Reprogrammierungstechniken für drahtlose Sensornetzwerke

Christian Sternecker
Betreuer: Christoph Söllner
Seminar Sensorknoten - Betrieb, Netze & Anwendungen SS2012
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitekturen
Fakultät für Informatik, Technische Universität München
Email: sterneck@in.tum.de

KURZFASSUNG

Aufgrund ihres Einsatzgebietes müssen drahtlose Sensornetze (WSNs) häufig drahtlose reprogrammiert werden. Dabei treten diverse Probleme auf, die durch Betriebssysteme und Datendisseminationsprotokolle gelöst werden können. Diese Arbeit stellt zunächst die Handhabung der Reprogrammierung bei den Betriebssystemen TinyOS und Contiki dar. Anschließend werden die Datendisseminationsprotokolle, die die neuen Programme im Netz verteilen, XNP, Trickle, Deluge, Zephyr und das Selective Reprogramming Scheme von Krüger, Pfisterer und Buschmann genauer beschrieben und soweit möglich miteinander verglichen. Während die vier erstgenannten in dieser Reihenfolge aufeinander aufbauen und jeweils mehr Funktionen und eine bessere Performanz als der Vorgänger liefern, verfolgt das Selective Reprogramming Scheme einen anderen Ansatz und ist schlecht mit den anderen vergleichbar.

Schlüsselworte

Drahtlose Sensornetzwerke, WSN, XNP, TinyOS, Contiki, Trickle, Deluge, Zephyr, Selective Reprogramming Scheme, data dissemination protocol

1. EINLEITUNG

Drahtlose Sensornetzwerke (wireless sensor networks, WSNs) bestehen aus einer oft grossen Anzahl einzelner Sensorknoten, die sehr weit verteilt sein können. Sie sind dazu gedacht, über einen längeren Zeitraum eingesetzt zu werden. Dadurch kann es notwendig werden, einzelne oder alle Knoten zu reprogrammieren. Durch die reine Anzahl und die Verteilung der Knoten ist eine Reprogrammierung jedes einzelnen Knotens mittels eines angeschlossenen Laptops meist nicht praktikabel. Deshalb ist es erforderlich, die Knoten über den Funkkanal zu reprogrammieren.

Dabei ergeben sich jedoch mehrere Schwierigkeiten. Eine Schwierigkeit sind die geringen Ressourcen, die den einzelnen Sensorknoten zur Verfügung stehen, da ein Knoten deshalb keine zu komplexen Algorithmen in vertretbarer Zeit ausführen kann. So hat ein Mica2-Knoten nur 128 kByte Programmspeicher, 4 kByte RAM, 512 kByte externen Flash-Speicher und einen 7 MHz Prozessor[1]. Ein etwas neuerer TelosB-Knoten verfügt über 48 kByte Programmspeicher, 10 kByte RAM, 1024 kByte externen Flash-Speicher und einen 8 MHz Prozessor[2]. Zwar existieren mittlerweile wesentlich leistungsfähigere Knoten, in etwa die Sun Spots mit 8 MByte Programmspeicher, 1 MByte RAM und 400 MHz Prozessor[3], jedoch befinden sich noch viele Mica2-Knoten im Einsatz. außerdem benötigt die Kommunikation zwischen den Knoten bis zu zehn mal mehr Energie[1, 2] als der Betrieb der Prozessoren. Diese Energie wird dem Knoten in der Regel nur von einzelnen Batterien oder durch Energy Harvesting, etwa mittels Sollarzellen, zur Verfügung gestellt. Jede Benutzung des Funkkanals verkürzt die Restlaufzeit des Knotens.

Eine weitere Schwierigkeit stellt noch der, im Gegensatz zum Kabel, unzuverlässige Funkkanal dar, der die fehlerfreie Verteilung des neuen Programms im Netzwerk behindert. Das Laden des neuen Programms wird zusätzlich zu den begrenzten Ressourcen noch durch die Architektur mancher Betriebssystem der Knoten, etwa TinyOS (siehe Abschnitt 3.1), erschwert.

Die Teilschritte der Reprogrammierung eines WSN sind die Verteilung des neuen Programmcodes auf die Knoten, das Laden des erhaltenen Codes in den Programmspeicher des Knotens und die Ausführung des neuen Codes. Die Verteilung ist Aufgabe von Datendisseminationsprotokollen (data dissemination protocols), Laden und Ausführen sind Aufgaben des Betriebssystems auf dem jeweiligen Knoten.

Diese Arbeit gibt einen Überblick darüber, wie diese Aufgaben in verschiedene Betriebssysteme und Datendisseminationsprotokollen gelösst werden.

Zuerst wird in Abschnitt 2 der grundlegende Aufbau eines drahtlosen Sensornetzwerks beschrieben.

Im Abschnitt 3 wird auf zwei Betriebssysteme, namentlich TinyOS (Abschnitt 3.1) und Contiki (Abschnitt 3.2) eingegangen. Dabei werden die Eigenschaften des Betriebssystems und die sich daraus ergebenden Lösungen zum Laden von neuem Code erörtert.

Der Abschnitt 4 beschreibt die Lösungen der Verteilung des Codes durch eine Auswahl an Datendisseminationsprotokollen und vergleicht deren Performanz miteinander soweit möglich.

Von den zahlreichen Protokollen ([4] beinhaltet eine unvollständige Liste) werden hier XNP (Abschnitt 4.1), Trickle (Abschnitt 4.2), Deluge (Abschnitt 4.3), Zephyr (Abschnitt 4.4) und das Selective Reprogramming Scheme von Krüger, Pfisterer und Buschmann (Abschnitt 4.5) besprochen.

Abschnitt 5 schließlich fasst die Arbeit zusammen.

2. AUFBAU EINES DRAHTLOSEN SENSORNETZWERKS

Ein drahtloses Sensornetz besteht aus einer Anzahl einzelner Sensorknoten. Mindestens einer dieser Knoten fungiert als Basisstation. Die Basisstation dient dazu, die vom gesamten Netwerk gemessenen Daten einzusammeln und auszuwerten. Oft werden die Daten zur Auswertung an fest mit der Basisstation verbundene, leistungfähigere Systeme weitergegeben. Die Basisstation dient auch als Ausgangspunkt, um Daten wie etwa Programmcode zur Reprogrammierung der restlichen Knoten im Netzwerk zu verteilen. Da meist nicht alle restlichen Knoten des Netzwerkes direkten Funkkontakt zur Basisstation haben, ist es nötig, Daten über mehrere Zwischenstationen (Hops) zu übertragen. Daher muss eines von zahlreichen Routingprotokollen[5] verwendet werden. Das verwendete Routingprotokoll bestimmt die Rolle der restlichen Knoten im Netzwerk. Zu beachten ist, dass die in Abschnitt 4 vorgestellten Protokolle nicht das vorhandene Routingprotokoll des Netzes verwenden, sondern ihre Daten auf eigene Weise verteilen.

3. BETRIEBSSYSTEME

In diesem Abschnitt wird die Reprogrammierung in den beiden Betriebssystemen TinyOS und Contiki beschrieben.

3.1 TinyOS

TinyOS[6] wird seit 1999 von der UC Berkeley als Betriebssystem für WSNs entwickelt. Es ist im C-Dialekt NesC (network embedded system C[7]) geschrieben und ist komponentenbasiert und ereignisgesteuert. TinyOS unterstützt seit Version 1.1 die drahtlose Reprogrammierung. Dafür wurde das von Crossbow Technologies Inc. entwickelte XNP (Xbow in-network programming)-Protokoll (siehe Abschnitt 4.1) verwendet. Seit Version 1.1.8 wird das Deluge-Protokoll (siehe Abschnitt 4.3) der UC Berkeley verwendet.
Bei der Reprogrammierung wird das neue Programm an die Knoten des Netzwerks verteilt und vom jeweilien Knoten in den externen Flash-Speicher des Knotens geschrieben. Sobald das neue Programm vollständig im Flash-Speicher ist, wird der Bootloader aufgerufen, der am Anfang des MCU-internen Programmspeichers steht. Er überprüft das neue Programm auf Vollständigkeit und Fehlerfreiheit und lädt es anschließend in den Programmspeicher. Abschließend startet er den Knoten neu. Bei TinyOS wird das Anwendungsprogramm immer zusammen mit dem Betriebssystem kompiliert und zusammen auf den Knoten aufgespielt. außerdem ist das Linking in TinyOS statisch. Aus diesen Gründen ist es nicht möglich, einen Knoten nur teilweise zu reprogrammieren. Bei jeder noch so kleinen Änderung, etwa dem Ändern eines Messintervalls, ist es nötig, das gesamte System des Knotens zu ersetzen, sofern diese Stelle im Programm nicht speziell programmiert wurde um von außen konfigurierbar zu sein. Um diesem Problem zu begegnen, wurde von der UC Berkeley die virtuelle Maschine Maté[8] entwickelt. Jedoch wurde Maté nicht für TinyOS 2.x portiert und ist daher nicht mehr relevant.

