote_id, adc_port);  
CREATE UNIQUE INDEX  
index_sensor_readings_on_sensor_id_and_timestamp ON  
sensor_readings(sensor_id, "timestamp");  
CREATE UNIQUE INDEX  
index_custom_radio_models_on_scenario_id_and_name ON  
custom_radio_models(scenario_id, name);  
CREATE INDEX  
index_tossim_messages_on_tossim_run_id_and_timestamp ON  
tossim_messages(tossim_run_id, "timestamp");  
CREATE INDEX  
index_energy_aggregates_on_pre_processing_id_and_timestamp  
ON energy_aggregates(pre_processing_id, "timestamp");
```

scenarios	
radio_range	cm
map_center_latitude	°
map_center_longitude	°
map_width	Pixel
map_height	Pixel
notes	
latitude	°
longitude	°
sensor_readings	
timestamp	Sekunden
tossim_runs	
simulation_duration	Sekunden
execution_time	Sekunden
tossim_messages	
mote	TOSSIM Knoten-ID
timestamp	CPU-Ticks
receiver	TOSSIM Knoten-ID
energy_aggregates	
min_aggregation_interval	Sekunden
energy_aggregates	
timestamp	CPU-Ticks
length	CPU-Ticks
mote	TOSSIM Knoten-ID
cpu, radio, led, eeprom, adc, sensor	μJ

Tabelle 8: Übersicht über verwendete Einheiten im Datenbankschema

C Rohdaten der Messungen

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
no tracer						
5	10	3				0,07
10	10	5				0,22
20	10	5				0,46
50	10	9				2
100	10	18				4,2
5	60	9				1,6
10	60	17				3,8
20	60	34				7
50	60	98				31
100	60	200				58
5	300	39				9
10	300	82				21
20	300	164				39
50	300	531				167
100	300	1.082				317
5	600	75				19
10	600	164				43
20	600	330				79
50	600	1.068				340
100	600	2.250				645
5	1800	223				56
10	1800	492				129
20	1800	988				240
50	1800	3.242				1.033
100	1800	6.548				1.955
nop tracer						
5	10	0	1.774	0		
10	10	2	5.312	0		
20	10	3	11.146	0		
50	10	9	45.675	0		
100	10	19	95.867	0		
5	60	7	38.148	0		
10	60	16	88.791	0		
20	60	33	163.929	0		
50	60	104	683.302	0		
100	60	214	1.302.011	0		

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
5	300	38	212.551	0		
10	300	88	489.656	0		
20	300	177	900.809	0		
50	300	569	3.730.203	0		
100	300	1.141	7.094.399	0		
5	600	78	430.984	0		
10	600	177	991.247	0		
20	600	356	1.820.847	0		
50	600	1.168	7.541.591	0		
100	600	2.329	14.325.795	0		
5	1800	235	1.304.234	0		
10	1800	526	2.997.744	0		
20	1800	1.070	5.501.087	0		
50	1800	3.498	22.801.928	0		
100	1800	7.013	43.251.509	0		
ro tracer						
5	10	0	1.809	0		
10	10	1	5.411	0		
20	10	2	11.092	0		
50	10	9	45.511	0		
100	10	20	95.790	0		
5	60	7	38.094	0		
10	60	16	88.944	0		
20	60	33	164.334	0		
50	60	106	681.674	0		
100	60	214	1.300.834	0		
5	300	39	212.731	0		
10	300	88	490.503	0		
20	300	178	899.809	0		
50	300	577	3.729.752	0		
100	300	1.175	7.088.506	0		
5	600	79	430.714	0		
10	600	179	993.173	0		
20	600	355	1.819.676	0		
50	600	1.175	7.547.935	0		
100	600	2.325	14.315.556	0		
5	1800	237	1.303.415	0		
10	1800	547	3.001.137	0		
20	1800	1.059	5.498.855	0		
50	1800	3.483	22.805.957	0		

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
100	1800	7.005	43.250.653	0		