3.2 Contiki

Seit 2003 wird Contiki[9, 10] am Swedish Institute of Computer Science als Betriebssystem für vernetzte Systeme mit begrenzten Ressourcen unter anderem Sensornetzen entwickelt. Es ist in C geschrieben und wie TinyOS ereignisgesteuert. Anders als TinyOS ist es in Contiki nicht notwendig, das Anwendungsprogramm zusammen mit dem Betriebssystem zu kompilieren. Das auf einem Knoten laufende System besteht bei Contiki aus zwei Teilen. Der eine ist der Kern, bestehend aus dem Kernel, dem Programloader, einigen Bibliotheken, Teilen des Runtimes und dem Kommunikationsstapel. Der zweite Teil ist das momentan laufende Programm. Abbildung 1 zeigt diesen Aufbau. Das Programm oder Teile davon können zur Laufzeit ausgetauscht werden. Auf diese Weise ist es möglich, die Datenmenge, die bei der Reprogrammierung durch das Netzwerk verbreitet werden muss, zu reduzieren. Die auszutauschenden Programmteile werden beim Empfangen in den Flash-Speicher geschrieben. Sobald das Programm vollständig empfangen ist, lädt der Programloader das Programm in den Programmspeicher des Knotens und ruft seine Initialisierungsfunktion auf. Sollen Teile des Kerns, etwa im Kommunikationsstapel geändert werden, so ist es auch bei Contiki erforderlich, das gesamte System des Knotens auszutauschen. Um die auszutauschenden Programmteile beziehungsweise das gesammte neue System im Netzwerk zu verteilen, benutzte Contiki in Version 1.0 ein XNP-ähnliches Protokoll. Seit Version 2.0 wird Trickle (siehe Abschnitt 4.2) verwendet.

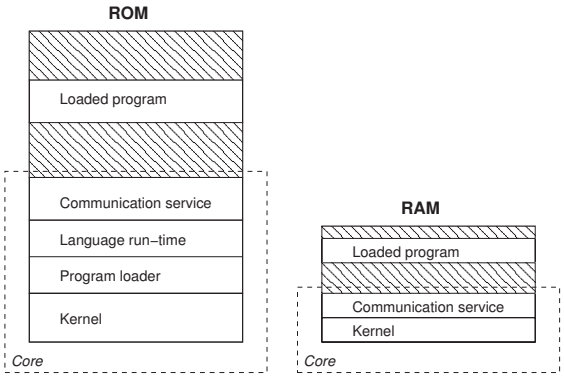


Abbildung 1: Teile eines laufenden Systems in Contiki. Entnommen aus [9].

4. DATENDISSEMINATIONS-PROTOKOLLE

Dieser Abschnitt beschreibt die Datendisseminationsprotokolle XNP, Trickle, Deluge, Zephyr und das Selective Reprogramming Scheme von Krüger, Pfisterer und Buschmann näher und vergleicht ihre Performanz soweit möglich. Da die Kommunikation zwischen den Knoten von allen Aktionen, die die Knoten bei diesen Protokollen ausführen, den Großteil der Energie verbraucht, ist der Energieverbrauch eines Protokolls in etwa proportional zu der Menge insgesamt an gesendeten Paketen.

4.1 XNP

XNP[11, 12] ist ein relativ einfaches Datendisseminationsprotokoll. Es wurde von den TinyOS Versionen 1.1 bis 1.1.7 verwendet. XNP kann nur zur Reprogrammierung von Knoten verwendet werden, die von der Basisstation aus ohne Zwischenstation erreichbar sind. Um das neue Programm im Netzwerk zu verteilen, wird es von einem PC auf die Basis-

station überspielt und von dort aus versendet. Das Protokoll unterscheidet bei der Reprogrammierung drei Phasen:

- die Downloadphase, in der das neue Programm im Netzwerk veteilt wird
- die Nachfragephase, in der das verteilte Programm auf vollständigkeit geprüft wird
- die Reprogrammierungsphase, in der die Knoten reprogrammiert werden

In der Downloadphase sendet die Basisstation eine Download-Start-Nachricht an alle Knoten des Netzwerks. Diese reservieren daraufhin die notwendigen Ressourcen für die Reprogrammierung. Das Senden der Download-Start-Nachricht wird mehrmals wiederholt, um die Wahrscheinlichkeit, dass ein Knoten sie nicht empfangen hat zu verringern. Anschließend werden die Nachrichten mit dem neuen Programm an alle Knoten im Netz gesendet. Jede Nachricht wird einmal versendet und beinhaltet 16 Byte des Programms. Die Knoten schreiben die empfangenen Teile in ihren externen Flash-Speicher. Sobald alle Teile des Programms versendet wurden, beginnt die Nachfragephase. Die Basisstation versendet eine Download-Terminate-Nachricht an alle Knoten. Die Knoten prüfen daraufhin, ob sie alle Teile des Programms erhalten haben. Die Basisstation sendet daraufhin eine Anfrage (Request) an alle Knoten, ob sie alle Programmteile erhalten haben. Vermisst ein Knoten einen Programmteil, so sendet er eine Anfrage für diesen Teil an die Basisstation. Diese versendet daraufhin den angefragten Teil erneut an alle Knoten. Anschließend sendet die Basisstation die Anfrage erneut an alle Knoten. Sobald die Basissation auf diese Anfrage wiederholt keine Antwort erhalten hat, beginnt die Reprogrammierungsphase. In dieser Phase sendet die Basisstation eine Reprogrammierungsanfrage an alle Knoten. Sobald ein Knoten diese Anfrage empfangen hat, ruft er seinen Bootloader auf. Dieser kopiert das neue Programm in den Programmspeicher und startet den Knoten neu. Abbildung 2 zeigt einen beispielhaften Ablauf von XNP bis zum Aufrufen des Bootloaders.

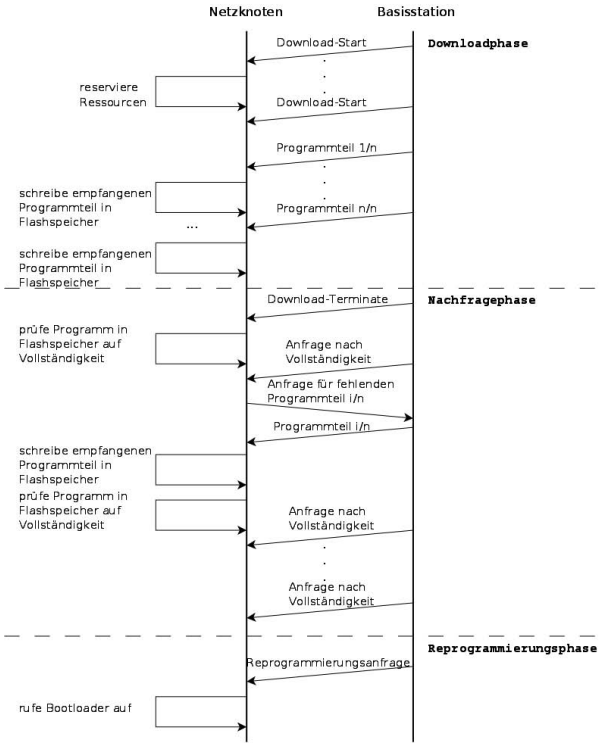


Abbildung 2: Beispielhafter Ablauf von XNP zwischen der Basisstation und einem Netzknoten. Dem Netzknoten fehlt am Ende der Downloadphase nur der Programmteil i. Die Basisstation versendet all ihre Nachrichten per Broadcast an alle Knoten und reagiert auf Anfragen aller Knoten des Netzes.

XNP hat mehrere offensichtliche Schwächen. Die erste ist, dass es keine Multi-Hop-Reprogrammierung unterstützt. Eine weitere ist, dass keine Bestätigungen für den Empfang der Programmteile verwendet wird, sondern nur Anforderungen nicht vorhandener Teile. Falls diese Anforderungen mehrfach verloren gehen, wird der Knoten entweder gar nicht, oder mit dem unvollständigen Programm reprogrammiert. In letzterem Fall läuft kein funktionierendes Programm mehr auf dem Knoten und es ist nicht möglich, ihn erneut zu reprogrammieren ohne ihn an einen PC anzuschließen. Laut[12] ist XNP jedoch sehr zuverlässig. Allerdings wurden nur relativ kleine WSNs mit höchstens 16 Knoten getestet. Bei größeren Netzwerken steigt das Risiko für verlorene Nachrichten jedoch an. Nach Messungen in[12] dauerte das Verteilen eines Programms der Grösse 37000 Byte inklusive dem erneuten Versenden fehlender Teile im Schnitt 159 Sekunden, im schlechtesten Fall betrug die Gesamtdauer 250 Sekunden.

4.2 Trickle

Trickle[13] wurde von der UC Berkeley entwickelt und wird von Maté und Contiki ab Version 2.0 verwendet. Es benutzt den so genannten "polite gossip"-Ansatz, um neue Programmteile im Netzwerk möglichst schnell und mit möglichst wenig Overhead zu verteilen. Dazu teilen die Knoten dem Netzwerk regelmäßig mit, auf welchem Stand sich ihr Programm befindet. Bekommt ein Knoten mit, dass ein anderer auf dem gleichen Stand ist, so geht er davon aus, dass er selbst auf dem neuesten Stand ist und sendet selbst keine neue Nachricht, da dies keinen Mehrwert an Information darstellen würde. Hört er von einem Knoten mit einer neueren Version des Programms, teilt er dem Netzwerk seinen veralteten Stand mit. Erhält ein Knoten die Nachricht, dass ein anderer auf einem älteren Stand ist, so sendet er seine Version des Prgramms an alle erreichbaren Knoten, die sich daraufhin reprogrammieren. Auf diese Weise verbreitet sich der neueste Stand im gesamten Netz.

Trickle funktioniert im Detail wie folgt: Jedes Programm auf einem Knoten trägt eine Versionsnummer. Version x eines Programms wird als ϕ_x bezeichnet. Jeder Knoten teilt die Zeit in Intervalle der Länge τ ein. außerdem besitzt jeder Knoten noch die Parameter c, k und t. Am Anfang jedes Zeitintervalls wird t zufällig auf einen Wert aus dem Intervall $[0, \tau]$ gesetzt, c wird auf 0 und k auf einen festen Wert (1 oder 2) gesetzt. Sobald der Zeitpunkt t erreicht ist, überprüft der Knoten, ob $c < k$ gilt. Ist das der Fall, sendet der Knoten eine Nachricht mit den Versionsnummern seines Programms per Broadcast an alle erreichbaren Knoten. Der Knoten hört zu jedem Zeitpunkt den Kanal ab. Empfängt er eine Nachricht von einem anderen Knoten mit den gleichen Versionsnummern, so wird c um eins erhöht. Wenn der Knoten eine Nachricht mit einer niedrigeren Versionsnummer empfängt, sendet er sein Programm per Broadcast an das

Netzwerk. Empfängt er hingegen eine Nachricht mit einer höheren Versionsnummer, so sendet er sofort eine Nachricht mit seiner älteren Versionsnummer, um so den anderen Knoten zu veranlassen, ihm die neue Version des Prgramms zu senden. Anschließend empfängt er die neue Version, schreibt sie in seinen externen Flash-Speicher und reprogrammiert sich möglichst sofort selbst.

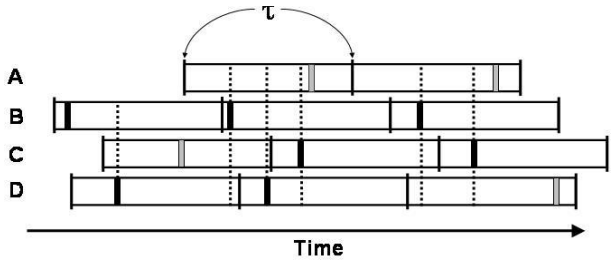
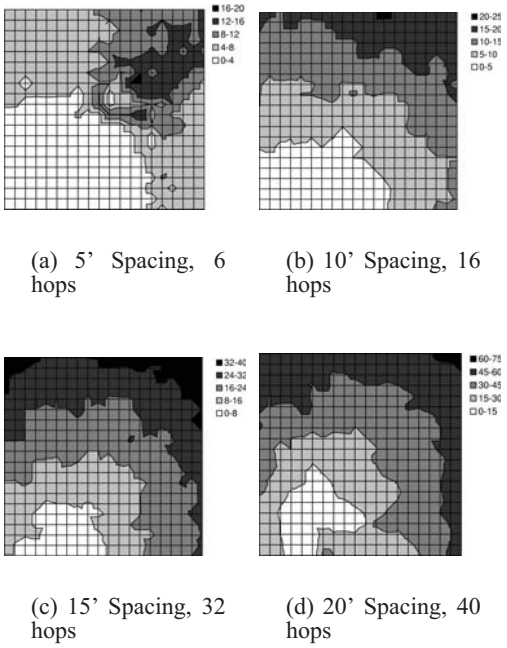


Abbildung 3: Beispiel für das short-listen-Problem, entnommen aus[13]. Die langen Striche sind die Intervallgrenzen. Dicke schwarze Striche sind die gesendeten Nachrichten, dicke helle Striche sind nicht gesendete Nachrichten. Die gestrichelten Linien verdeutlichen die Unterdrückung der Nachrichten der anderen Knoten im momentanen Intervall.

In nicht zeitsynchronisierten Netzen kann es mit Trickle zum so genannten "short-listen"-Problem kommen. Dabei führt die Verschiebung der Intervalle τ zwischen den Knoten dazu, dass mehr Nachrichten mit Metadaten gesendet werden als es in einem synchronen Netz der Fall wäre. Abbildung 3 zeigt ein Beispiel für das Problem. Die redundanten Nachrichten behindern die restliche Kommunikation im Netz und führen so zu einem eventuell höheren Energieverbrauch der Knoten. Um das Problem zu reduzieren, wird t in der Regel auf einen Wert im Intervall $[\frac{\tau}{2}, \tau]$ statt $[0, \tau]$ gesetzt. Dadurch reduziert sich die Zahl der redundanten Nachrichten fast auf den Wert von synchronen Netzen.
Eine weitere Methode, um die Zahl der gesendeten Nachrichten zu reduzieren, ist die dynamische Anpassung des Zeitintervalls τ. Dabei werden für das Intervall eine Untergrenze τ_l und eine Obergrenze τ_h festgelegt. τ wird zu Beginn auf τ_l gesetzt. Nach dem Ablauf eines Intervalls wird τ bis auf maximal τ_h verdoppelt. Beim Empfang einer Nachricht mit einer neueren Versionsnummer oder einer neuen Version des Programms wird τ wieder auf τ_l gesetzt. Auf diese Weise senden Knoten, die keine neuen Informationen haben mit der Zeit immer weniger Nachrichten. Das Verteilen einer neuen Version des Programms wird so nicht beeinträchtigt.
Zur Performanz von Trickle wurden in[13] Messungen sowohl mit dem zu TinyOS gehörenden Simulator TOSSIM[14] als auch in einem echten Netzwerk durchgeführt. Gemessen wurde jeweils die Zeit, die benötigt wurde, um alle Knoten zu reprogrammieren in Abhängigkeit von der Anzahl Hops des Netzwerkes und der Einstellung für τ_h.
Auf den Knoten war Maté im Einsatz, der neue Programmteil bestand aus einem Paket der Größe 30 Byte. Das echte Netzwerk bestand aus 14 Mica2-Knoten und war mehrere Hops breit. Die Simmulierten Netzwerke bestanden aus einem Gitternetz mit 20×20 Knoten mit variablem Abstand und damit variabler Hopzahl. Alle Knoten konnten immer erfolgreich reprogrammiert werden. Die Erhöhung von τ_h von einer Minute auf 20 Minuten führte bei den simulierten Netzen zu keinem nennenswerten Unterschied bei der gemesssenen Zeit. Im realen Netzwerk führte der höhere Wert zu einer starken Erhöhung des Durchschnitts der gemessenen Zeit von 22 auf 32 Sekunden. In den simulierten Netzwerken stieg die Zeit wie erwartet mit der Anzahl der Hops. Das neue Programm breitete sich immer außer im dichtesten Netz immer in der erwarteten Wellenform aus. Abbildung 4 veranschaulicht diese Ausbreitung und die dazu notwendige Zeit.



(a) 5' Spacing, 6 hops

(b) 10' Spacing, 16 hops

(c) 15' Spacing, 32 hops

(d) 20' Spacing, 40 hops

Abbildung 4: Ausbreitung des neuen Programms mit Trickle im simulierten Netzwerk entnommen aus[13]. Die angegebene Anzahl der Hops bezieht sich auf den Abstand zwischen dem Knoten ganz links unten und dem ganz links oben. Die Ausbreitungszeit ist in Sekunden angegeben.