no brc tracer						
5	10	1	1.602	1.394	28	0
10	10	1	4.651	4.164	42	1
20	10	3	9.580	8.538	43	2
50	10	9	37.988	35.248	70	7
100	10	19	80.008	74.061	74	15
5	60	8	32.261	28.987	97	6
10	60	17	74.498	67.873	113	14
20	60	33	138.523	125.199	104	26
50	60	107	557.177	523.717	175	109
100	60	217	1.066.050	998.675	166	207
5	300	40	179.801	161.779	108	34
10	300	90	410.244	374.123	125	78
20	300	176	757.403	685.090	114	142
50	300	573	3.045.010	2.864.085	191	594
100	300	1.155	5.801.793	5.439.482	181	1.129
5	600	80	363.981	327.527	109	68
10	600	179	830.485	757.500	126	157
20	600	359	1.531.405	1.385.364	115	288
50	600	1.163	6.161.086	5.795.832	193	1.203
100	600	2.365	11.715.589	10.984.620	183	2.280
5	1800	242	1.101.303	991.112	110	206
10	1800	538	2.509.345	2.288.883	127	475
20	1800	1.090	4.627.205	4.186.225	116	869
50	1800	3.446	18.613.964	17.511.382	195	3.635
100	1800	6.951	35.391.718	33.186.084	184	6.888
current						
5	10	0	1.809	1.545	31	0
10	10	3	5.411	4.749	47	1
20	10	3	11.092	9.693	48	2
50	10	11	45.511	41.306	83	8
100	10	24	95.790	88.396	88	18
150	10	38	151.125	142.751	95	29
5	60	10	38.094	31.391	105	6
10	60	19	88.944	75.831	126	15
20	60	35	164.334	139.871	117	28
50	60	112	681.674	600.841	200	122
100	60	226	1.300.834	1.165.400	194	236

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
150	60	361	2.092.876	1.904.661	212	386
5	300	42	212.731	174.996	117	35
10	300	93	490.503	417.472	139	85
20	300	182	899.809	764.649	127	155
50	300	601	3.729.752	3.282.349	219	666
100	300	1.215	7.088.506	6.337.629	211	1.285
150	300	1.936	11.396.383	10.348.788	230	2.098
5	600	83	430.714	354.263	118	72
10	600	185	993.173	845.090	141	171
20	600	375	1.819.676	1.546.028	129	313
50	600	1.228	7.547.935	6.640.503	221	1.347
100	600	2.479	14.315.556	12.797.042	213	2.595
150	600	3.981	23.035.158	20.911.527	232	4.240
5	1800	249	1.303.415	1.071.921	119	217
10	1800	562	3.001.137	2.553.441	142	518
20	1800	1.116	5.498.855	4.671.317	130	947
50	1800	3.669	22.805.957	20.061.731	223	4.068
100	1800	7.433	43.250.653	38.655.660	215	7.838
150	1800	12.090	69.568.942	63.145.538	234	12.804
5	3600	496	2.611.475	2.147.634	119	435
10	3600	1.113	6.010.369	5.113.850	142	1.037
20	3600	2.225	11.018.149	9.359.659	130	1.898
50	3600	7.379	45.681.860	40.184.934	223	8.149
100	3600	14.876	86.646.124	77.438.024	215	15.703
150	3600	23.510	139.386.327	126.509.168	234	25.653
am brc tracer						
5	10	1	7.272	1.545	31	0
10	10	3	22.519	4.749	47	1
20	10	6	46.999	9.693	48	2
50	10	21	203.874	41.306	83	8
100	10	36	415.980	88.396	88	18
5	60	14	169.663	31.391	105	6
10	60	33	408.770	75.831	126	15
20	60	62	705.067	139.871	117	28
50	60	229	3.185.578	600.841	200	120
100	60	432	5.685.972	1.165.400	194	233
5	300	75	946.731	174.996	117	35
10	300	177	2.244.150	417.472	139	84
20	300	335	3.865.347	764.649	127	153
50	300	1.241	17.456.705	3.282.348	219	657

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
100	300	2.340	30.963.848	6.337.629	211	1.268