4.3 Deluge

Während Trickle primär dazu gedacht ist, kleinere Programmteile im Netzwerk zu verteilen, ist das darauf basierende Protokoll Deluge[15] auch zur Verteilung größerer Datenmengen geeignet. Deluge wurde ebenfalls an der UC Berkeley entwickelt und ist seit Version 1.1.8 Bestandteil von TinyOS. Es funktioniert ähnlich wie der "polite gossip"-Ansatz von Trickle. Auch hier sendet jeder Knoten den Stand seines Programms periodisch an alle Knoten in Reichweite. Knoten, die auf dem selben Stand sind, halten sich auch hier mit dem Senden zurück. Ein Knoten, der eine Nachricht von einem Knoten auf einem älteren Stand mithört, sendet hier nach einer kurzen, zufälligen Wartezeit, um Kollissionen mit anderen Knoten mit der selben Version zu vermeiden, eine neue Nachricht mit seinem Stand und genaueren Informationen über sein Programm an das gesamte Netz. Ein Knoten, der von einer neueren Version des Programms erfährt, versendet nun eine Nachricht, in der er die ihm fehlenden Programmteile spezifisch anfordert. Ein Knoten, der diese Anforderung hört und erfüllen kann, sendet die angeforderten Teile an das Netz. Knoten, die die gesendeten Teile ebenfalls brauchen, speichern sie genau wie der anfordernde Knoten in ihrem externen Flash-speicher. Durch das stückweise Versenden ist es

doi: 10.2313/NET-2012-08-2_03

in Deluge möglich, die Verbreitung des Programms mittels Pipelining zu beschleunigen, wenn das Netzwerk über mindestens drei Hops verfügt.
Deluge funktioniert im Detail wie folgt: Jedes Programm ist ein Objekt. Jedes Objekt wird in Seiten gleicher Länge eingeteilt, die wiederum aus Paketen gleicher Länge bestehen. Pakete und Seiten verfügen über einen CRC der Länge 16 Bit. Jeder Knoten speichert zu jedem Objekt ein Objektprofil bestehen aus einer Versionsnummer v und einen Altersvektor $\mathbf{a} =< a_0, a_1, .., a_{p-1} >$, der angibt, welche Seite wann zuletzt geändert wurde. Seite i wurde in Version $v - a_i$ geändert. v wird bei jeder Änderung des Programms um eins erhöht. Wie in Trickle wird die Zeit auch hier in Intervalle unterteilt. Jeder Knoten besitzt die Variablen $\tau_{m,i}$, die die Länge des i-ten Intervalls angibt, τ_l und τ_h, die die Ober- und Untergrenze der Intervalllänge angeben, den festen Wert k und eine Zusammenfassung $\phi = \{v, \gamma\}$ seines Programmstandes. Dabei ist v die Versionsnummer seines Programms und γ die Nummer der verfügbaren Seite mit der höchsten Nummer. Eine Seite ist bei Deluge verfügbar, wenn alle Pakete davon vorhanden und fehlerfrei sind und alle Seiten mit niedriegeren Nummern ebenfalls verfügbar sind. außerdem befindet sich jeder Knoten immer in einem von drei Zuständen, Maintain, Receive(RX) oder Transmit(TX). Ein Knoten startet im Zustand Maintain. Nur in diesem Zustand teilt der Knoten die Zeit in Intervalle ein. Der Intervall i beginnt zum Zeitpunkt $t_i = t_{i-1} + \tau_{m,i-1}$. Zu Beginn jedes Intervalls überprüft der Knoten, ob er im letzten Intervall eine Nachricht mit einer Zusammenfassung $\phi' \neq \phi$ oder eine Anfragenachricht oder Seitenpakete empfangen oder mitgehöht hat. Ist nichts davon der Fall, so wird $\tau_{m,i} = min(2\tau_{m,i-1}, \tau_h)$ gesetzt, also die Länge dieses Intervalls bis zur Obergrenze verdoppelt. außerdem wird ein Zufallswert r_i aus dem Intervall $[\frac{\tau_h}{2}, \tau_h]$ erstellt. Dies geschieht aus dem selben Grund wie bei Trickle, um das short-listen-Problem zu reduzieren. Hat der Knoten jedoch eine der oben genannten Nachrichten empfangen oder mitgehörtr, so setzt er $\tau_{m,i}$ auf τ_l zurück und beginnt sofort ein neues Intervall. Zum Zeitpunkt $t_i + r_i$ im Intervall i überprüft der Knoten, ob und wie viel Zusammenfassungen $\phi' = \phi$ er in diesem Intervall bereits empfangen hat. Hat er k oder weniger empfangen, sendet er selbst seine Zusammenfassung ϕ. Hat der Knoten jedoch vor dem Zeitpunkt $t_i + r_i$ eine Zusammenfassung ϕ' mit $v' < v$, also einer kleineren Versinsnummer als seine eigene, empfangen, so sendet er bei $t_i + r_i$ sein Objektprofil an das Netzwerk, sofern er nicht vorher k oder mehr Objektprofile mit der gleichen Versionsnummer mitgehört hat, die ander Knoten bereits geschickt haben. Empfängt ein Knoten ein Objektprofil, so benutzt er es, um seine Zusammenfassung zu aktualisieren. v wird auf $v\prime$ gesetzt und mittels $\mathbf{a}'$ wird der höchstmögliche Wert für γ gesetzt. Ein Knoten, der im Intervall i eine Zusammenfassung ϕ' mit $v' = v$ und $\gamma' > \gamma$ empfängt, prüft einerseits, ob er innerhalb der Zeit $2\tau_{m,i}$ zuvor eine Anfrage für eine Seite mit Nummer $p \leq \gamma$ empfangen hat und andererseits, ob er ein Paket einer Seite mit Nummer $p \leq \gamma$ erhalten hat. Trifft keiner dieser Fälle zu, so wechselt er in den Zustand RX. Empfängt ein Knoten im Zustand Maintain eine Anfrage für eine Seite mit Versionsnummer v und Seitennummer $p \leq \gamma$, so wechselt er in den Zustand TX. Ist der Knoten im Zustand RX, so wartet er für einen Zeitraum $\omega T_{tx} + r$. ω ist dabei eine festgelegte Anzahl an Paketen, T_{tx} ist die Zeit, die ein Paket zum senden braucht, r ist ein Zufallswert in der Größenordnung des festgelegten Wertes τ_r. Hat er bis dahin kein Paket und keine Anfrage erhalten, so sendet er eine Anfrage nach dem nächsten benötigten Paket. Erhält er ein Paket, so überprüft er den CRC und wartet erneut für den Zeitraum $\omega T_{tx} + r$ bis er die nächste Anfrage schickt. Sobald der Knoten eine feste Anzahl λ an Anfragen verschickt hat, auf die er fehlerfreie Pakete nur mit einer Rate unter einem Schwellwert α empfangen hat, so wechselt er trotz unvollständigem Objekt wieder in den Zustand Maintain. Diese Verhalten dient dazu, asymmetrische Links tolerieren zu können. Ein Knoten im Zustand RX, der alle Pakete für die angeforderten Seiten erhalten und auf Fehlerfreiheit überprüft hat, aktualisiert sein Objektprofil und wechselt in den Zustand Maintain zurück. Ein Knoten im Zustand TX verfügt über eine Menge Π_{tx} an angeforderten und noch zu versendenden Paketen. Sobald er eine Anforderung erhält, die die Menge Π'_{rx} anfordert, wird diese Menge der der schon zu versendenden Pakete zugeschlagen $\Pi_{tx} = \Pi_{tx} \cup \Pi'_{rx}$. Der Knoten versendet alle Pakete aus Π_{tx} im round-robin Verrfahren und entfernt sie dann aus Π_{tx}. Sobald $\Pi_{tx} = \emptyset$ ist, wechselt der Knoten in den Zustand Maintain zurück. Abbildung 5 stellt ein vereinfachtes Zustandsdiagramm des Protokolls dar.

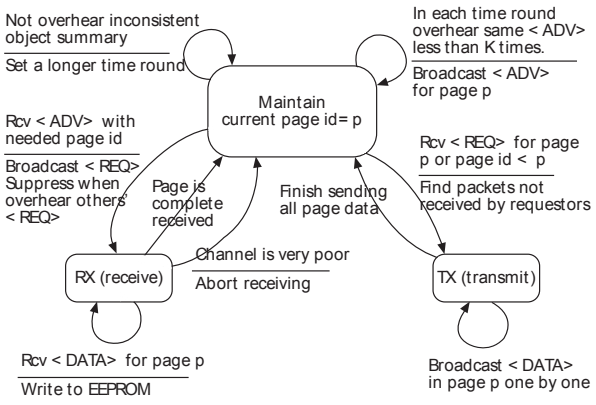


Abbildung 5: Vereinfachtes Zustandsdiagramm für Deluge, entnommen aus [4]. <ADV> (Advertisement) steht für Nachrichten mit Zusammenfassung, <REQ> (Request) für Anfragen.

Wie Trickle wurde auch Deluge sowohl in einem realen Netwerk als auch in der Simulation mit TOSSIM getestet. Das reale Netz bestand aus 75 Mica2-Knoten. τ_l wurde auf 2 Sekunden, τ_h auf 60 Sekunden und k auf 1 gesetzt. Diese Werte entsprechen denen aus den Tests für Trickle in Abschnitt 4.2. τ_r wurde auf 0.5 Sekunden gesetzt, λ auf 2 und ω auf 8. Für den Test wurde das reale Netzwerk mit einer variierenden Anzahl an Seiten reprogrammiert. Eine Seite hatte die Größe 1104 Byte und war in 48 Pakete zu je 23 Byte aufgeteilt. In jedem Durchlauf konnten alle Knoten fehlerfrei reprogrammiert werden. Die Zeit um das Netzwerk zu reprogrammieren stieg nicht linear mit der Anzahl der Seiten, sondern langsamer. Der Grund dafür war das Pipelining von Deluge. Jeder Knoten kann bereits erhaltene Seiten weiter verbreiten, ohne auf die restlichen Seiten des Objekts warten zu müssen, was die Ausbreitung vor allem bei größeren WSNs stark beschleunigt. Deluge erreichte in diesem Test eine durchschnittliche Datenübertragungs-

rate von 88,4 Byte/ Sekunde. Dies ist wesentlich schneller als die etwa 1,5 Byte/Sekunde von Trickle. Der Overhead an Zusammenfassungs- und Anfragenachrichten betrug zusammen 18.18%. Der Energieverbrauch ist bei Trickle und Deluge in etwa gleich, da alle Knoten permanent eingeschaltet bleiben müssen.
Die simulierten Netzwerke bestanden bei Deluge wie bei Trickle aus einem Gitternetz von 20×20 Knoten mit variablem Abstand. Hier wurde eine variable Zahl von Seiten der Größe 552 Byte aus 24 Paketen zu je 23 Bytes zum reprogrammieren des Netzes verwendet. Der Abstand zwischen zwei Knoten betrug 10 beziehungsweise 15 Fuß. Wie in Abbildung 6 zu sehen ist, geschieht die Ausbreitung bei 15 Fuß Abstand im erwarteten Wellenmuster, während das bei 10 Fuß Abstand nicht der Fall ist. Dies ist auf das hiddenterminal-problem zurückzuführen. Dabei senden zwei Knoten, die sich gegenseitig nicht sehen können, gleichzeitig an den gleichen Knoten, was zu Kollissionen und damit zum Paketverlust führt. Die Datenrate, mit der sich das neue Programm ausbreitet ist auch hier im Vergleich zu Trickle wesentlich höher. Wegen der Unterstützung größerer Datenmenge und der höheren Datenrate ist Deluge Trickle wesentlich überlegen und daher vorzuziehen.

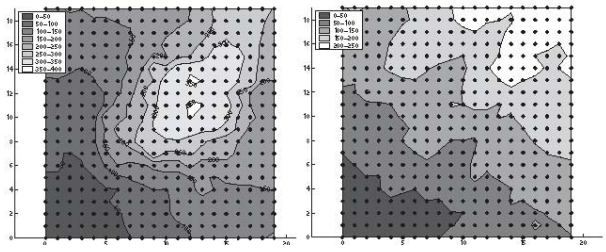


Abbildung 6: Ausbreitung einer neuen Seite bei Deluge. Links mit 10 Fuß Abstand zwischen den Knoten rechts mit 15 Fuß. Zeitangaben in Sekunden. Einzelbilder entnommen aus[15]

Parallel zu Deluge wurde für TinyOS der Bootloader TOSBoot entwickelt. Dieser erlaubt einem Knoten, mehrere Programmabbilder im externen Flash-Speicher zu halten und jeweils eins davon nach eine Neustart in den Programmspeicher zu laden. Deluge unterstützt dies, indem es bei seinen Zusammenfassungsnachrichten immer die Objektprofile aller Programme mitschickt.

4.4 Zephyr

Zephyr[16] wurde 2009 an der School of Electrical and Computer Engeneering der Purdue University entwickelt. Aufbauend auf Deluge beschleunigt Zephyr die Reprogrammierung von WSNs, indem es die Menge der zu versendenden Daten auf ein Minimum zu reduzieren versucht. Dazu bedient es sich einerseits einer Methode aus dem Deluge ähnlichen Protokoll Stream[17] und andererseits einer optimierten Variante des Rsync-Algorithmus[18]. Zephyr wurde zuerst für TinyOS implementiert, kann jedoch ohne Probleme von Contiki verwendet werden.
Stream reduziert die Größe der zu versenden Programmabbilder, indem es die Komponenente, die für das Ausführen der Reprogrammierung notwendig ist, nicht als Bestandteil des Programms sondern als seperate Komponente behandelt. Diese Komponenete wird von den einzelnen Knoten nicht im Programmspeicher sondern im externen Flash-Speicher gespeichert, das Programm enthält nur den Code, der notwendig ist, um Aktuallisierungen zu erkennen. Sobald der Knoten reprogrammiert werden muss, ruft das Programm des Knotens den Bootloader auf. Dieser lädt nun die Reprogrammierkomponente in den Programmspeicher und startet den Knoten neu. Die Reprogrammierkomponente führt nun die Reprogrammierung durch und ruft erneut den Bootloader auf, der daraufhin das neue Programm in den Programmspeicher lädt und den Knoten erneut neu startet. Auf diese Weise kann das neue Programm ohne die Komponente im Netz verteilt werden. Dieser Ansatz ermöglicht es ein einerseits, die Größe des Programmabbildes zu reduzieren und dadurch sowohl die Verteilung des Programms im Netzwerk als auch die Reprogrammierung der einzelnen Knoten zu beschleunigen, andererseits verhindert er aber Änderungen an der Reprogrammierkomponente und belegt permanent einen Platz im externen Flash-Speicher der Knoten.
Zephyr versendet jedoch nicht das kürzere Programmabbild von Stream, sondern nur ein so genanntes delta script, das nur den Unterschied auf zwischen zwei Versionen des Abbildes beinhaltet. Das neue Abbild wird mit Hilfe des delta scripts aus dem alten Abbild erstellt. Um das delta script zu erstellen, verwendet Zephyr eine angepasste Version des Rsync Algorithmus. Dieser wurde 2000 an der Australien National University entwickelt und diente ursprünglich dazu, Daten zwischen zwei Rechnern über ein Netzwerk mit geringer Bandbreite zu synchronisieren. Zephyr modifiziert Rsync so dass teuere Rechenoperationen nicht auf den Knoten, sondern nur auf dem an die Basisstation angeschlossenen Rechner ausgeführt werden müsen. Der modifizierte Algorithmus unterteilt das alte und das neue Abbild in Blöcke fester Länge. Anschließend berechnet er für jeden Block des alten Abbildes zwei Hashwerte. Der erste wird mit einer in[18] spezifizierten schnellen Methode berechnet, der zweite mittels der kryptographischen Hashfunktion MD4[19]. Beide Werte werden für jeden Block in einer Hashtabelle gespeichert. Dann wird für jeden Block im neuen Abbild der schnelle Hashwerte berechnet. Für jeden Block wird mittels der Hashtabelle nach einem gleichen Block im alten Abbild gesucht. Bei einem Treffer wird auch der MD4 Wert des neuen Blocks berechnet. Dies dient dazu, Kollissionen möglichst auszuschließen. MD4 wird nur bei möglichen Treffern berechnet um Rechenzeit zu sparen. Anschließend wird das delta script erstellt. Für jeden Block aus dem neuen Abbild, der eine Entsprechung im alten hat, wird die Codezeile
`COPY <oldOffset> <newOffset> <len>`
erstellt, für jeden Block ohne Entsprechung die Zeile
`INSERT <newOffset> <len>`.
Das delta script wird nun optimiert indem der längste mit dem alten Abbild übereinstimmende Abschnitt im neuen Abbild gesucht wird. Dies erreicht man durch suchen der längsten ununterbrochenen Folgen von `COPY` -Anweisungen. Diese `COPY`-Anweisungen werden nun zu einer verschmolzen.

```
COPY <old> <new> <B>
COPY <old+1> <new+1> <B>
...
COPY <old+k> <new+k> <B>
```