5	600	153	1.917.297	354.263	118	71
10	600	358	4.541.723	845.090	141	169
20	600	659	7.828.333	1.546.028	129	309
50	600	2.583	35.325.644	6.640.503	221	1.329
100	600	4.703	62.546.538	12.797.042	213	2.561
5	1800	454	5.807.544	1.071.921	119	215
10	1800	1.079	13.753.171	2.553.441	142	511
20	1800	2.018	23.628.985	4.671.317	130	935
50	1800	7.834	106.759.301	20.061.731	223	4.015
100	1800	14.610	189.025.659	38.655.660	215	7.736
sync write tracer						
5	10	1	1.809	1.545	31	0
10	10	3	5.411	4.749	47	1
20	10	6	11.092	9.693	48	2
50	10	24	45.511	41.306	83	8
100	10	48	95.790	88.396	88	18
5	60	19	38.094	31.391	105	6
10	60	42	88.944	75.831	126	15
20	60	81	164.334	139.871	117	28
50	60	308	681.674	600.841	200	120
100	60	611	1.300.834	1.165.400	194	233
5	300	99	212.731	174.996	117	35
10	300	231	490.503	417.472	139	84
20	300	431	899.809	764.649	127	153
50	300	1.712	3.729.752	3.282.348	219	657
100	300	3.492	7.088.506	6.337.629	211	1.268
5	600	201	430.714	354.263	118	71
10	600	483	993.173	845.090	141	169
20	600	899	1.819.676	1.546.028	129	309
50	600	3.589	7.547.935	6.640.503	221	1.329
100	600	6.873	14.315.556	12.797.042	213	2.561
5	1800	634	1.303.415	1.071.921	119	215
10	1800	1.477	3.001.137	2.553.441	142	511
20	1800	2.754	5.498.855	4.671.317	130	935
50	1800	10.941	22.805.957	20.061.731	223	4.015
100	1800	20.770	43.250.653	38.655.660	215	7.736
vmware						
5	10	1	1.809	1.545	31	0

Kno- ten	SimT	WallT	Simulationsereignisse			Daten- bank- größe in MB
			vom Simulator	in Datenbank	pro Knoten- sekunde	
10	10	3	5.411	4.749	47	1
20	10	8	11.092	9.693	48	2
50	10	25	45.511	41.306	83	8
100	10	55	95.790	88.396	88	18
5	60	23	38.094	31.391	105	6
10	60	52	88.944	75.831	126	15
20	60	100	164.334	139.871	117	28
50	60	339	681.674	600.841	200	120
100	60	670	1.300.834	1.165.400	194	233
5	300	129	212.731	174.996	117	35
10	300	280	490.503	417.472	139	84
20	300	539	899.809	764.649	127	153
50	300	1.833	3.729.752	3.282.348	219	657
100	300	3.667	7.088.506	6.337.629	211	1.268
5	600	269	430.714	354.263	118	71
10	600	570	993.173	845.090	141	169
20	600	1.100	1.819.676	1.546.028	129	309
50	600	3.711	7.547.935	6.640.503	221	1.329
100	600	7.280	14.315.556	12.797.042	213	2.561
5	1800	788	1.303.415	1.071.921	119	215
10	1800	1.707	3.001.137	2.553.441	142	511
20	1800	3.318	5.498.855	4.671.317	130	935
50	1800	11.066	22.805.957	20.061.731	223	4.015
100	1800	21.693	43.250.653	38.655.660	215	7.736

Tabelle 9: Daten der mit SenseToRfm durchgeführten Messreihen (alle Zeiten in Sekunden)

In den Tabellen 10, 11 und 12 beziehen sich die “adc” und “crc” Zahlen notwendigerweise auf die vom Simulator generierten Ereignisse. Bei den “boot”, “power”, “other” Zahlen gibt es keinen Unterschied zwischen den vom Simulator erzeugten Ereignissen und den in die Datenbank geschriebenen. Aber bei den “radio” Zahlen wurden die in die Datenbank geschriebenen Ereignisse gezählt, deren Menge deutlich über der vom Simulator erzeugten liegt.