wird zu

```
COPY <old> <new> <(k+1)*B> .
```

So wird das delta script wesentlich verkleinert. Jedoch kann

es trotzdem auch bei kleinen Änderungen im Programm zu großen delta scripts kommen. Werden etwa durch die Veränderungen am Programm die Anfangsadressen von Funktionen verschoben, so ändern sich im binären Abbild alle Aufrufe dieser Funktionen, was das delta script stark aufblähen kann. Um solchen und ähnlichen Problemen entgegenzuwirken nimmt Zephyr auch Änderungen des Programms auf höherer Ebene vor. Diese Änderungen geschehen mit allen Versionen des Programms, auch mit der ersten und immer vor der Erstellung des delta scripts. Zephyr modifiziert dafür das Linking des Programms. Eine dieser Modifikationen sorgt dafür, dass die Unterbrechungsbehandlung des Programms trotz Änderungen immer an der gleichen Stelle im Speicher steht. Die zweite Änderung ist das Erstellen einer Tabelle für indirekte Funktionsaufrufe. Funktionen werden in einem Programm in der Regel mehr als einmal aufgerufen. Um die dadurch nötigen Änderungen am Programm auf ein Minimum zu beschränken, erstellt Zephyr am Ende des Programmspeichers eine Tabelle mit den Startadressen aller Funktionen. Alle Funktionsaufruf werden so geändert, dass sie die entsprechende Stelle in dieser Tabelle aufrufen, von wo aus dann wiederum die Funktionen selbst aufgerufen werden. Abbildung 7 stellt ein Beispiel für eine solche Tabelle dar.

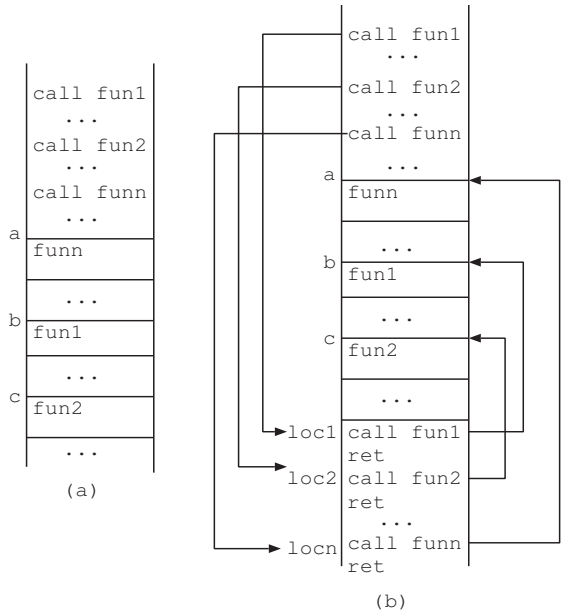


Abbildung 7: Beispielprogramm ohne(a) und mit(b) Tabelle für indirekte Funktionsaufrufe. Entnommen aus[16]

Ändert sich nun die Anfangsadresse einer Funktion, so muss nur das Ziel des Aufrufs in der Tabelle geändert werden, was nur eine Zeile im delta script ausmacht. Nach den Änderungen auf höhere Ebene, dem Erstellen und der ersten Optimierung des delta scripts führt Zephyr noch weitere Optimierungen durch. Die erste Optimierung ist das ersetzen von Anweisungsblocks bestimmter Form durch Metaanweisungen. Blöcke, bei denen sich mehrmals `COPY`-Anweisungen großer Länge mit `INSERT`-Anweisungen kleiner Länge abwechseln, werden durch `CIW(COPY_WITH_INSERTS)`-Anweisungen ersetzt, die aus einem Kopierbefehl und den Adressen und Daten für die Einfügungen bestehen. So wird das delta script um mehrere `COPY`-Anweisungen kürzer. Blöcke, in denen bestimmte Folgen mit leichten Unterschieden hintereinander eingefügt werden, werden durch `REPEAT`-Anweisungen ersetzt, die die unveränderliche und die sich ändernden Folgen mit Adressen beinhalten. So werden mehrere `INSERT`-Anweisungen eingespart. Die letzte Optimierung ist das entfernen aller `<newOffset>` Teile aller Anweisungen. Dies ist möglich, da alle Anweisungen der Reihe nach ausgeführt werden und die Länge ihrer Daten beinhalten.
Um die Reprogrammierung des Netzwerks zu starten, sendet die Basisstation eine Rebootanweisung an das Netz und ruft ihren Bootloader auf. Dieser lädt die Reprogrammierkomponente in den Programmspeicher und startet den Knoten neu. Jeder Knoten, der die Rebootanweisung erhält, sendet sie an das gesamte Netzwerk weiter und verfährt dann ebenso wie die Basisstation. Auf diese Weise wird das Netz mit der Rebootanweisung kontrolliert geflutet. Sobald alle Knoten die Reprogrammierkomponente gestartet haben, erhält die Basissation das delta script und verteilt es per Deluge beziehungsweise Stream im Netz. Dabei wird eine dynamische Seitengröße benutzt um nicht unnötig Bandbreite durch Padding zu verbrauchen. Ein Knoten, der das delta script vollständig erhalten hat, verwendet es, um aus dem Abbild des alten Programms in seinem externen Flash-Speicher das Abbild des neuen Programms zu erstellen. Abschließend wird das Netz mit einer erneuten Rebootanweisung geflutet, woraufhin alle Knoten mit dem neuen Programm rebooten. Abbildung 8 zeigt die Belegung des Speichers eines Knotens während der Reprogrammierung.

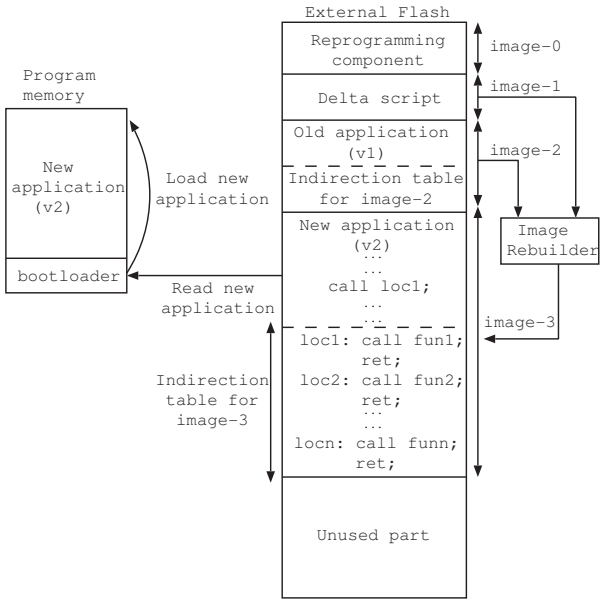


Abbildung 8: Belegung des Speichers eines Knotens während der Reprogrammierung mit Zephyr. Entnommen aus[16]

Um die Performanz von Zephyr zu testen, wurden in [16] reale und simulierte WSNs aus Mica2-Knoten verwendet. Die realen Netze waren einerseits 2×2, 3×3 und 4×4 große Gitter und andererseits Ketten von 2 bis 10 Knoten Länge. Die mit TOSSIM simulierten Netze waren Gitter der Größe 6×6 bis 14×14. Als Testfälle wurden Änderungen an den bei TinyOS mitgelieferten Standardprogrammen

doi: 10.2313/NET-2012-08-2_03

wie CntToLeds, das einen internen Zähler über die Leds des Kontens ausgibt, und an dem real verwendetem Programm eStadium[20] mit unterschiedlicher Tragweite, von einzelnen neuen Parametern bis zum Austausch des kompletten Programms, verwendet. Alle Netze wurden mit allen Testfällen reprogrammiert. Jede Reprogrammierung wurde mit Deluge, Stream, dem einfachen Rsync-Algorithmus, dem optimierten Rsync-Algorithmus ohne Änderungen auf höherer Ebene und Zephyr durchgeführt. Gemessen wurden jeweils die Zeit zum Reprogrammieren des gesamten Netzes und die Menge der insgesamt versendeten Pakete. Zephyr stellt sich als wesentlich schneller als die anderen Ansätze heraus. Um wie viel schneller Zephyr ist, hängt von der Tragweite der Änderungen am Programm ab. Abbildung 9 zeigt die benötigte Zeit zur Reprogrammierung und die Menge der gesendeten Pakete für eine möderate Änderung an eStadium in den simulierten WSNs. Das Verhältniss der Performanz zwischen den Verfahren ist in allen Fällen ähnlich, jedoch unterscheidet sich der maximale Vorteil von Zephyr stark. So ist Zephyr in der Simmulation mit moderaten Änderungen am Programm bis zu 92,9 mal schneller als Deluge, während es in realen Netzen bei sehr großen Änderungen im Durchschnitt nur 2,1 mal so schnell ist. Auffällig ist, das Zephyr gegenüber dem optimierten Rsync-Algorithmus bei größeren Änderungen schneller wird, während bei allen anderen Verfahren das Gegenteil der Fall ist. Die Zahl der versendeten Pakete ist bei Deluge zwischen 2,5 und 146 mal so groß wie bei Zephyr.

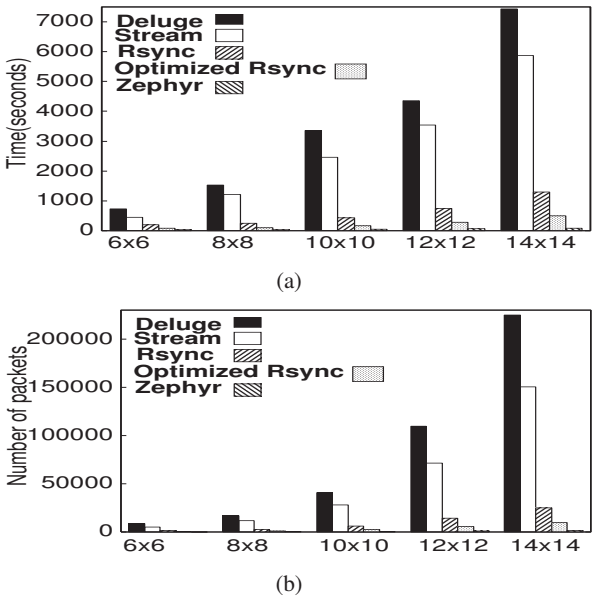


Abbildung 9: Reprogrammierungszeit(a) und Anzahl der gesendeten Pakete(b) bei moderater Änderung an eStadium in den simulierten Netzen. Entnommen aus[16]

4.5 Selective Reprogramming Scheme von Krüger, Pfisterer und Buschmann

In einem WSN läuft nicht zwangsläufig das selbe Programm auf jedem Knoten. Zum Beispiel ist es nicht nötig, Knoten am Rand des Netzes mit Aggregationsfunktionen auszustatten. Neue Versionen dieser Funktionen müssen so auch nur an bestimmte Knoten verteilt werden. Ein neues Protokoll, dass diese Reprogrammierung einer Untergruppe von Knoten eines WSNs unterstützt ist das von Krüger, Pfisterer und Buschmann an der Universität Lübeck entwickelte Selective Reprogramming Scheme[21].
Das Protokoll unterteilt die Reprogrammierung in vier Phase: Die erste ist die Node-Discovery-Phase, in der die zu reprogrammierenden Knoten ausgewählt und ein Routing-Baum zu ihnen eingerichtet wird. Die zweite ist die Code-Desssemination-Phase, in der das neue Programm an die ausgewählten Knoten verteilt wird. In der dritten Phase, Request Missing Parts, fragen die Knoten nach den nicht erhaltenen Programmteilen. Die letzte Phase ist Finalization and Reboot, in der die Knoten mit dem neuen Programm neu gestartet werden.
Die Node-Discovery-Phase läuft wie folgt ab: Die Basisstation flutet das Netzwerk mit einem Weckruf, so dass alle Knoten aktiv werden. Der Weckruf beinhaltet den Zeitpunkt, an dem der nächste Schritt durchgeführt wird. Dieser Schritt besteht darin, dass die Basisstation ein Präsenzerkennungspaket versendet, mit dem ein Routing-Baum im Netz aufgebaut wird. Ein Knoten, der das Präsenzerkennungspaket erhält, sendet ein Präsenzantwortpaket mit seiner ID über den Routing-Baum an die Basisstation. Dabei werden auf der Verbindungsschicht Acknowledgements verwendet, um alle Antwortpakete zuverlässig zu übermitteln. Die IDs in den Antwortpaketen werden an jedem Hop aggregiert. Erhält die Basisstation keine Antwort, so wird das Präsenzerkennungspaket erneut gesendet. Sobald die Basisstation über die IDs aller Knoten verfügt, werden die zu reprogrammierenden Knoten ausgewählt. Die Basisstation flutet das Netz mit einem Selektionsanzeigepaket mit ihren IDs. Die ausgewählten Knoten senden jeder ein Selektionsbestätigungspaket über den Routing-Baum zurück. Jeder dabei beteiligte Knoten wird zu einem Weiterleitungsknoten. Sind alle Selektionsbestätigungspakete an der Basisstation angekommen, so beginnt die Code-Desssemination-Phase.
In der zweiten Phase wird das Netzwerk mit dem neuen, in Segmente unterteilten Programm probabilistisch geflutet. Das bedeutet, dass jeder Knoten ein neu erhaltenes Paket nicht wie beim einfachen Fluten immer per Broadcast im Netzwerk verteilt, sondern nur mit einer bestimmten Wahrscheinlichkeit. Jedes Paket enthält neben seinem Segment auch die Gesamtzahl alle Segmente. Anhand dieser Information und dem Zeitunterschied zwischen der Ankunft der Pakete schätzt jeder Knoten die restliche Dauer dieser Phase bei jedem erhaltenen Paket neu. Nach Ablauf der geschätzten Zeit gehen alle Knoten in die dritte Phase über.
In der Request-Missing-Parts-Phase überprüft jeder Knoten, welche Segmente er erhalten hat und welche nicht. Daraufhin sendet er eine Nachricht mit einem Vektor mit dieser Information an alle erreichbaren Knoten. Nachdem er die entsprechenden Nachrichten der Nachbarknoten erhalten hat, sendet er zufällig bis zu 15 Segmente, die er hat und die mindestens ein Nachbar braucht an das Netz. Dieses abwechselnde Senden des Vektors und der Segmente wird wiederholt, bis der Knoten alle bei den Nachbarn verfügbaren Segmente erhalten hat. Die noch fehlenden Segmente werden über den Routing-Baum bei der Basisstation angefordert. Über den Routing-Baum wird die Basisstation auch über den die in jedem ausgewählten Knoten vorhandenen Segmente informiert. Sobald die Basisstation weiß, dass alle ausgewählten

Knoten das gesamte Programm haben, beginnt Phase vier. In der letzten Phase, Finalization and Reebot, flutet die Basisstation das Netz mehrmals mit Rebootpaketen. Jeder ausgewählte Knoten, der ein solche Paket erhält, lädt das neue Programm und startet sich neu.
Das Selective Reprogramming Scheme wurde in[21] evaluiert. Dabei wurde ein reales WSN aus 22 iSense-Knoten[22] und mehrere mit Shawn[23] simmulierte Netze aus den gleichen Knoten der Größen 15, 20×20 und 30×30 verwendet. Da die iSense-Knoten in verschieden Konfigurationen mit stark unterschiedlichen Leistungswerten vorkommen[24, 25] und die Konfiguration in[21] nicht angegeben ist, sind die Ergebnisse schlecht mit den anderen Datendisseminationsprotokollen in dieser Arbeit vergleichbar. Außerdem wurden kaum mit den Test der anderen Protokolle vergleichbare Werte gemessen. Hier werden nur die für den Vergleich interessanten Messungen aufgeführt. Für den Test wurde jedes Netzwerk mit einem Programm aus 1200 Paketen nicht näher definierter Größe aufgeteilt auf 600 Segmente reprogrammiert. Es wurden verschieden Algorithmen für das Senden der Pakete der Knoten entlang des Routing-Baums verglichen. Auf diese Algorithmen wird hier nicht näher eingegangen.
Bei allen Tests war die Reprogrammierung aller Knoten erfolgreich. Dauer und Anzahl gesendeter Pakete in Phase 1 abhängig von Algorithmus und Anteil der ausgewählten Knoten am Netzwerk wurden im realen WSN gemessen. Die Zeitdauer dieser Phase stieg kaum mit dem Anteil am Netzwerk, sie lag bis auf wenige Ausreisser unter 625 ms. Die Menge an der Basisstation empfangenen Pakete stieg mit dem Anteil der ausgewählten Knoten von 2 auf maximal 10. Eine weitere interessante Vergleichsgröße ist die Dauer von Phase 3 im simulierten 15 Netzwerk und die Anzahl der dabei versendeten Nachrichten, beide abhängig vom verwendeten Algorithmus und dem Anteil der ausgewählten Knoten am Netz. Bei 5% Anteil der ausgewählten Knoten betrug die Dauer beim schnellsten Algorithmus 50 Sekunden, beim langsamsten 175 Sekunden. Bei steigendem Anteil näherten sich die beiden Dauern an bis sie bei 100% beide bei 135 Sekunden lagen. Die Menge der gesendeten Nachrichten lag beim langsamsten Algorithmus bei 5% Anteil bei 50000, während er beim schnellsten bei 150000 lag. Mit steigendem Anteil näherten sich beide Mengen wieder an und stiegen beide bis auf 375000 bei 100% Anteil. Aus diesen Messwerten kann der Overhead wie folgt berechnet werden:
$$\frac{\#gesendete\ Pakete - 225 \times Anteil\ ausgewaehlte\ Knoten \times 1200}{\#gesendete\ Pakete}$$
Der Overhead ist also im ungünstigsten Fall 90%, im günstigsten 18%.