Knoten	SimT	Simulationsergebnisse		Ereigniskategorien					
		vom Simulator	in Datenbank	adc	crc	boot	power	radio	other
5	10	1.809	1.545	208	207	10	1.287	218	30
10	10	5.411	4.749	487	760	20	3.924	745	60
20	10	11.092	9.693	1.042	1.512	40	8.034	1.499	120
50	10	45.511	41.306	2.740	7.523	100	33.941	6.965	300
100	10	95.790	88.396	5.947	15.782	200	71.294	16.302	600
5	60	38.094	31.391	3.274	5.833	10	27.856	3.495	30
10	60	88.944	75.831	6.625	14.446	20	65.585	10.166	60
20	60	164.334	139.871	13.324	25.811	40	120.600	19.111	120
50	60	681.674	600.841	33.460	124.497	100	512.169	88.272	300
100	60	1.300.834	1.165.400	67.375	234.784	200	975.425	189.175	600
5	300	212.731	174.996	18.022	32.930	10	155.733	19.223	30
10	300	490.503	417.472	36.121	80.259	20	362.004	55.388	60
20	300	899.809	764.649	72.313	142.406	40	660.828	103.661	120
50	300	3.729.752	3.282.349	180.925	684.741	100	2.803.382	478.566	300
100	300	7.088.506	6.337.629	362.311	1.286.713	200	5.317.922	1.018.907	600
5	600	430.714	354.263	36.454	66.733	10	315.336	38.887	30
10	600	993.173	845.090	72.985	162.688	20	733.092	111.918	60
20	600	1.819.676	1.546.028	146.041	288.271	40	1.336.525	209.343	120
50	600	7.547.935	6.640.503	365.254	1.386.849	100	5.673.687	966.416	300
100	600	14.315.556	12.797.042	730.969	2.599.967	200	10.740.174	2.056.068	600
5	1800	1.303.415	1.071.921	110.191	202.112	10	954.342	117.539	30
10	1800	3.001.137	2.553.441	220.462	491.792	20	2.215.317	338.044	60
20	1800	5.498.855	4.671.317	440.980	871.650	40	4.039.073	632.084	120
50	1800	22.805.957	20.061.731	1.102.582	4.191.993	100	17.143.460	2.917.871	300
100	1800	43.250.653	38.655.660	2.205.634	7.858.935	200	32.450.085	6.204.775	600
Summe		115.821.865	102.333.114	6.235.726	20.779.194	1.850	86.721.075	15.604.638	5.550

Tabelle 10: Aufschlüsselung der Simulationsergebnisse nach Kategorien (in absoluten Zahlen)

Knoten	Simulations- zeit	Ereigniskategorien					
		adc	crc	boot	power	radio	other
5	10	11,5	11	0,55	71	12	1,66
10	10	9,0	14	0,37	73	14	1,11
20	10	9,4	14	0,36	72	14	1,08
50	10	6,0	17	0,22	75	15	0,66
100	10	6,2	16	0,21	74	17	0,63
5	60	8,6	15	0,03	73	9	0,08
10	60	7,4	16	0,02	74	11	0,07
20	60	8,1	16	0,02	73	12	0,07
50	60	4,9	18	0,01	75	13	0,04
100	60	5,2	18	0,02	75	15	0,05
5	300	8,5	15	0	73	9	0,01
10	300	7,4	16	0	74	11	0,01
20	300	8,0	16	0	73	12	0,01
50	300	4,9	18	0	75	13	0,01
100	300	5,1	18	0	75	14	0,01
5	600	8,5	15	0	73	9	0,01
10	600	7,3	16	0	74	11	0,01
20	600	8,0	16	0	73	12	0,01
50	600	4,8	18	0	75	13	0
100	600	5,1	18	0	75	14	0
5	1800	8,5	16	0	73	9	0
10	1800	7,3	16	0	74	11	0
20	1800	8,0	16	0	73	11	0
50	1800	4,8	18	0	75	13	0
100	1800	5,1	18	0	75	14	0
Summe		5,4	18	0	75	13	0,0048

Tabelle 11: Aufschlüsselung der Simulationsereignisse nach Kategorien in %
(relativ zu den Simulationsereignissen vom Simulator)

Knoten	Simulations- zeit	Ereigniskategorien					
		adc	crc	boot	power	radio	other
5	10	13,5	13	0,65	83	14	1,94
10	10	10,3	16	0,42	83	16	1,26
20	10	10,8	16	0,41	83	15	1,24
50	10	6,6	18	0,24	82	17	0,73
100	10	6,7	18	0,23	81	18	0,68
5	60	10,4	19	0,03	89	11	0,1
10	60	8,7	19	0,03	86	13	0,08
20	60	9,5	18	0,03	86	14	0,09
50	60	5,6	21	0,02	85	15	0,05
100	60	5,8	20	0,02	84	16	0,05
5	300	10,3	19	0,01	89	11	0,02
10	300	8,7	19	0	87	13	0,01
20	300	9,5	19	0,01	86	14	0,02
50	300	5,5	21	0	85	15	0,01
100	300	5,7	20	0	84	16	0,01
5	600	10,3	19	0	89	11	0,01
10	600	8,6	19	0	87	13	0,01
20	600	9,4	19	0	86	14	0,01
50	600	5,5	21	0	85	15	0
100	600	5,7	20	0	84	16	0
5	1800	10,3	19	0	89	11	0
10	1800	8,6	19	0	87	13	0
20	1800	9,4	19	0	86	14	0
50	1800	5,5	21	0	85	15	0
100	1800	5,7	20	0	84	16	0
Summe		6,1	20	0	85	15	0,0054

Tabelle 12: Aufschlüsselung der Simulationsereignisse nach Kategorien in %
(relativ zu den Simulationsereignissen in der Datenbank)