5. ZUSAMMENFASSUNG

In dieser Arbeit wurde ein Überblick über die Reprogrammierung von drahtlosen Sensornetwerken gegeben. Zuerst wurde der grundlegende Aufbau eines drahtlosen Sensornetzwerks beschrieben. Daraufhin wurden die Handhabung der Reprogrammierung auf den einzelnen Knoten durch die Betriebssysteme TinyOS und Contiki vorgestellt. TinyOS macht es erforderlich, bei jeder Änderung das gesamte Programm des Knotens auszutauschen, während Contiki Teile des Programms ohne Neustart des Knotens austauschen kann. Anschließend wurden die für die Verteilung des neuen Programms im Netzwerk verantwortlichen Datendisseminationsprotokolle XNP, Trickle, Deluge, Zephyr und das Selective Reprogramming Scheme von Krüger, Pfisterer und Buschmann im einzelnen beschrieben und deren Performanz soweit möglich miteinander verglichen.
XNP ist ein relativ einfaches Protokoll, das eine rudimentäre Unterstützung der drahtlosen Reprogrammierung für alte Versionen von TinyOS darstellte. Trickle verwendet den "polite gossip"-Ansatz, um einzelne neue Programmteile zu verteilen. Die Knoten versenden dabei möglichst nur neue Informationen über den Stand des Programms. Wird bei einem Nachbarknoten ein alter Stand festgestellt, so wird er sofort auf den neuseten gebracht. Deluge baut auf Trickle auf und erweitert es um die Unterstützung für die Verteilung größerer Datenmengen. Einzelne Pakete des neuen Programms werden hier gezielt angefordert. Deluge unterstützt auch das Pipelining bei der Verteilung des neuen Programms und kann gegenüber Trickle eine höhere Veteilungsgeschwindigkeit vorweisen. Zephyr baut wiederum auf Deluge auf und versendet keine ganzen Abbilder des neuen Programms sondern nur delta scripts, mit denen das neue Abbild aus dem alten erstellt werden kann. Zephyr verwendet Ansätze aus dem Protokoll Stream, eine optimierte Version des Rsync-Algorithmus und Änderungen des Programmabbildes auf höherer Ebene um die Größe der delta scripts stark zu verringern. Durch die geringe zu sendende Datenmenge hat Zephyr gegenüber Deluge einen starken Geschwindigkeitsvorteil. Das Selective Reprogramming Scheme schließlich dient dazu, nur Teile des Netzwerks zu reprogrammieren. Dazu wählt es zuerst die zu reprogrammierenden Knoten aus und baut einen Routing-Baum mit ihnen auf. Das Netz wird mit den neuen Paketen geflutet, die Rückmeldungen erfolgen über den Routing-Baum. Das Selective Reprogramming Scheme ist wegen seines anderen Ansatzes und der verwendeten Testmethoden schlecht mit den anderen Protokollen vergleichbar. Fast alle Protokolle konnten alle Knoten eines Netzwerks zuverlässig reprogrammieren, Fehler gab es nur in geringem Ausmaß bei XNP.
Insgesamt ist zu sehen, dass sich der Funktionsumfang und die Performanz von Datendisseminationsprotokollen im Laufe der Zeit stetig verbessert haben. Beides ging jedoch stets mit einer Steigerung bei der Komplexität und dem Ressourcenbedarf (Rechenzeit, Speicherplatz auf dem Knoten) einher. Für reale Sensornetze erscheint es daher sinnvoll, ein Datendisseminationsprotokoll zu verwenden, das die Anforderungen an die Reprogrammierung des Netwerks erfüllt und dabei selbst möglichst simpel und Ressourcensparend ist. So genügt es, für Netzwerke mit nur einem Hop und wenigen Knoten, XNP zu verwenden. Bei größeren Netzen ist mindestens Trickle erforderlich, da jedoch nicht nur kleine Teile eines Programms ausgetauscht werden, wird meist Deluge notwendig. Zephyr bietet gegenüber Deluge zwar einen starke Performanzvorteil, benötigt jedoch viel Platz im externen Flash-Speicher des Knotens und kann nur verwendet werden, wenn dieser nicht anderweitig gebraucht wird. Wenn der Speicher zur Verfügung steht, rechtfertigt die höhere Performanz Zephyrs Komplexität. Das Selective Reprogramming Scheme ist komplexer als alle anderen Protokolle in dieser Arbeit und sollte nur verwendet werden, wenn die Reprogrammierung einzelner Knoten notwendig ist.

6. LITERATUR

[1] Crossbow Technology, Inc., *Mica2 Wireless Measurement System Datasheet*, 2003.

[2] Crossbow Technology, Inc., *TelosB Datasheet*, 2004.

[3] Oracle America, Inc., *Sun SPOT Main Board Technical Datasheet*, 2010

[4] Q. Wang, Y. Zhu, L. Chen *Reprogramming Wireless Sensor Networks: Challenges and Approaches*, In IEEE Network, Seiten 48-55, Mai/Juni 2006.

[5] S. Singh, M. Singh, D. Singh *Routing Protocols in Wireless Sensor Networks - A Survey* In International Journal of Computer Science & Engineering Survey, Seiten 63-83, November 2010.

[6] TinyOS Community, *TinyOS FAQ* `http://docs.tinyos.net/tinywiki/index.php/FAQ`.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler *The nesC Language: A Holistic Approach to Networked Embedded Systems*, In Proceedings of SIGPLAN'03, 2003.

[8] P. Lewis, D. Culler *Maté: A Tiny Virtual Machine for Sensor Networks*, In Proceedings of ASPLOS-X, October 2002.

[9] A. Dunkels, B. Grönvall, T. Voigt *Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors*, IEEE Emnets, Seiten 455-462, 2004.

[10] A. Dunkels *The Contiki OS* `http://www.contiki-os.org/`

[11] Crossbow Technology, Inc., *Mote In-Network Programming User Reference*, 2003.

[12] J. Jeong, S. Kim, A. Broad *Network Reprogramming*, 2003.

[13] P. Levis, N. Patel, S. Shenker, D. Culler *Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks* In Proceedings of 1st Symposium Networked System Design and Implementation, Seiten 15-28, 2004.

[14] P. Lewis, N. Lee, M. Welsh, D. Culler *TOSSIM: Simulating large wireless sensor networks of TinyOS motes* In Proceedings of the First ACM Conference on Embeded Networked Sensor Systems(SenSys 2003), 2003.

[15] J. Hui, D. Culler *The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale*, In J. Stancovic, A. Arora, R.Govindan(Editoren), SenSys, Seiten 81-94, ACM, 2004.

[16] R. Panta, S. Bagchi, S. Midkiff *Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation* In Proceedings of Proceedings of the 2009 conference on USENIX Annual technical conference, Seiten 32-32, 2009

[17] R. Panta, I. Khalil, S. Bagchi *Stream: Low Overhead Wireless Reprogramming for Sensor Networks*, IEE Infocomm, Seiten 928-936, 2007

[18] A. Trygdell *Efficient Algorithms for Sorting and Synchronization*, Dissertation, Australian National University, 2000

[19] R. Rivest, *RFC 1320*, 1992, `http://tools.ietf.org/html/rfc1320`

[20] `http://estadium.perdue.edu`

[21] D. Krüger, D. Pfisterer, C. Buschmann *Selective Reprogramming in Sensor Networks* In Proceedings of 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS),Seiten 1-5, 2011.

[22] C. Buschmann, D. Pfisterer *iSense: A modular hardware and software platform for wireless sensor networks* Technical report, 6.Fachgespräche Drahtlose Sensornetze der GI/ITG-Fachgruppe Kommunikation und Verteilte systeme, 2007.

[23] S. Fekete, A. Kröller, S. Fischer, D. Pfisterer *Shawn: The fast, highly cunstomizable sensor network simulator*, In Proceedings of the Fourth International Conference on Networked Sensing Systems (INSS'07), 2007

[24] `http://www.coalesenses.com/index.php?page=core-module-2`

[25] `http://www.coalesenses.com/index.php?page=core-module-3`